Mastering Deep Neural Network Training

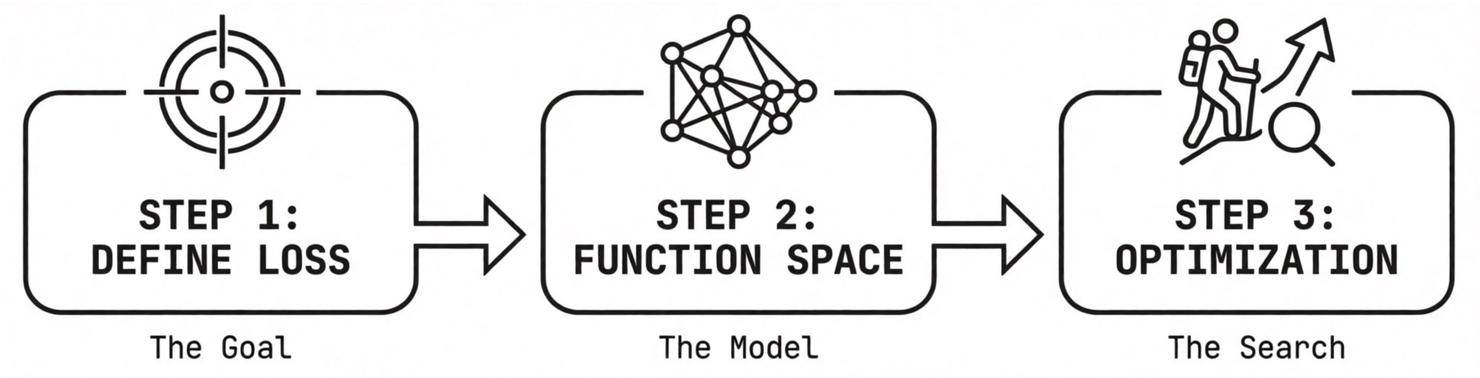
Structured Blueprint for Optimization and Generalization

AI with Deep Learning
EE4016

Prof. Lai-Man Po
Department of Electrical Engineering
City University of Hong Kong

A Three-Step Framework for Model Training

Navigating the Deep Learning Workflow



Training can be viewed as a structured search for optimal parameters θ^* that minimize a cost/loss function $\mathcal{L}(\theta)$:

1. **Define what you want**: Specify the objective function that quantifies error on training data.
2. **Explore the choices**: Define the hypothesis space by selecting a model architecture (e.g., MLP, CNN, RNN, Transformer).
3. **Pick the best**: Optimize parameters within this space, typically using gradient-based methods.

Step1: Define Objective

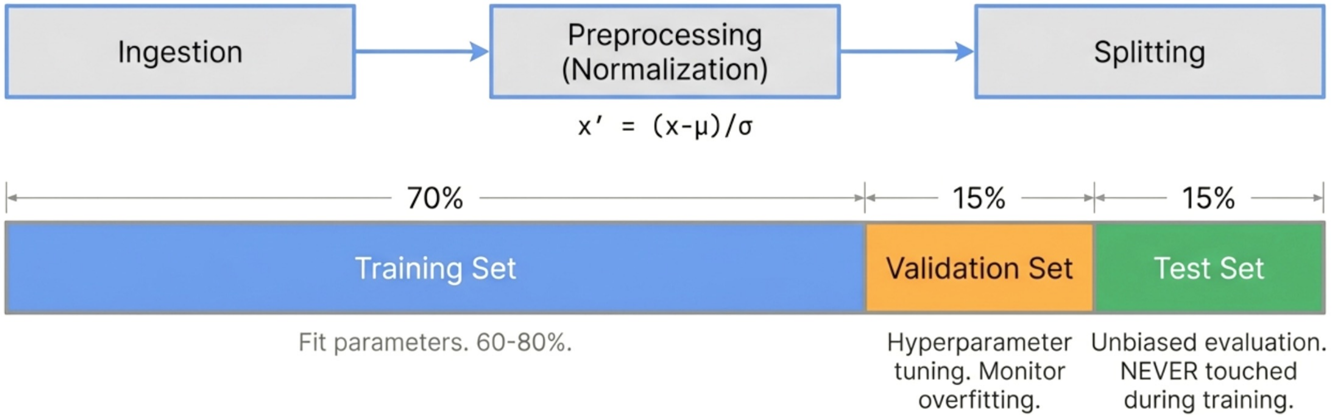
Loss Function & Data Prep

Step1: Define Objective

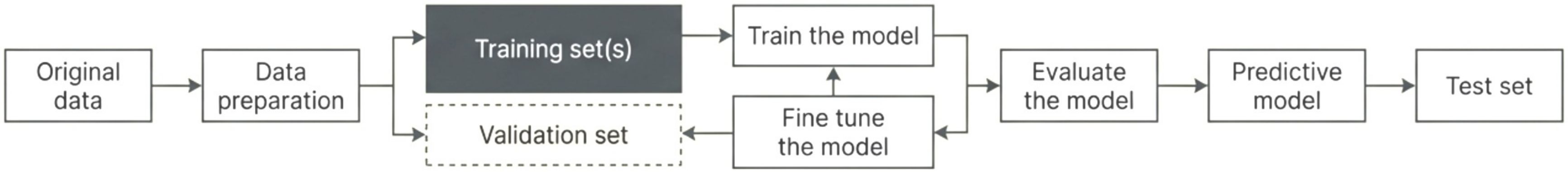
- For a given application with a dataset $\mathcal{D} := \left\{\left(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\right)\right\}_{i=1}^{N}$, specify **Loss/Cost Function** $\mathcal{L}(\theta)$ to quantify error
  - **Regression**
    - MSE Loss: $\mathcal{L}_{\text{MSE}} = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}\left\|\hat{\mathbf{y}}^{(i)} - \mathbf{y}^{(i)}\right\|_2^2 = \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{j=1}^{d}\left(y_j^{(i)} - \hat{y}_j^{(i)}\right)^2\right)$
    - MAE Loss: $\mathcal{L}_{\text{MAE}} = \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{j=1}^{d}\left|y_j^{(i)} - \hat{y}_j^{(i)}\right|\right)$
  - **Classification**
    - Binary Cross Entropy (BCE): $\mathcal{L}_{BCE} = -\sum_{i=1}^{N}\left[y^{(i)}\log\left(\hat{y}_k^{(i)}\right) + \left(1 - y^{(i)}\right)\log\left(1 - \hat{y}_k^{(i)}\right)\right]$
    - Categorical Cross Entropy (CCE): $\mathcal{L}_{CCE} = \sum_{i=1}^{N}\sum_{k=1}^{K} -y_k^{(i)}\log\left(\hat{y}_k^{(i)}\right)$

Data is the Foundation

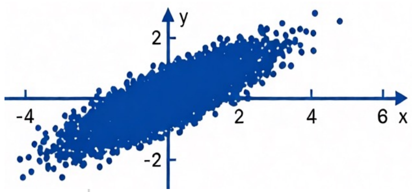
The Pipeline and The Golden Rule of Splitting



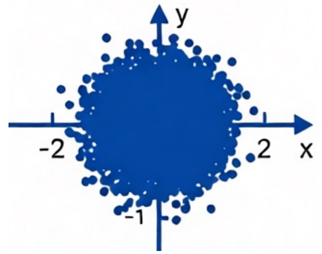
- **Ingestion & Preprocessing**: Normalization, Cropping, Filtering.
- **Strict Separation**: Validation is for tuning; Test is for unbiased evaluation.
- **WARNING**: Data leakage into the test set invalidates the entire blueprint.



```python
# Define data transformations for the training and test sets
train_transform = transforms.Compose([
    transforms.ToTensor(),  # Convert images to tensors
    transforms.Normalize((0.5,), (0.5,))])  # Normalize the image data

test_transform = transforms.Compose([
    transforms.ToTensor(),  # Convert images to tensors
    transforms.Normalize((0.5,), (0.5,))])  # Normalize the image data
```



Before Normalization)



Normalized (Mean =0, Var=1)

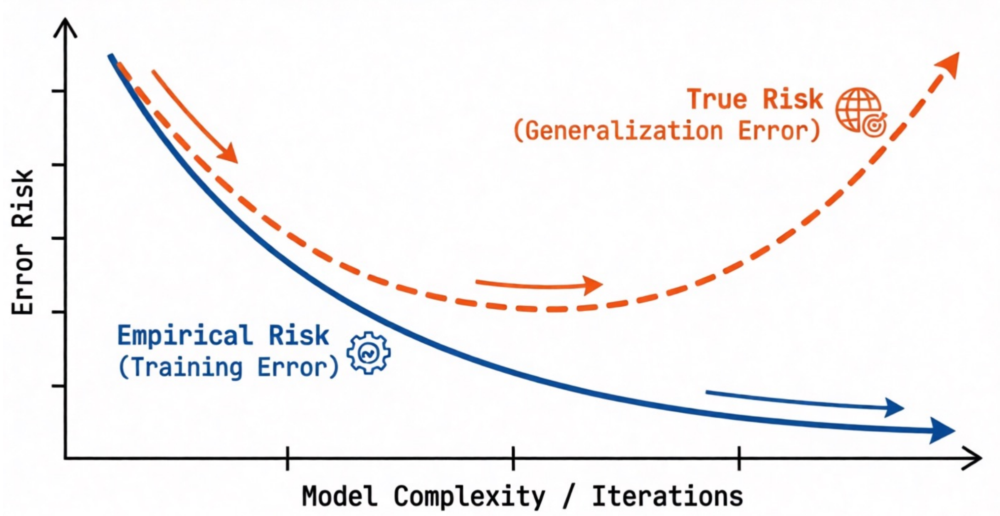
```python
# Create the Fashion MNIST dataset for the training set with 60,000 images
train_set = torchvision.datasets.FashionMNIST(root='./data', train=True, download=True, transform=train_transform)

# Create the Fashion MNIST dataset for the test set with 10,000 images
test_dataset = torchvision.datasets.FashionMNIST(root='./data', train=False, download=True, transform=test_transform)

# Split the original test set into a validation set with 5,000 samples and a test set with 5,000 samples
val_set, test_set = torch.utils.data.random_split(test_dataset, [5000, 5000])
```

Data Splitting

```python
# Define the data loaders for the training, validation, and test sets
train_loader = torch.utils.data.DataLoader(train_set, batch_size=256, shuffle=True, num_workers=2)
val_loader = torch.utils.data.DataLoader(val_set, batch_size=256, shuffle=False, num_workers=2)
test_loader = torch.utils.data.DataLoader(test_set, batch_size=256, shuffle=False, num_workers=2)

# Define the classes for the Fashion MNIST dataset
classes = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

The Two-Front War

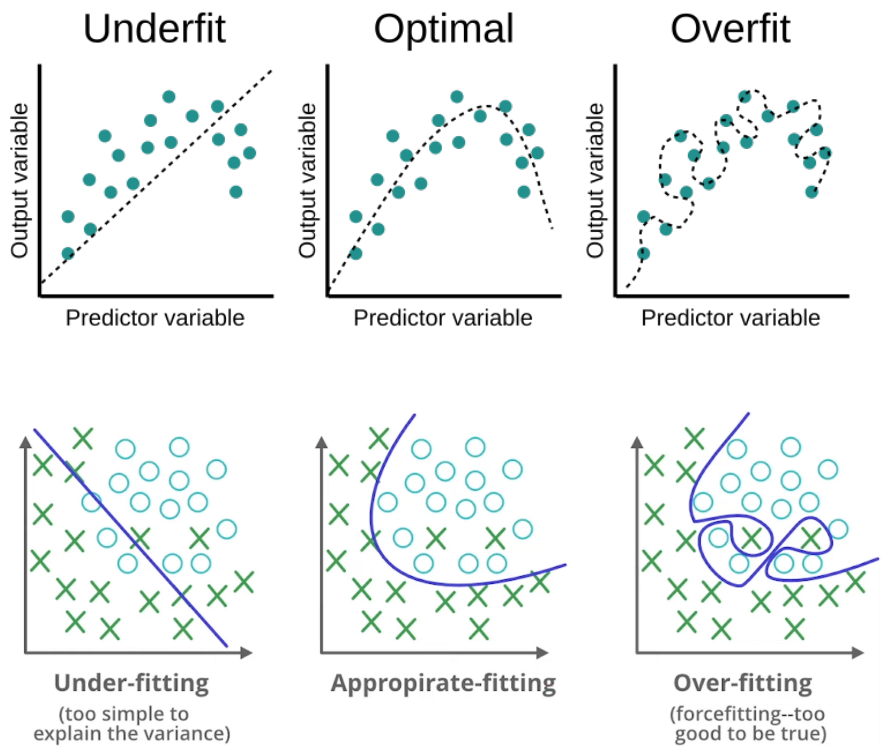
- Deep learning optimization is not a simple descent into a valley. It is navigation through a high-dimensional, non-convex landscape dominated by saddle points and plateaus.
- We face a two-front war: minimizing Empirical Risk **(the training error)** while simultaneously minimizing True Risk **(the generalization error).**



Minimizing the training error does not guarantee that we find the best set of parameters to minimize the generalization error.

Underfit vs Overfit

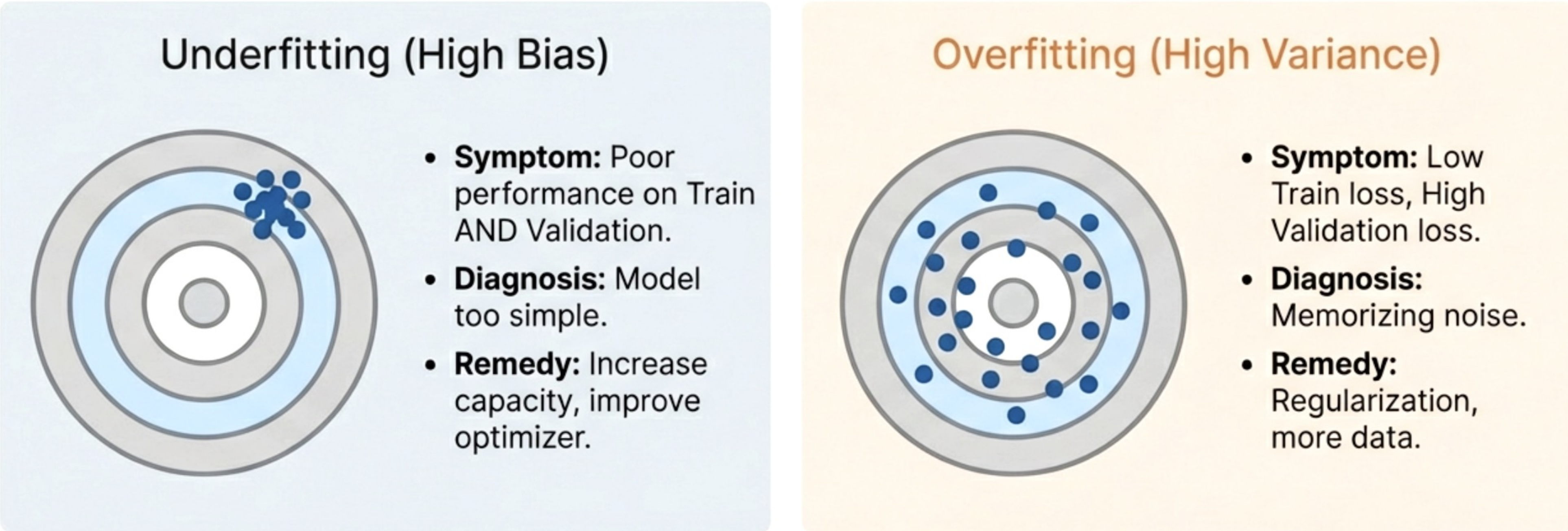
- **Underfitting (High Bias):**
  - Model is too simple. Fails to capture structure.
  - Symptom: Poor Training & Validation Performance.
- **Overfitting (High Variance):**
  - Model memorizes noise.
  - Symptom: Low Training Loss, Degrading Validation.
- **The Goal:**
  - Minimize Total Error (Bias + Variance).



Key Insight: Most training techniques are attempts to shift the balance of Bias and Variance in a controlled manner at one of these three stages.

Diagnosing Failure

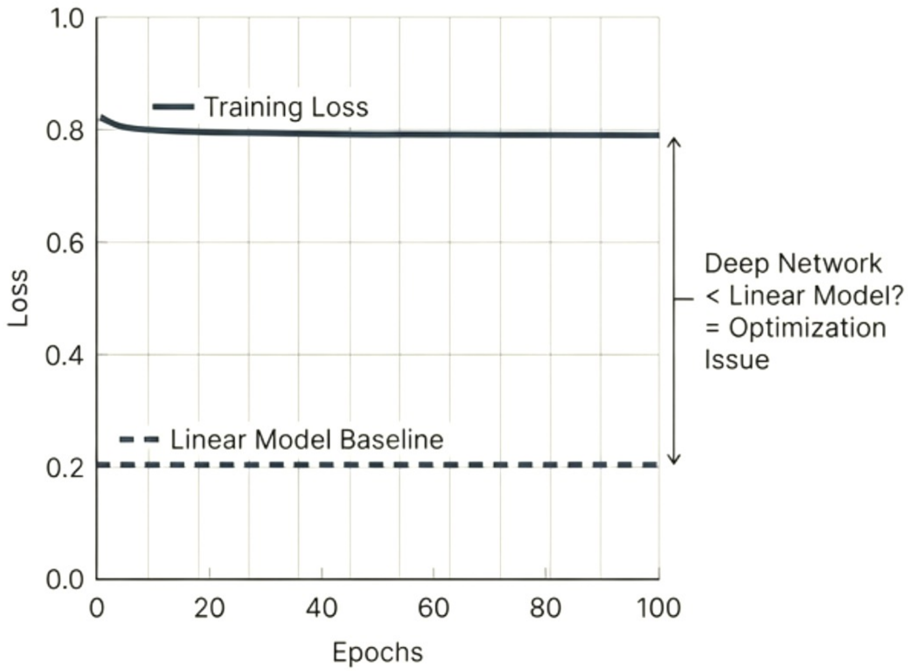
The Bias-Variance Tradeoff



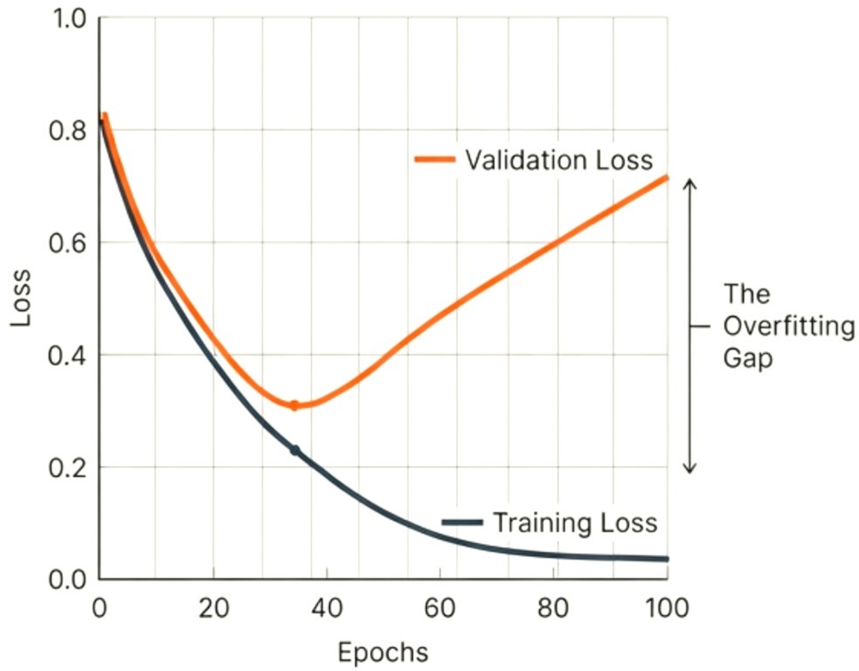
The Sweet Spot: Minimizing Total Error (Bias + Variance)

Diagnosing the Villain

CASE A: OPTIMIZATION FAILURE



CASE B: GENERALIZATION FAILURE



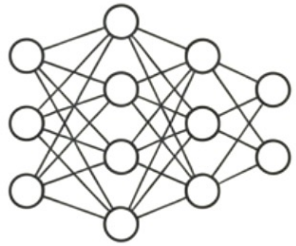
Step 2: Defining the Hypothesis Space

Architecture Design to match the Data Types

Step 2: Architecture & Inductive Bias

Selecting the right hypothesis space for the data

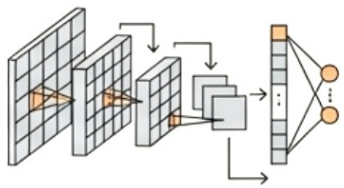
MLP (Multi-Layer Perceptron)



Inductive Bias:
Independence

Use Case:
Tabular Data

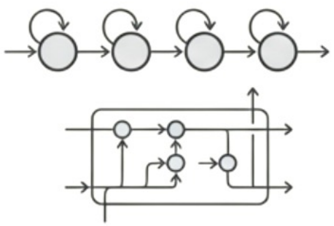
CNN (Convolutional Network)



Inductive Bias:
Spatial Locality & Invariance

Use Case:
Image Data

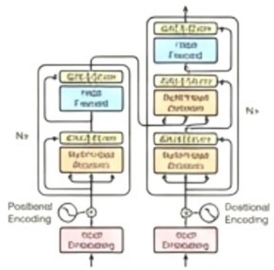
RNN / LSTM



Inductive Bias:
Sequentiality

Use Case:
Time-series, Sequence Data

Transformer



Inductive Bias:
Global Context (Self-Attention)

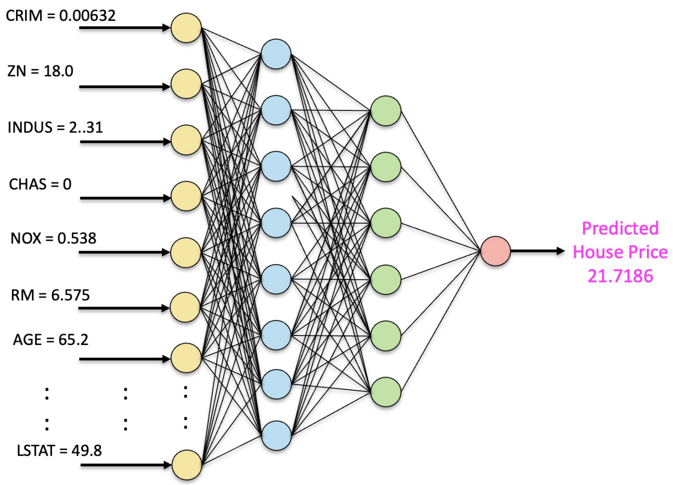
Use Case:
NLP, Seq2Seq, Vision

Data Types and Neural Network Architectures

- **Tabular Data**: Initially, the focus was on utilizing **Multilayer Perceptrons (MLPs)** to process tabular data. This approach evolved into deep learning, increasing the model's capacity to capture complex patterns by adding more layers.
- **Image Data**: **Convolutional Neural Networks (CNNs)** emerged to interpret and analyze visual information in grid formats, outperforming MLPs.
- **Sequential Data**: Sequences with meaningful order (e.g., textual or time-series data) require specialized models, which led to the development of Recurrent Neural Networks (RNNs), which can model and learn from sequential patterns.
- **Seq2Seq Data**: Specialized architectures were created to handle sequence-to-sequence data, such as machine translation tasks, due to the complexities involved in aligning variable-length input and output sequences.

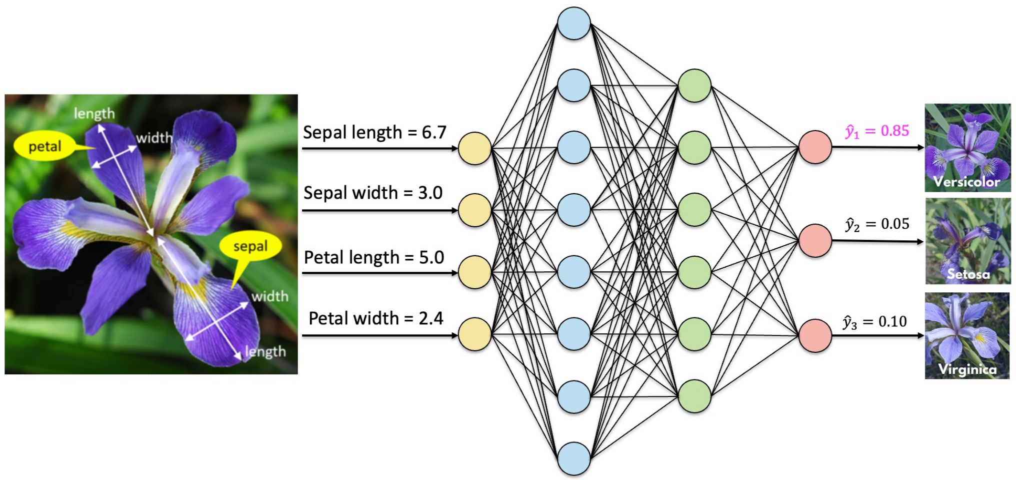
MLPs for Simple Regression and Classification

Regression



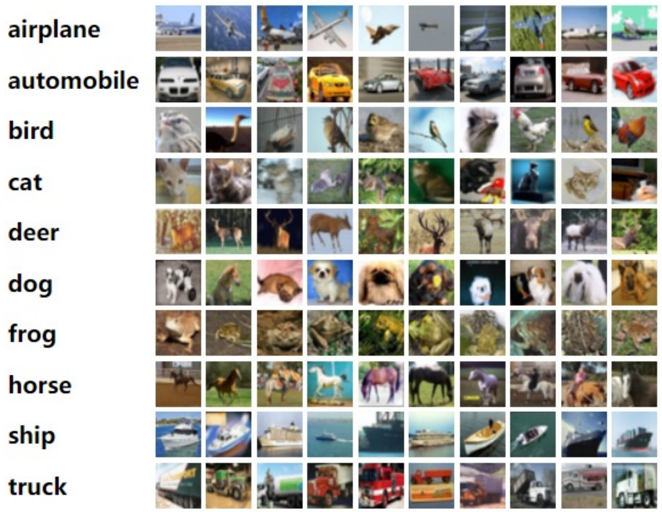
- **Boston Housing Dataset**
  - 13 features and **506** records
  - A 3-Layer MLP (13-8-6-1)
  - **Cost Function: MSE**
  - Performance: **RMSE = 3.97**

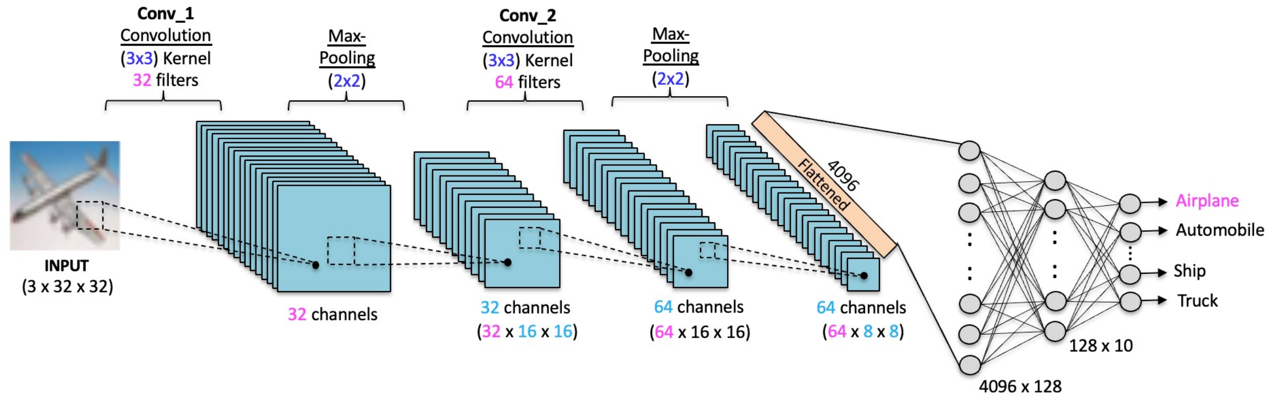
Classification



- **Iris Flower Dataset**
  - 4 features and **150** records
  - A 3-Layer MLP (4-8-6-3)
  - **Cost Function: CCE**
  - **Performance: 98% Accuracy**

CNN for CIFAR-10 Image Classification





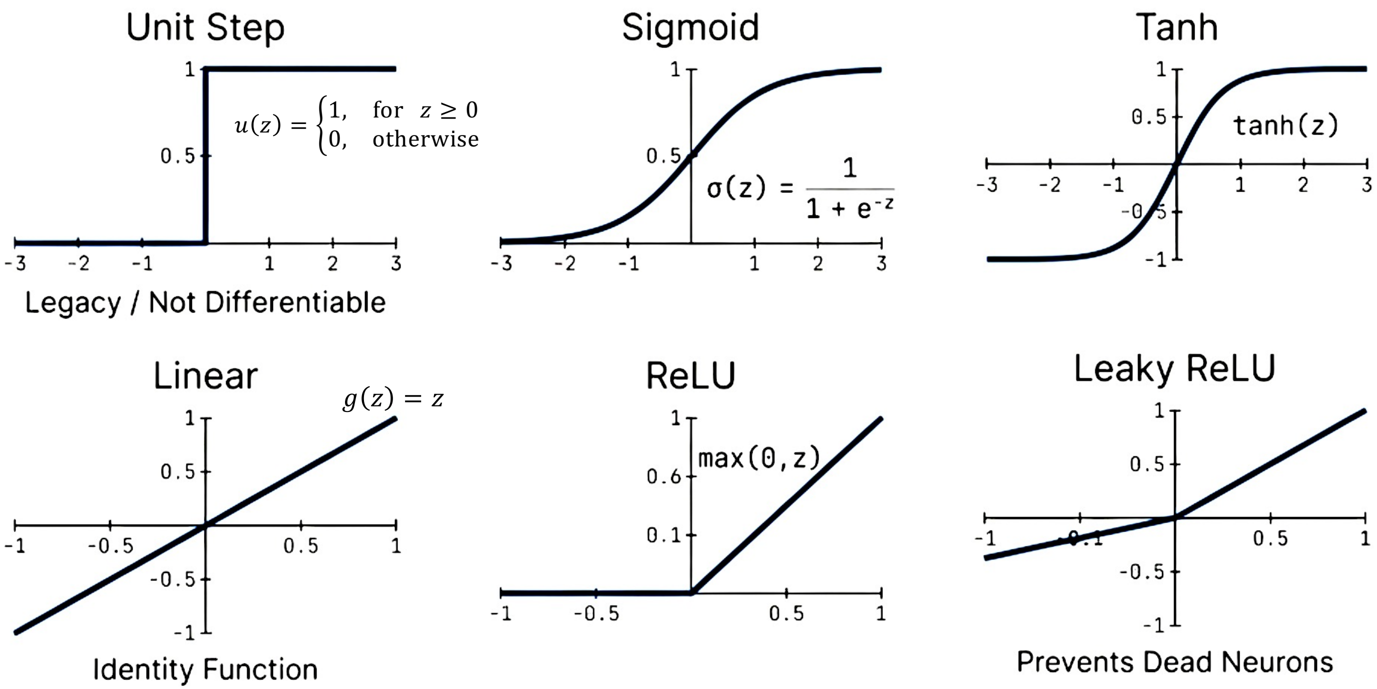
- **CIFAR-10 Color Image Dataset**
  - 60,000 32x32x3 RGB-Color Images
  - 5-Layer CNN (3x3-32, 3x3-64, 128-10)
  - **Cost Function: CCE**
  - **Performance: 78% Accuracy**

A simple Convolutional Neural Network (CNN) can achieve **70-80% accuracy**. State-of-the-art is **above 97%.**

Micro-Architecture: Activation Function

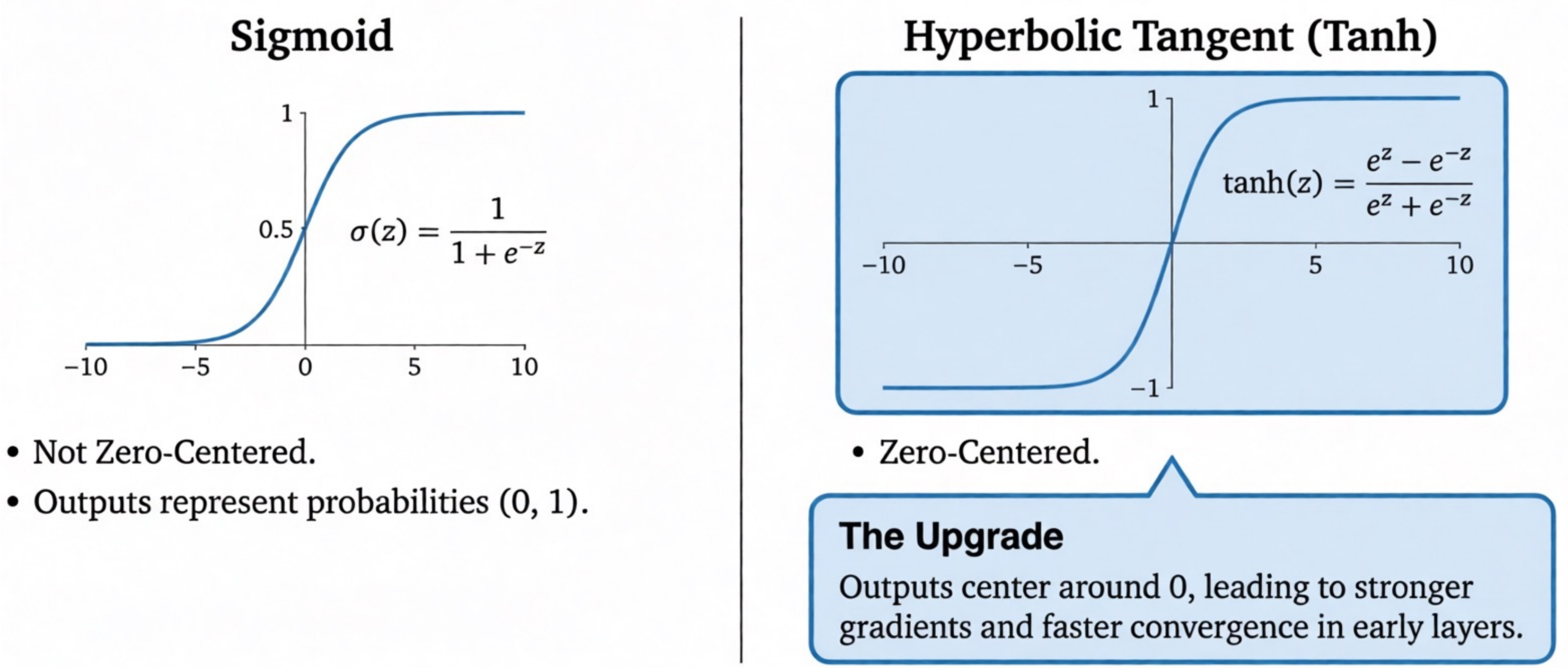
The Engine of Non-Linearity

- Activation functions decide whether a neuron 'fires". They **introduce non-linearity,** preventing the network from collapsing into a simple linear regression.



The Differentiable Era

Smoothing the Curve (1980s – 1990s)



These smooth curves enabled the first generation of functional Multi-Layer Perceptrons via Backpropagation

Derivative of the Sigmoid Function $\sigma'(z)$

$$\sigma'(z) = \frac{d}{dz}\sigma(z) = \frac{d}{dz}\left[\frac{1}{1+e^{-z}}\right]$$

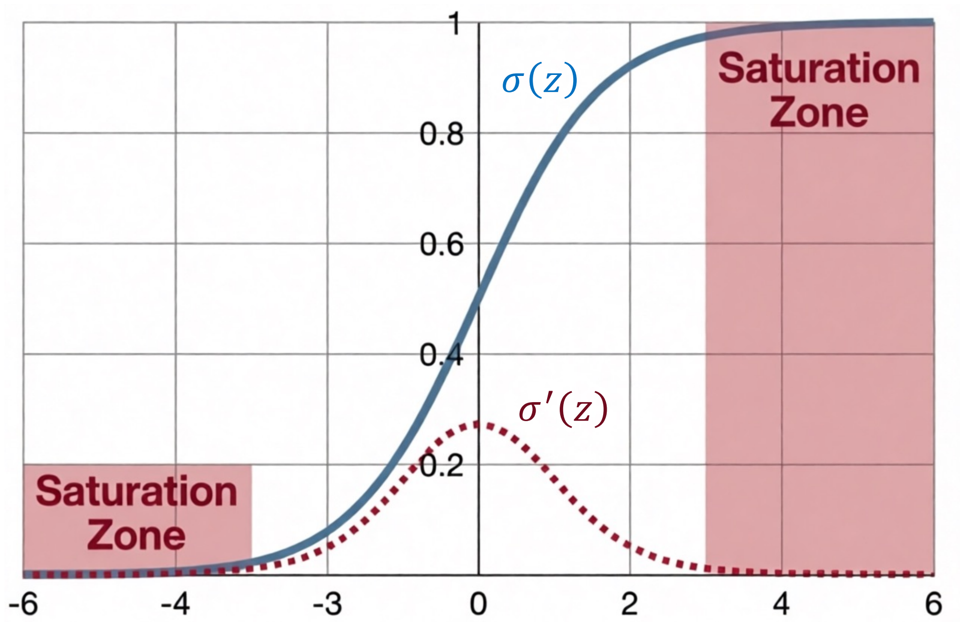
$$= \frac{d}{dz}(1+e^{-z})^{-1} = -(1+e^{-z})^{-2}(-e^{-z})$$

$$= \frac{e^{-z}}{(1+e^{-z})^2} = \frac{1}{1+e^{-z}} \cdot \frac{e^{-z}}{1+e^{-z}}$$

$$= \frac{1}{1+e^{-z}} \cdot \frac{(1+e^{-z})-1}{1+e^{-z}}$$

$$= \frac{1}{1+e^{-z}} \cdot \left(\frac{1+e^{-z}}{1+e^{-z}} - \frac{1}{1+e^{-z}}\right)$$

$$= \frac{1}{1+e^{-z}} \cdot \left(1 - \frac{1}{1+e^{-z}}\right) = \boxed{\sigma(z) \cdot (1-\sigma(z))}$$

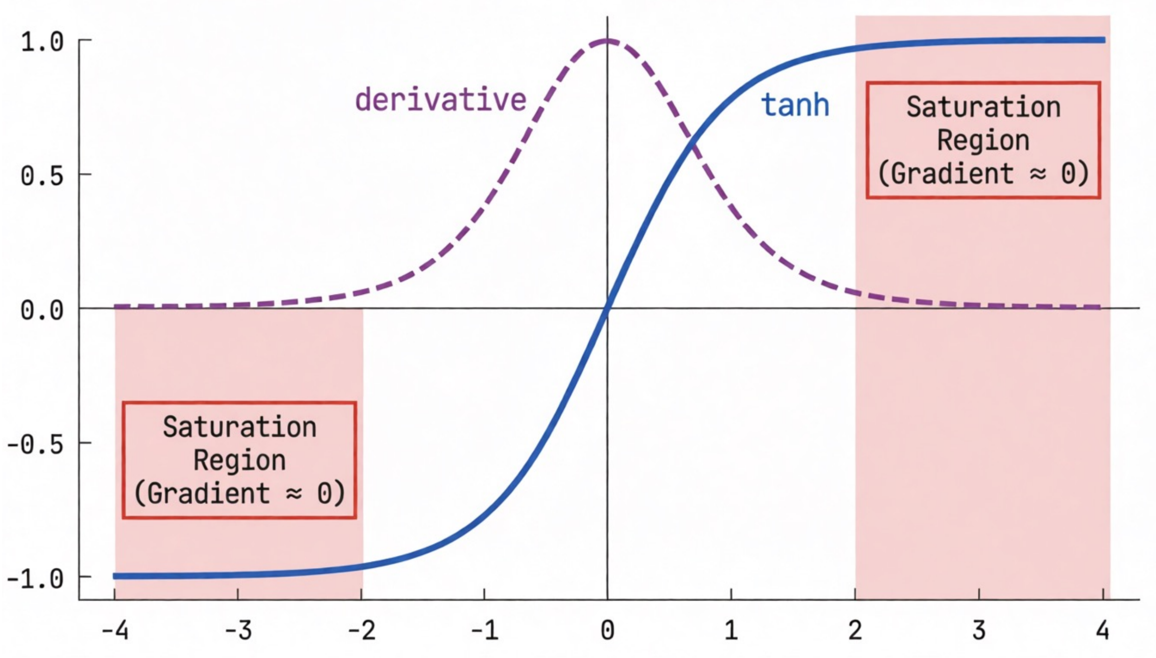


Vanishing Gradients

At extreme input values, the curve flattens. The derivative approaches zero. As these tiny gradients **multiply backward through layers, the signal disappears, and the network stops learning**.

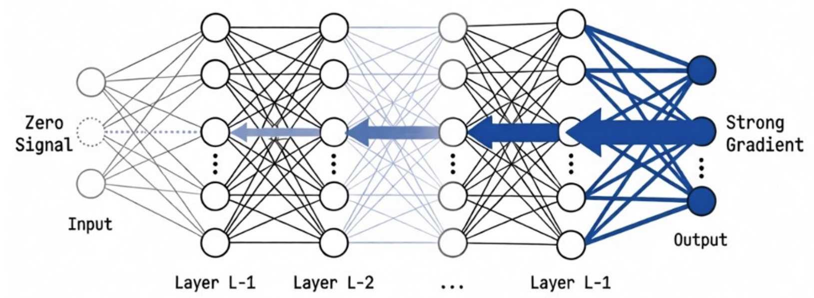
The Saturation Crisis of tanh

The Vanishing Gradient Problem



THE MECHANISM

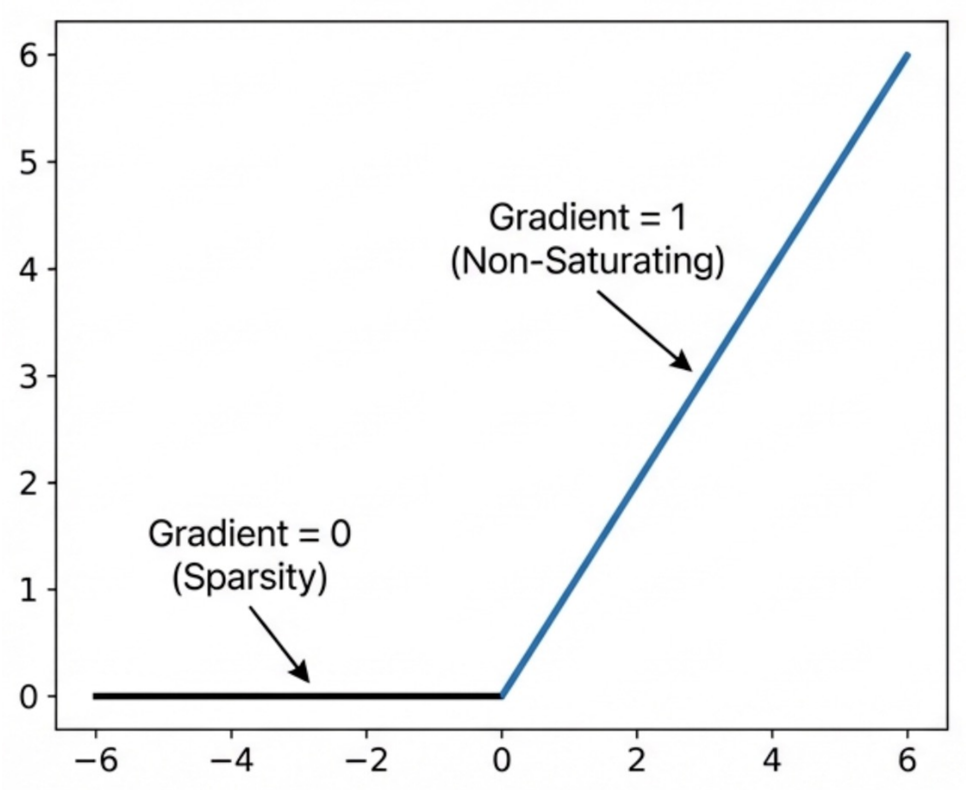
- When inputs are large/small, the curve flattens.
- The slope (gradient) becomes near-zero.
- Result: Error signals "vanish" during backpropagation. Deep Network stop to learn.



Tanh is just a scaled Sigmoid. It still suffers from saturation and vanishing gradients in deep networks.

The ReLU Revolution (2010s)

Abandoning the curve to solve the depth problem



The Hockey Stick Shape

ReLU(z) = max(0, z)

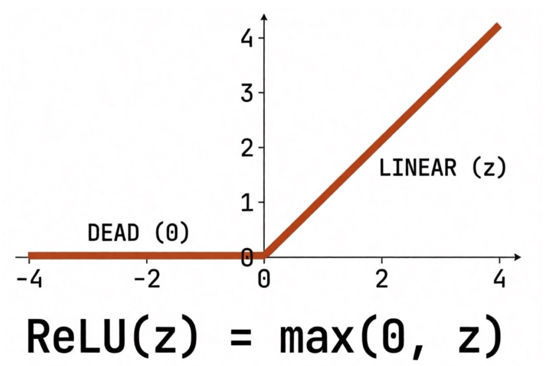
Wins

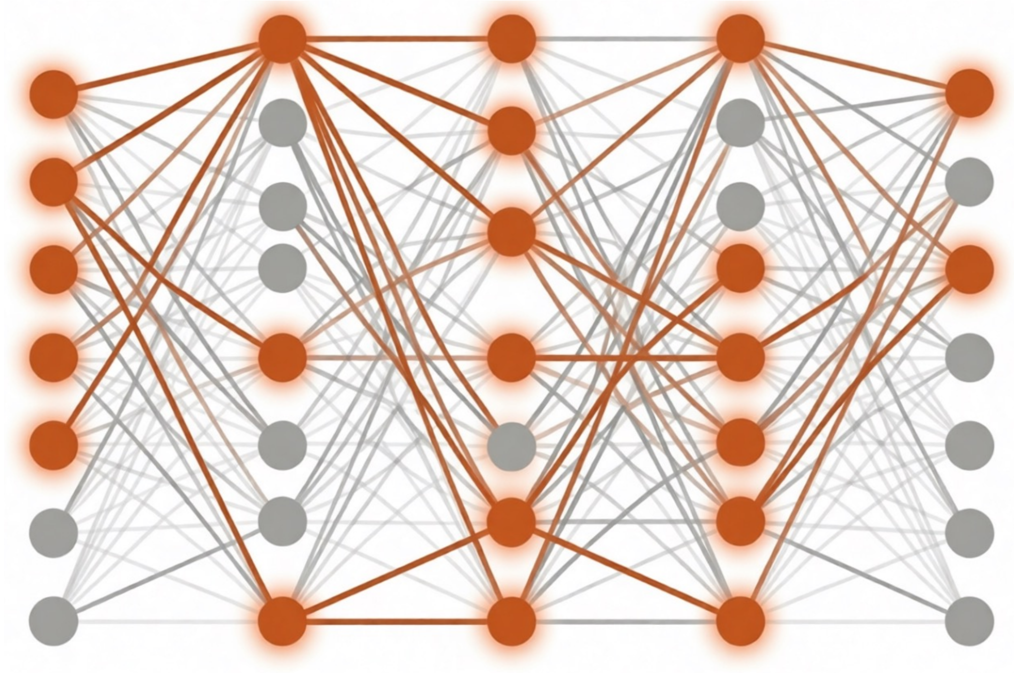
1. Non-saturating: Positive inputs never cause vanishing gradients. Allowed AlexNet (2012) to train deep models.
2. Sparsity: Zeros out negative inputs, making the network computationally efficient.
3. Speed: Computational Cheap

The New Flaw: DYING ReLU

Dying ReLU: If inputs are negative, the gradient is 0.

- A neuron can get stuck in this 'off' state and never learn again.

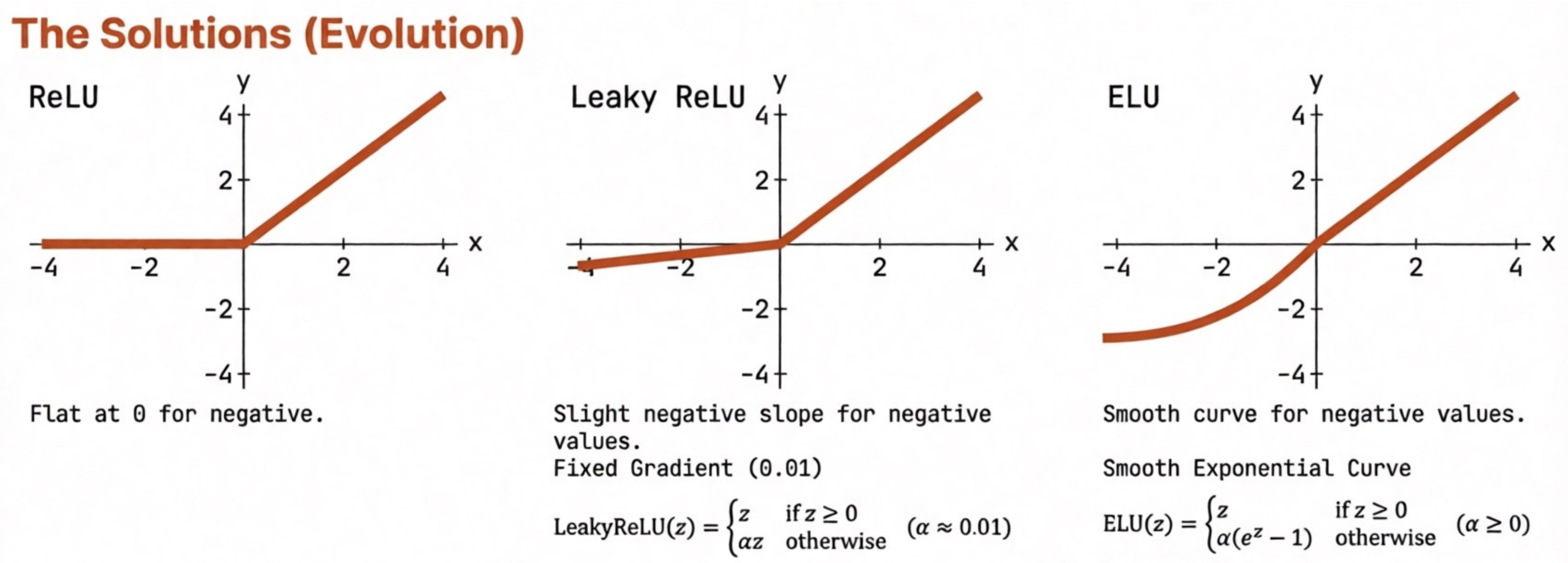




Fixing the "Dying ReLU"

Leaking Information on Purpose

Neurons stuck in the negative range have a gradient of 0 and never update again.



GELU (Gaussian Error Linear Unit, 2016)

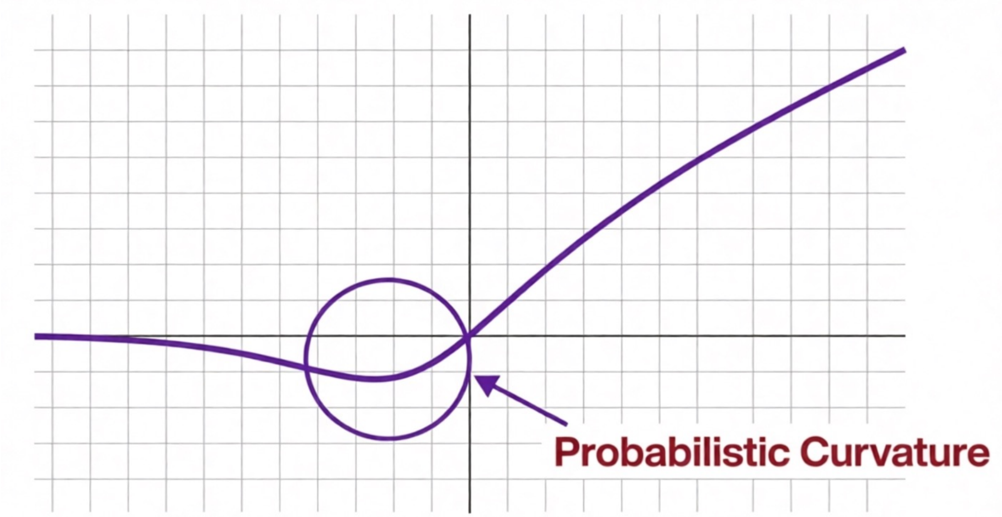
The Transformer Standard

The Probabilistic Switch

$$\mathrm{GELU}(z) = z \cdot \Phi(z)$$

where $\Phi(z)$ is the cumulative distribution function (CDF) of the standard normal distribution $N(0,1)$.

- GELU can be viewed as a **smoother version of ReLU** that also incorporates stochastic regularization



Used in BERT & GPT. Shifts from binary thresholding to weighting inputs by their magnitude relative to a Gaussian distribution.

"Gaussian Error Linear Units (GELUs)" by Dan Hendrycks and Kevin Gimpel.

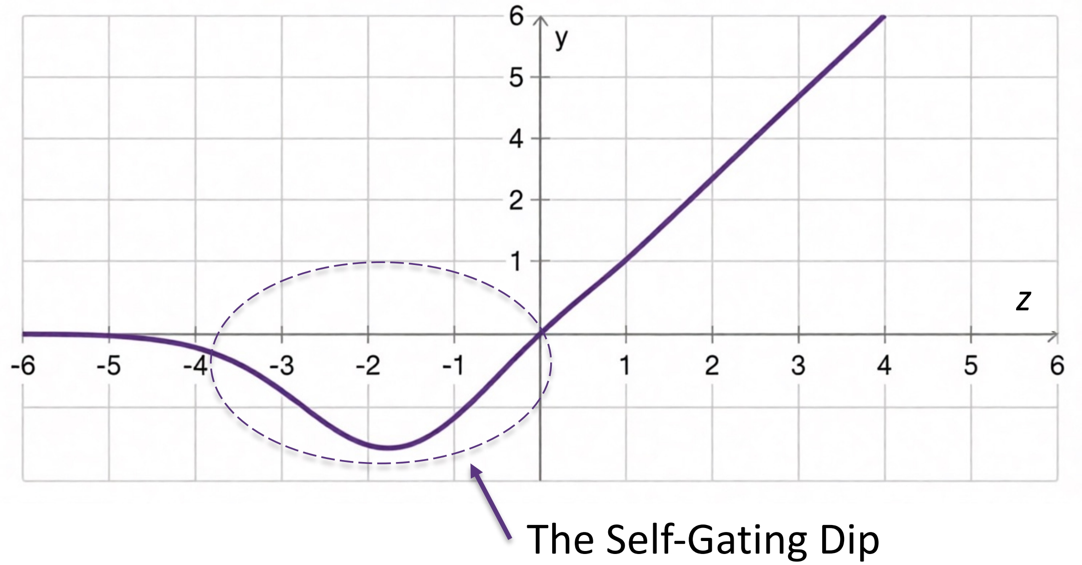
Swish and SiLU (2017)

The Power of Non-Monotonicity

- Discovered by **Google Brain** is automated search. The function is "self-gated," allowing the input to determine its own passage magnitude.
- **Essential for complex feature capture.**

$$\text{Swish}(z) = z \cdot \sigma(\beta \cdot z)$$

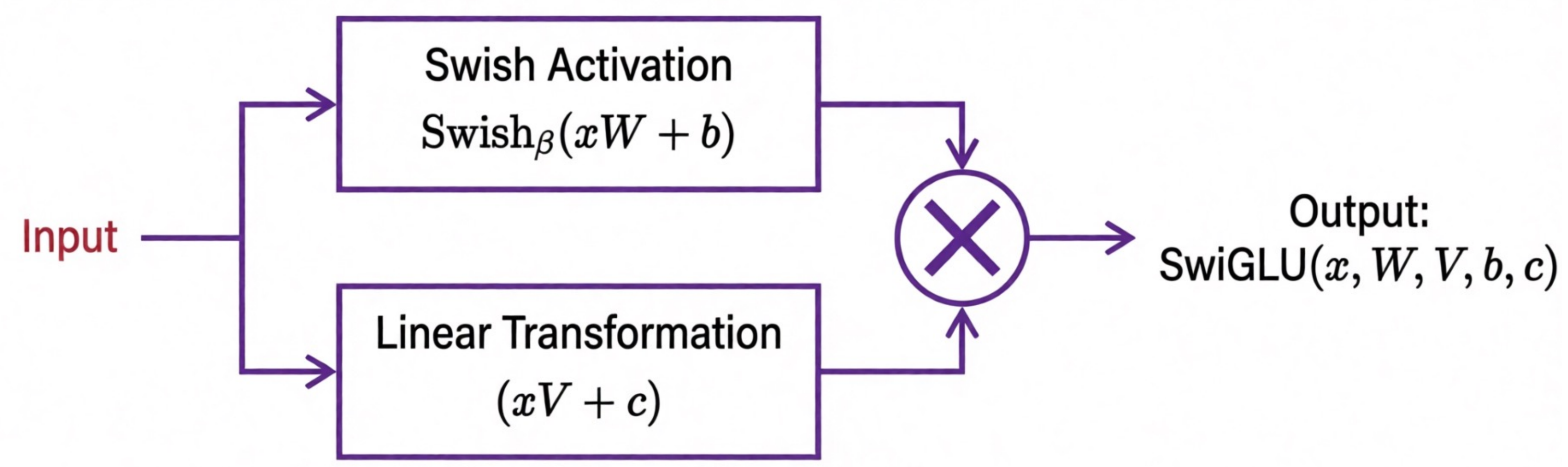
- **SiLU** is Swish where β=1



"Swish: a Self-Gated Activation Function" by researchers Prajit Ramachandran, Barret Zoph, and Quoc V. Le.

State of the Art: SwiGLU (2020)

- Combines the smoothness of Swish with learnable flexibility of Gated Linear Unit
  - Powering Giants: PaLM, LLaMA-2



Micro-Architecture: Activation Function

The Engine of Non-Linearity

ReLU (Rectified Linear Unit) : ReLU(x) = max(0,x)

- **Pros**: Efficient computation.
- **Cons**: "Dying ReLU" problem where gradient is 0 for negative inputs.

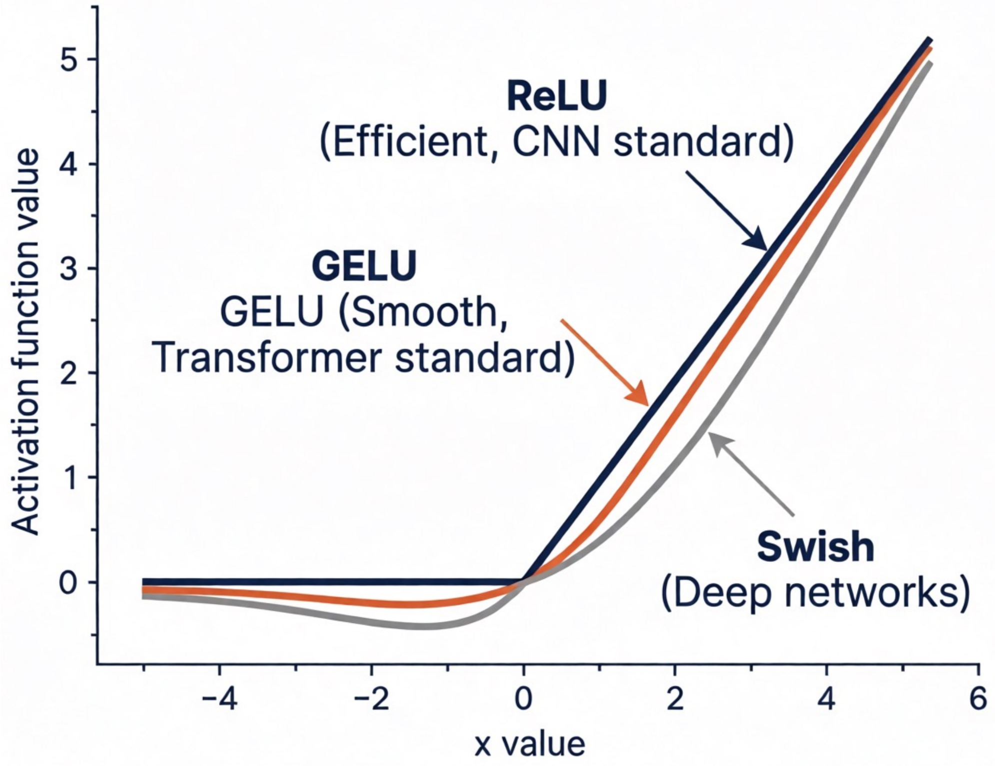
GELU (Gaussian Error Linear Unit) : gelu(x) =xΦ(x)

- **Pros**: Smooth approximation, prevents dead neurons. Standard for Transformers.

Swish : swish(x) =x• σ(Bx)

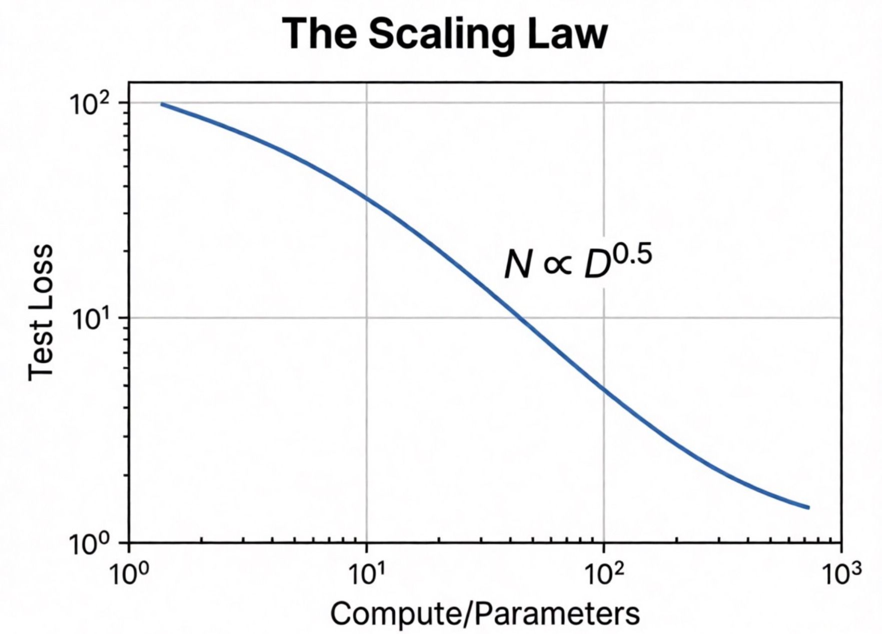
- **Pros**: Self-gated adaptive non-linearity. Outperforms ReLU in deep networks.

Diagnostic Tip: If gradients vanish, check for saturating functions (Sigmoid/Tanh) and switch to non-saturating alternatives.



Depth, Width, and Scaling

- Model capacity grows with **depth** (more layers) and **width** (more neurons per layer).
  - **Depth** enables hierarchical feature composition but can cause vanishing gradients.
  - **Width** offers parallel representational paths, often yielding flatter minima and better generalization.
- **Scaling Laws**: Performance improves predictably with model size, data, and compute.
  - *Chinchilla scaling*: For fixed compute, optimal model size N and dataset size D follow $N \propto D^{0.5}$.

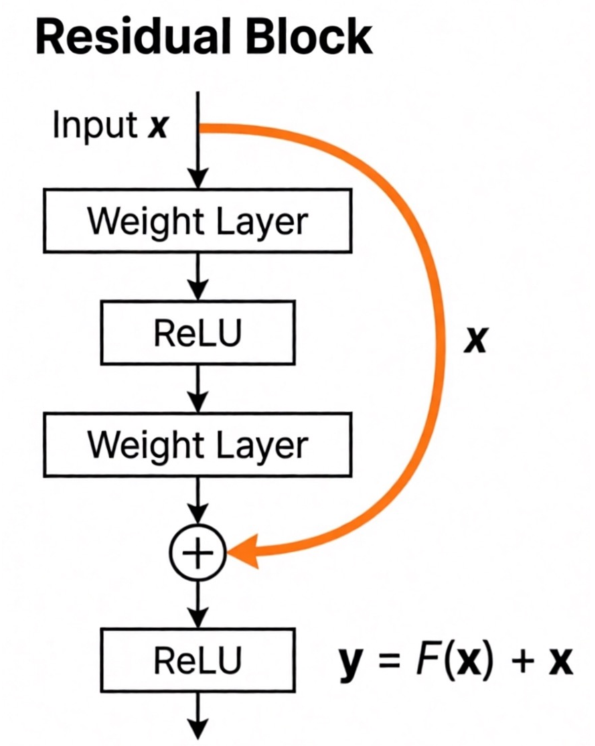


This graph illustrates scaling laws, plotting model performance (Test Loss) against parameters or Compute

Residual (or Skip) Connections

Solving the Depth Problem with Residuals

- **Residual connections** address degradation in deep networks by adding identity shortcuts:
  - **Formula**: $\mathbf{y} = F(\mathbf{x}) + \mathbf{x}$
  - **Mechanism**: The 'Identity Shortcut' creates a direct super-highway for gradient flow during backpropagation.
  - **Result**: Enables training of networks with 1000+layers (ResNets).

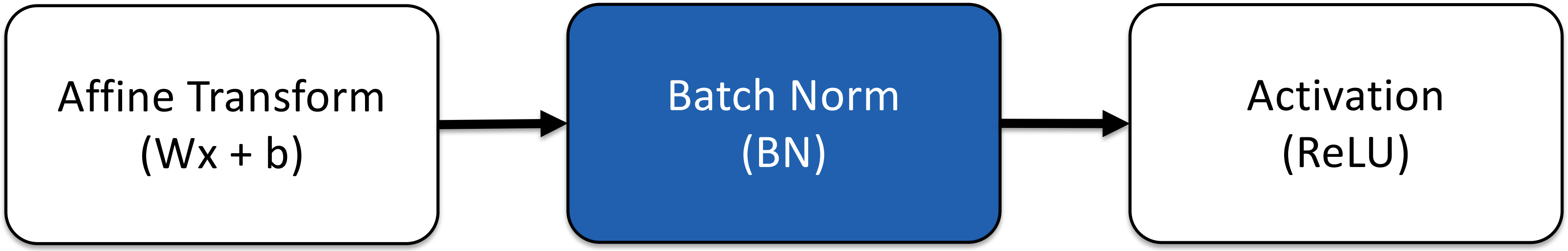


Why it works: The identity shortcut creates a direct path for gradient flow, enabling the training of networks with 100+ layers.

Batch Normalization (BN, 2015)

The Mechanism That Enabled Deep Architectures

- **Batch Normalization (BN)**: The 2015 breakthrough that allowed 100+ layer networks.



Note: A batch is a small set of samples of the dataset.

Mechanism

1. Normalization: Forces layer inputs to mean 0 and variance 1.
2. Re-calibration: Introduces learnable parameters to shift and scale data back if required.

The Mechanics: How BN Works

- **Calculate Batch Statistics** for the net input z at layer l :

$$\mu_j = \frac{1}{M}\sum_{i=1}^{M} z_j^{(i)} \qquad \sigma_j = \sqrt{\frac{1}{M}\sum_{i=1}^{M}\left(z_j^{(i)} - \mu_j\right)^2 + \epsilon}$$

$$\mathbf{Z} = [\mathbf{z}^{(1)} \quad \mathbf{z}^{(2)} \quad \cdots \quad \mathbf{z}^{(M)}] = \begin{bmatrix} z_1^{(1)} & z_1^{(2)} & \cdots & z_1^{(M)} \\ z_2^{(1)} & z_2^{(2)} & \cdots & z_2^{(M)} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n_l}^{(1)} & z_{n_l}^{(2)} & \cdots & z_{n_l}^{(M)} \end{bmatrix}$$

A mini batch with M net input $\mathbf{z}^{(i)}$

- **Normalize with μ_j & σ_j** :

$$z_j'^{(i)} = \frac{z_j^{(i)} - \mu_j}{\sigma_j}$$

- **De-normalize (Scale & Shift)**:

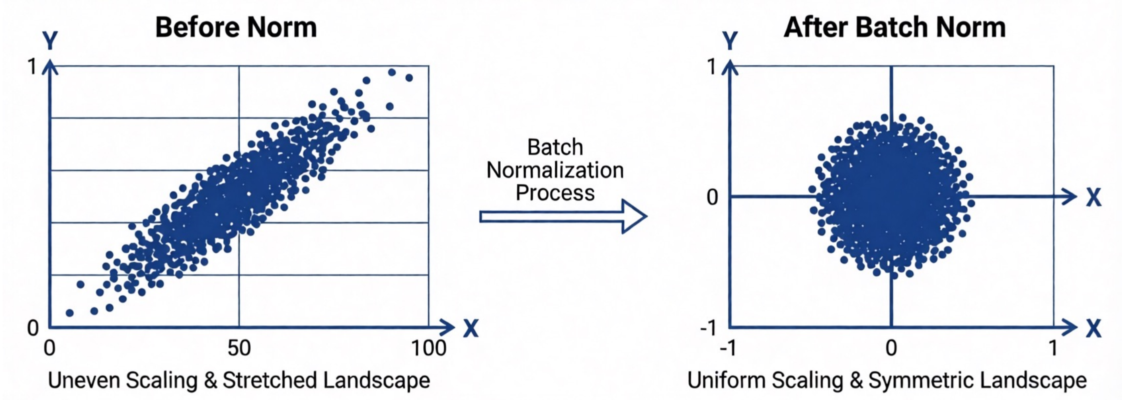
$$\hat{z}_j^{(i)} = \gamma_j \cdot z_j'^{(i)} + \beta_j$$

The Learnable Parameters

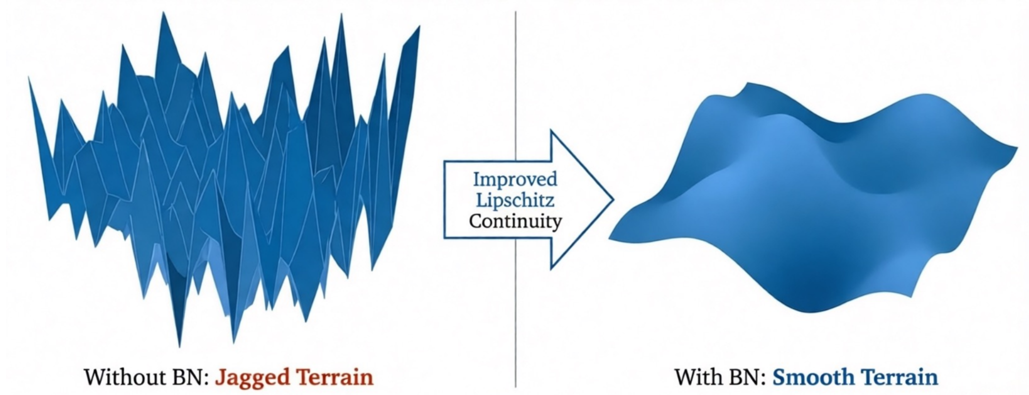
- γ_j (scale) and β_j (shift) are learned during training.
- They allow the network to 'undo' the normalization if optimal.
- This preserves the network's capacity to represent complex functions (expressivity).

Batch Norm: Smoothing the Landscape

- Batch Normalization:
  - Constrains layer outputs to a standard distribution (Mean=0, Var=1).
  - Prevents 'Internal Covariate Shift’.



- Benefit: Makes the loss landscape symmetric and smoother, allowing higher learning rates and faster optimization.



The 'Jekyll & Hyde' Problem: Training vs. Inference

Training Mode

Uses Mini-Batch Statistics (μ_B, σ_B).

Behavior: Stochastic / Noisy.

Dependency: Dependent on other samples in the batch.

Inference Mode

Uses Running Average Statistics (μ_{global}, σ_{global}).

Behavior: Deterministic.

Dependency: Independent processing.

Common Pitfall: Forgetting to switch to 'model.eval()' during validation leads to catastrophic failure.

How to use BatchNorm in Practice and During Inference

```python
class MultilayerPerceptron(torch.nn.Module):

    def __init__(self, num_features, num_classes, drop_proba,
                 num_hidden_1, num_hidden_2):
        super().__init__()

        self.my_network = torch.nn.Sequential(
            # 1st hidden layer
            torch.nn.Flatten(),
            torch.nn.Linear(num_features, num_hidden_1, bias=False)
            torch.nn.BatchNorm1d(num_hidden_1),
            torch.nn.ReLU(),
            # 2nd hidden layer
            torch.nn.Linear(num_hidden_1, num_hidden_2, bias=False)
            torch.nn.BatchNorm1d(num_hidden_2),
            torch.nn.ReLU(),
            # output layer
            torch.nn.Linear(num_hidden_2, num_classes)
        )

    def forward(self, x):
        logits = self.my_network(x)
        return logits
```

https://github.com/rasbt/stat453-deep-learning-ss21/blob/main/L11/code/batchnorm.ipynb

BatchNorm Variants

Pre-Activation

compute net inputs

↓

BatchNorm

↓

apply activation function

↓

compute next-layer net inputs

Post-Activation

compute net inputs

↓

apply activation function

↓

BatchNorm

↓

compute next-layer net inputs

How to use BN

before activation, no bias

```
self.my_network = torch.nn.Sequential(
    # 1st hidden layer
    torch.nn.Flatten(),
    torch.nn.Linear(num_features, num_hidden_1, bias=False),
    torch.nn.BatchNorm1d(num_hidden_1),
    torch.nn.ReLU(),
    # 2nd hidden layer
    torch.nn.Linear(num_hidden_1, num_hidden_2, bias=False),
    torch.nn.BatchNorm1d(num_hidden_2),
    torch.nn.ReLU(),
    # output layer
    torch.nn.Linear(num_hidden_2, num_classes)
)
```

before activation, with bias

```
self.my_network = torch.nn.Sequential(
    # 1st hidden layer
    torch.nn.Flatten(),
    torch.nn.Linear(num_features, num_hidden_1),
    torch.nn.BatchNorm1d(num_hidden_1),
    torch.nn.ReLU(),
    # 2nd hidden layer
    torch.nn.Linear(num_hidden_1, num_hidden_2),
    torch.nn.BatchNorm1d(num_hidden_2),
    torch.nn.ReLU(),
    # output layer
    torch.nn.Linear(num_hidden_2, num_classes)
)
```

after activation, with bias

```
self.my_network = torch.nn.Sequential(
    # 1st hidden layer
    torch.nn.Flatten(),
    torch.nn.Linear(num_features, num_hidden_1, bias=True),
    torch.nn.ReLU(),
    torch.nn.BatchNorm1d(num_hidden_1),
    # 2nd hidden layer
    torch.nn.Linear(num_hidden_1, num_hidden_2, bias=True),
    torch.nn.ReLU(),
    torch.nn.BatchNorm1d(num_hidden_2),
    # output layer
    torch.nn.Linear(num_hidden_2, num_classes)
)
```

before activation + dropout

```
self.my_network = torch.nn.Sequential(
    # 1st hidden layer
    torch.nn.Flatten(),
    torch.nn.Linear(num_features, num_hidden_1, bias=False),
    torch.nn.BatchNorm1d(num_hidden_1),
    torch.nn.ReLU(),
    torch.nn.Dropout(drop_proba),
    # 2nd hidden layer
    torch.nn.Linear(num_hidden_1, num_hidden_2, bias=False),
    torch.nn.BatchNorm1d(num_hidden_2),
    torch.nn.ReLU(),
    torch.nn.Dropout(drop_proba),
    # output layer
    torch.nn.Linear(num_hidden_2, num_classes)
)
```

after activation + dropout

```
self.my_network = torch.nn.Sequential(
    # 1st hidden layer
    torch.nn.Flatten(),
    torch.nn.Linear(num_features, num_hidden_1, bias=True),
    torch.nn.ReLU(),
    torch.nn.BatchNorm1d(num_hidden_1),
    torch.nn.Dropout(drop_proba),
    # 2nd hidden layer
    torch.nn.Linear(num_hidden_1, num_hidden_2, bias=True),
    torch.nn.ReLU(),
    torch.nn.BatchNorm1d(num_hidden_2),
    torch.nn.Dropout(drop_proba),
    # output layer
    torch.nn.Linear(num_hidden_2, num_classes)
)
```

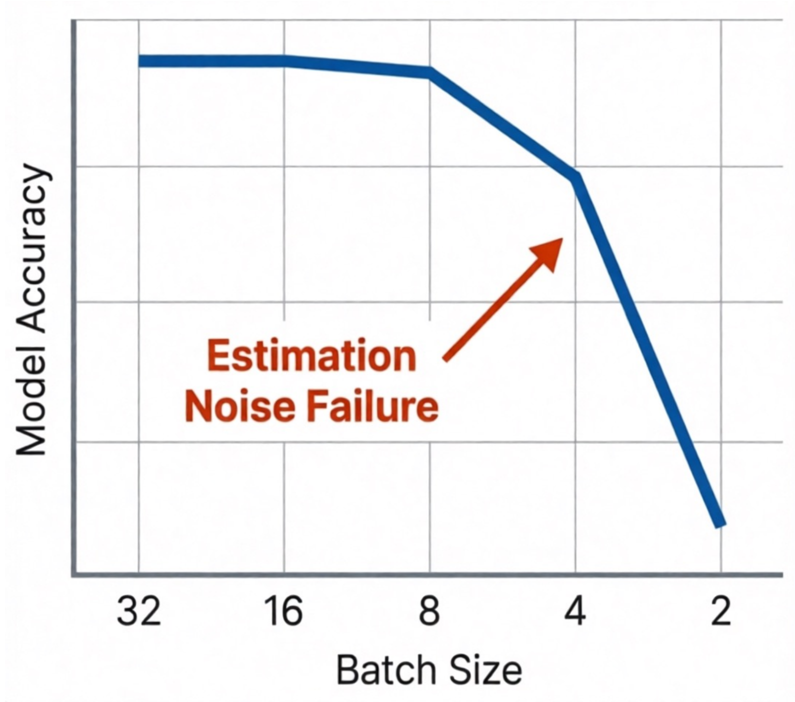
The Limitations of Batch Norm

- **Weakness 1: Small Batch Sizes**

  BN relies on batch stats to estimate the population. If Batch Size < 8, statistics are noisy and error rates spike.

- **Weakness 2: RNNs & Sequences**

  Variable sequence lengths make tracking statistics computationally messy.



The Challenger: Layer Normalization (2016)

- Instead of normalizing across the batch, Layer Norm normalizes across the features of a single sample.

$$\mu_j = \frac{1}{n_l}\sum_{j=1}^{n_l} z_j^{(i)} \qquad \sigma_j = \sqrt{\frac{1}{n_l}\sum_{j=1}^{n_l}\left(z_j^{(i)} - \mu_j\right)^2 + \epsilon}$$

An individual net input sample $\mathbf{z}^{(i)}$

$$\mathbf{z}^{(i)} = \begin{bmatrix} z_1^{(i)} \\ z_2^{(i)} \\ \vdots \\ z_{n_l}^{(i)} \end{bmatrix}$$

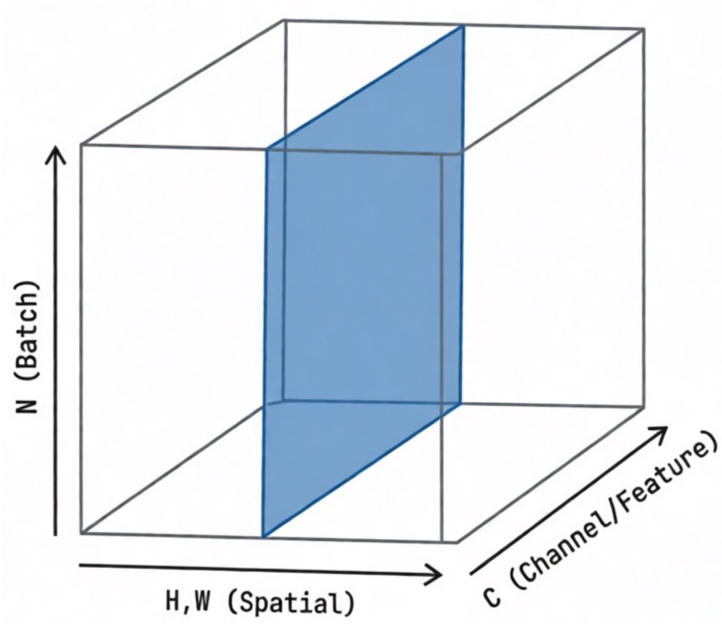
$$z_j'^{(i)} = \frac{z_j^{(i)} - \mu_j}{\sigma_j} \qquad \hat{z}_j^{(i)} = \gamma_j \cdot z_j'^{(i)} + \beta_j$$

Batch Independent: Works perfectly with Batch Size = 1.

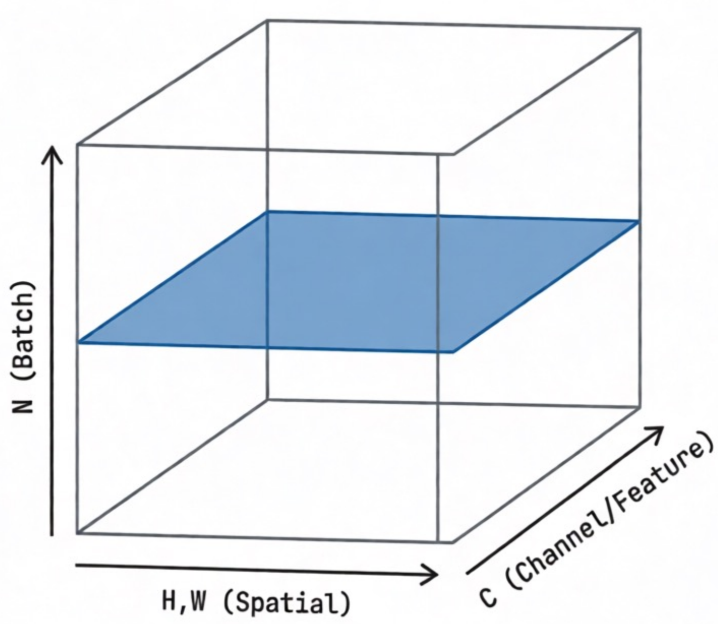
Deterministic: Identical behavior in Training and Inference.

Sequence Friendly: Ideal for RNNs and Transformers.

Visualization: Slicing the Data Cube



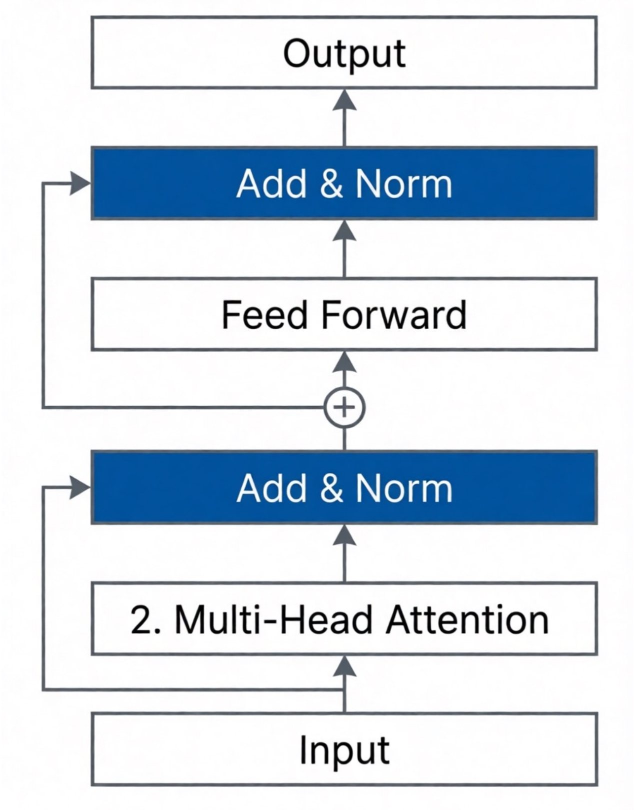
Batch Norm: Global stats from the crowd.



Layer Norm: Individual stats from the self.

Transformers & The Dominance of Layer Norm

- Modern NLP (BERT, GPT, T5) relies almost exclusively on Layer Norm. In NLP, batch dimensions are arbitrary, but the relationships between features (embeddings) within a token are critical.



Head-to-Head: Choosing Your Norm

Feature	Batch Normalization	Layer Normalization
Best For	MLPs and CNNs (Computer Vision)	Transformers / RNNs
Batch Dependency	High (needs large batches)	None (Works with Batch =1)
Training/Inference	Different modes required	Same mode
Regularization	Adds noise (beneficial)	Deterministic (little noise)
Structure Use	Use spatial structure	Isotropic (treats feature same)

Step 3: Optimization Strategies

Navigating the Loss Landscape

Step 3: Optimization Strategies

- **Goal: Minimize cost function** $\mathcal{L}(\theta)$ across a set of model parameters $\theta \coloneqq \{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$

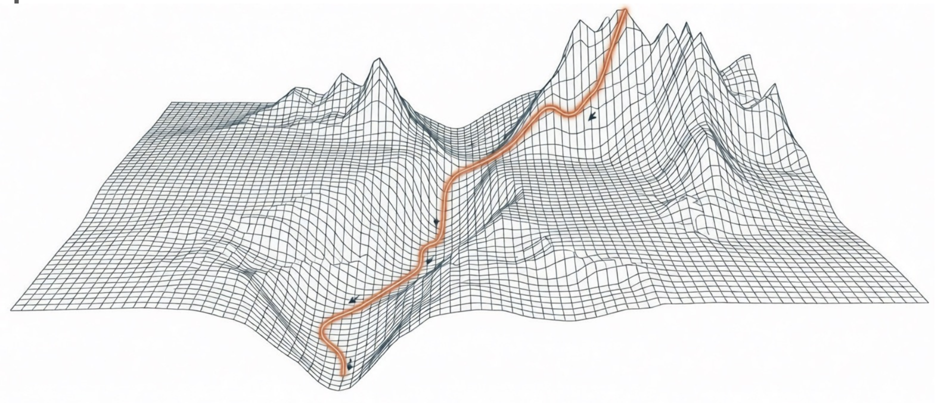
$$\theta^* = \arg\min_{\theta} \mathcal{L}(\theta)$$

- **Gradient Descent:** Simple equation, complex implementation:

$$\theta_{new} = \theta_{old} - \eta \cdot \nabla\mathcal{L}(\theta)$$

Follow negative gradient → reach global minimum

Cauchy's Steepest Descent (1840)



- *The invisible engine powering modern AI systems*

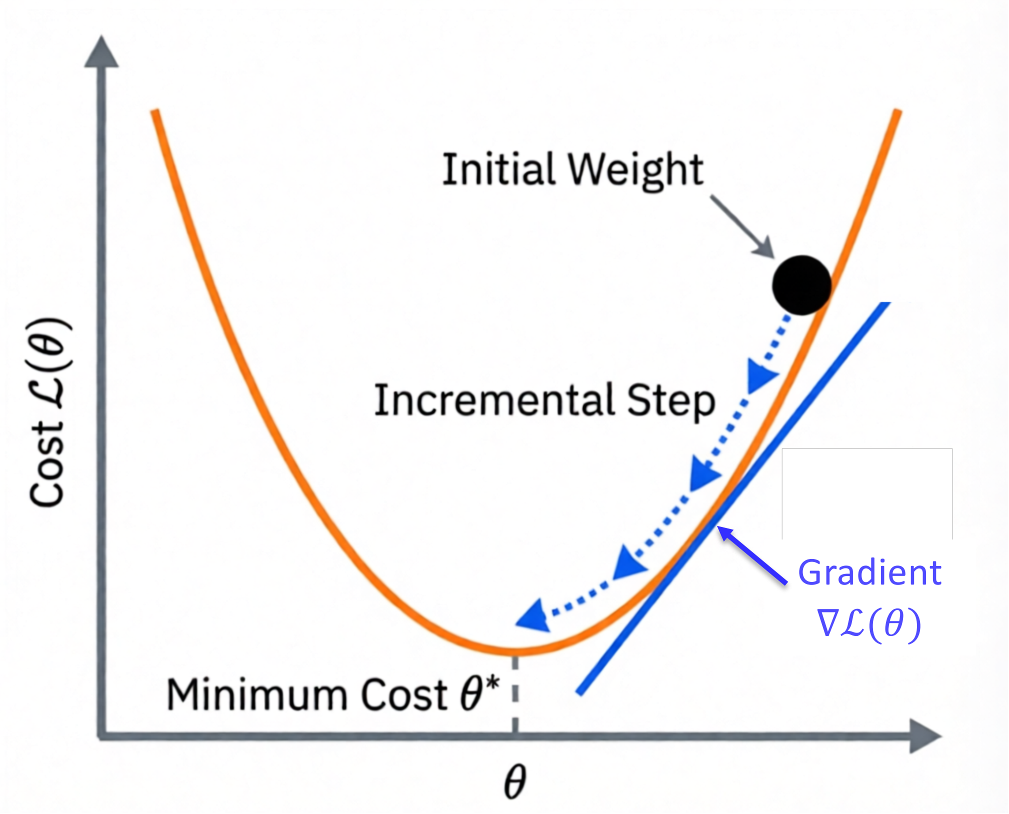
Recap: Gradient Descent Algorithm

1. **Initialize**: Randomly set weights θ
2. **Compute Cost**: Measure performance $\mathcal{L}(\theta)$.
3. **Find Gradient**: Calculate $\nabla\mathcal{L}(\theta)$ (direction of steepest ascent).
4. **Update**: Step down the hill.

$$\theta_{new} = \theta_{old} - \eta \cdot \nabla\mathcal{L}(\theta)$$

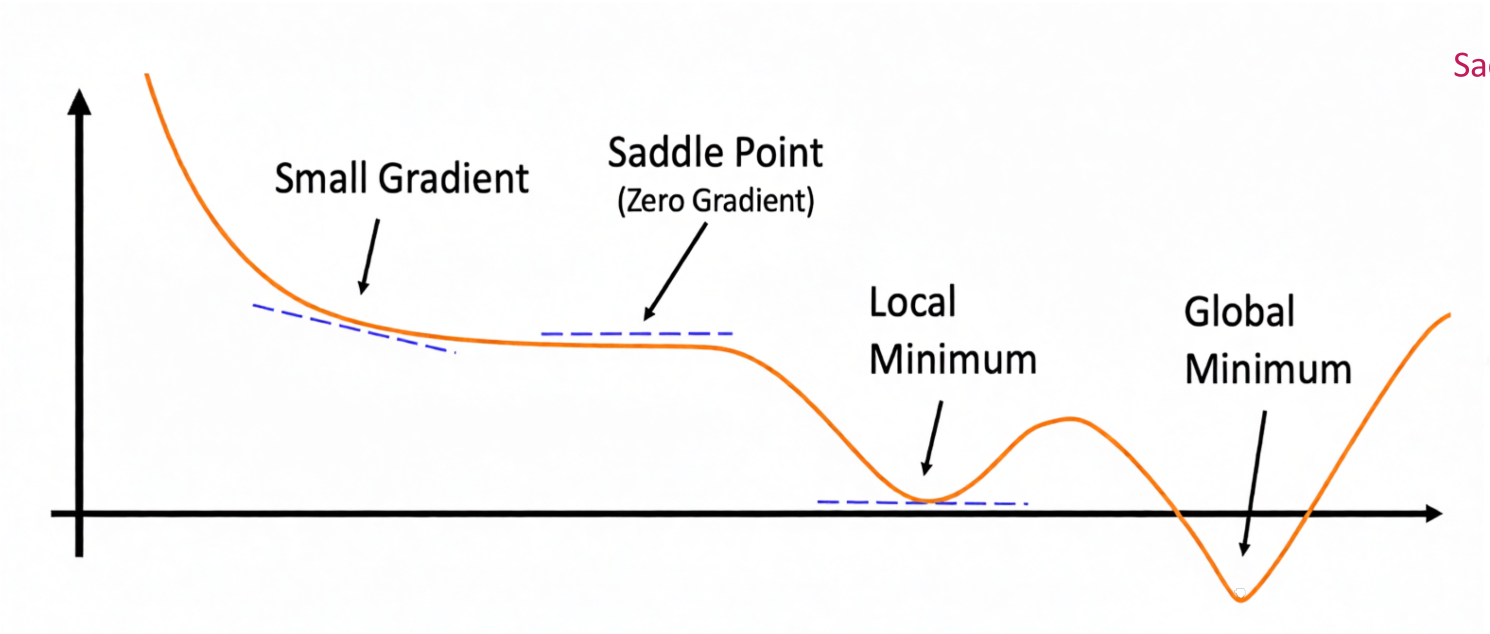
η = Learning Rate (e.g. 0.001)

- **Repeat** steps 2 to 4, unit the cost is low enough or convergence.

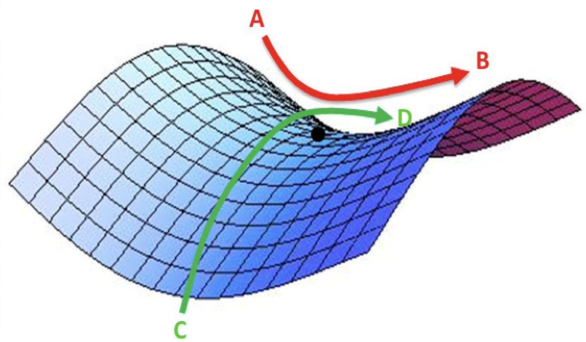


The Challenge of Local Minima

Deep learning optimization is hindered not by local minima but by saddle points and flat plateaus where gradients vanish, causing standard methods to stall or falsely appear converged.



Saddle point — simultaneously a local minimum and a local maximum.

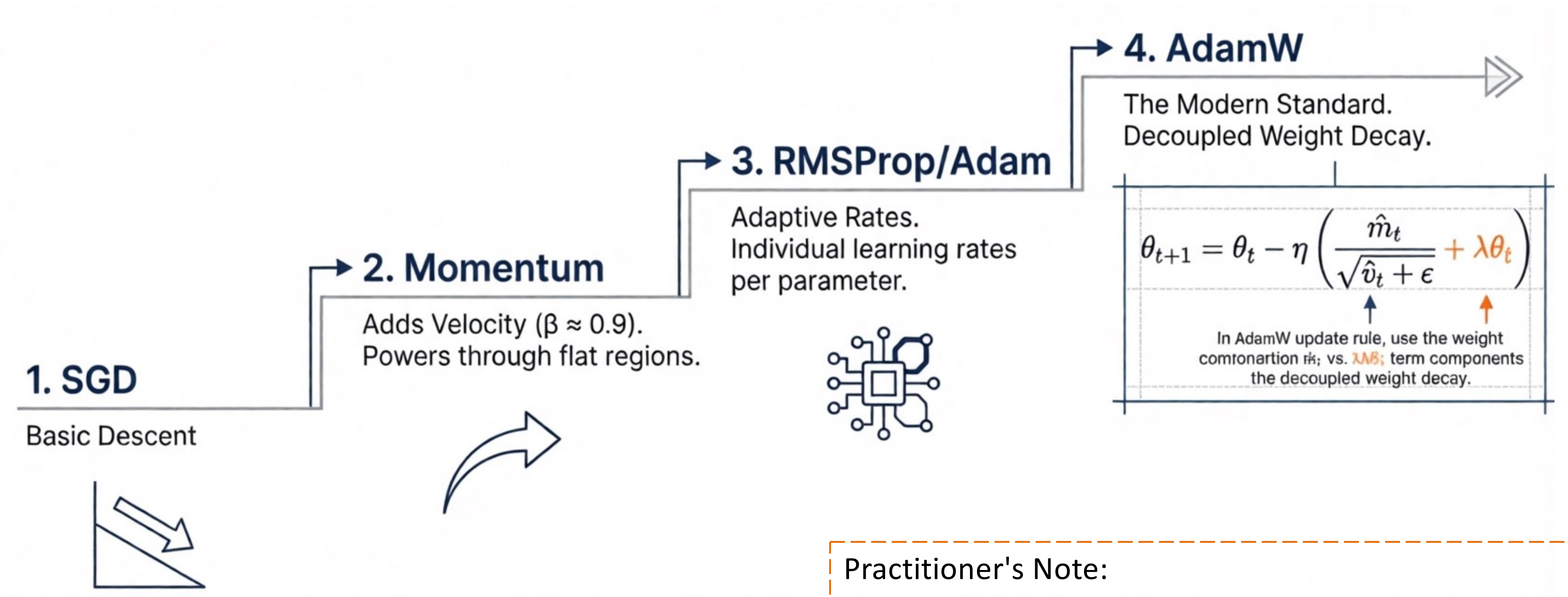


Historical Timeline (1840-2025)

Evolution of Gradient Descent

- **1840s:** Cauchy's Steepest Descent (Theoretical Foundation)
- **1950s**: Batch Gradient Descent (First Implementation)
- **1951:** Stochastic Gradient Descent (Efficiency Revolution)
- **1964:** Momentum (Adding Memory)
- **1990s:** Mini-batch Gradient Descent (Balance & Parallelization)
- **2011:** AdaGrad (Adaptive Learning Rates)
- **2012:** RMSprop (Solving Vanishing LR)
- **2014:** Adam (The Crown Jewel)
- **2017:** AdamW (Regularization Fix)
- **2023:** Lion & Sophia (Modern Breakthroughs)
- **2024-2025:** Continuous Innovation

Evolution of the Optimizer



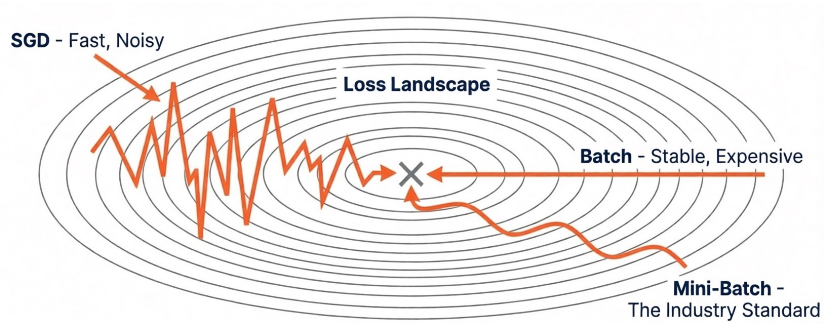
Practitioner's Note:

Selection Guide:

- Use AdamW for Transformers & CNNs.
- Use SGD+Momentum for simple streaming tasks.

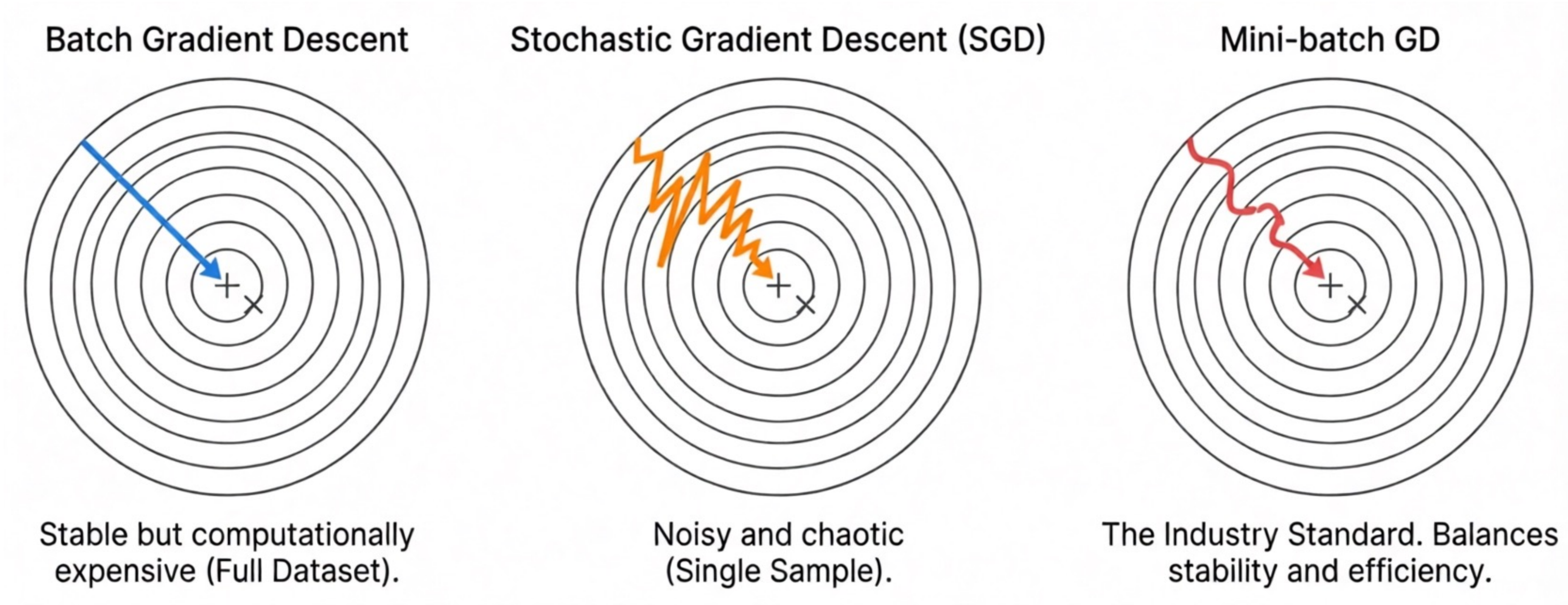
Basic Gradient Descent Algorithms

- **Batch Gradient Descent** (BGD, 1950s)
  - Uses the full dataset per update; stable but computationally expensive.
  - $\theta_{t+1} = \theta_t - \eta \cdot \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \ell\left(\theta_t; \mathbf{x}^{(i)}, \mathbf{y}^{(i)}\right)$
- **Stochastic Gradient Descent** (SGD, 1951)
  - Uses a single sample; fast but noisy.
  - $\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta \ell(\theta_t; \mathbf{x}, \mathbf{y})$



- **Mini-batch Gradient Descent** (MBGD, 1990s)
  - Uses small batches ($M \ll N$); balances efficiency and stability and is the industry standard.
  - $\theta_{t+1} = \theta_t - \eta \cdot \frac{1}{M} \sum_{i=1}^{M} \nabla_\theta \ell\left(\theta_t; \mathbf{x}^{(i)}, \mathbf{y}^{(i)}\right)$

BGD vs MBGD vs SGD



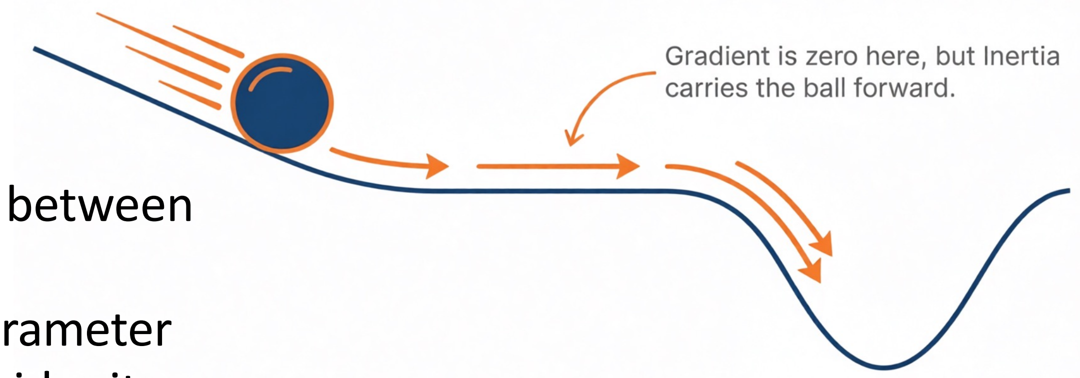
SGD + Momentum (1964)

- Like a ball rolling downhill - **builds momentum in consistent directions**

$$m_t = \beta m_{t-1} + (1-\beta)\nabla\mathcal{L}(\theta_t)$$

Momentum = exponential average of the gradients

$$\theta_{t+1} = \theta_t - \eta \cdot m_t$$

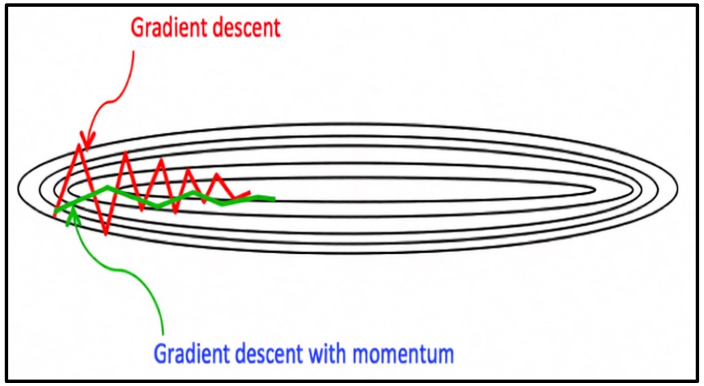


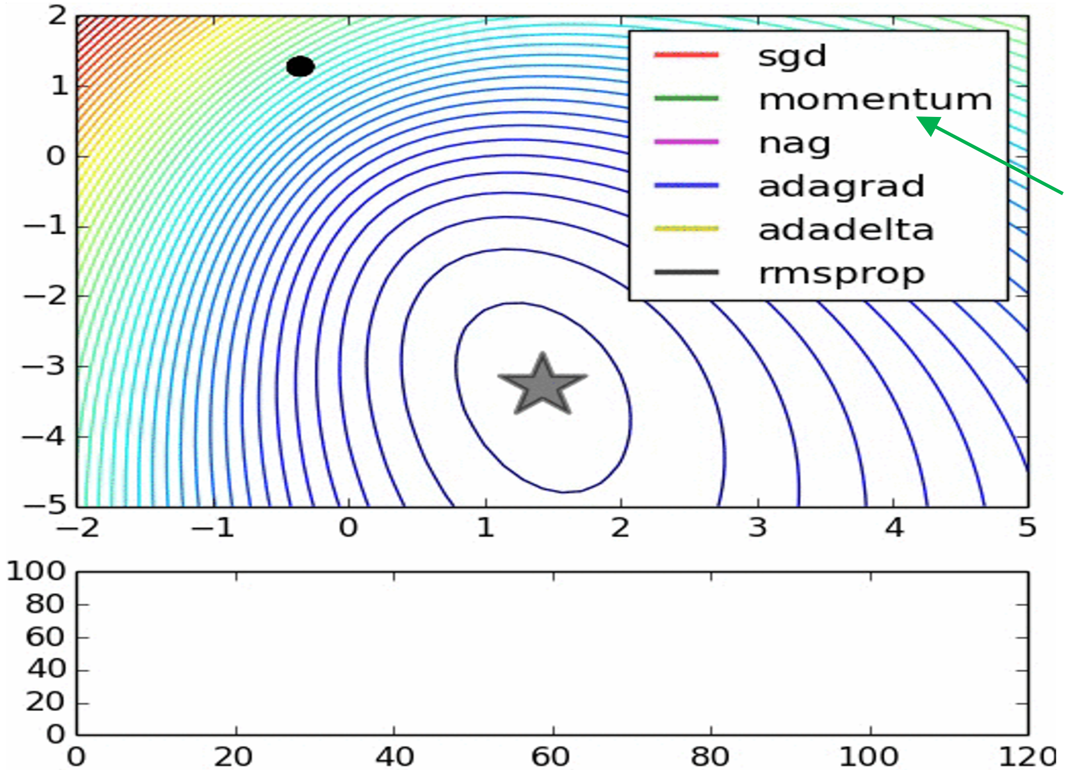
- The momentum rate β is usually chosen between 0.9 and 0.999.
  - You can think of it a “dampening” parameter
  - On the other hand, you can also consider it as an exponential moving average parameter.

Polyak (1964) “Some methods of speeding up the convergence of iteration methods”

SGD vs. Momentum GD

Momentum
(dampening oscillations)

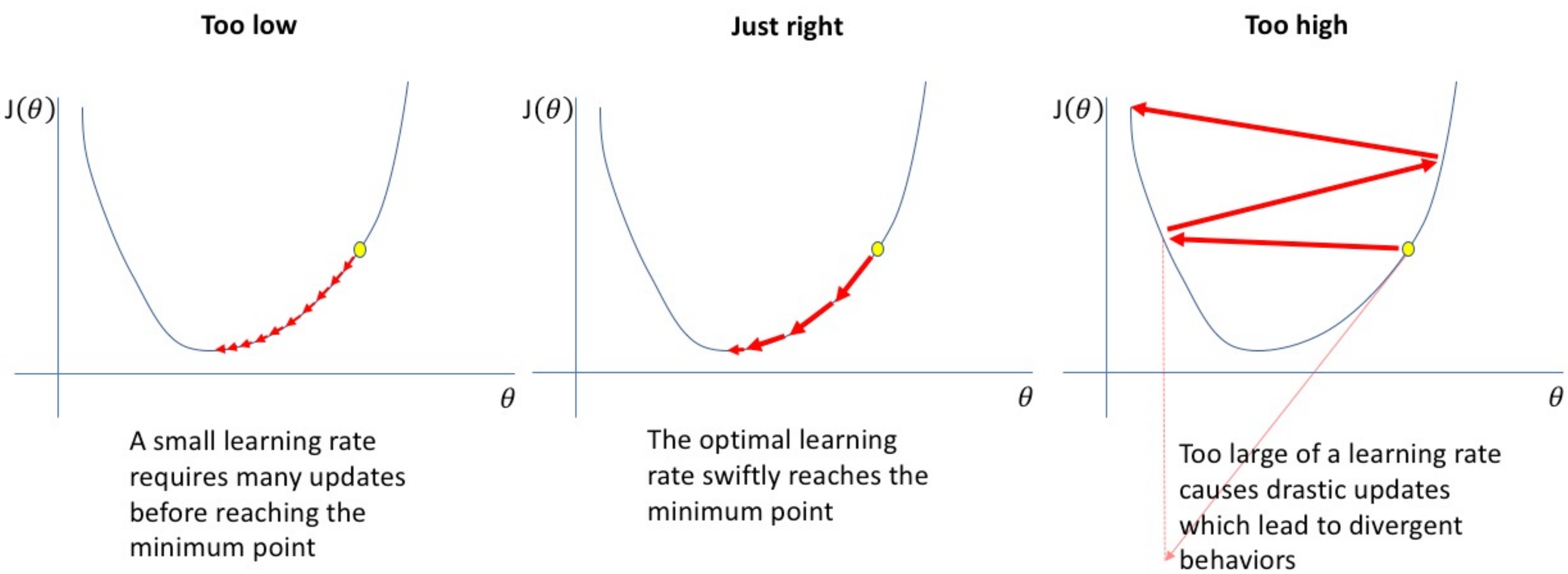




Momentum GD

Learning Rate Issues

- The learning rate is a crucial hyperparameter that controls the step size in Gradient Descent optimizers.
- **Too low**: training becomes painfully slow. **Too high**: the optimizer becomes unstable



Adaptive Learning Rate Optimizers

- The idea behind adaptive learning rates is to address the issue where **sparse but important features can have small gradients**, **leading to slow learning in those directions.**
- To remedy this, we can assign different learning rates to each feature, giving higher rates to sparse features.
- This approach involves adjusting the learning rate based on the gradient's behavior:
  - **Decreasing the rate** when the gradient changes rapidly (indicating large gradients)
  - **Increasing the rate** when the gradient remains consistent (indicating small gradients).

AdaGrad: Adaptive Gradient (2011)

AdaGrad uses a cumulative sum of **squared of historical gradients** $G_t = \sum_{k=1}^{t}[\nabla\mathcal{L}(\theta_k)]^2$ **to adapt the learning rate** η_t for each parameter.

- Parameters with large gradients have their learning rates reduced
- Parameters with small gradients have their learning rates increased

$$G_t = \sum_{k=1}^{t}[\nabla\mathcal{L}(\theta_k)]^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \varepsilon}} \cdot \nabla\mathcal{L}(\theta_t)$$

Small ε term to avoid division by zero

AdaGrad eliminates the need to manually tune the learning rate with default rate of η = 0.01 and a common default value for ε is 1e-8 (10^{-8}).

Duchi et. Al (2011) "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization.

RMSProp (2012)

- **RMSProp** solved AdaGrad's limitation through exponentially decaying averages => **RMS (Root-Mean-Squared) gradient**

$$v_t = \beta v_{t-1} + (1 - \beta)[\nabla \mathcal{L}(\theta_k)]^2 \qquad (0 \leq \beta \leq 1)$$

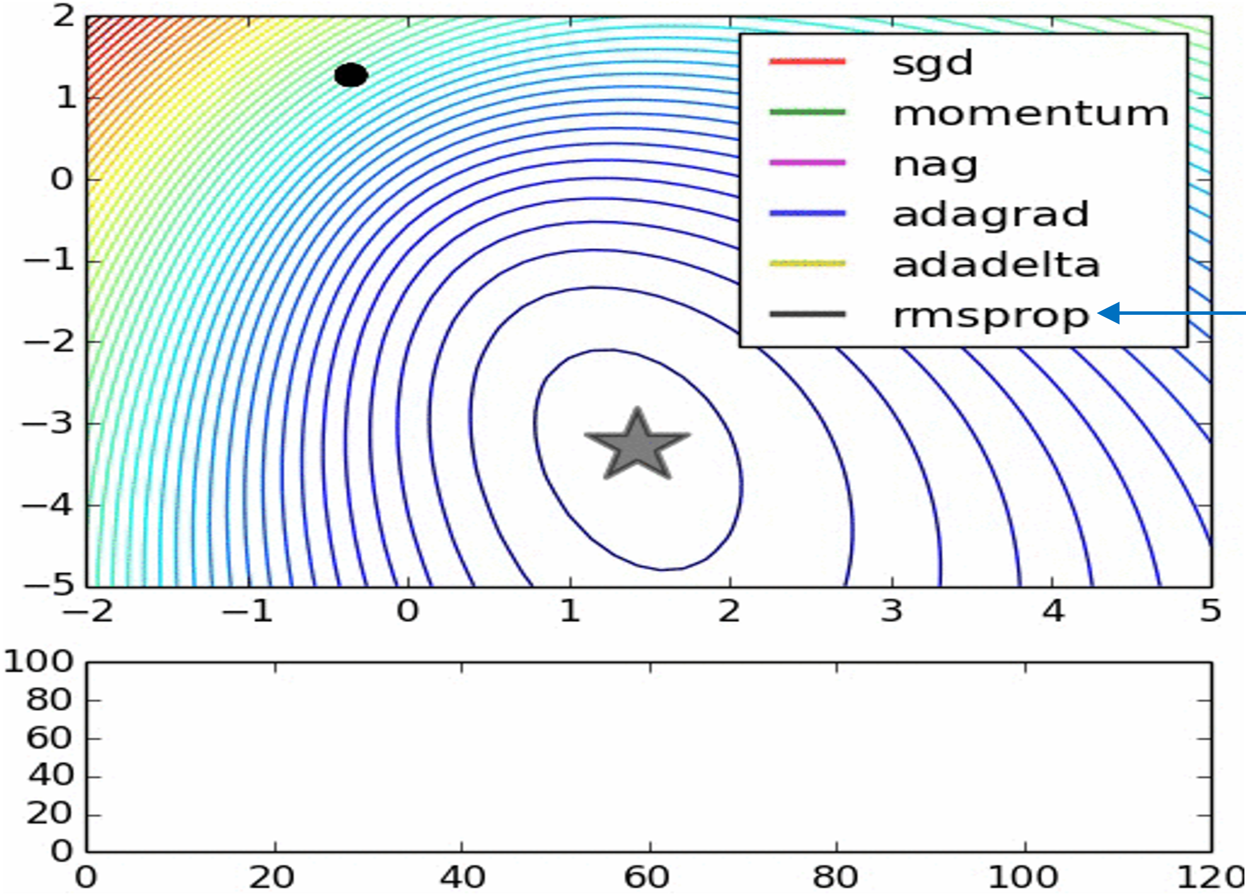
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t + \varepsilon}} \cdot \nabla \mathcal{L}(\theta_t)$$

The decay parameter β (typically 0.9 - 0.95) prevents indefinite accumulation.

- **Strengths**: RMSprop provides more stable learning rates and faster convergence compared to Adagrad.
- **Limitations**: RMSprop can still suffer from some of the limitations of Adagrad, such as the need for careful tuning of the decay rate.

Teleman & Hinton (2012) "Neural Networks for Machine Learning", Lecture 6, Coursera.

RMSProp



RMSProp

Adam: Adaptive Moment Estimation (2014)

- **Adam** combines the strengths of **AdaGrad** and **RMSProp**.
  - AdaGrad is good for sparse gradients, while RMSprop is good for online and changing situations.
  - **Adam** adapts learning rates per parameter by tracking two exponential moving averages:

Momentum: $m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla\mathcal{L}(\theta_t),$ $\longrightarrow$ $\hat{m}_t = \frac{m_t}{(1 - \beta_1^t)}$

Bias-corrected

RMSProp: $v_t = \beta_2 v_{t-1} + (1 - \beta_2)[\nabla\mathcal{L}(\theta_k)]^2,$ $\longrightarrow$ $\hat{v}_t = \frac{v_t}{(1 - \beta_2^t)}$

Kingma & Ba (2014) "Adam: A Method for Stochastic Optimization": https://arxiv.org/pdf/1412.6980.pdf

Adam ≈ Momentum + RMSProp

- The moving average is initialized to 0, causing the moment estimate bias to be around 0, especially during the initial time step. **This initialization bias can be easily offset, yielding bias-corrected estimates**

$$\hat{m}_t = \frac{m_t}{(1-\beta_1^t)}$$

Momentum
(dampening oscillations)

$$\hat{v}_t = \frac{v_t}{(1-\beta_2^t)}$$

RMSProp
(Adaptive learning rate)

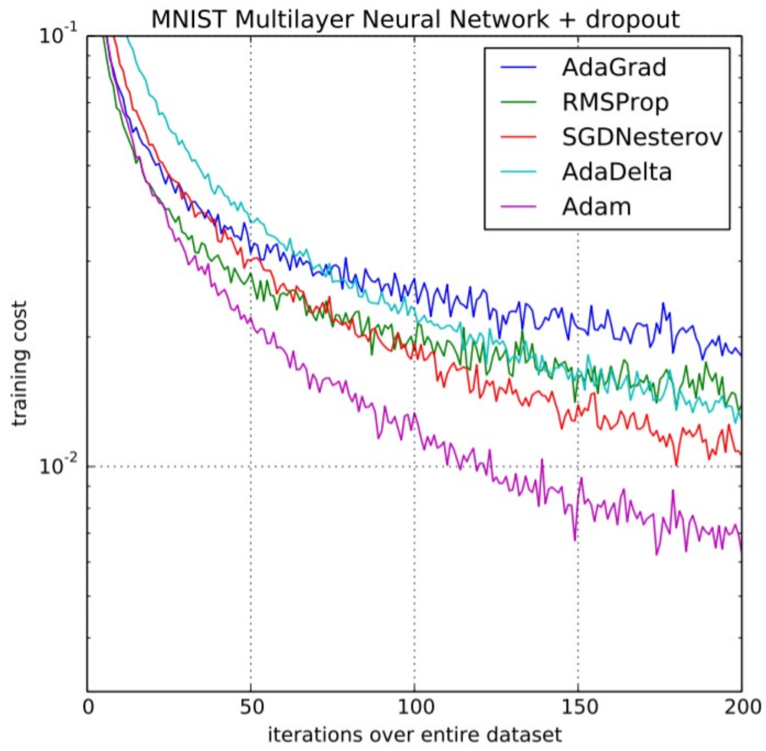
When $t \to \infty, \beta \to 0$

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \varepsilon}}$$

- Important practical point: β_1 typically 0.9 while β_2 typically much closer to 1, e.g. 0.999

Adam Optimizer

- Adam is a widely used optimization algorithm for gradient descent.
- It is efficient in terms of computational resources and does not require much memory.
- Adam works well for problems that involve a large amount of data or parameters.
- It is also suitable for problems with noisy or sparse gradients.
- Most popular libraries, like PyTorch, use the default hyperparameters from the original paper for Adam:
  - Learning rate η = 0.001, β_1 = 0.9, β_2 = 0.999, ε =1e-08, weight decay = 0.0.



Issue of Adam with Weight Decay

- In the standard Adam optimizer, weight decay is typically implemented by adding an **L2 regularization term** to the loss function:

$$\mathcal{L}(\theta) = \mathcal{L}_{org}(\theta) + \lambda \frac{1}{2} \|\theta\|^2$$

- However, incorporating this term directly into the loss affects the adaptive learning rates computed by Adam, which can interfere with optimal convergence and degrade performance.

AdamW: The Regularization Fix (2017)

- **AdamW (Adam with Weight decay)** addressed a subtle but critical issue with Adam's weight decay implementation:

$$\theta_{t+1} = \theta_t - \eta \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t + \varepsilon}} + \lambda \theta_t \right)$$

where λ is the weight decay coefficient.

- By decoupling weight decay from gradient-based optimization, AdamW achieves improved generalization, especially in transformer architectures.

Loshchilov, I., & Hutter, F. (2017). Decoupled weight decay regularization.

Adam vs AdamW

The Standard (Adam) and the Refinement (AdamW)

Adam (2014)

Momentum + RMSprop

- $\mathcal{L}(\theta) = \mathcal{L}_{org}(\theta) + \lambda \frac{1}{2} \|\theta\|^2$
  - $m_t = \beta_1 m_{t-1} + (1 - \beta_1)\big(\nabla \mathcal{L}(\theta_t)\big)$
  - $v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla \mathcal{L}(\theta_t)\,)^2$
- Updated: $\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \varepsilon}}$

L2 Regularization is entangled with the gradient adaptation

AdamW (2017)

The Moden Corw Jewl

- $\mathcal{L}(\theta) = \mathcal{L}_{org}(\theta)$
  - $m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla \mathcal{L}(\theta_t)$
  - $v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla \mathcal{L}(\theta_t)\,)^2$
- Updated: $\theta_{t+1} = \theta_t - \eta \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t + \varepsilon}} + \lambda \theta_t \right)$

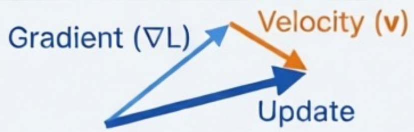
Weight decay is applied diretly to the parameters, decopled from the gradient update. This yields significantly better generation.

The Evolution of Optimizers

From Momentum to AdamW

1. SGD + Momentum

Adds velocity to dampen oscillations.



2. RMSprop

Adaptive learning rates. Normalizes by recent gradient magnitude.

3. Adam (Adaptive Moment Estimation)

Momentum + RMSprop. The default starting point.

4. AdamW (The Standard)

Decoupled Weight Decay. Fixes regularization in Adam.

Adam	AdamW
$L(\theta) = L_{org}(\theta) + \lambda\frac{1}{2}\|\theta\|^2$	$\theta_{t+1} = \theta_t - \eta(\ldots) - \eta\lambda\theta_t$
(L2 Mixed in gradient)	(Decay separate)

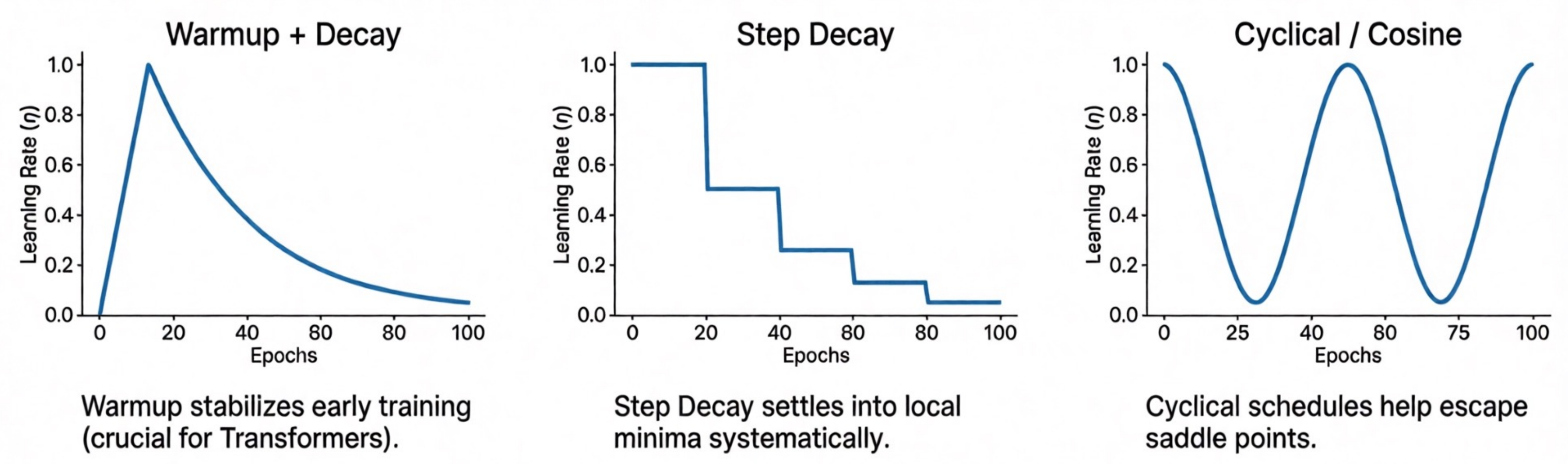
Practical Implementation: From Theory to Code

- Usage is the as for vanilla SGD, which we used before, you can find an overview at: https://pytorch.org/docs/stable/optim.html

```
# SGD with Momentum (1964)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9, weight_decay=0)

# SGD with NAG (1983)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9, nesterov=True, weight_decay=0)

# Adagrad (2011)
optimizer = torch.optim.Adagrad(model.parameters(), lr=0.01, eps=1e-10, weight_decay=0)

# RMSprop (2012)
optimizer = torch.optim.RMSprop(model.parameters(), lr=0.01, alpha=0.99, eps=1e-08, weight_decay=0, momentum=0)

# Adam (2014)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0)

# AdamW (2017)
optimizer = torch.optim.AdamW(model.parameters(), lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0.01)
```

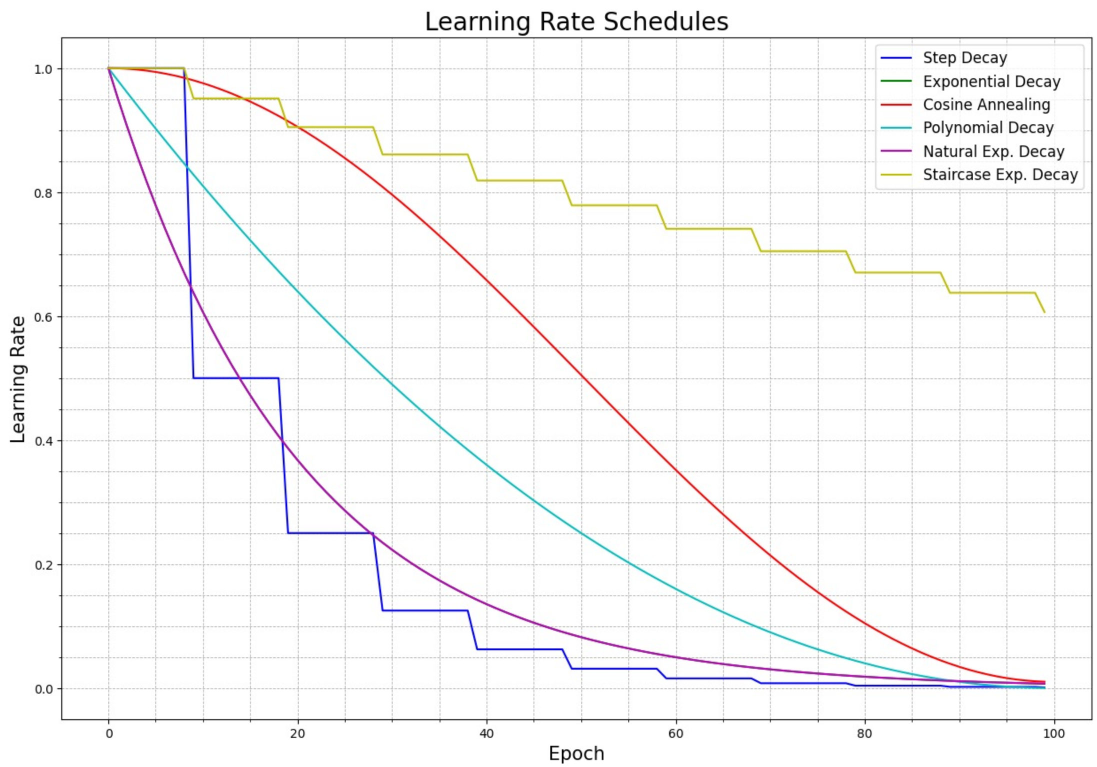
Learning Rate Scheduling

Dynamic Control of the Optimization Speed Limit



The 'Goldilocks' Zone: We need high LR for exploration and low LR for refinement.

Other Learning Rate Schedulers



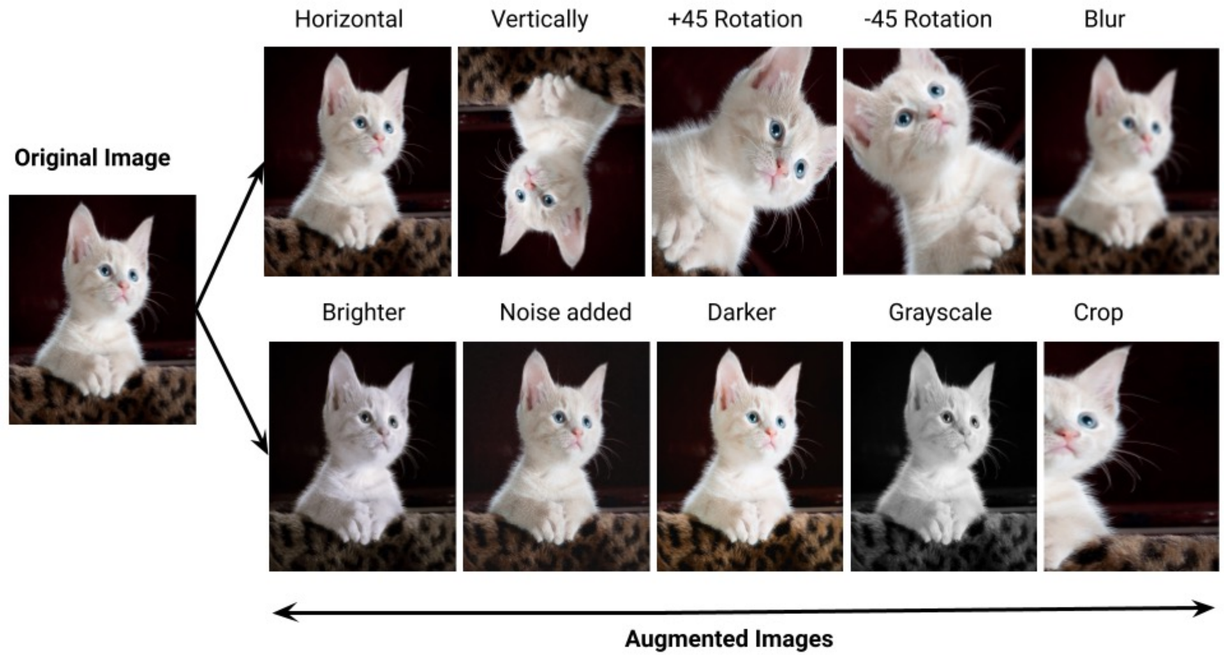
- Step Decay
  - $\eta_t = \eta_0 \cdot \gamma^{\lfloor t/T \rfloor}$
- Exponential Decay
  - $\eta_t = \eta_0 \cdot \gamma^t$
- Inverse Time Decay
  - $\eta_t = \frac{\eta_0}{1+\gamma^t}$
- Cosine Annealing
  - $\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{T_{cur}}{T_{max}}\pi\right)\right)$

https://medium.com/@theom/a-very-short-visual-introduction-to-learning-rate-schedulers-with-code-189eddffdb00

Generalization: Data Augmentation

Free data to reduced variance

- **Objective**: Enforce invariance and simulate real-world diversity.
- **Standard Techniques:**
  - Geometric (Flip, Rotate, Crop) & Photometric (Noise, Color).



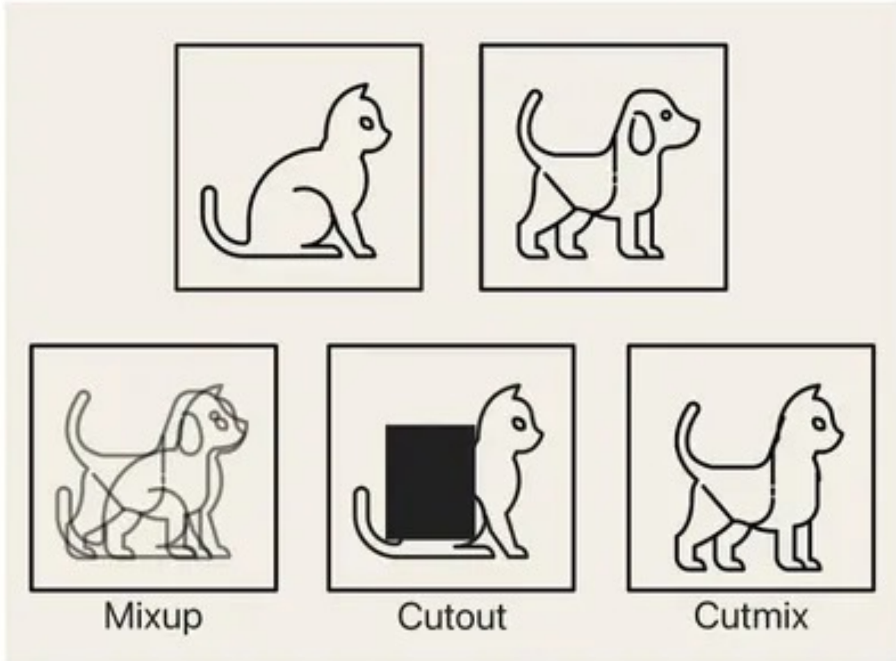
PyTorch Image Data Augmentation

```python
from torchvision import datasets,transforms

transform_train = transforms.Compose([
                                  transforms.RandomHorizontalFlip(p=0.5), # Randomly flip the image horizontally
                                  transforms.RandomCrop(28, padding=2, padding_mode='edge'), # Randomly crop the image with padding
                                  transforms.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1), # color jitter
                                  transforms.RandomRotation(5),
                                  transforms.RandomAffine(degrees=3, translate=(0.1, 0.1)),
                                  transforms.RandomPerspective(),
                                  transforms.RandomVerticalFlip(p=0.5),
                                  transforms.ToTensor(), # Convert the image to a PyTorch tensor
                                  transforms.Normalize((0.5,),(0.5,))  # Normalize the image with mean and std
                              ])

transform_test = transforms.Compose([
                                  transforms.ToTensor(), # Convert the image to a PyTorch tensor
                                  transforms.Normalize((0.5,),(0.5,))  # Normalize the image with mean and std
                              ])

# Download the Fashion MNIST dataset and apply transformations
train_dataset=datasets.FashionMNIST('FMNIST/',
                              train=True,
                              download=True,
                              transform=transform_train)

test_dataset=datasets.FashionMNIST('FMNIST/',
                              train=False,
                              download=True,
                              transform=transform_test)
```

https://towardsdatascience.com/a-comprehensive-guide-to-image-augmentation-using-pytorch-fb162f2444be

Advanced Data Augmentation Techniques

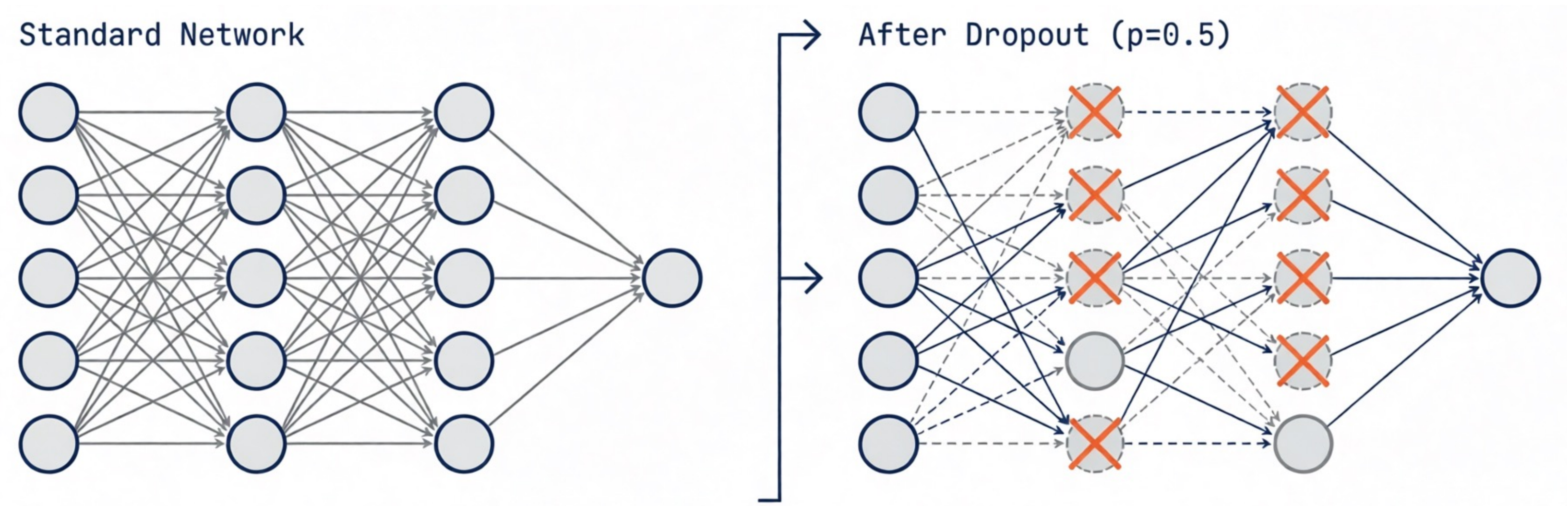
1. **Mixup**: Creates new examples by interpolating between pairs of examples and labels, enhancing generalization.
2. **Cutout**: Randomly masks out square regions of input images, promoting focus on remaining areas.
3. **AutoAugment**: Uses reinforcement learning to automatically discover the best augmentation policies, yielding improved performance on various datasets.



Mixp, Cutout and Cutmix Data Augmentations. (Source)

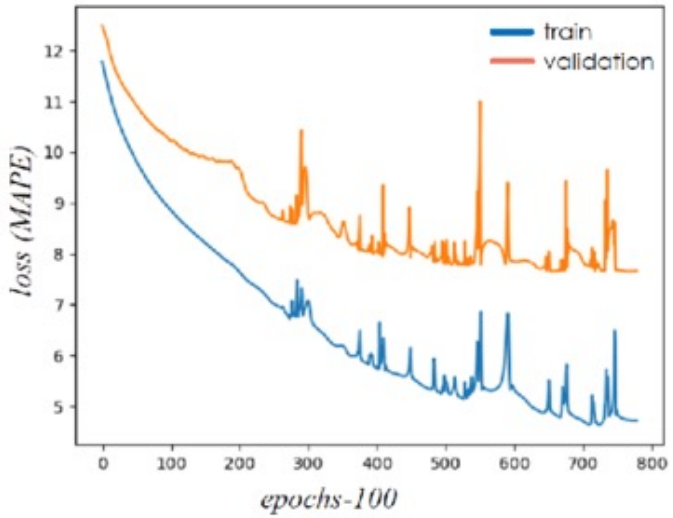
Generalization II: Dropout

Implicit Ensembling. Forces redundancy.

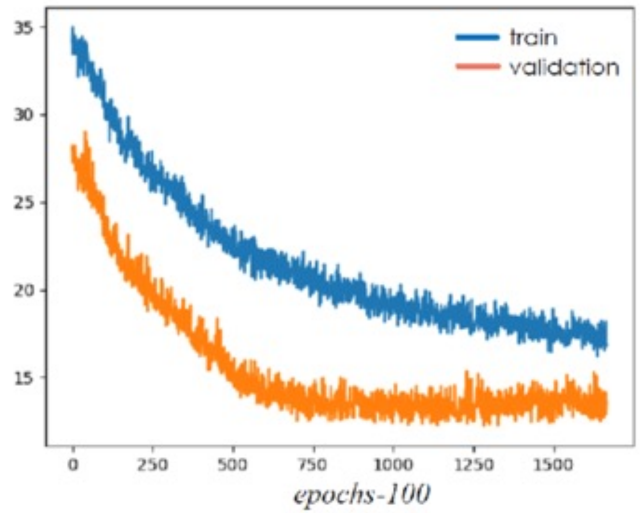


- **Dropout**: Randomly masks neurons (e.g. $p = 0.5$) **during training** to prevent feature co-adaptation. The network cannot rely on any single feature.
  - RESULT: Training Loss UP (Harder), Validation Loss DOWN (Generalization).
  - WARNING: Only use for Overfitting.

Without vs With Dropout



Without Dropout



10% Dropout

Drop-out effect on the loss function during training in parallel. The model with drop-out exhibits a validation loss history with lower values than the corresponding train curve.

```python
class Net(nn.Module):
  def __init__(self, input_shape=(3,32,32)):
    super(Net, self).__init__()

    self.conv1 = nn.Conv2d(3, 32, 3)
    self.conv2 = nn.Conv2d(32, 64, 3)
    self.conv3 = nn.Conv2d(64, 128, 3)

    self.pool = nn.MaxPool2d(2,2)

    n_size = self._get_conv_output(input_shape)

    self.fc1 = nn.Linear(n_size, 512)
    self.fc2 = nn.Linear(512, 10)

    # Define proportion or neurons to dropout
    self.dropout = nn.Dropout(0.25)

  def forward(self, x):
    x = self._forward_features(x)
    x = x.view(x.size(0), -1)
    x = self.dropout(x)
    x = F.relu(self.fc1(x))
    # Apply dropout
    x = self.dropout(x)
    x = self.fc2(x)
    return x
```

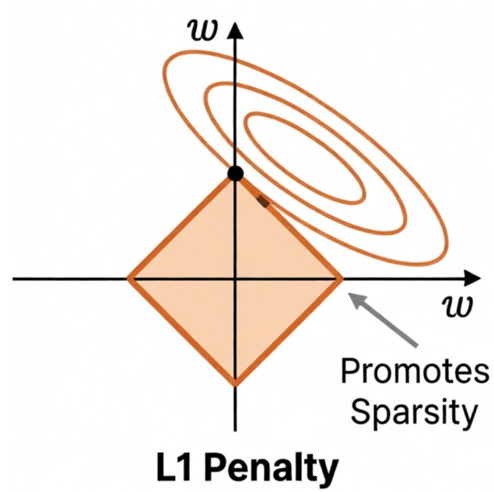
Generalization III: Regularized Objectives

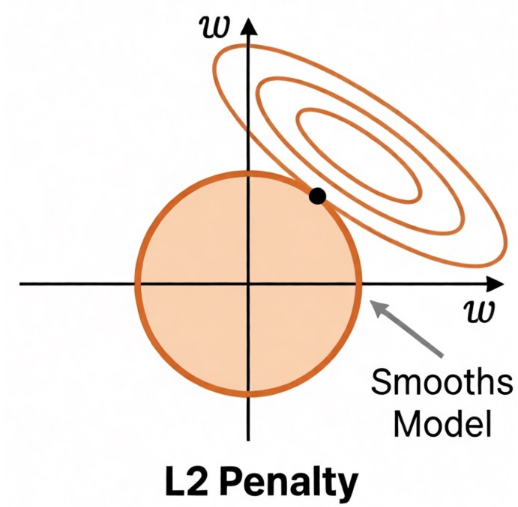
L1/L2 Regularizations

Constraining complexity via the Cost Function:

$$\mathcal{L}_{total} = \mathcal{L}_{data} + \lambda R(\theta)$$

- **L1 (Lasso)**: Adds $\sum|\theta|$
  - $\mathcal{L}_{total} = \mathcal{L}_{data} + \lambda \sum|\theta|$
  - Effect: Promotes sparsity (drives weights to exactly 0).
- **L2 (Ridge)**: Adds $\sum \theta^2$
  - $\mathcal{L}_{total} = \mathcal{L}_{data} + \lambda \sum \theta^2$
  - Effect: Promotes Smoothness (prevents large weights).





L2 Regularization vs Weight Decay in PyTorch

- **L2 regularization** and **Weight decay** are often used interchangeably in PyTorch.
  - **Adam with L2 regularization,** in which weight decay is added a penalty term proportional to the square of the L2 norm to the loss function.

```
optimizer = optim.Adam(model.parameters(), lr=0.01, weight_decay=0.001)
```

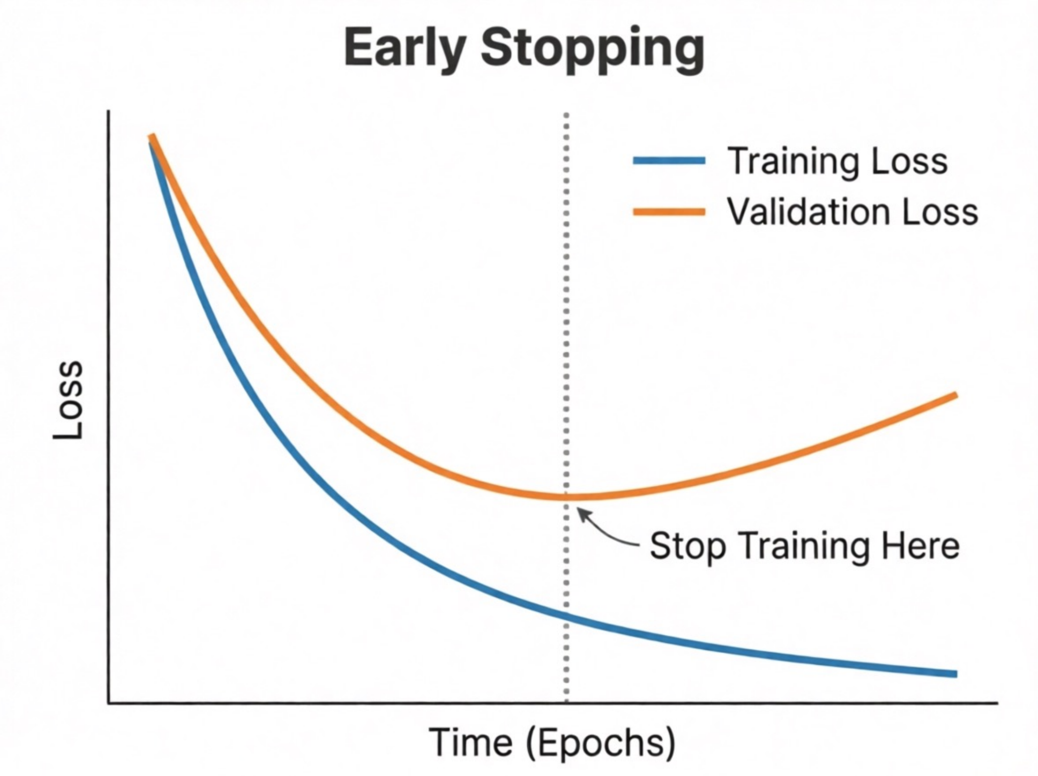
  - **AdamW** is a specific implementation of weight decay regularization in PyTorch, in which weight decay is only applied during parameter update.

```
optimizer = optim.AdamW(model.parameters(), lr=0.01, weight_decay=0.001)
```

Early Stopping: Convergence Strategies

The "Free Lunch". Stop when generalization degrades.

- The simplest, most effective regularizer.
- Stop when validation loss stops improving, even if training loss is still falling.



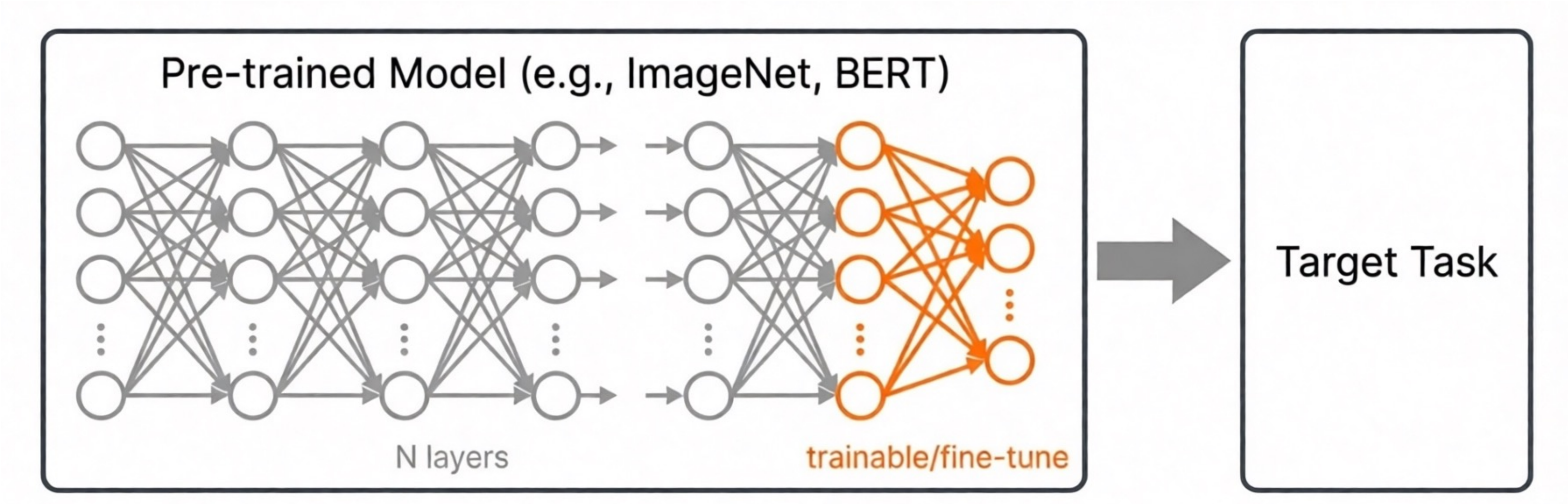
Starting Right: Weight Initialization

Good weight initialization can prevent vanishing/exploding gradients at Step 0.

- **Xavier (Glort) Init**: Best for symmetric activations (Sigmoid Tanh)
  - $w_{ij}^{(l)} \sim \mathcal{U}\left(-\sqrt{\frac{6}{\text{fan}_{in}+\text{fan}_{out}}}, \sqrt{\frac{6}{\text{fan}_{in}+\text{fan}_{out}}}\right)$ OR $w_{ij}^{(l)} \sim \mathcal{N}\left(0, \sqrt{\frac{2}{\text{fan}_{in}+\text{fan}_{out}}}\right)$
- **Kaiming (He) Init**: Best for ReLU family. Acconts for ReLU zeroing half the input
  - $w_{ij}^{(l)} \sim \mathcal{U}\left(-\sqrt{\frac{6}{\text{fan}_{in}}}, \sqrt{\frac{6}{\text{fan}_{in}}}\right)$ OR $w_{ij}^{(l)} \sim \mathcal{N}\left(0, \sqrt{\frac{2}{\text{fan}_{in}}}\right)$

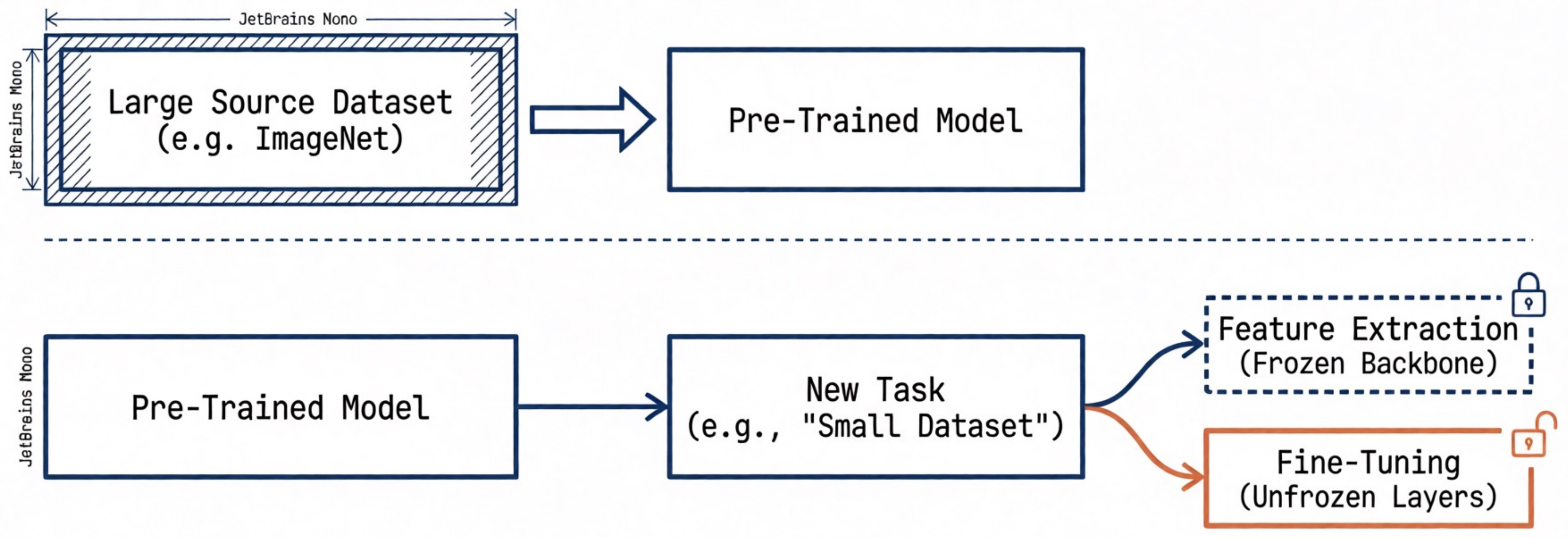
Transfer Learning

Don't start from scratch.



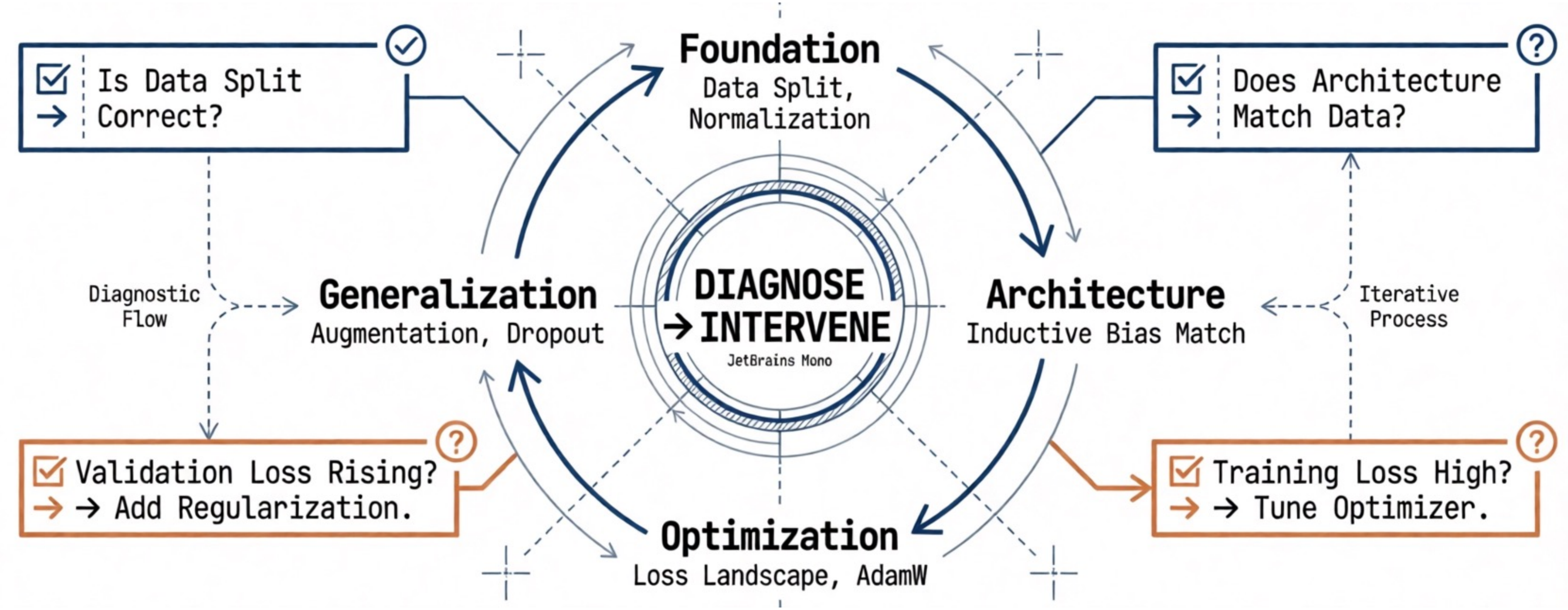
- **Strategy A**: Feature Extraction (Freeze Backbone, train Head).
- **Strategy B**: Fine-Tuning (Unfreeze layers with Discriminative Learning Rates).

Transfer Learning: The Shortcut



- **Why:** 5-10x faster convergence. Mitigates poor initialization.
- **Strategy:** Use pre-trained backbones for small or data-scarce domains.

The Practitioner's Mental Model



Deep Learning is a structured search for the sweet spot between optimization and generalization.

Assignment 1 Section B to Practice these Skills

Image Classification with Multi-Layer Perceptron

- The assignment 1 is now available in the schedule webpage for download. The deadline for the assignment 1 is **Saturday of Week 5 (Feb 21, 2026).**
  - https://www.ee.cityu.edu.hk/~lmpo/ee4016/pdf/2026_EE4016_Ass01.pdf
  - **Colab:** https://colab.research.google.com/drive/1zSe-32cpojFYT2oxySvrAdSbMLr4vY9I#scrollTo=hjkFuokaRv3G
- **The answers of the section A must be handwritten** and then scan the answer sheets into a single pdf file.
- Submit the answer sheets and Colab notebook of the Assignment 1 as a zip file to this CANVAS assignment 1:
  - Filename format : Assignment01_StudentName_StudentID.zip
  - Filename example: Assignment01_Chen_Hoi_501234567.zip