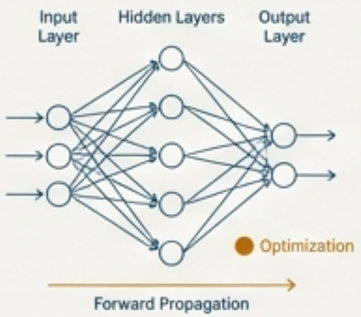
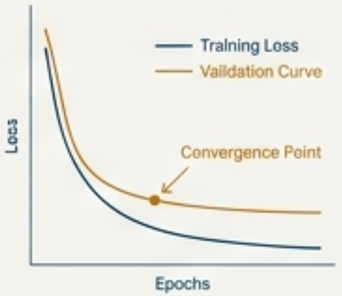
The Deep Learning Architect's Handbook

A Review of Advanced Architectures and Training Techniques





Based on Lecture Notes for IN2064: Deep Learning II

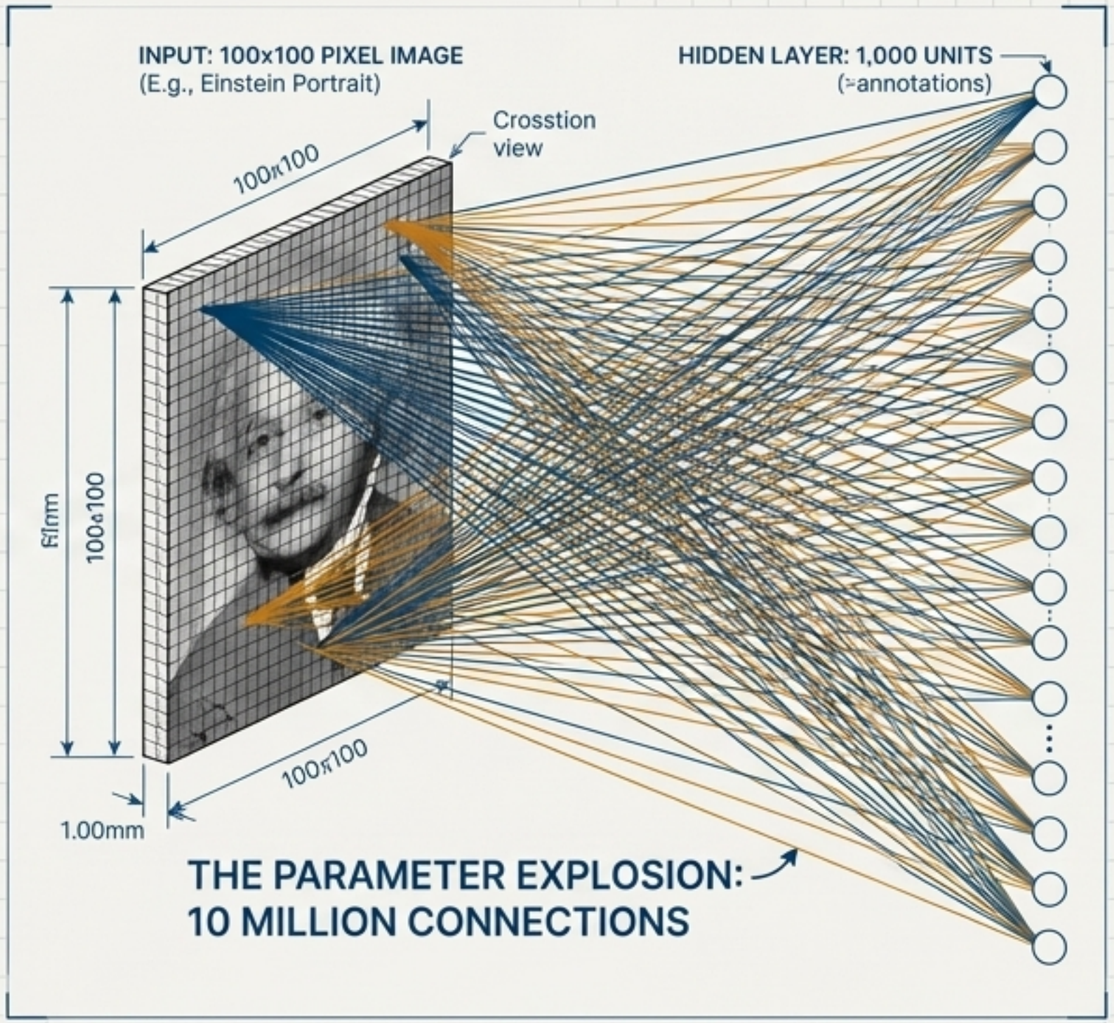
Beyond Fully-Connected: Why We Need Specialized Blueprints

Problem Statement

- Standard feed-forward networks treat all inputs equally, ignoring any underlying structure (e.g., spatial relationships in an image).
- **The Parameter Explosion:** Processing a modest 100x100 pixel image with a single 1,000-unit hidden layer in a fully-connected network results in 10 million parameters. This is computationally expensive and prone to overfitting.

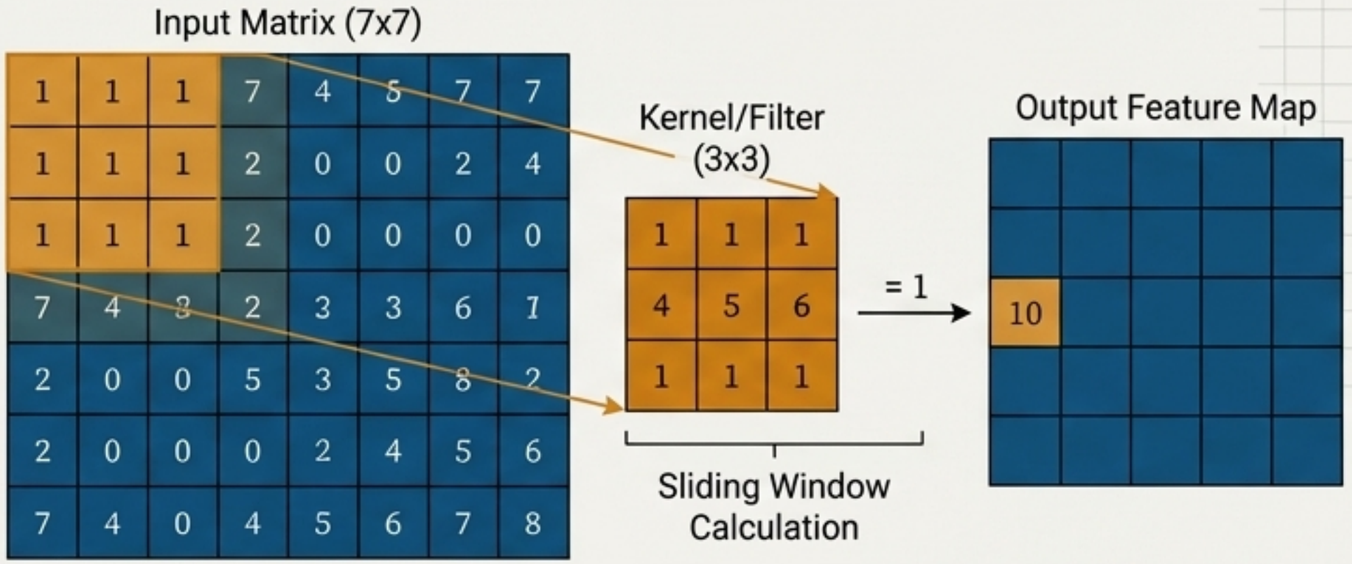
The Architect's Solution

- Leverage the known structure of the data to create an *inductive bias*.
- Introduce specialized layers as new building blocks:
  - **Convolutional Layers** for images (our focus).
  - Recurrent Layers for sequences.
  - Graph Convolutional Layers for graphs.



Visualization of the overwhelming number of parameters in a fully-connected network processing an image, highlighting the lack of structural awareness.

The Convolution Operation: A Smarter Way to See



Core Concept

Convolution is a weighted average of an input signal, using a sliding window called a **kernel** or **filter**. This exploits the high local correlation of pixel values in natural images.

Mathematical Foundation

- While continuous convolution is defined as $(x * k)(t) = \int x(\tau)k(t-\tau)d\tau$, CNNs use a discrete 2D variant.
- In practice, deep learning libraries implement **cross-correlation**, which is functionally similar:

$$\hat{x}(i,j) = \sum_{l=1}^{L}\sum_{m=1}^{M} x(i+l, j+m)k(l,m)$$

The Power of Parameter Sharing

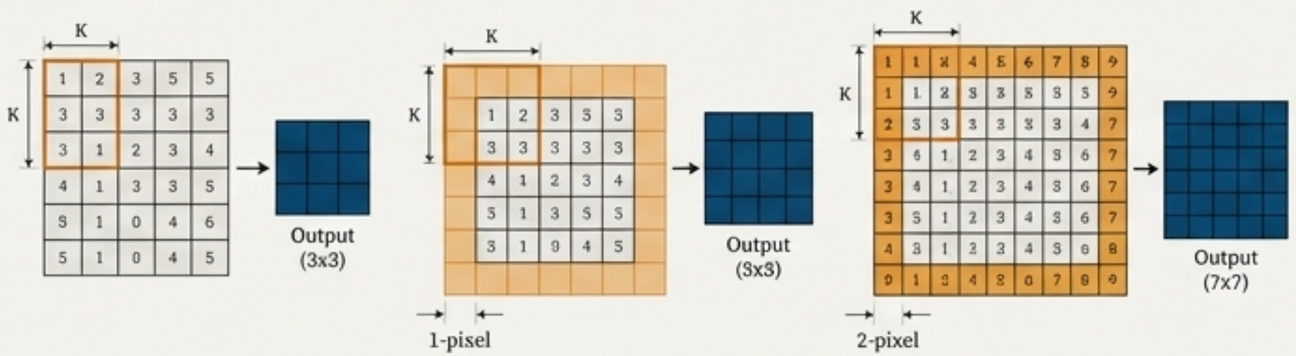
- The same kernel (a small set of weights) is applied across every patch of the input.
- **Example:** Using 1,000 learnable 5x5 convolutional filters on a 100x100 image requires only 25,000 parameters, a dramatic reduction from the 10 million needed for a fully-connected layer.

Engineering the Convolution: Padding and Strides

Handling Boundaries: Padding

Question: What happens when the kernel reaches the edge of the input?
Solutions:

- **VALID Padding:** No padding is used. The output size shrinks.
  $D_{l+1} = (D_l - K) + 1$
- **SAME Padding:** Padding is added to preserve the input dimensions.
  Add $P = \lfloor K/2 \rfloor$ values on each side.
- **FULL Padding:** Padding is added to increase the output size.
  Add $K - 1$ values on each side.



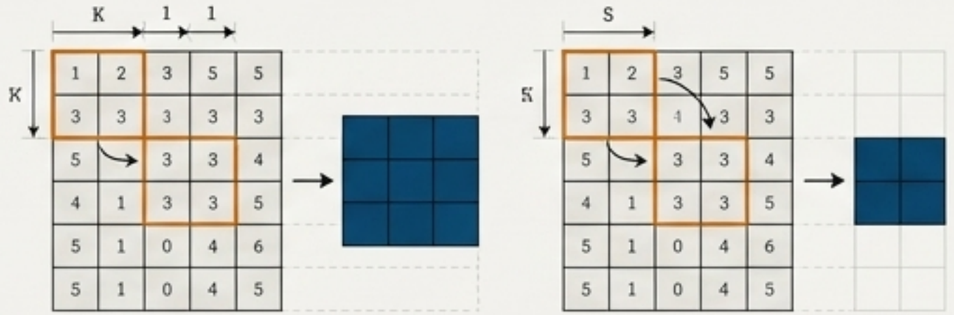
VALID PADDING SAME PADDING FULL PADDING

Controlling Resolution: Strides

Definition: The stride S is the step size the kernel takes as it moves across the input.

Function: Strides greater than 1 are a form of downsampling.

Output Size Formula: $D_{l+1} = \left[\frac{D_l + 2P - K}{S} \right] + 1$



STRIDE=1 STRIDE=2

Assembling the Blueprint: Pooling and a Complete CNN

Downsampling with Pooling

- Purpose: To calculate a summary statistic over a sliding window, reducing dimensionality and creating invariance to small translations.
- Common Types:
  - Max Pooling: Selects the maximum value in the window.
  - Mean Pooling: Calculates the average value in the window.

A Complete CNN Architecture (Example)

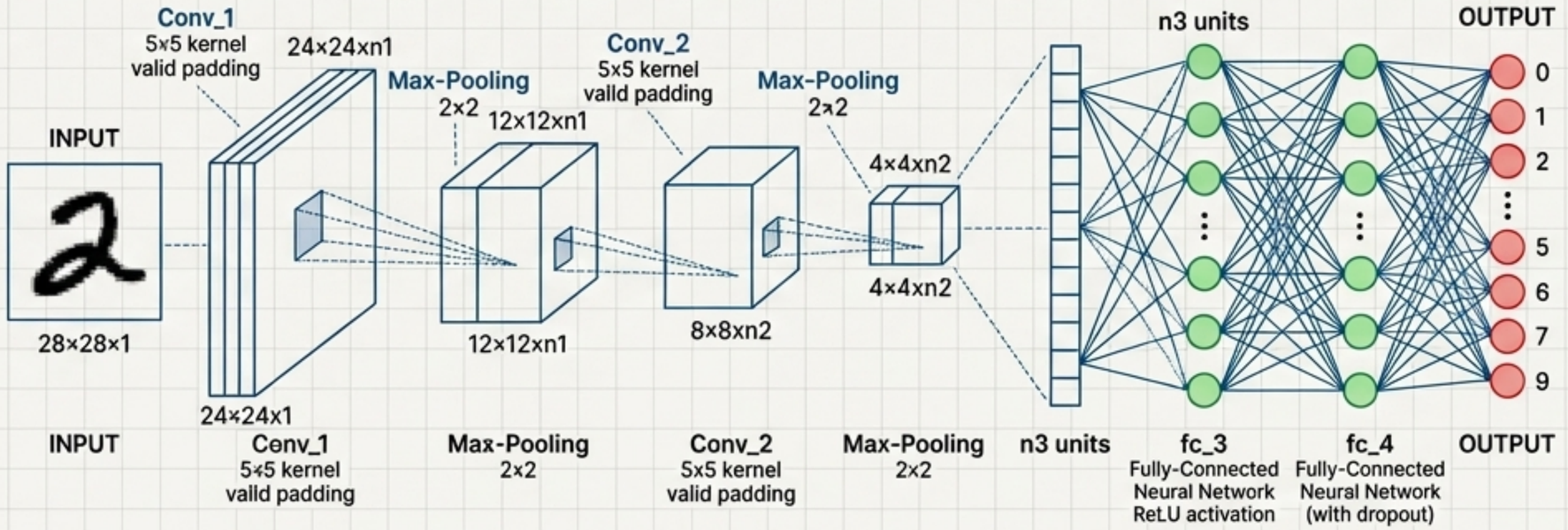
Stage 1:
Input -> Conv Layer
(e.g., 5x5 kernel, valid padding)
-> Max Pooling (e.g., 2x2)

Stage 2:
-> Conv Layer 2
-> Max Pooling 2

Stage 3:
-> Flatten the feature map into a vector.

Stage 4:
-> Fully-Connected Layers with ReLU activation (and dropout for regularization).

Stage 5:
-> Output Layer (e.g., Softmax for classification).



The Modern Training Puzzle: Double Descent

The Classical View (Bias-Variance Tradeoff)

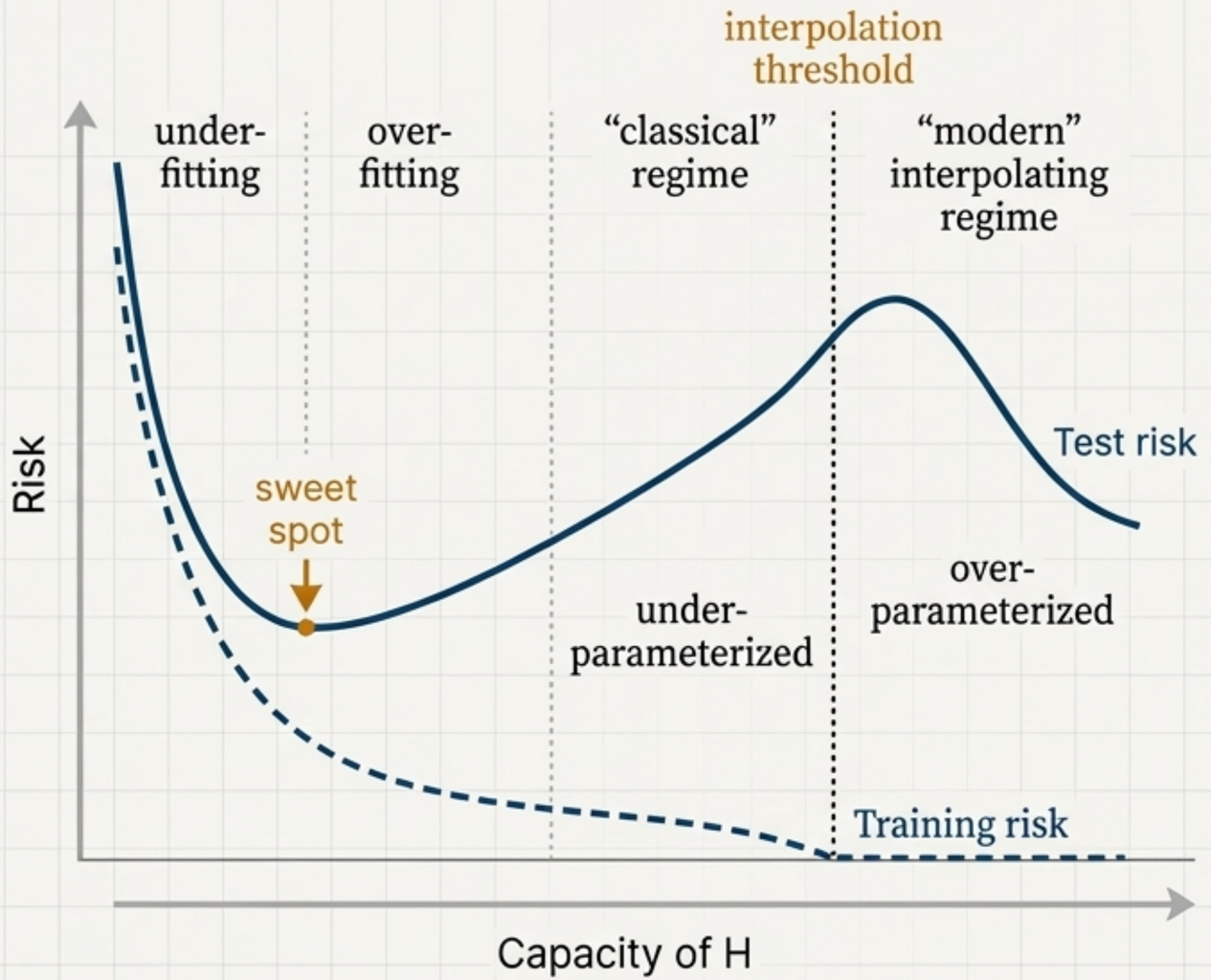
- Increasing model capacity (more parameters) decreases bias but increases variance.
- There's a "sweet spot" before overfitting dominates and test error increases.

The Modern Observation

- Deep neural networks often operate in an over-parameterized, "interpolating regime" where they have far more parameters than training examples.
- Double Descent: As model capacity continues to increase past the interpolation threshold (where training error hits zero), the test error decreases again.
- This behavior, where "bigger is better," is not yet fully understood but is a key characteristic of modern deep learning.

Implication:

Training these massive models successfully requires careful engineering.



The First Step is Critical: Proper Weight Initialization

The Problems with Naive Initialization

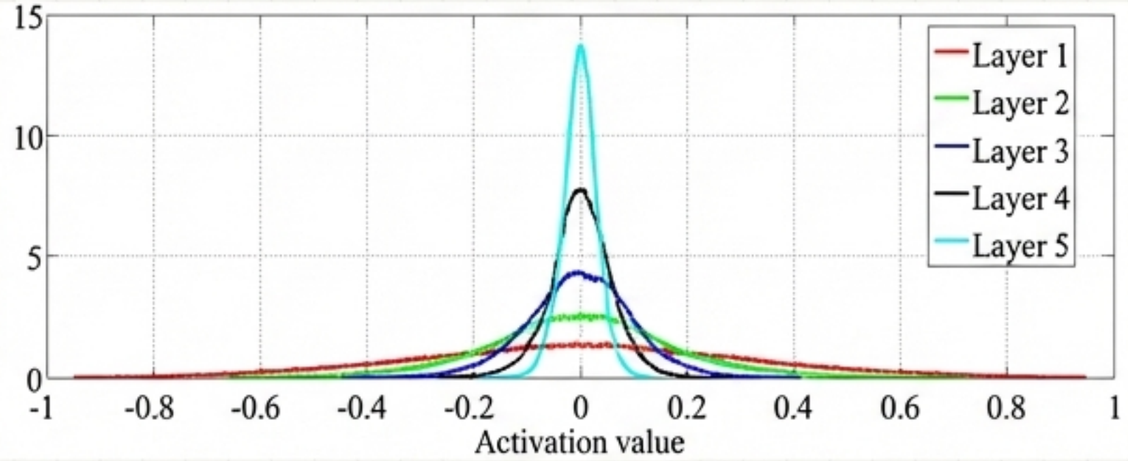
1. **Weight Symmetry:** If all weights start the same, every neuron in a layer learns the same features. They never differentiate.
   - **Solution:** Break symmetry with small *random* values.
2. **Incorrect Weight Scale:**
   - **Vanishing Gradients:** If weights are too small, activations shrink through layers until gradients are nearly zero.
   - **Exploding Gradients:** If weights are too large, activations grow exponentially until they saturate or cause numerical overflow.

The Solution: Xavier (Glorot) Initialization

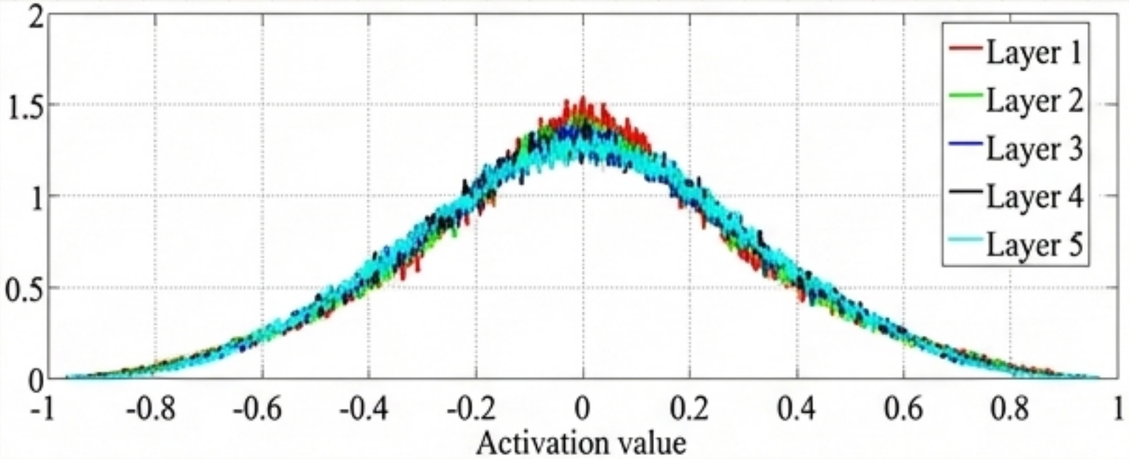
- **Goal:** Preserve the mean (zero) and variance of the signal in both the forward and backward passes.
- **Method:** Initialize weights from a distribution with zero mean and a specific variance: $Var(W) = \frac{2}{\text{fan-in} + \text{fan-out}}$
- **Example Implementation:**

$$W \sim \text{Uniform}\left(-\sqrt{\frac{6}{\text{fan-in} + \text{fan-out}}}, \sqrt{\frac{6}{\text{fan-in} + \text{fan-out}}}\right)$$

Unnormalized



Glorot Initialization



Building Robust Structures: Regularization and Dropout

Preventing Overfitting in High-Capacity Models

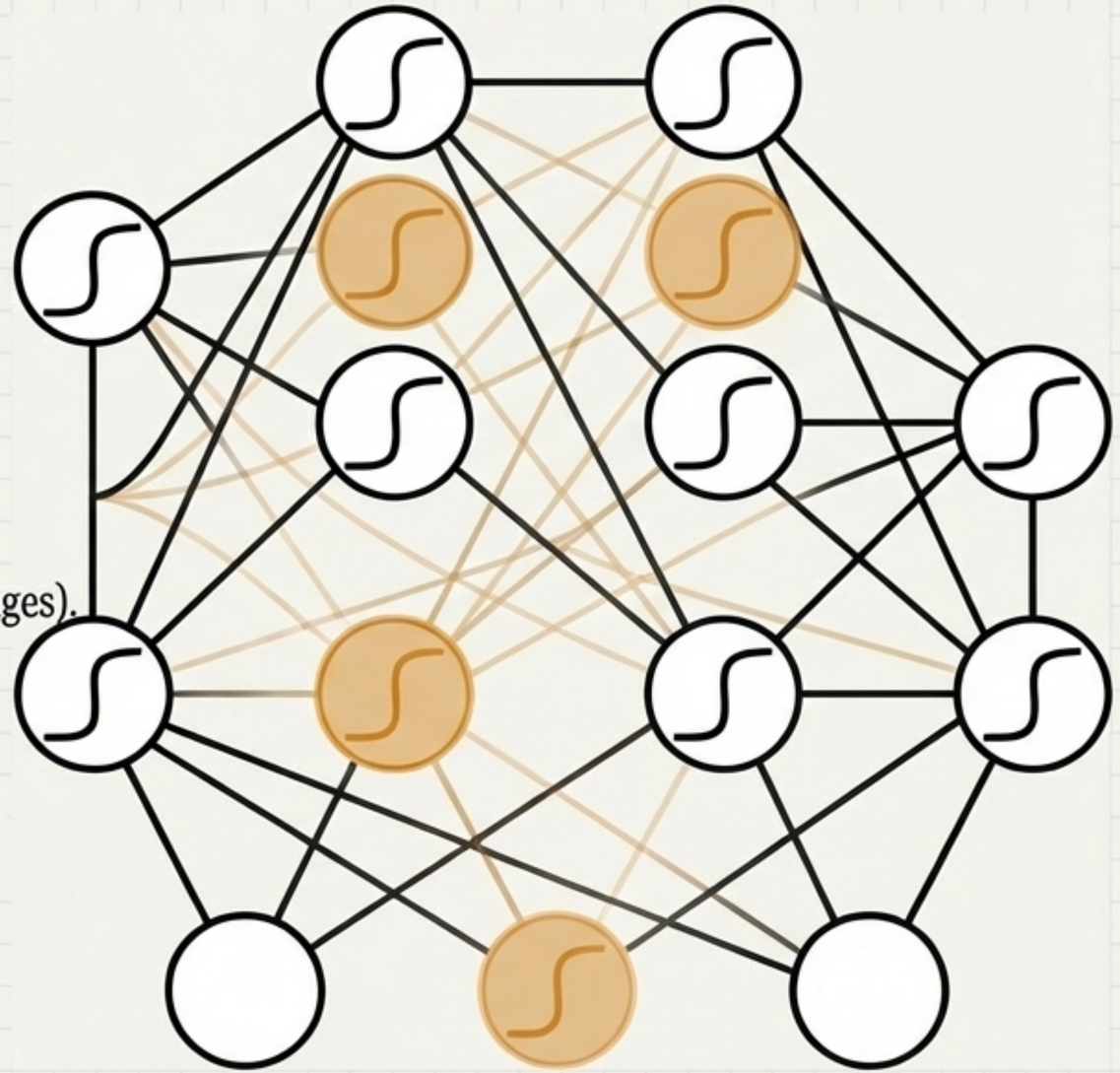
High-capacity neural networks can easily memorize the training data. Regularization is essential for good generalization.

Standard Techniques

- **L2 Parameter Norm Penalty** (Weight Decay)
- **L1 Parameter Norm Penalty** (Promotes Sparsity)
- **Early Stopping**: Monitor validation loss and stop training when it begins to increase.
- **Dataset Augmentation**: Create more training data by transforming existing samples (e.g., rotating, skewing, or changing the lighting of images).

A Neural Network-Specific Technique: Dropout

- **Mechanism**: During training, for each sample, randomly set the output of each hidden unit to zero with some probability p (e.g., p=0.5).
- **Intuition:** This prevents complex co-adaptations where neurons rely on specific other neurons being present. It forces the network to learn more robust, redundant features.
- **Effect**: It's like training an ensemble of 2^H smaller networks that share weights, all at once.



Fine-Tuning the Design: Hyperparameter Optimization

The Architect's Control Panel

Squeezing performance out of a neural network requires tuning numerous hyperparameters:

- **Architecture:** Number of layers, number of units per layer, activation function (ReLU, Sigmoid, etc.).
- **Optimization:** Optimizer (SGD, Adam, etc.), learning rate schedule (warmup, decay).
- **Regularization:** Dropout rate, weight decay strength.
- **Data Handling:** Preprocessing, augmentation strategies.

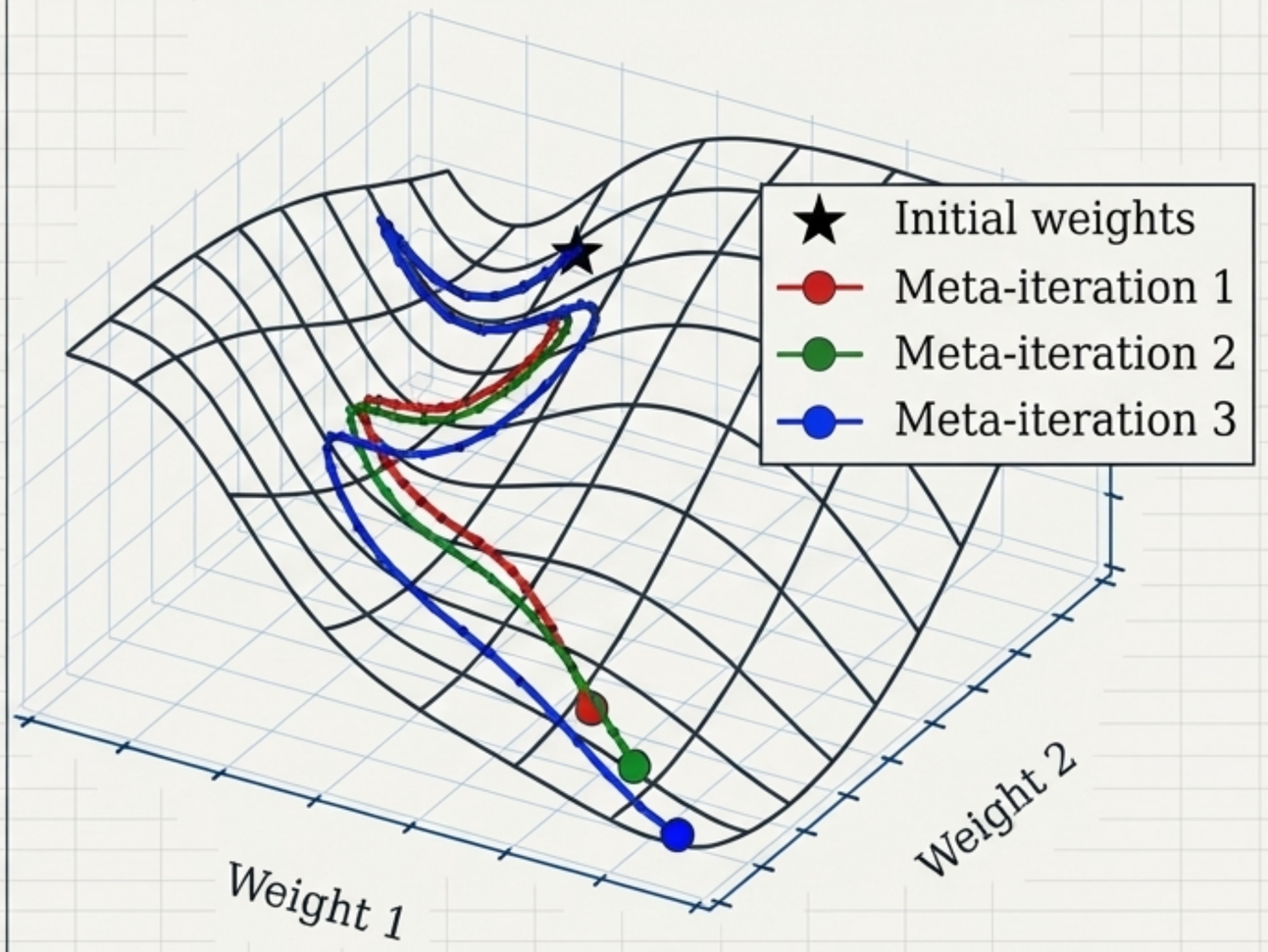
Systematic Approaches

While initial tuning can be done by "playing around," systematic methods are more effective.

- **Random Search:** Often more effective than grid search for finding good hyperparameter combinations.
- **Bayesian Optimization:** A model-based approach that intelligently chooses the next hyperparameters to test based on past results.

Advanced Concept: Meta-Learning

Instead of searching, we can learn optimal hyperparameters (like learning rates) using gradient descent itself. This involves backpropagating through the entire training process, which is computationally expensive but powerful.



The Modern Workshop: Deep Learning Frameworks

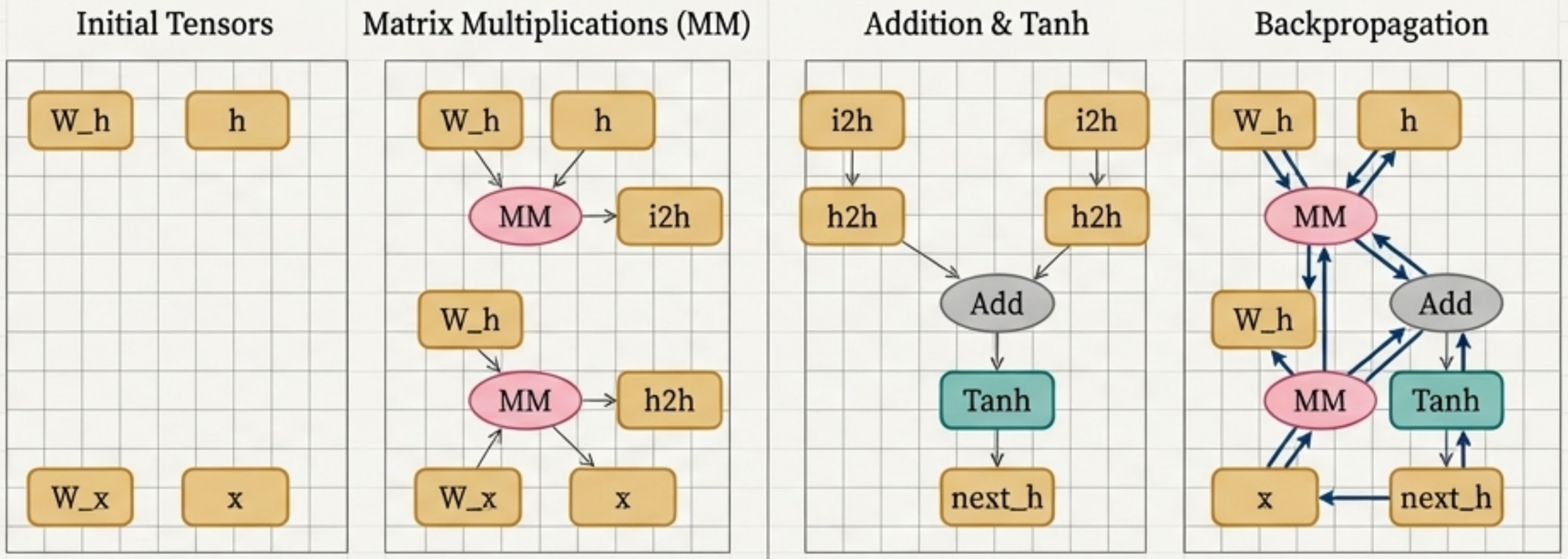
Why Use a Framework?

- Programming neural networks from scratch in Python is educational but inefficient for real-world applications.
- Frameworks provide optimized, GPU-accelerated building blocks for rapid development.
- **Key Players:** PyTorch, TensorFlow, JAX.

Core Abstraction: The Computational Graph

Frameworks build a graph that defines the sequence of operations (e.g., matrix multiply, add, apply activation). Backpropagation is then performed automatically on this graph.

Dynamic Graph Construction (Define-by-Run)




```
x = Variable(...)
prev_h = Variable(...)
W_h = Variable(...)
W_x = Variable(...)
```

```
h2h = torch.mm(W_x, x.t())
i2h = torch.mm(W_h, prev_h.t())
```

```
next_h = i2h + h2h
next_h = next_h.tanh()
```

Back-propagation uses the dynamically built graph

```
next_h.backward(torch.ones(1, 20))
```

Two Paradigms:

1. **Static Graphs ('Define-and-Run'):** The entire graph is defined first, then executed. **Pro:** Allows for compiler-like optimizations. **Con:** Inflexible.
2. **Dynamic Graphs ('Define-by-Run'):** The graph is created on the fly as operations are executed. **Pro:** Highly flexible and intuitive. **Trend:** Most frameworks have moved to a dynamic-first approach.

Stabilizing the Build: Batch Normalization

Goal: Accelerate and stabilize the training of deep networks.

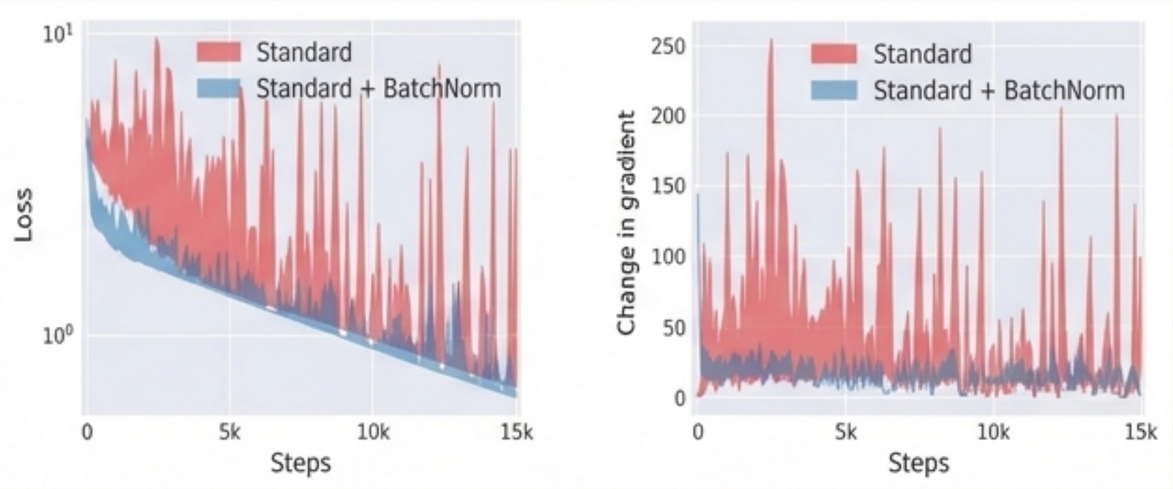
- **Original Hypothesis (Internal Covariate Shift):** BN was thought to work by reducing changes in the distribution of layer inputs during training.
- **Modern Understanding:** Its effectiveness is primarily due to smoothing the loss landscape.

Mechanism:

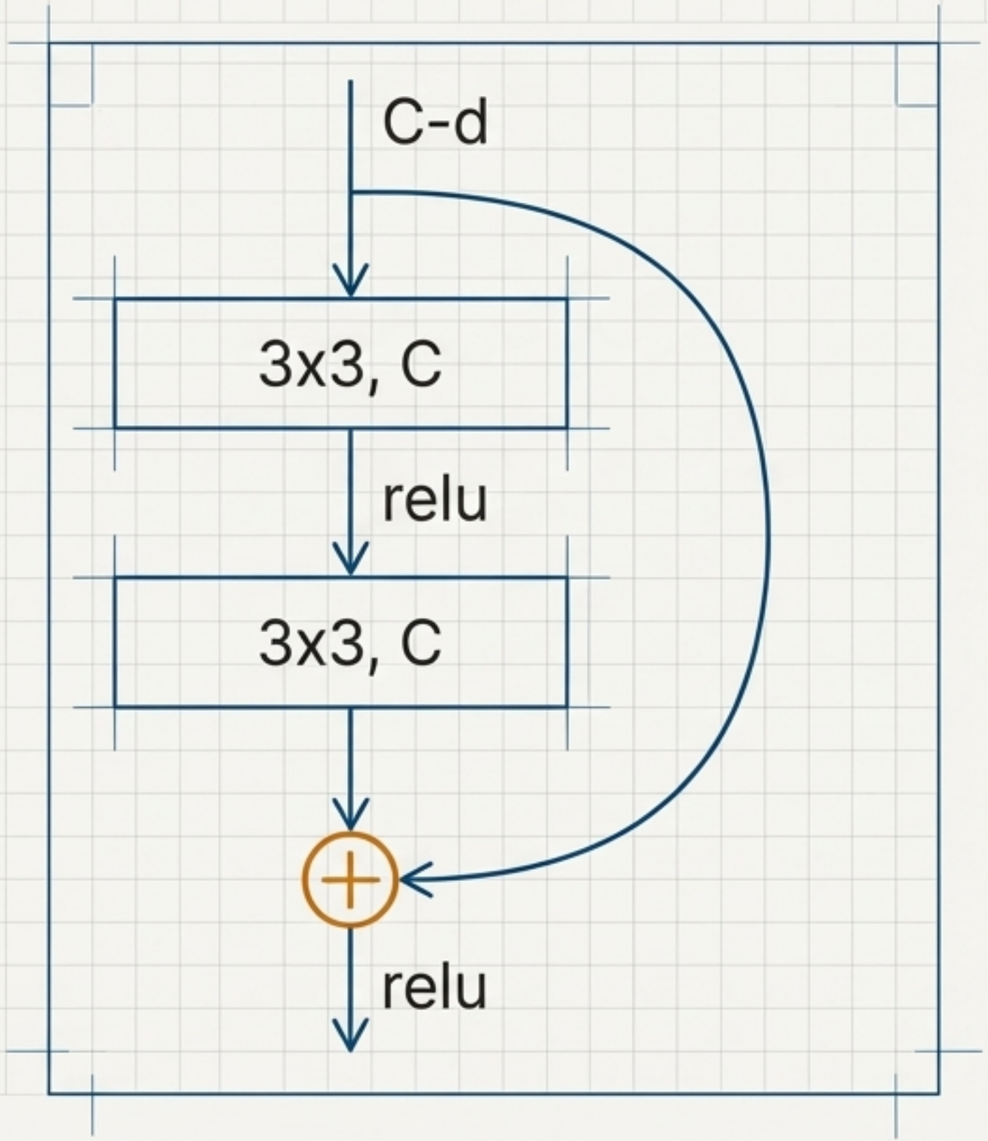
1. For each mini-batch B, normalize the layer's activations to have zero mean and unit variance.
$$\hat{x} = \frac{x - E_B[x]}{\sqrt{Var_B[x] + \epsilon}}$$
2. Introduce learnable scale (γ) and shift (β) parameters to restore the network's representational power.
$$y = \gamma\hat{x} + \beta$$
3. This layer is typically inserted after a linear/convolutional layer and *before* the non-linear activation function.

The Benefit:

Reduces the dependence of gradients on the parameter scale and and initialization. Leads to more predictive and stable gradients, allowing for higher learning rates and faster convergence.



Enabling Depth: Skip Connections



The Problem of Depth

- As networks get deeper, it becomes harder for gradients to propagate back to the initial layers (vanishing gradient problem).
- Information can be lost or "washed out" as it passes through many layers.

The Solution: Create Shortcuts

Skip connections improve information flow by adding the output of a previous layer to a subsequent layer.

Key Variants

- **Residual Connections (ResNet)**: The input x to a block of layers is added directly to the output of the block: **output** = $F(x) + x$. This allows the network to easily learn an identity function if a block is not needed, which simplifies optimization.
- **Highway Networks**: A more complex version that uses a learnable gating mechanism T to control how much information passes through the shortcut versus the transformation: $y = f(x,W)T(x,W\_T) + x(1-T(x,W\_T))$.

Impact

These connections are the key innovation that allowed for the successful training of networks with hundreds or even thousands of layers.

A Foundational Question: When is a Neural Net Truly More Powerful?

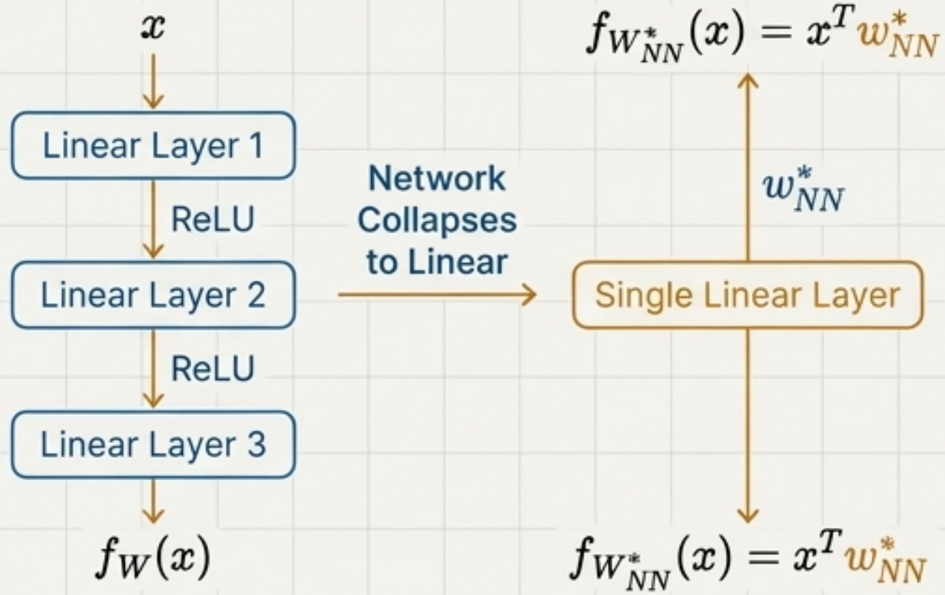
The Scenario

We are solving a regression task on data with non-negative features ($x_n \in \mathbb{R}^D_{\geq 0}$). We compare two models:

1. **Linear Regression**: Minimizes $L_{LS}(w) = \frac{1}{2}\sum(w^T x_n - y_n)^2$. Optimal weights are w^*_{LS}.
2. **Deep ReLU Network**: A multi-layer network with ReLU activations and no biases. Minimizes $L_{NN}(W) = \frac{1}{2}\sum(f_W(x_n) - y_n)^2$. Optimal weights are W^*_{NN}.

Case 1: The Network Collapses

- **Assumption**: The optimal NN weights (W^*_{NN}) happen to be non-negative.
- **Insight**: Since features x and weights W are non-negative, $\text{ReLU}(xW) = xW$. Every ReLU layer becomes a linear layer.
- The entire network simplifies to a linear model: $f_{W^*_{NN}}(x_i) = x_i^T(W_1^* W_2^* \cdots W_{L+1}^*) = x_i^T w^*_{NN}$.
- **Conclusion**: In this specific case, the network *is* a linear model. The optimal losses will be equal: $L_{NN}(W^*_{NN}) = L_{LS}(w^*_{LS})$.



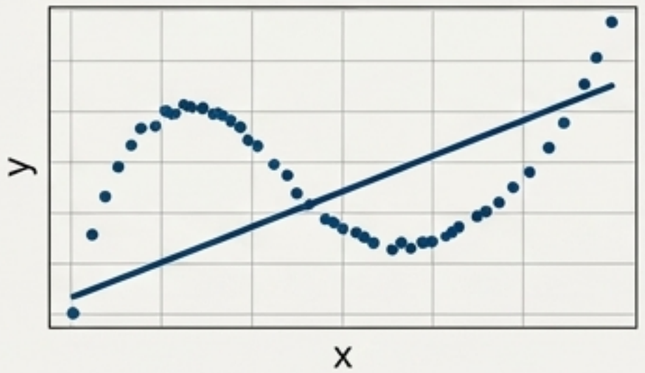
The Expressive Power of Non-Linearity

The Scenario (Revisited)

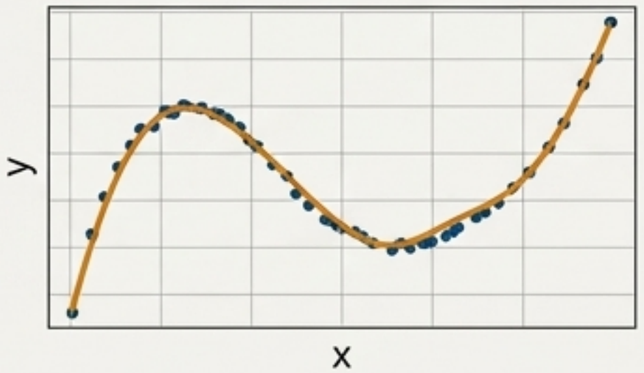
Case 2: The Power of Superset Functions

- **Assumption:** The optimal *linear regression* weights (w^*_{LS}) are non-negative.
- **Insight:** This assumption tells us nothing about the optimal NN weights W^*_{NN}, which could be negative. The NN is free to use its non-linear capabilities.
- Linear regression is a *special case* of the neural network. We can always construct NN weights W_{NN} to perfectly replicate *any* linear model (as shown in Case 1).
- However, the reverse is not true. The NN can learn more complex, non-linear functions that a linear model simply cannot represent.
- **Conclusion:** The neural network's loss will be less than or equal to the linear model's loss. It has the potential to find a better fit for the data.
  - $L_{NN}(W^*_{NN}) \leq L_{LS}(w^*_{LS})$

Linear Regression



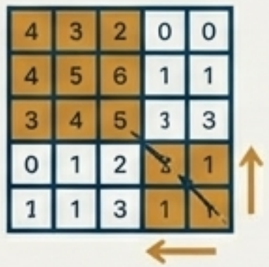
Neural Network



The Takeaway

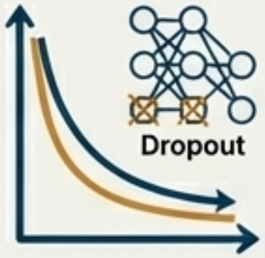
The power of a deep network lies in its ability to learn a much richer class of functions. It contains linear models as a subset but can go far beyond them by leveraging non-linear activations.

The Architect's Handbook: A Summary



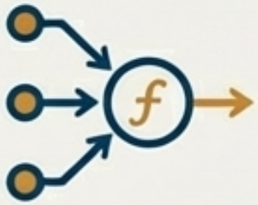
PART 1: The Blueprint (New Components)

- CNNs: The essential tool for structured data like images, using convolution, padding, strides, and pooling to efficiently extract features with shared weights.



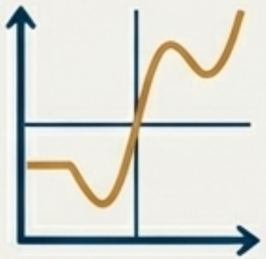
PART 2: Mastering the Build (Training Techniques)

- Initialization: Use Xavier/Glorot initialization to prevent vanishing/exploding gradients.
- Regularization: Employ Dropout, weight decay, and data augmentation to combat overfitting in high-capacity models.



PART 3: Advanced Engineering (Modern Tools)

- Frameworks: Leverage PyTorch/TensorFlow for efficient development via computational graphs.
- Batch Normalization: Stabilize and accelerate training by normalizing layer activations.
- Skip Connections: Enable the construction of very deep networks by improving gradient flow.



PART 4: Theoretical Foundations (Model Power)

- A deep ReLU network's expressive power allows it to outperform linear models, as it can represent a strictly larger family of functions.