CS-470: Machine Learning

Week 11 - Training Neural Networks: A Deep Dive

Instructor: Dr. Sajjad Hussain

Department of Electrical and Computer Engineering
SEECS, NUST

November 27th, 2025

Overview

The Vanishing Gradient Problem

- In deep networks, gradients become extremely small as they are backpropagated from the output layer to the initial layers.
- This means **early layers learn very slowly** or not at all, while later layers converge.
- The core of the problem lies in the **chain rule** and the **activation function**.

The Chain Rule in Backpropagation

For a weight $w^{(1)}$ in the first layer, the gradient is:

$$\frac{\partial \mathcal{L}}{\partial w^{(1)}} = \underbrace{\frac{\partial \mathcal{L}}{\partial a^{(L)}}}_{\text{Output Grad}} \cdot \underbrace{\frac{\partial a^{(L)}}{\partial a^{(L-1)}} \cdots \frac{\partial a^{(2)}}{\partial a^{(1)}}}_{\text{Many terms}} \cdot \frac{\partial a^{(1)}}{\partial z^{(1)}} \cdot \frac{\partial z^{(1)}}{\partial w^{(1)}}$$

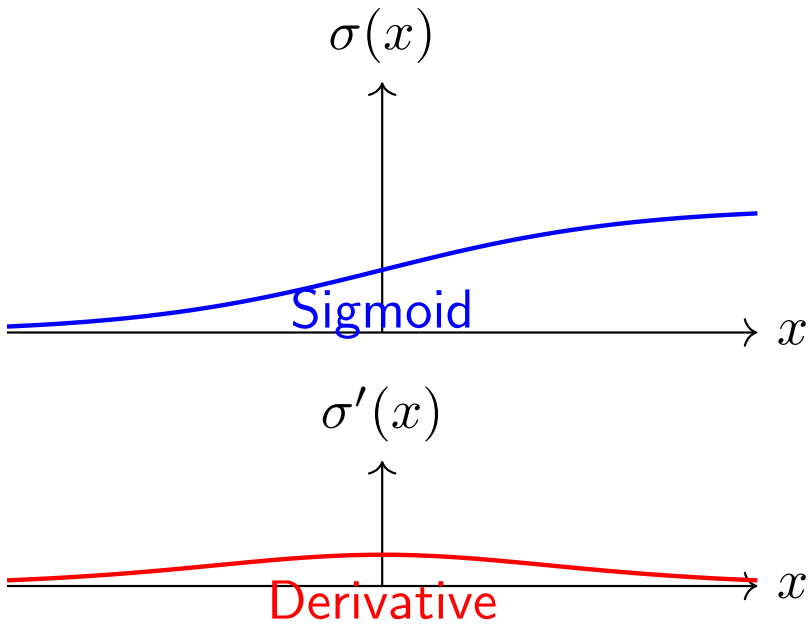
Each term $\frac{\partial a^{(l)}}{\partial a^{(l-1)}}$ depends on the derivative of the activation function, $g'(z^{(l-1)})$.

Why Sigmoid Causes Vanishing Gradients

The Sigmoid function and its derivative:

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

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$



- Maximum value of $\sigma'(x)$ is 0.25.
- For most inputs ($|x| > 4$), $\sigma'(x) \approx 0$.

The Problem: $g'(z) \leq 0.25$. In a deep network, multiplying many such small numbers (< 1) causes the product to **vanish exponentially**.

$$\frac{\partial \mathcal{L}}{\partial w^{(1)}} \propto (0.25 \times 0.25 \times \cdots \times 0.25) \approx 0$$

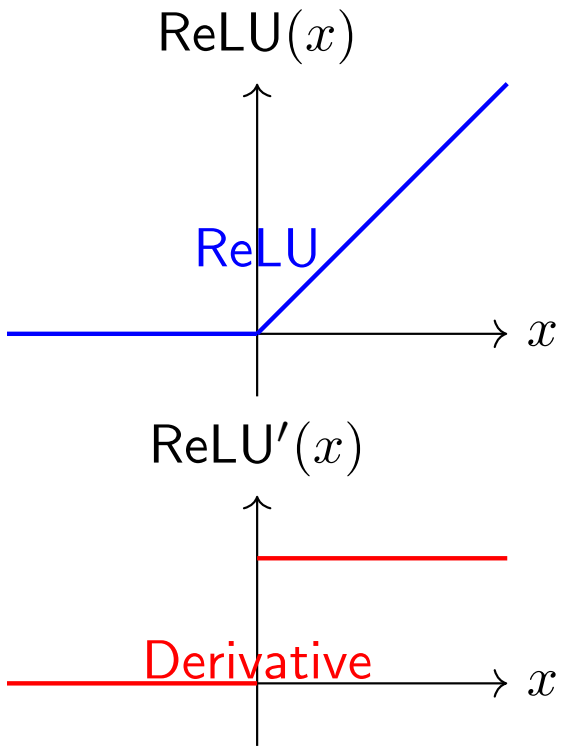
The ReLU Solution

The ReLU (Rectified Linear Unit) function:

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

Its derivative:

$$\text{ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$



How ReLU Helps:

- For active neurons ($x > 0$), the derivative is **exactly 1**.
- This prevents the gradient from shrinking *purely* due to the activation function when the neuron is active.
- The product in the chain rule is now $\propto 1 \times 1 \times \cdots \times 1 = 1$, preventing exponential vanishing.

Glorot (Xavier) and He Initialization

The Goal: Prevent the outputs of the layers from exploding or vanishing *during the forward pass*, even in deep networks.

Intuition

We want the **variance** of the outputs of each layer to be equal to the variance of its inputs. Similarly, we want the gradients to have the same variance during backpropagation.

For a linear layer: $z = Wx + b$. Assuming $E[W] = 0$, $E[x] = 0$:

$$\text{Var}(z) = n_{\text{in}} \cdot \text{Var}(W) \cdot \text{Var}(x)$$

To have $\text{Var}(z) = \text{Var}(x)$, we need:

$$n_{\text{in}} \cdot \text{Var}(W) = 1 \quad \Rightarrow \quad \text{Var}(W) = \frac{1}{n_{\text{in}}}$$

Considering backpropagation as well, Glorot et al. proposed a compromise.

Glorot vs. He Initialization

Glorot/Xavier Initialization (for Tanh/Sigmoid)

Balances the variance for both forward and backward passes.

$$W \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}\right) \quad \text{or} \quad \mathcal{U}\left(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right)$$

He Initialization (for ReLU and variants)

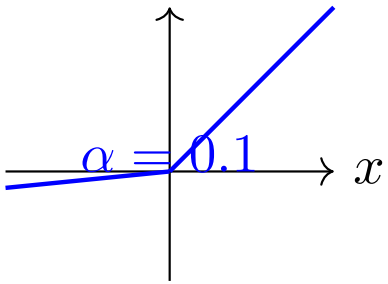
Accounts for the fact that ReLU sets half its inputs to zero, effectively halving the variance. It focuses on the forward pass variance.

$$W \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{\text{in}}}}\right) \quad \text{or} \quad \mathcal{U}\left(-\sqrt{\frac{6}{n_{\text{in}}}}, \sqrt{\frac{6}{n_{\text{in}}}}\right)$$

Summary: Use the right initialization for your activation function to get stable training from the start.

Variants of ReLU

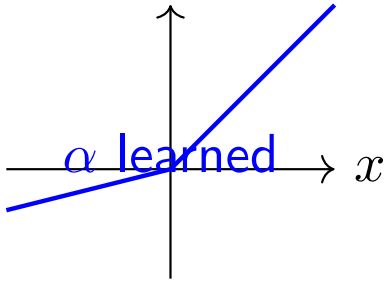
Leaky ReLU



$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$$

Prevents "dying ReLU". α is a small constant (e.g., 0.01).

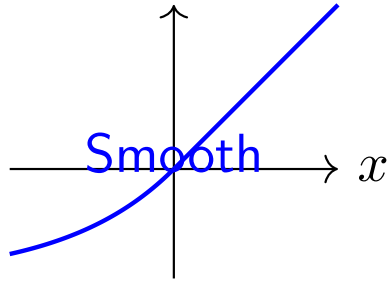
Parametric ReLU (PReLU)



$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$$

The slope α is a **learnable parameter**.

Exponential LU (ELU)



$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}$$

Smooth transition, helps push mean activations towards zero.

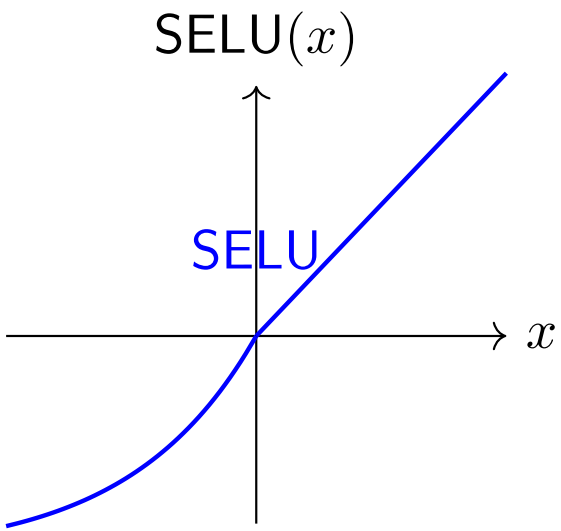
Scaled Exponential LU (SELU)

SELU is a self-normalizing activation function.

$$\text{SELU}(x) = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}$$

With $\alpha \approx 1.6733$ and $\lambda \approx 1.0507$.

- Under specific conditions (LeCun initialization), SELU networks **self-normalize**: the outputs of each layer tend to preserve zero mean and unit variance.
- This inherently avoids vanishing/exploding gradients.
- Very effective for deep feedforward networks.



Batch Normalization: The Idea

Internal Covariate Shift: The change in the distribution of network activations due to the change in network parameters during training. This slows down training.

BatchNorm Solution

Normalize the inputs of each layer to have **zero mean** and **unit variance**, for each mini-batch. This stabilizes the distribution of activations.

Figure: BatchNorm is applied *after* the linear transformation and *before* the non-linear activation.

Batch Normalization: The Algorithm

For a layer output (pre-activation) $\mathbf{z}$ over a mini-batch $\mathcal{B} = \{z_1, \ldots, z_m\}$:

1. Calculate the mean and variance of the batch:

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} z_i \qquad \sigma_{\mathcal{B}}^2 = \frac{1}{m}\sum_{i=1}^{m}(z_i - \mu_{\mathcal{B}})^2$$

2. Normalize the values:

$$\hat{z}_i = \frac{z_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$

   (ϵ is a small constant for numerical stability)

3. Scale and shift (this is crucial!):

$$\text{BN}(z_i) = \gamma \hat{z}_i + \beta$$

Key: γ (scale) and β (shift) are **learnable parameters**. They allow the network to *decide* if normalization is useful, and to represent the identity function if that's optimal.

Why BatchNorm Works So Well

- **Allows Higher Learning Rates:** By stabilizing gradients, it prevents updates from blowing up the model.
- **Reduces Sensitivity to Initialization:** Less careful initialization is needed.
- **Acts as a Regularizer:** The output for a given example depends on the statistics of the entire mini-batch, adding noise. This reduces overfitting.
- At test time, we use **running averages** of the mean and variance computed during training, not the batch statistics.

Transfer Learning: Leveraging Pre-trained Models

"Why train from scratch when someone has already done the hard work?"

Scenarios:

1. **Small target dataset, similar to source:** Freeze all convolutional layers, train only the classifier head.
2. **Medium target dataset, similar to source:** Freeze early layers, fine-tune later layers and the classifier.
3. **Large target dataset, similar to source:** Fine-tune the entire network (this is just using the pre-trained weights as a very good initialization).
4. **Dataset not similar to source:** Transfer learning might not help much. Consider training from scratch.

Transfer Learning

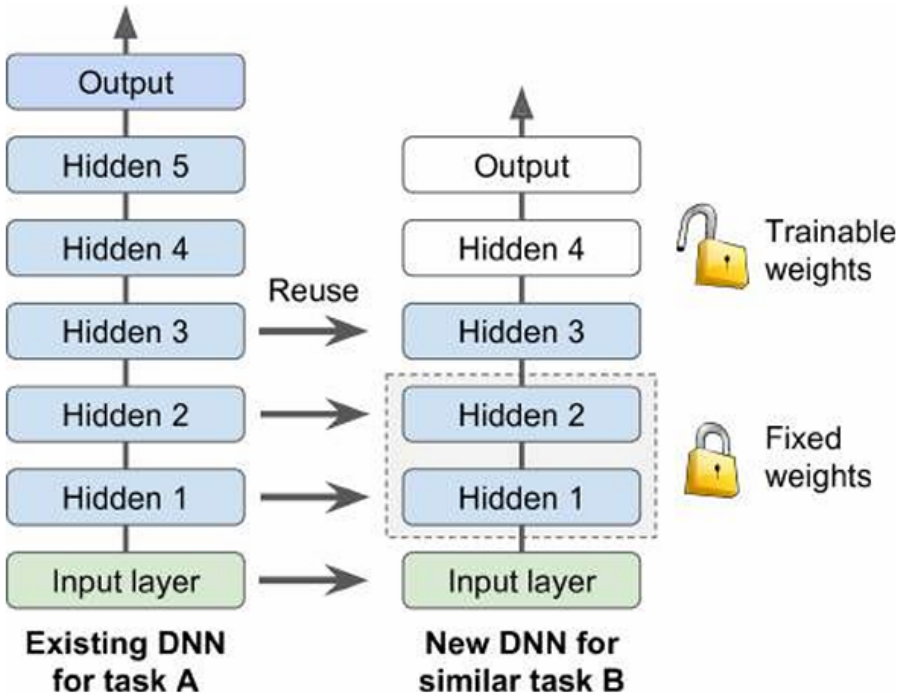


Figure: Transfer Learning[1]

[1] Image adapted from Aurélien Géron, "Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow"

Optimizers: Vanilla Stochastic Gradient Descent (SGD)

The Core Update Rule

The simplest form of gradient descent updates the parameters θ using only the current gradient $\nabla J(\theta_t)$:

$$\theta_{t+1} = \theta_t - \eta g_t \nabla_\theta J(\theta_t)$$

where $g_t = \nabla_\theta J(\theta_t)$ is the current gradient and η is the learning rate.

The Problem:

- The update is *myopic*; it only looks at the gradient **at this very moment**.
- As we approach a minimum, the gradient $\nabla J(\theta_t)$ naturally becomes smaller.
- Therefore, the update step $-\eta \nabla J(\theta_t)$ also becomes smaller, leading to very slow convergence near the optimum.
- It has no memory of past gradients, making it susceptible to getting stuck in saddle points and sensitive to noisy gradients.

Summary: SGD slows down near the minimum because its update is directly proportional to the gradient, which vanishes at the optimum.

Optimizers: SGD with Momentum

The Idea: Simulate a heavy ball rolling downhill. It doesn't just follow the current slope but maintains **momentum** from previous gradients.

The Algorithm

We introduce a velocity vector v_t that accumulates past gradients.

$$v_t = \beta v_{t-1} + \eta g_t$$
$$\theta_{t+1} = \theta_t - v_t$$

- β is the momentum term (e.g., 0.9). It dictates how much of the past velocity is retained.
- v_t is the **exponentially decaying average** of all past gradients.

Mathematical Insight: Let's unroll the velocity term for the first few steps:

$$v_0 = 0, \qquad v_1 = \beta v_0 + \eta g_1 = \eta g_1$$
$$v_2 = \beta v_1 + \eta g_2 = \beta \eta g_1 + \eta g_2$$

SGD with Momentum: Why It Works

$$v_3 = \beta v_2 + \eta g_3 = \beta(\beta \eta g_1 + \eta g_2) + \eta g_3 = \beta^2 \eta g_1 + \beta \eta g_2 + \eta g_3$$
$$v_t = \eta(g_t + \beta g_{t-1} + \beta^2 g_{t-2} + \dots)$$

The update is a weighted sum of **all previous gradients**, with more recent gradients weighted more heavily.

Benefits:

- **Faster Convergence:** In directions with a consistent gradient, momentum adds up, leading to larger updates (v_t is large).
- **Reduces Oscillations:** In ravines (steep walls, gentle slope), gradients oscillate across the sides. The momentum term cancels out these opposing gradients, leading to a smoother, faster path down the gentle slope.
- **Escapes Flat Regions:** Even if the current gradient is near zero, the accumulated momentum can carry the parameters through.

Nesterov Accelerated Gradient (NAG)

"Look-ahead" Momentum: A smarter version of momentum.

Nesterov Momentum

1. First, make a **"look-ahead"** jump using the old velocity: $\theta_{\text{look-ahead}} = \theta_t - \beta v_{t-1}$.
2. Then, calculate the gradient **at this future position**, $g_t = \nabla J(\theta_t - \beta v_{t-1})$.
3. Finally, update the velocity and parameters:

$$v_t = \beta v_{t-1} + \eta\, g_t$$
$$\theta_{t+1} = \theta_t - v_t$$

Why it's Better: By calculating the gradient *after* the momentum jump, NAG gets a "preview" of where it's going. This creates a **corrective term**. If the momentum jump was too big and is about to increase the loss, the gradient at the look-ahead point will correct the velocity, preventing overshooting and leading to more responsive behavior.

RMSProp: Adaptive Learning Rates

The Idea: Adjust the learning rate for **each parameter** individually based on the history of its gradients.

The Algorithm

Accumulate an exponentially decaying average of **squared gradients**.

$$E[g^2]_t = \beta E[g^2]_{t-1} + (1-\beta) g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Effect:

- Parameters with large, frequent gradients $\Rightarrow$ Large $E[g^2]_t$ $\Rightarrow$ **Small effective learning rate** $\frac{\eta}{\sqrt{E[g^2]_t}}$.
- Parameters with small, infrequent gradients $\Rightarrow$ Small $E[g^2]_t$ $\Rightarrow$ **Large effective learning rate**.

This automatically normalizes the step size for each parameter, which is very effective for problems with sparse gradients or ill-conditioned landscapes.

Adam: Adaptive Moment Estimation

The Best of Both Worlds: Adam combines the concepts of **Momentum** (first moment) and **RMSProp** (second moment).

The Algorithm

Compute gradient: $g_t = \nabla_\theta J(\theta_t)$

Update biased first moment (momentum): $m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$

Update biased second moment (RMS): $v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$

Compute bias-corrected first moment: $\hat{m}_t = \dfrac{m_t}{1-\beta_1^t}$

Compute bias-corrected second moment: $\hat{v}_t = \dfrac{v_t}{1-\beta_2^t}$

Update parameters: $\theta_{t+1} = \theta_t - \dfrac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$

Typical values: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$.

Adam: Understanding Bias Correction

- $\hat{m}_t$ provides a smooth, momentum-like direction.
- $\hat{v}_t$ adapts the learning rate for each parameter, like RMSProp.
- **Bias Correction** is crucial in early steps when m_t and v_t are initialized to 0.

What are β_1^t and β_2^t?

- They represent β_1 and β_2 **raised to the power of** t
- t is the current time step (iteration number)
- Used in the bias correction terms:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \quad \text{and} \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Why Bias Correction is Needed

- At $t = 0$, $m_0 = 0$ and $v_0 = 0$
- Early estimates are **biased toward zero**
- Example: $m_1 = 0.9 \cdot 0 + 0.1 \cdot g_1 = 0.1 g_1$ (too small!)
- Without correction, early updates would be too small

Adam is robust and works well out-of-the-box on a wide variety of problems.

L1 and L2 Regularization

Goal: Prevent overfitting by penalizing large weights, encouraging a simpler model.

L2 Regularization (Weight Decay, Ridge)

Adds the squared magnitude of weights to the loss function.

$$J_{\text{reg}}(\theta) = J(\theta) + \frac{\alpha}{2}\|\theta\|_2^2$$

Effect: Shrinks all weights proportionally towards zero. The gradient is $\nabla J(\theta) + \alpha\theta$.

L1 Regularization (Lasso)

Adds the absolute magnitude of weights to the loss function.

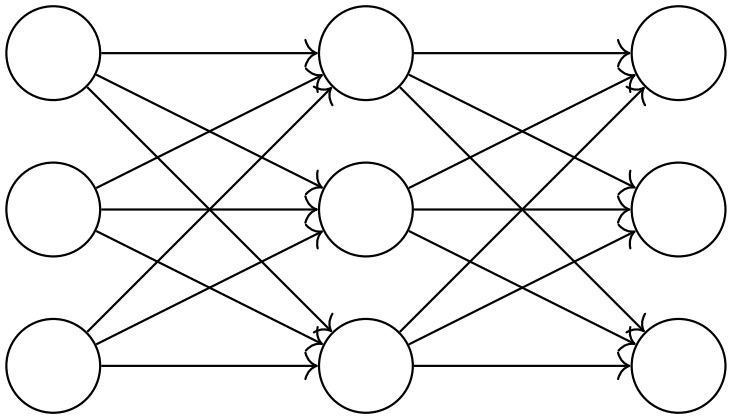
$$J_{\text{reg}}(\theta) = J(\theta) + \alpha\|\theta\|_1$$

Effect: Can drive some weights **exactly to zero**, creating a sparse model. Useful for feature selection.

Dropout

Idea: During training, randomly "drop out" (set to zero) a fraction p of the neurons in a layer for each forward pass. This prevents complex co-adaptations on training data.

Standard Network



Network with Dropout Applied

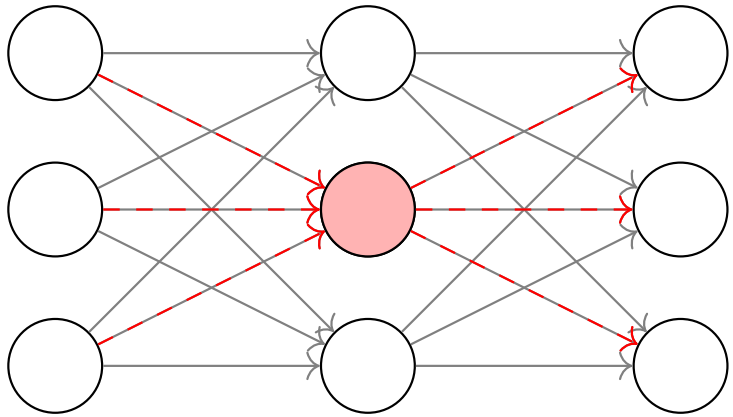


Figure: Dropout forces the network to learn redundant representations.

How Dropout Works

- During each training iteration (for each mini-batch), we randomly **"drop"** (set to zero) a fraction p of the neurons in the dropout-applied layers.
- This creates a different **random sub-network** for each training example, where only a subset of neurons are active and contribute to the forward and backward passes.
- At test time, we use the **full network**. To compensate for the fact that more neurons are active, the weights of the trained network are **scaled down by** $1 - p$ (or activations are scaled up by $1/(1 - p)$ during training).
- **Effect:** It's like training a large ensemble of many different sub-networks and averaging their predictions at test time.
- This prevents overfitting very effectively and is a key technique in modern deep learning.

Summary

- **Vanishing Gradients:** Solved by ReLU and proper initialization (He/Xavier).
- **Initialization:** Critical for stable training. Use He for ReLU, Glorot for Tanh.
- **Activation Functions:** ReLU is standard; Leaky ReLU/ELU/SELU solve the "dying ReLU" problem.
- **BatchNorm:** Normalizes layer inputs, allowing higher learning rates and acting as a regularizer.
- **Transfer Learning:** Use pre-trained models and fine-tune based on your data size and similarity.
- **Optimizers:** Adam is a good default, combining Momentum and RMSProp.
- **Regularization:** L1/L2 penalize weights; Dropout prevents feature co-adaptation.