Lecture 7: Training deep neural network

谢丹
清华大学数学系

November 12, 2025

Chapter 7: Technique of training deep network

Training Deep Neural Networks I

The Power and Complexity of Deep Networks

Deep neural networks have demonstrated remarkable success across diverse machine learning applications, enabling the modeling of highly complex probability relationships. However, their training presents significant challenges that we will address in three key areas:

1. **Advanced Optimization Methods**
   - Overcoming non-convex loss landscapes
   - Adaptive learning rate techniques
   - Momentum-based optimization
2. **Vanishing and Exploding Gradients**
   - Normalization techniques (BatchNorm, LayerNorm)
   - Residual connection
   - Careful initialization strategies
3. **Overfitting Mitigation**
   - Regularization methods (Dropout, Weight decay)

Training Deep Neural Networks II

- Early stopping and model selection

4. **Architecture**
   - Convolutional neural network (CNN): Image
   - Recurrent Neural Network (RNN): Sequential data
   - Transformer: Natural language
   - Graph Neural Network: data with graph structure

Section 1: Optimization methods

The Optimization Challenge in Deep Learning

Basic Stochastic Gradient Descent (SGD)

$$\theta_{t+1} = \theta_t - \eta \nabla J(\theta_t)$$

$J(\theta)$ is the loss function.

- **Problems with basic SGD**:
  - Fixed learning rate for all parameters
  - No memory of past gradients
  - Sensitive to feature scaling
  - Slow convergence on ravines
  - Oscillations in high-curvature directions

The Evolution of Gradient-Based Optimization

- 1960s: Momentum
- 1980s: Nesterov Acceleration
- 2010s: Adaptive methods (AdaGrad, RMSProp, Adam)
- 2010s+: Advanced variants (AdamW, Lookahead, etc.)

Momentum SGD

Key Insight

Accumulate velocity in directions of persistent gradient reduction

Algorithm:

$$v_t = \gamma v_{t-1} + \eta \nabla J(\theta_t)$$
$$\theta_{t+1} = \theta_t - v_t$$

Parameters:

- η: Learning rate
- γ: Momentum factor (0.9 typical)

Physical Analogy of Momentum

Ball Rolling Down Hill:

- Gradient: Slope of hill
- Momentum: Ball's velocity
- Learning rate: Time step
- Momentum factor: Friction

Benefits:

- Smoother updates
- Faster convergence
- Escapes shallow local minima

Nesterov Accelerated Gradient (NAG)

Key Improvement Over Momentum

"Look ahead" to the future position before computing gradient

Algorithm:

$$v_t = \gamma v_{t-1} + \eta \nabla J(\theta_t - \gamma v_{t-1})$$
$$\theta_{t+1} = \theta_t - v_t$$

Intuition:

- Compute gradient at approximate future position
- More responsive to landscape changes
- Better convergence guarantees

AdaGrad (Adaptive Gradient)

Key Idea

Adapt learning rate per parameter based on historical gradient magnitudes

$$G_t = G_{t-1} + (\nabla J(\theta_t))^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla J(\theta_t)$$

Advantages:

- Automatic learning rate tuning
- Great for sparse data
- No manual learning rate scheduling needed

Limitations:

- Learning rate decreases too much
- Not suitable for non-convex problems
- Memory grows with time

RMSProp (Root Mean Square Propagation)

Fixing AdaGrad's Aggressive Decay

Use exponentially weighted moving average of squared gradients

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

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

- γ: Decay rate (typically 0.9)
- Prevents aggressive learning rate decrease
- Suitable for non-stationary problems
- Works well for online learning

Visualizing Adaptive Learning Rates

- **SGD**: Constant learning rate
- **AdaGrad**: Rapid decrease, then very small
- **RMSProp**: Stable adaptation
- **Adam**: Combines momentum and adaptation

Adam (Adaptive Moment Estimation)

The Best of Both Worlds

Combines momentum (like RMSProp) and adaptive learning rates (like Momentum)

Key Components:

- First moment estimate (mean of gradients) - momentum
- Second moment estimate (variance of gradients) - adaptive LR
- Bias correction for initialization

Adam Algorithm Details

Complete Adam Algorithm

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \quad \text{(Update first moment)}$$

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

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

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

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \quad \text{(Parameter update)}$$

Default Parameters:

- $\beta_1 = 0.9$ (momentum decay)
- $\beta_2 = 0.999$ (squared gradient decay)
- $\epsilon = 10^{-8}$ (numerical stability)
- $\eta = 0.001$ (learning rate)

Adam Bias Correction

Why Bias Correction?

Initial moments are biased toward zero, especially early in training

Without Correction:

$$E[m_t] = E[g_t](1 - \beta_1^t)$$
$$\approx 0 \text{ when } t \text{ is small}$$

With Correction:

$$E[\hat{m}_t] = E[g_t]$$

AdamW: Adam with Weight Decay

Fixing Weight Decay in Adam

Original Adam implements L2 regularization incorrectly

Original Adam (incorrect):

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

AdamW (correct):

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

- Proper weight decay separation
- Better generalization performance
- Now standard in most deep learning frameworks

Practical Techniques: Gradient Clipping

Preventing Exploding Gradients

Especially important in RNNs and Transformers

Value Clipping:

$$g_t = \max(\min(g_t, \text{clip\_value}), -\text{clip\_value})$$

Norm Clipping:

$$g_t = \begin{cases} g_t & \text{if } \|g_t\| \leq \text{clip\_norm} \\ g_t \cdot \frac{\text{clip\_norm}}{\|g_t\|} & \text{otherwise} \end{cases}$$

Learning Rate Scheduling

Step Decay:

- Reduce by factor every N epochs
- Simple and effective

Exponential Decay:

- Continuous decrease
- $\eta_t = \eta_0 e^{-kt}$

Cosine Annealing:

- Smooth periodic restarts
- Better for finding good minima

Warmup Strategies

Gradual Learning Rate Increase

Prevents early instability in adaptive methods

Linear Warmup:

$$\eta_t = \eta_{\text{final}} \times \frac{t}{T_{\text{warmup}}}$$

Cosine Warmup:

$$\eta_t = \eta_{\text{final}} \times \frac{1}{2}(1 + \cos(\pi \times (1 - \frac{t}{T_{\text{warmup}}})))$$

Optimizer Performance Comparison

Optimizer	Convergence	Stability	Memory	Hyperparams	Generalization
SGD	Slow	High	Low	Sensitive	Good
SGD+Momentum	Fast	High	Low	Moderate	Good
Adam	Very Fast	Medium	Medium	Robust	Medium
AdamW	Very Fast	High	Medium	Robust	Good
RMSProp	Fast	Medium	Medium	Moderate	Medium

Table: Optimizer characteristics comparison

When to Use Which Optimizer?

Computer Vision:

- SGD with momentum
- Good generalization
- Large batches work well

Natural Language Processing:

- AdamW
- Transformers prefer Adam variants
- Good for sparse gradients

Reinforcement Learning:

- Adam/RMSProp
- Stable online learning
- Handles non-stationarity

Recommendations:

1. Start with AdamW
2. Try SGD+Momentum for CV
3. Use Lookahead for stability

Practical Guidelines

Hyperparameter Tuning Tips

- **Learning Rate**: Use learning rate finder (LR range test)
- **Batch Size**: Larger batches allow higher learning rates
- **Weight Decay**: AdamW: 0.01-0.1, SGD: 0.0001-0.001
- **Warmup**: Essential for large models and adaptive methods
- **Gradient Clipping**: Use norm clipping (1.0-5.0) for RNNs/Transformers

Common Pitfalls

- Using Adam without weight decay correction

Section 2: Deal with vanishing and explosive gradient

Addressing Gradient Challenges in Deep Networks

The Gradient Problem in Deep Learning

Deep neural networks often suffer from **vanishing/exploding gradients**, which impede training stability and convergence. We present three effective solutions:

1. **Initialization**
2. **Batch Normalization**
   - Normalizes activations across mini-batches
   - Reduces internal covariate shift
   - Enables higher learning rates
3. **Layer Normalization**
   - Normalizes across features within each sample
   - Independent of batch size
   - Ideal for recurrent networks and small batches
4. **Residual Connections**
   - Provides skip connections with identity mapping
   - Enables gradient flow through shortcut paths
   - Facilitates training of very deep networks

Why Weight Initialization Matters I

The Problem

Poor initialization causes:

- **Vanishing gradients**: Signals disappear in deep networks
- **Exploding gradients**: Signals grow uncontrollably
- **Slow convergence**: Training takes too long
- **Training failure**: Network doesn't learn at all

Good Initialization Provides

- Stable forward/backward signal flow
- Faster convergence
- Better final performance
- Consistent training across runs

Key Principle

Why Weight Initialization Matters II

Maintain consistent variance of activations and gradients across all layers

Popular Initialization Methods I

Xavier/Glorot Initialization

For: Tanh, Sigmoid activations

$$\text{Scale} = \sqrt{\frac{2}{n_{in} + n_{out}}}$$

- Maintains activation variances
- Good for smooth activations

He Initialization

Popular Initialization Methods II

For: ReLU, Leaky ReLU activations

$$\text{Scale} = \sqrt{\frac{2}{n_{in}}}$$

- Accounts for ReLU "dead zone"
- Default choice for modern networks

Xavier Initialization: Problem Setup

Forward Propagation

For layer l with linear activation:

$$z^l = W^l a^{l-1} + b^l$$

Assuming:

- W_i^l are i.i.d. with $\mathbb{E}[W^l] = 0$
- a_i^{l-1} are i.i.d. with $\mathbb{E}[a^{l-1}] = 0$
- W^l and a^{l-1} are independent

Then:

$$\text{Var}(z^l) = n_{in} \cdot \text{Var}(W^l) \cdot \text{Var}(a^{l-1})$$

Backward Propagation

For gradients:

$$\frac{\partial L}{\partial a^{l-1}} = (W^l)^T \frac{\partial L}{\partial z^l}$$

$$\text{Var}\left(\frac{\partial L}{\partial a^{l-1}}\right) = n_{out} \cdot \text{Var}(W^l) \cdot \text{Var}\left(\frac{\partial L}{\partial z^l}\right)$$

Xavier Initialization: Derivation

Variance Preservation Goal

We want to maintain stable signal flow:

$$\text{Var}(z^l) = \text{Var}(z^{l-1}) \quad \text{(forward)}$$

$$\text{Var}\left(\frac{\partial L}{\partial a^l}\right) = \text{Var}\left(\frac{\partial L}{\partial a^{l-1}}\right) \quad \text{(backward)}$$

Dual Constraints

From forward propagation:

$$n_{in} \cdot \text{Var}(W^l) = 1$$

From backward propagation:

$$n_{out} \cdot \text{Var}(W^l) = 1$$

We cannot satisfy both simultaneously!

Xavier's Compromise

Take the average of both constraints:

$$\text{Var}(W^l) = \frac{2}{n_{in} + n_{out}}$$

Xavier Initialization: Implementation

For Uniform Distribution

For $W \sim U[-a, a]$, we have $\text{Var}(W) = \frac{a^2}{3}$

Set $\frac{a^2}{3} = \frac{2}{n_{in}+n_{out}}$:

$$a = \frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}$$

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}\right]$$

For Normal Distribution

For $W \sim \mathcal{N}(0, \sigma^2)$, we have $\text{Var}(W) = \sigma^2$

Set $\sigma^2 = \frac{2}{n_{in}+n_{out}}$:

$$W \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{in} + n_{out}}}\right)$$

Best For

Tanh and Sigmoid activations (smooth, approximately linear near zero)

He Initialization: ReLU Challenge

ReLU's Effect on Variance

For ReLU activation: $a = \max(0, z)$
If z has symmetric distribution around 0:

$$\mathbb{E}[a^2] = \mathbb{E}[\max(0, z)^2] = \frac{1}{2}\mathbb{E}[z^2]$$

Therefore:

$$\text{Var}(a) = \frac{1}{2}\text{Var}(z)$$

Implication for Forward Pass

From earlier: $\text{Var}(z^l) = n_{in} \cdot \text{Var}(W^l) \cdot \text{Var}(a^{l-1})$
With ReLU: $\text{Var}(a^{l-1}) = \frac{1}{2}\text{Var}(z^{l-1})$
So:

$$\text{Var}(z^l) = n_{in} \cdot \text{Var}(W^l) \cdot \frac{1}{2}\text{Var}(z^{l-1})$$

R

eLU effectively halves the variance at each layer!

He Initialization: Derivation

Variance Preservation with ReLU

We want: $\text{Var}(z^l) = \text{Var}(z^{l-1})$

From previous slide:

$$\text{Var}(z^l) = \frac{1}{2} n_{in} \cdot \text{Var}(W^l) \cdot \text{Var}(z^{l-1})$$

Setting $\text{Var}(z^l) = \text{Var}(z^{l-1})$:

$$1 = \frac{1}{2} n_{in} \cdot \text{Var}(W^l)$$

$$\text{Var}(W^l) = \frac{2}{n_{in}}$$

Backward Pass Consideration

He initialization primarily addresses the forward pass variance reduction caused by ReLU. In practice, this works well for both forward and backward propagation.

He Initialization: Implementation

For Uniform Distribution

Set $\frac{a^2}{3} = \frac{2}{n_{in}}$:

$$a = \frac{\sqrt{6}}{\sqrt{n_{in}}}$$

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_{in}}}, \frac{\sqrt{6}}{\sqrt{n_{in}}}\right]$$

For Normal Distribution

Set $\sigma^2 = \frac{2}{n_{in}}$:

$$W \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{in}}}\right)$$

Variants

- **Forward pass only**: Use n_{in} in denominator
- **Backward pass only**: Use n_{out} in denominator
- **Default**: Use n_{in} (works well in practice)

Best For

ReLU and its variants (Leaky ReLU, PReLU, etc.)

Comparison and Summary

Key Formulas

Method	Uniform	Normal
Xavier	$U\left[-\frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}\right]$	$\mathcal{N}\left(0, \sqrt{\frac{2}{n_{in}+n_{out}}}\right)$
He	$U\left[-\frac{\sqrt{6}}{\sqrt{n_{in}}}, \frac{\sqrt{6}}{\sqrt{n_{in}}}\right]$	$\mathcal{N}\left(0, \sqrt{\frac{2}{n_{in}}}\right)$

Theoretical Basis

- **Xavier**: Balances forward/backward variance for linear-like activations
- **He**: Compensates for ReLU's variance reduction in forward pass
- Both assume zero-centered symmetric weight distributions

Practical Recommendation

- **Tanh/Sigmoid**: Use Xavier initialization
- **ReLU family**: Use He initialization
- **Modern default**: He initialization (most networks use ReLU)

Normalization

- **Internal Covariate Shift**: Change in input distribution during training
- **Training Challenges**:
  - Vanishing/exploding gradients
  - Slow convergence
  - Sensitivity to initialization
- **Benefits of Normalization**:
  - Faster training convergence
  - Higher learning rates
  - Better generalization
  - Reduced sensitivity to initialization

Batch Normalization (BN)

The input is $B \times X$, with B the batch size.

Key Idea

Normalize activations across the batch dimension for each feature

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$y_i = \gamma \hat{x}_i + \beta$$

- μ_B: Batch mean
- σ_B^2: Batch variance
- γ, β: Learnable parameters

Batch Normalization: Properties

Advantages:

- Faster convergence
- Higher learning rates
- Reduces overfitting
- Stable gradients

Limitations:

- Batch size dependent
- Problematic for RNNs
- Different train/test behavior

Layer Normalization (LN)

Key Idea

Normalize across feature dimensions for each sample independently

$$\hat{x}_i = \frac{x_i - \mu_L}{\sqrt{\sigma_L^2 + \epsilon}}$$

- Normalizes across all features of a single sample
- Independent of batch size
- Suitable for recurrent networks

x_i are the values on the neurons in a single layer.

The Deep Learning Paradox

- **Intuition**: Deeper networks should perform better
- **Reality**: Very deep networks often perform worse
- **Observation**: Training error increases with depth!

The Vanishing Gradient Problem

Chain Rule in Backpropagation:

$$\frac{\partial L}{\partial W^{(1)}} = \frac{\partial L}{\partial a^{(L)}} \prod_{k=2}^{L} \frac{\partial a^{(k)}}{\partial a^{(k-1)}} \frac{\partial a^{(1)}}{\partial W^{(1)}}$$

Problem:

- Many small derivatives multiplied together
- Gradients approach zero exponentially
- Weights in early layers don't update effectively

The Fundamental Insight

Key Question

Is it easier to learn an identity mapping or a zero mapping?

- Traditional approach: Learn $H(x)$ directly
- Residual approach: Learn $F(x) = H(x) - x$
- If identity mapping is optimal: $F(x) = 0$
- Easier to push residuals to zero than learn identity

$$H(x) = F(x) + x$$

Residual Block Diagram

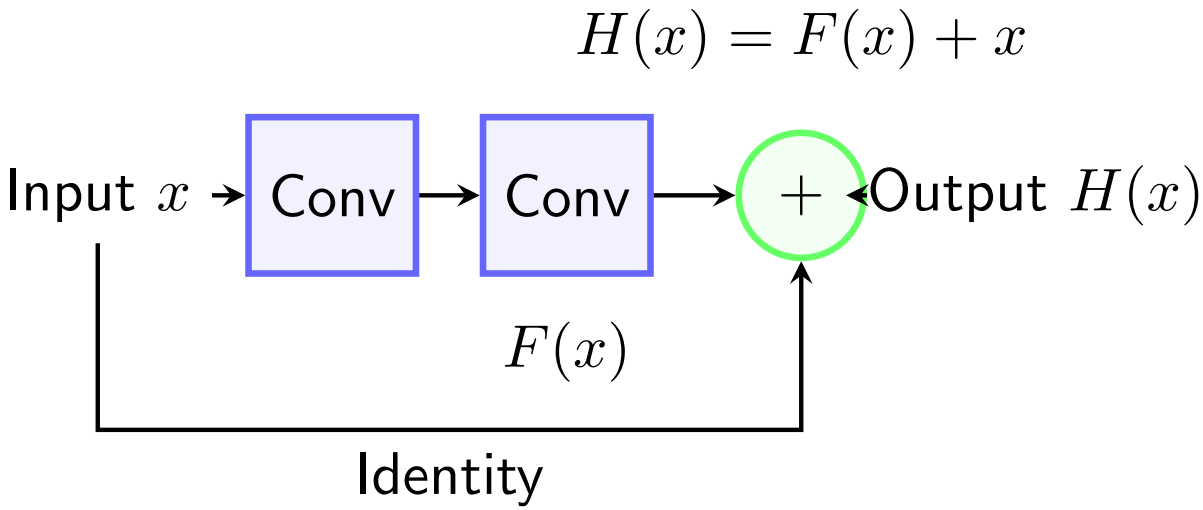


Figure: Basic residual block architecture

Section 3: Deal with Overfitting problem

The Problem: Overfitting

- Model learns training data *too* well, including noise.
- Performs poorly on new, unseen data (test set).
- Complex models (e.g., large neural nets) are highly susceptible.

Method 1: Regularization

Goal: Prevent overfitting to improve generalization.

Occam's Razor

"Among competing hypotheses, the one with the fewest assumptions should be selected."

In ML: **Prefer simpler models.**

- How do we define a "simple" model?
- One where the weights ($\mathbf{w}$) have smaller magnitudes.
- Large weights can make a model overly sensitive to its inputs.

The Mathematics: Modifying the Loss

Original Loss Function

$$L(\mathbf{w})$$

Measures how well the model fits the data (e.g., MSE, Cross-Entropy).

Regularized Loss Function

$$L_{\text{reg}}(\mathbf{w}) = L(\mathbf{w}) + \lambda R(\mathbf{w})$$

- $R(\mathbf{w})$: Regularization term
- λ: Regularization strength (hyperparameter)

We now minimize *both* the original loss *and* the size of the weights.

L2 Regularization: The Most Common Choice

The most common form is **L2 Regularization**:

$$R(\mathbf{w}) = \frac{1}{2}||\mathbf{w}||_2^2 = \frac{1}{2}\sum_{j=1}^{p} w_j^2$$

- Penalizes large weights *more severely* (because of the square).
- Leads to diffuse, smaller weights across the model.
- The $\frac{1}{2}$ term is used to simplify the derivative.

Final L2-Regularized Loss

$$L_{\text{reg}}(\mathbf{w}) = L(\mathbf{w}) + \frac{\lambda}{2}\sum_{j=1}^{p} w_j^2$$

Connection to Gradient Descent

Let's see how the weight update rule changes.

Standard Update

$$w_j \leftarrow w_j - \eta \frac{\partial L}{\partial w_j}$$

With L2 Regularization The new gradient is: $\frac{\partial L_{\text{reg}}}{\partial w_j} = \frac{\partial L}{\partial w_j} + \lambda w_j$

$$w_j \leftarrow w_j - \eta \left(\frac{\partial L}{\partial w_j} + \lambda w_j \right)$$

This can be rearranged to reveal the decay:

$$w_j \leftarrow (1 - \eta\lambda) w_j - \eta \frac{\partial L}{\partial w_j}$$

The weight is *decayed* by a factor of $(1 - \eta\lambda)$ before the main update!

Why Does This Help?

- **Promotes Simplicity:** Encourages the model to use all inputs weakly rather than a few inputs strongly.
- **Improves Generalization:** A smoother model is less sensitive to small fluctuations/noise in the input data.
- **Numerical Stability:** Can help stabilize the optimization process.

The Hyperparameter: λ

λ controls the strength of the penalty.

$\lambda = 0$	: No effect. Model may overfit.
λ too small	: Minimal effect. May still overfit.
λ just right	: Good generalization.
λ too large	: Underfitting! Weights are forced to near zero, model cannot learn complex patterns.

λ must be tuned carefully (e.g., via cross-validation).

Summary

- Weight decay is a regularization technique to combat overfitting.
- It works by adding a penalty term (usually L2) to the loss function.
- This penalty encourages smaller weights, leading to simpler models.
- During gradient descent, it acts as a multiplicative decay of the weights at each step.
- The key hyperparameter λ controls regularization strength and must be tuned.

Method 2: Dropout

- Inspiration: Not all neurons in the brain fire at once.
- Idea: Randomly "drop" neurons during training.
- Effect: Forces the network to learn redundant, robust pathways.
- Prevents any single neuron from becoming a critical bottleneck.

The Dropout Operation

Core Concept

For each training example, randomly set a fraction p of a layer's neurons to zero. The remaining ones are scaled up by $\mathbf{1/(1\text{-}p)}$.

Training Phase with Dropout

For each hidden layer l and each training example in a mini-batch:

1. **Sample a mask vector r:**

$$r_j^{(l)} \sim \text{Bernoulli}(1-p) \quad \text{for each neuron } j$$

2. **Apply the mask to the layer's output y:**

$$\tilde{y}^{(l)} = r^{(l)} \odot y^{(l)}$$

3. **Scale the activations (Inverted Dropout):**

$$\tilde{y}^{(l)} = \frac{r^{(l)} \odot y^{(l)}}{1-p}$$

4. **Proceed** with forward and backward pass using $\tilde{y}^{(l)}$.

We are training a different "thinned" sub-network every time!

Inference/Test Phase

Dropout is turned OFF.

- We want to use the full network's capacity for prediction.
- All neurons are active. No random sampling.
- **Crucial:** Because we used **Inverted Dropout** (scaling *during training*), we need to do **nothing special** at test time.
- The weights are already in their final, correctly scaled state.
- We simply run the forward pass as usual.

Why the scaling works

Scaling during training ensures the *expected* output at test time (with all neurons active) is the same as the expected output during training (with only some active).

The Magic: Why Dropout is Effective

1. Prevents Co-adaptation

Neurons can't rely on their neighbors. They must become independently useful.

2. Approximate Bagging

Training 2^N sub-networks and averaging their predictions. A form of model ensemble.

3. Robustness

The network learns to make accurate predictions even with missing data, making it robust to noise.

Using Dropout in Practice

- **Where to Place It:** Typically after fully connected layers (the most common use case) and sometimes after convolutional layers (though BatchNorm is often preferred here).
- **Dropout Rate (p):**
  - **Hidden Layers:** A good default is $\mathbf{p = 0.5}$.
  - **Input Layer:** A much lower rate, e.g., $\mathbf{p = 0.1}$ **or** $\mathbf{0.2}$.
- **Interaction with Other Techniques:**
  - Works very well in combination with other regularizers like L2 Weight Decay.
  - Often used alongside Batch Normalization, though the ordering (Dropout before or after BN) can be a design choice.
- **Effect on Training Time:** Training takes about 2-3 times longer because the forward/backward pass is still done over the entire network (even though parts are masked). However, it dramatically reduces overfitting.

Summary

- Dropout is a regularization technique to combat overfitting and co-adaptation.
- It works by randomly dropping neurons during training, forcing the network to learn robust features.
- It uses inverted dropout (scaling during training) for efficient inference.
- Its power comes from training an ensemble of many sub-networks simultaneously.
- It is a simple, highly effective, and widely used tool in deep learning.

What is Early Stopping? I

Core Idea

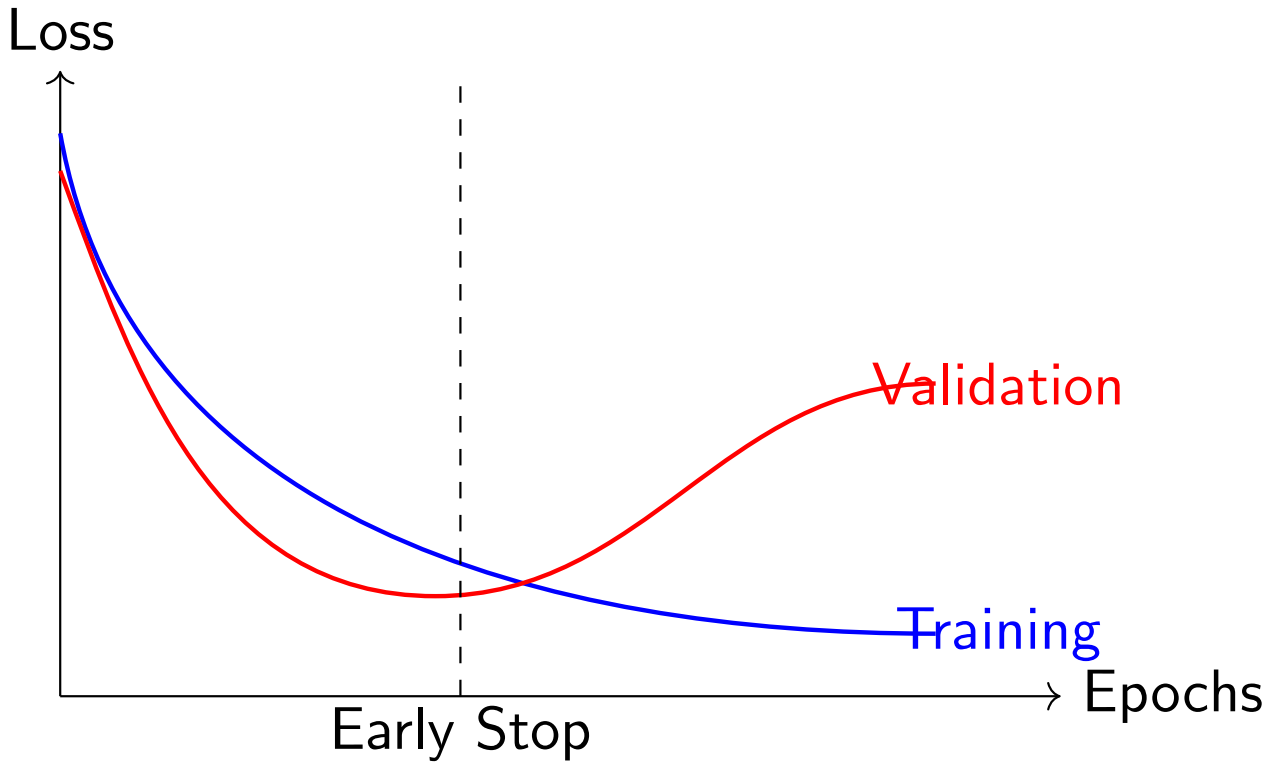
Monitor validation performance during training and stop when it begins to degrade, preventing the model from overfitting to the training data.

Key Components

- **Training set**: Used for weight updates
- **Validation set**: Used for monitoring performance
- **Patience**: How long to wait after last improvement
- **Best weights**: Model snapshot at optimal validation performance

Visual Concept

What is Early Stopping? II



How Early Stopping Prevents Overfitting

Theoretical Interpretation

- Limits effective model complexity
- Acts as implicit regularization
- Controls the optimization process
- Prevents over-specialization to training data

Comparison with Other Regularization Methods

Method	Computation	Hyperparameters	Effectiveness
Early Stopping	**Low**	Few	High
L1/L2 Regularization	Medium	Moderate	Medium
Dropout	Medium	Few	High
Data Augmentation	High	Many	High
Batch Normalization	Low	Few	Medium

Table: Comparison of regularization techniques

Unique Advantages of Early Stopping

- **No inference overhead**: Model unchanged at test time
- **Automatic**: Reduces need for manual epoch selection
- **Universal**: Works with any architecture
- **Computational savings**: Stops unnecessary training