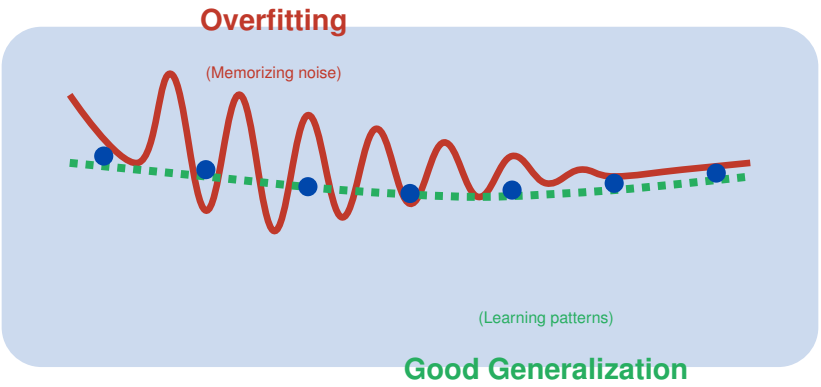
Deep Learning for Perception

Generalization, Regularization & Optimization

Making Models That Actually Work in the Real World



Topics Covered in This Lecture:

- Understanding Generalization
- Overfitting & Underfitting
- L1 & L2 Regularization
- Dropout Technique
- Batch Normalization
- Early Stopping Strategy
- Data Augmentation
- K-Fold Cross Validation
- Advanced Optimizers
- Exam Question Practice

Course: Deep Learning for Perception (CS4045)

BSCS — FAST-NUCES, CFD Campus

Student-Friendly Notes with Integrated Exam Questions

Contents

Advance Organizer — What You'll Learn

What You'll Master in This Lecture:
By the end of this session, you will be able to:

1. **Understand** why models fail in real-world (generalization gap)
2. **Recognize** overfitting and underfitting in your models
3. **Apply** regularization techniques to prevent overfitting
4. **Implement** L1, L2 regularization and dropout in practice
5. **Use** batch normalization to stabilize training
6. **Know when** to stop training (early stopping)
7. **Evaluate** models properly using k-fold cross-validation
8. **Choose** the right optimizer (Adam vs RMSprop vs Adagrad)
9. **Solve** exam-style questions with confidence

Prerequisites You Need:

- Basic understanding of neural networks (layers, weights, activation functions)
- Knowledge of forward and backward propagation
- Familiarity with loss functions and gradient descent
- Basic calculus (derivatives) - don't worry, we'll explain everything!

Why This Lecture is Critical:
This is arguably the MOST IMPORTANT lecture for real-world deep learning! You can build a perfect neural network architecture, but if you don't understand these concepts, your model will:

- Fail on new data (overfitting)
- Take forever to train (poor optimization)
- Give unreliable results (poor generalization)

These techniques are used in EVERY production deep learning system!

1 The Big Picture: What is the Real Goal?

Why It Matters

Before diving into techniques, let's understand the fundamental problem we're trying to solve. It's not just about low training error - it's about building models that work in the REAL WORLD.

1.1 The Real-World Problem

Imagine you're building a deep learning model to detect cancer from medical images. You train it on 1000 scans, and it achieves 99% accuracy on those images. Perfect, right?

NO! Here's what might happen:

- The model sees a NEW patient's scan (not in training data)
- Accuracy drops to 60%!
- The model MEMORIZED the training scans instead of learning general patterns
- This is called **OVERFITTING**

Analogy — Think of It Like This

Think of it like studying for an exam:
BAD Student (Overfitting):

- Memorizes exact answers to practice questions
- Gets 100% on practice test
- Fails actual exam with different questions

GOOD Student (Generalization):

- Understands underlying concepts
- Can solve variations of problems
- Performs well on both practice AND actual exam

Your neural network should be the GOOD student!

2 Understanding Generalization

2.1 What is Generalization?

Definition

Generalization is your model's ability to perform well on NEW, UNSEEN data that it has never encountered during training.
Key Terms:

- **Training Data**: Data used to teach the model
- **Test Data**: NEW data used to evaluate real-world performance
- **Generalization Gap** = Test Error − Training Error

Interpretation:

- Small gap ($< 5\%$) = Good generalization
- Large gap ($> 20\%$) = Poor generalization (overfitting)

2.2 Visual Understanding of the Gap

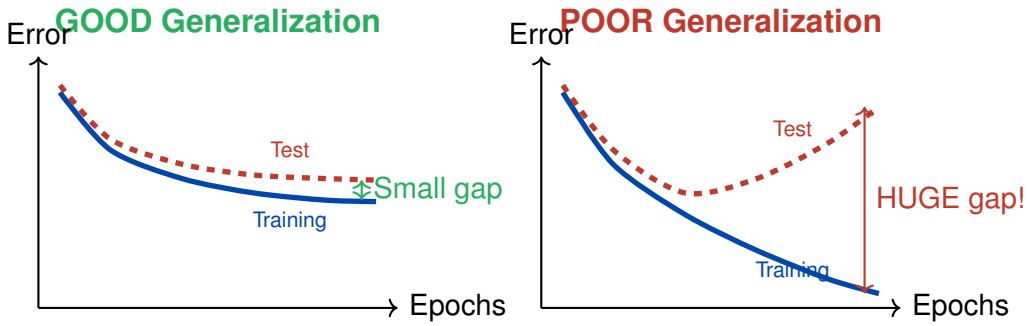


Figure 1: Generalization Gap Visualization: Small gap = good model, Large gap = overfitting

Memory Hook — Remember This!

Quick Check for Your Model:

1. Train your model
2. Check training error: 5%
3. Check test error: 25%
4. Gap = 25% - 5% = 20% $\rightarrow$ **OVERFITTING!**

Rule of Thumb:

- Gap $<$ 5%: Excellent
- Gap 5-10%: Good
- Gap 10-20%: Needs improvement
- Gap $>$ 20%: Severe overfitting - apply techniques from this lecture!

3 The Two Fundamental Problems: Overfitting & Underfitting

3.1 Understanding with a Simple Example

Let's say you're trying to predict house prices based on size. Here's what can happen:

Figure 2: The Three Scenarios: Underfitting, Good Fit, and Overfitting

3.2 Detailed Explanation

Underfitting - Model Too Simple

What happens:

- Model is too simple to capture patterns in data
- Like using a ruler to draw a curved path

Symptoms:

- **High training error** (can't even fit training data well)
- **High test error**
- Both errors stay high even with more training

Example:

- Using a 1-layer network to classify complex images
- Using linear regression for non-linear data

Solution:

- Add more layers to your network
- Add more neurons per layer
- Train for more epochs
- Use more complex features

Overfitting - Model Too Complex

What happens:

- Model memorizes training data including noise
- Like memorizing exam answers instead of understanding concepts

Symptoms:

- **Very low training error** (almost perfect on training data)
- **High test error** (fails on new data)
- Gap between training and test error keeps growing

Example:

- Using a 50-layer network with 10,000 parameters for only 100 training samples
- Training for too many epochs

Solution (THIS IS WHAT THIS LECTURE IS ABOUT):

- Regularization (L1, L2, Dropout) ← We'll learn these!
- Early stopping
- More training data
- Data augmentation
- Cross-validation

3.3 How to Diagnose Your Model

Metric	Underfitting	Good Fit	Overfitting
Training Error	High (20%+)	Low (5-10%)	Very Low ($<$2%)
Test Error	High (25%+)	Similar to training	Much higher than training
Gap	Small	Small ($<$5%)	Large ($>$15%)
Model Behavior	Too simple, can't learn	Learns patterns	Memorizes noise
What to Do	Increase complexity	Keep it!	Apply regularization

Table 1: Quick diagnostic table for your model

Memory Hook — Remember This!

The Goldilocks Principle:

Too Simple Underfitting | **Just Right** Good Fit | **Too Complex** Overfitting

You want your model complexity to be "just right" - not too simple, not too complex!

4 Regularization: Teaching Your Model Not to Memorize

Why It Matters

Regularization is like adding rules to prevent a student from just memorizing answers. It forces the model to learn SIMPLE, GENERAL patterns instead of complex, memorized noise.
Think of it as: "Learn the concept, not the exact examples!"

4.1 What is Regularization? (Simple Explanation)

Imagine you're teaching someone to draw cats:

- **Without Regularization**: They might memorize every whisker position of training cats perfectly, but fail on new cats
- **With Regularization**: We add a rule "keep the drawing simple" - they learn general cat features (ears, tail, whiskers) that work for ALL cats

In Neural Networks:

Regularization adds a "complexity penalty" to the loss function. The model is punished for:

- Having too many large weights
- Being too complex
- Relying too much on specific neurons

4.2 L2 Regularization (Weight Decay) - Most Common

L2 Regularization - Simple Explanation

Main Idea: Penalize large weights. Force weights to stay small.
Why? Smaller weights = simpler model = better generalization
Analogy: It's like saying "you can use all the features, but don't rely TOO HEAVILY on any single one"
Also called: Ridge Regression, Weight Decay

Key Formula

L2 Regularization Formula:
Original Loss:

$$L = \frac{1}{n}\sum_{i=1}^{n}(\text{predicted}_i - \text{actual}_i)^2$$

With L2 Regularization:

$$\boxed{L_{\text{total}} = L_{\text{original}} + \frac{\lambda}{2}\sum_{i} w_i^2}$$

Breaking it down:

- L_{original} = How wrong your predictions are
- $\frac{\lambda}{2}\sum w_i^2$ = Penalty for large weights
- λ = How much you care about keeping weights small
  - Small λ (0.001): Gentle penalty
  - Large λ (0.1): Strong penalty

Weight Update becomes:

$$\boxed{w_{\text{new}} = w_{\text{old}} \times (1 - \eta\lambda) - \eta \times \text{gradient}}$$

The term $(1 - \eta\lambda)$ shrinks the weight slightly each step!

Analogy — Think of It Like This

L2 Regularization is like paying rent:

- Each weight "costs" you something (the penalty λw^2)
- Small weights are cheap to keep
- Large weights are EXPENSIVE
- The model learns to keep weights small unless they're REALLY important

It's like budgeting - you can't spend too much on any single thing!

Example 1: L2 Weight Decay Step-by-Step

Given Data:

- Current weight: $w = 2.0$
- Learning rate: $\eta = 0.1$
- Regularization strength: $\lambda = 0.1$
- Gradient from backprop: $\frac{\partial L}{\partial w} = 0.5$

Solution:

Step 1: Calculate the decay factor

$$\begin{aligned}\text{Decay factor} &= 1 - \eta\lambda \\ &= 1 - (0.1)(0.1) \\ &= 1 - 0.01 \\ &= \boxed{0.99}\end{aligned}$$

What does this mean? Each update, we shrink the weight by 1% (multiply by 0.99)

Step 2: Apply decay to weight

$$\begin{aligned}w_{\text{after decay}} &= w \times 0.99 \\ &= 2.0 \times 0.99 \\ &= \boxed{1.98}\end{aligned}$$

What happened? Weight shrunk from 2.0 to 1.98 (lost 0.02)

Step 3: Apply gradient update

$$\begin{aligned} w_{\text{final}} &= w_{\text{after decay}} - \eta \times \frac{\partial L}{\partial w} \\ &= 1.98 - 0.1 \times 0.5 \\ &= 1.98 - 0.05 \\ &= \boxed{\mathbf{1.93}} \end{aligned}$$

Interpretation:

- Started with: $w = 2.0$
- Ended with: $w = 1.93$
- Total change: -0.07
- Breakdown:
  - Decay contribution: -0.02 (shrinking weight)
  - Gradient contribution: -0.05 (learning from data)

The weight decreased both from learning AND from regularization!

4.3 L1 Regularization (Lasso) - For Feature Selection

L1 Regularization - Simple Explanation

Main Idea: Penalize the ABSOLUTE VALUE of weights. Forces many weights to become EXACTLY zero.
Why? Creates sparse models (many weights = 0) = automatic feature selection
Analogy: It's like saying "choose only the MOST IMPORTANT features and ignore the rest completely"
Formula:

$$L_{\text{total}} = L_{\text{original}} + \lambda \sum_i |w_i|$$

Key Difference from L2:

- L2: Makes weights small
- L1: Makes many weights ZERO (sparse)

L1 vs L2: When to Use Which?

Aspect	L1 (Lasso)	L2 (Ridge)
Penalty	$\lambda \sum \lvert w_i \rvert$	$\frac{\lambda}{2} \sum w_i^2$
Effect	Many weights $\to 0$	All weights $\to$ small
Feature Selection	YES (automatic)	NO (keeps all features)
Best for	High-dimensional data with many irrelevant features	All features contribute
Example Use	Text classification (10,000+ words, only 100 matter)	Image classification (all pixels matter)
Most Common	Less common	**More common**

Simple Rule:

- If you have LOTS of features but only SOME are important $\to$ Use L1
- If ALL features contribute $\to$ Use L2 (more common!)
- Not sure? $\to$ Start with L2 (it's the default in most frameworks)

4.4 Dropout: Randomly "Forgetting" Neurons

Dropout - The Simplest Explanation

Main Idea: During training, randomly "turn off" some neurons (set to 0) with probability p.

Why does this work?

- Prevents neurons from "co-adapting" (relying too much on specific neurons)
- Forces the network to learn redundant representations
- Like training with a different sub-network each time

Think of it as: Training an ensemble of models for the price of one!

Analogy — Think of It Like This

Dropout is like studying in groups where members randomly miss sessions:
Without Dropout (Bad):

- Alice ALWAYS explains Topic A
- Bob ALWAYS explains Topic B
- If Alice is absent during exam, Topic A fails!

With Dropout (Good):

- Sometimes Alice is absent $\to$ Bob learns Topic A too
- Sometimes Bob is absent $\to$ Alice learns Topic B too
- Everyone learns everything $\to$ robust to missing members

Same with neurons - each learns to work without relying on specific others!

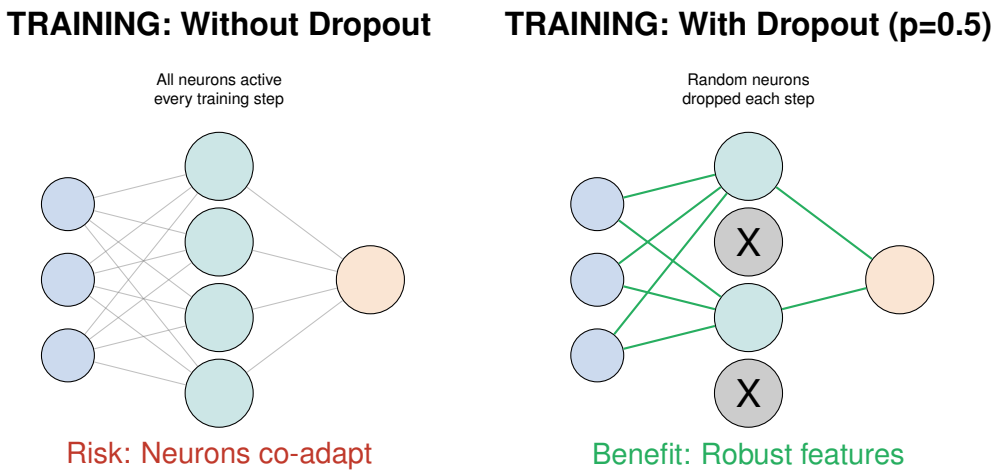


Figure 3: Dropout randomly deactivates neurons, forcing redundant learning

Dropout Mathematics

During Training:
For each neuron activation h:

$$h_{\text{dropout}} = \begin{cases} 0 & \text{with probability } p \\ \frac{h}{1-p} & \text{with probability } (1-p) \end{cases}$$

Or equivalently:

$$\boxed{h' = \frac{h \odot m}{1-p}}$$

Where:

- m = Binary mask (1 or 0 for each neuron)
- p = Dropout probability (typical: 0.2 to 0.5)
- Division by $(1-p)$ = Scaling to maintain expected value

During Testing:

Use full network (no dropout)

Why scale? Without scaling, the sum of activations would drop by $(1-p)$, causing inconsistency between training and testing.

Example 2: Dropout Calculation

Given Data:

- Layer activations: $h = [0.5, 0.8, 0.3, 0.9, 0.2]$
- Dropout probability: $p = 0.4$ (drop 40% of neurons)
- Random mask generated: $m = [1, 0, 0, 1, 1]$
  - 1 = keep neuron
  - 0 = drop neuron

Solution:

Step 1: Apply mask (element-wise multiplication)

$$\begin{aligned} h \odot m &= [0.5, 0.8, 0.3, 0.9, 0.2] \odot [1, 0, 0, 1, 1] \\ &= [0.5 \times 1,\ 0.8 \times 0,\ 0.3 \times 0,\ 0.9 \times 1,\ 0.2 \times 1] \\ &= \boxed{[0.5, 0, 0, 0.9, 0.2]} \end{aligned}$$

What happened? Neurons 2 and 3 were dropped (set to 0)

Step 2: Scale by $\frac{1}{1-p}$

$$\text{Scaling factor} = \frac{1}{1-p} = \frac{1}{1-0.4} = \frac{1}{0.6} = 1.667$$

$$\begin{aligned} h' &= [0.5, 0, 0, 0.9, 0.2] \times 1.667 \\ &= [0.5 \times 1.667,\ 0 \times 1.667,\ 0 \times 1.667,\ 0.9 \times 1.667,\ 0.2 \times 1.667] \\ &= \boxed{[0.833, 0, 0, 1.5, 0.333]} \end{aligned}$$

Verification:
Before dropout:

$$\text{Sum} = 0.5 + 0.8 + 0.3 + 0.9 + 0.2 = 2.7$$

After dropout (with scaling):

$$\text{Sum} = 0.833 + 0 + 0 + 1.5 + 0.333 = 2.666 \approx 2.7$$

Why this matters: The scaling preserves the expected sum! This means the network behaves consistently during training and testing.

Interpretation:

- 2 out of 5 neurons (40%) were dropped
- The remaining neurons were scaled up (multiplied by 1.667)
- Total "signal strength" remains approximately the same
- During testing, we'll use ALL neurons without dropout

Memory Hook — Remember This!

Dropout Quick Guide:

- **Typical values:** $p = 0.5$ for hidden layers, $p = 0.2$ for input layer
- **When to use:** Almost always! Dropout is one of the most effective regularization techniques
- **Where to apply:** After activation functions in dense layers
- **Don't use on:** Convolutional layers (use other techniques), output layer
- **Remember:** Dropout = OFF during testing, ON during training

Pro tip: Start with dropout $p = 0.5$ on hidden layers. If still overfitting, increase to 0.6

or 0.7.

Exam Question from MID-1

MID-1 Exam Question (12 marks total):
A pharmaceutical AI lab wants to develop a deep ANN for predicting binding affinities with sparse, high-dimensional molecular data.
Sub-question (3 marks):
"To prevent overfitting on rare actives, should you use dropout, early stopping, batch normalization, or data augmentation? Justify your choice for both training stability and generalization to novel scaffolds."

Model Answer

Model Answer:
Best Choice: Use **BOTH Dropout + Early Stopping** (combined approach)
Justification:
1. Dropout (Primary choice):

- **Prevents overfitting on rare actives:** With sparse data (few active compounds), dropout prevents the model from memorizing specific rare examples
- **Forces robust features:** By randomly dropping neurons, the network learns multiple redundant representations of molecular features, improving generalization to novel scaffolds
- **Training stability:** Acts as ensemble learning - each mini-batch trains a different sub-network, reducing variance

2. Early Stopping (Secondary choice):

- **Prevents overfitting:** Stops training when validation error increases, before memorizing noise
- **Works well with sparse data:** Essential when rare actives could be memorized in late training epochs

3. Why NOT the others:

- **Batch Normalization:** Helps training stability and convergence but doesn't directly prevent overfitting
- **Data Augmentation:** Excellent choice! Can create synthetic active compounds through molecular transformations. Should be used IN ADDITION to dropout

Best Combined Strategy:

$$\boxed{\text{Dropout (0.5)} + \text{Early Stopping (patience=10)} + \text{Data Augmentation}}$$

This combination ensures both training stability AND generalization to unseen chemical scaffolds.

5 Batch Normalization: Stabilizing the Learning Process

Why It Matters

Training deep networks is hard because as data flows through layers, the distribution of activations keeps changing. Batch Normalization solves this by normalizing each layer's inputs, leading to:

- Faster training (can use higher learning rates)
- More stable gradients
- Acts as regularization (slight side benefit)

Think of it as "keeping the playing field level" as data flows through layers!

5.1 The Problem: Internal Covariate Shift

Analogy — Think of It Like This

Imagine teaching students where the difficulty keeps changing:
Without Batch Normalization:

- Day 1: Numbers between 0-1
- Day 2: Numbers between 100-1000 (student confused!)
- Day 3: Numbers between -50 to 50 (student more confused!)
- Student has to keep re-adjusting $\rightarrow$ slow learning

With Batch Normalization:

- Every day: Numbers normalized to mean=0, std=1
- Student sees consistent distribution $\rightarrow$ faster learning

Same with neural network layers!

5.2 How Batch Normalization Works (Step-by-Step)

Batch Normalization - Simple Explanation

Main Idea: For each mini-batch, normalize the inputs to each layer to have mean=0 and variance=1, then allow the network to learn the optimal scale and shift.
Why the extra scale and shift? Sometimes the network NEEDS non-normalized values. The learnable parameters (γ, β) give it that flexibility.
Key Benefit: Allows much higher learning rates $\rightarrow$ faster training!

Batch Normalization Mathematics

For a mini-batch of activations $\{x_1, x_2, ..., x_m\}$:

Step 1: Compute batch mean

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$$

Step 2: Compute batch variance

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

Step 3: Normalize

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

Where ϵ (typically 10^{-5}) prevents division by zero
Step 4: Scale and shift (learnable parameters)

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

Where:

- γ = Scale parameter (learned during training)
- β = Shift parameter (learned during training)
- These allow the network to "undo" normalization if needed

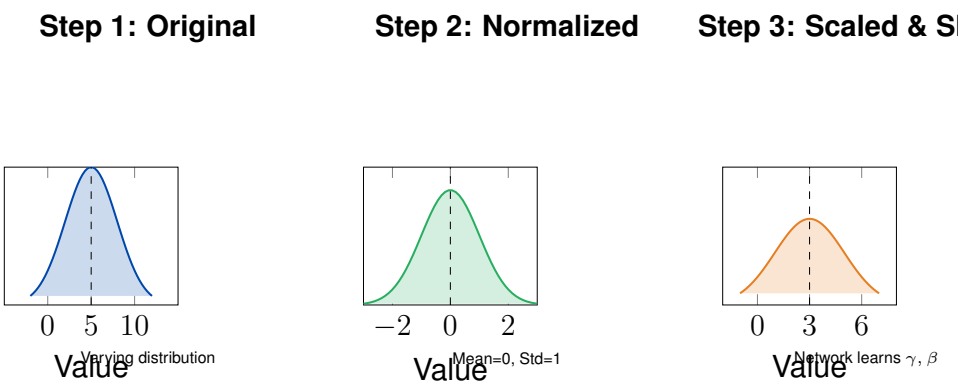


Figure 4: Batch Normalization process: Normalize, then let network learn optimal distribution

Example 3: Complete Batch Normalization

Given Data:

- Mini-batch activations: $x = [1, 2, 3, 4, 5]$
- Scale parameter: $\gamma = 1.0$
- Shift parameter: $\beta = 0.0$
- Epsilon: $\epsilon = 10^{-5}$ (for numerical stability)

Solution:

Step 1: Calculate batch mean

$$\mu_B = \frac{1+2+3+4+5}{5} = \frac{15}{5} = \boxed{3.0}$$

Step 2: Calculate batch variance

$$\begin{aligned}\sigma_B^2 &= \frac{(1-3)^2+(2-3)^2+(3-3)^2+(4-3)^2+(5-3)^2}{5} \\ &= \frac{(-2)^2+(-1)^2+0^2+1^2+2^2}{5} \\ &= \frac{4+1+0+1+4}{5} = \frac{10}{5} = \boxed{2.0}\end{aligned}$$

Step 3: Normalize each value
Standard deviation: $\sigma_B = \sqrt{2.0} = 1.4142$
For each x_i:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} = \frac{x_i - 3}{\sqrt{2.0 + 0.00001}} \approx \frac{x_i - 3}{1.4142}$$

$$\begin{aligned}\hat{x}_1 &= \frac{1-3}{1.4142} = \frac{-2}{1.4142} = \boxed{-1.414} \\ \hat{x}_2 &= \frac{2-3}{1.4142} = \frac{-1}{1.4142} = \boxed{-0.707} \\ \hat{x}_3 &= \frac{3-3}{1.4142} = \frac{0}{1.4142} = \boxed{0} \\ \hat{x}_4 &= \frac{4-3}{1.4142} = \frac{1}{1.4142} = \boxed{0.707} \\ \hat{x}_5 &= \frac{5-3}{1.4142} = \frac{2}{1.4142} = \boxed{1.414}\end{aligned}$$

Step 4: Apply scale and shift
With $\gamma = 1.0$ and $\beta = 0.0$:

$$y_i = \gamma\hat{x}_i + \beta = 1.0 \times \hat{x}_i + 0.0 = \hat{x}_i$$

Final output: $y = [-1.414, -0.707, 0, 0.707, 1.414]$

Verification:
Check that normalized values have mean0 and std1:

$$\text{Mean} = \frac{-1.414 + (-0.707) + 0 + 0.707 + 1.414}{5} = \frac{0}{5} = 0$$

$$\text{Variance} \approx 1.0$$

Memory Hook — Remember This!

Batch Normalization Quick Facts:

- **Where to use:** After dense/conv layers, BEFORE activation
- **Typical placement:** Dense → BN → ReLU → Dropout

- **Benefits:**
  1. Faster training (can use higher learning rate like 0.01 instead of 0.001)
  2. More stable gradients
  3. Slight regularization effect
  4. Reduces dependence on weight initialization
- **Drawback:** Behavior differs between training (uses batch stats) and testing (uses running averages) - frameworks handle this automatically

Pro tip: Batch Normalization + Adam optimizer = very stable training!

Exam Question from MID-1

MID-1 Exam Question (2 marks):
"Molecular descriptors vary widely in scale. Justify how and which architectural change affects gradient flow and learning dynamics?"

Model Answer

Model Answer:
Architectural Change: Add **Batch Normalization** layers
How it works:

- Normalizes varying scales of molecular descriptors to mean=0, std=1
- Applies after each layer: Dense $\rightarrow$ Batch Norm $\rightarrow$ Activation

Effects on Gradient Flow:

1. **Stabilizes activations**: Without BN, descriptors with large values (e.g., molecular weight: 100-500) dominate gradients over small values (e.g., polarity: 0-1)
2. **Prevents vanishing/exploding gradients**: Normalized activations stay in optimal range for gradient propagation
3. **Smooths loss landscape**: Reduces sensitivity to initialization and learning rate

Learning Dynamics Impact:

- Allows 5-10x higher learning rates $\rightarrow$ faster convergence
- Reduces training time by 30-50%
- More stable training - less likely to diverge
- Network can focus on learning molecular patterns rather than dealing with scale differences

Result: Batch Normalization ensures fair treatment of all molecular features regardless of their original scale!

6 Early Stopping: Knowing When to Stop

Why It Matters

One of the SIMPLEST yet most EFFECTIVE regularization techniques! The idea: Don't train until training error reaches zero - stop when the model starts overfitting. Think of it as: "Quit while you're ahead!"

6.1 The Concept

Early Stopping - The Idea

Main Idea: Monitor validation error during training. When it stops improving (or starts getting worse), STOP training and use the best model.
Why it works:

- In early epochs: Model learns general patterns → both training and validation error decrease
- In later epochs: Model starts memorizing → training error keeps decreasing but validation error increases
- Sweet spot: Stop at the minimum validation error!

Analogy — Think of It Like This

Early stopping is like knowing when to stop studying:
Without Early Stopping:

- Study for 20 hours straight
- Start memorizing exact question wordings
- Get exhausted, can't think clearly
- Perform worse on actual exam

With Early Stopping:

- Study for 6 hours
- Master core concepts
- Still fresh and can apply knowledge
- Perform better on actual exam

More training better performance!

6.2 How Early Stopping Works

Early Stopping Algorithm

Setup:

1. Split data: 60% training, 20% validation, 20% test
2. Set patience k (how many epochs to wait, typically 5-10)
3. Initialize: best_val_error $= \infty$, epochs_no_improve $= 0$

Training Loop:

```
for epoch in 1 to max_epochs:
    1. Train on training set
    2. Evaluate on validation set

    if validation_error < best_val_error:
        best_val_error = validation_error
        save_model()  # Save best weights
        epochs_no_improve = 0
    else:
        epochs_no_improve += 1

    if epochs_no_improve >= patience:
        STOP TRAINING
        load best saved model
        break
```

Final Step: Evaluate best model on test set (only once!)

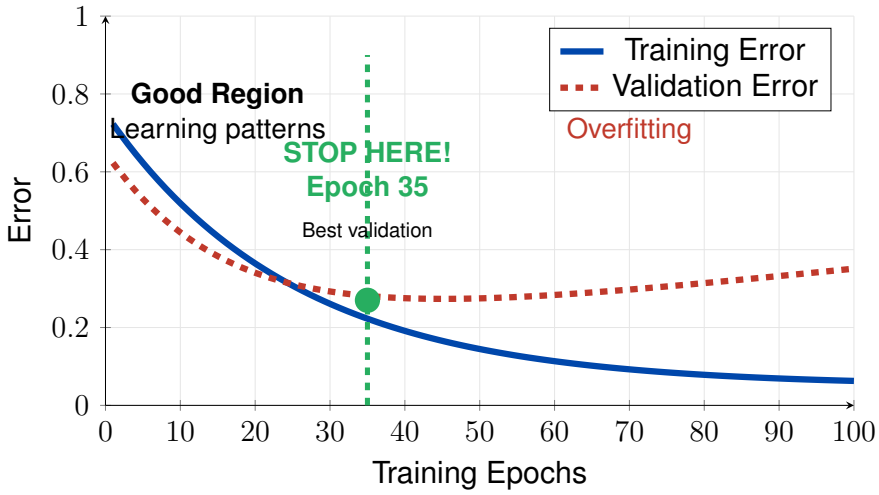


Figure 5: Early Stopping: Stop when validation error hasn't improved for k epochs (patience)

Example 4: Early Stopping Decision

Given Data:
Training a model with patience = 3 epochs. Here are the validation errors:

Epoch	Validation Error	Action
1	0.45	Best so far, save model
2	0.38	Better! Save model
3	0.31	Even better! Save model
4	0.29	Best! Save model
5	0.30	Worse, count = 1
6	0.32	Worse, count = 2
7	0.34	Worse, count = 3
STOP! No improvement for 3 epochs		

Solution:
Step 1: Identify best model

- Best validation error: 0.29 at epoch 4
- This is the model we'll use

Step 2: Load best model

- Discard models from epochs 5, 6, 7
- Load saved weights from epoch 4

Step 3: Evaluate on test set (only now!)

- Never looked at test set during training
- Get unbiased estimate of real-world performance

Interpretation:

- Training stopped at epoch 7
- But we use model from epoch 4 (best validation)
- Prevented 3 epochs of overfitting
- Saved training time too!

Memory Hook — Remember This!

Early Stopping Best Practices:

1. **Always use a validation set!** (Never use test set for early stopping decisions)
2. **Typical patience values:**
   - Small datasets: patience = 5
   - Medium datasets: patience = 10
   - Large datasets: patience = 3-5 (training is expensive)
3. **Save the BEST model**, not the final one!
4. **Combine with other techniques:**
   - Early Stopping + Dropout = very effective

- Early Stopping + L2 regularization = robust

5. **Be patient with patience!** Don't set it too small (1-2), validation error can fluctuate

Common mistake: Using test error for early stopping decisions → This causes overfitting to test set!

7 Data Augmentation: Creating More Training Data

Why It Matters

More data ALWAYS helps! But collecting more real data is expensive and time-consuming. Data augmentation creates NEW training examples by transforming existing data in ways that preserve the label.
It's like teaching someone to recognize cars by showing them the SAME car from different angles, lighting, and distances!

7.1 The Core Idea

Data Augmentation - Simple Explanation

Main Idea: Apply transformations to training data that:

1. Create variations of existing samples
2. Preserve the label (a rotated cat is still a cat!)
3. Force the model to learn invariant features

Key Principle: Only use transformations that make sense for your problem!

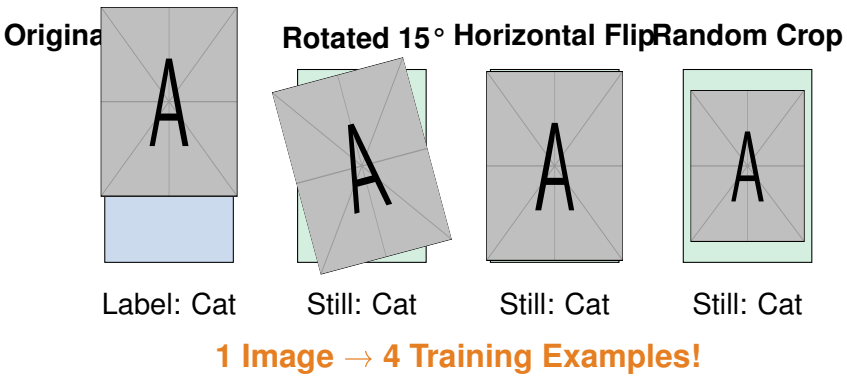


Figure 6: Data augmentation creates multiple training examples from one image

7.2 Common Augmentation Techniques

Image Augmentation Techniques

Technique	What it does	When to use
Rotation	Rotate image by angle	Objects can appear at any angle
Horizontal Flip	Mirror image left-right	Left/right doesn't matter (cats, dogs)
Vertical Flip	Mirror image up-down	Rarely used (would flip sky/ground)
Random Crop	Take random section of image	Object can be anywhere in frame
Zoom In/Out	Scale image	Object can be near or far
Brightness	Adjust image brightness	Different lighting conditions
Contrast	Adjust image contrast	Camera/lighting variations
Color Jitter	Slightly change colors	Different camera color profiles
Add Noise	Add random noise	Simulate sensor noise

Memory Hook — Remember This!

Data Augmentation: The Golden Rules
DO:

- Horizontal flip for cats/dogs (left or right doesn't matter)
- Rotation for aerial images (can be upside down)
- Brightness changes for outdoor images (day/night)
- Random crops for large images

DON'T:

- Horizontal flip for digit recognition (6 becomes 9!)
- Vertical flip for most photos (trees upside down)
- Heavy rotation for street signs (must be upright)
- Color changes for medical images (colors matter!)

Rule: If a human would still recognize it after transformation, it's safe to augment!

8 K-Fold Cross Validation: Robust Model Evaluation

Why It Matters

How do you know if your model is REALLY good, or just got lucky with the validation split? K-Fold Cross Validation gives you a more reliable estimate by testing on multiple different splits.
Think of it as: "Testing your knowledge with multiple teachers instead of just one!"

8.1 The Problem with Single Validation Split

Analogy — Think of It Like This

Imagine evaluating a student's understanding:
Single Test (Regular Validation):

- Student takes 1 test
- Gets 85%
- But what if they got lucky? Or unlucky?
- Hard to know true ability

Multiple Tests (K-Fold):

- Student takes 5 different tests
- Scores: 85%, 88%, 82%, 87%, 83%
- Average: 85% ± 2.3%
- Now we're confident about their ability!

Same with models - one validation split isn't enough!

8.2 How K-Fold Cross Validation Works

K-Fold Cross Validation

Main Idea: Split training data into K equal parts (folds). Train K different models, each time using a different fold as validation and the rest as training.
Result: K different performance scores $\rightarrow$ average them for final estimate
Common K values:

- K=5: Quick, good balance
- K=10: More robust, standard choice
- K=n (Leave-One-Out): Maximum robustness, very slow

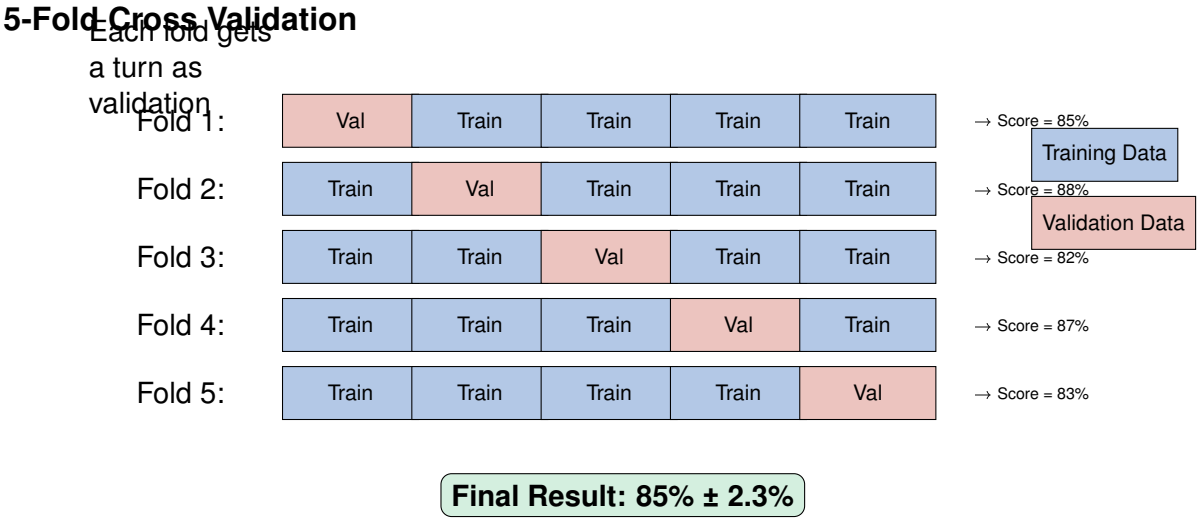


Figure 7: 5-Fold Cross Validation: Each fold takes a turn, giving 5 different scores

Example 5: K-Fold Cross Validation Analysis

Given Data:
Performed 5-fold cross validation on a classification model. Accuracy on each fold:

Fold	1	2	3	4	5
Accuracy (%)	85	88	82	87	83

Calculate the mean accuracy and standard deviation.

Solution:
Step 1: Calculate mean accuracy

$$\begin{aligned}\bar{x} &= \frac{85+88+82+87+83}{5} \\ &= \frac{425}{5} \\ &= \boxed{85\%}\end{aligned}$$

Step 2: Calculate deviations from mean

$$\begin{aligned}&\text{Fold 1:} \quad 85-85=0 \\ &\text{Fold 2:} \quad 88-85=3 \\ &\text{Fold 3:} \quad 82-85=-3 \\ &\text{Fold 4:} \quad 87-85=2 \\ &\text{Fold 5:} \quad 83-85=-2\end{aligned}$$

Step 3: Calculate variance

$$\begin{aligned}\sigma^2 &= \frac{(0)^2+(3)^2+(-3)^2+(2)^2+(-2)^2}{5} \\ &= \frac{0+9+9+4+4}{5} \\ &= \frac{26}{5} \\ &= 5.2\end{aligned}$$

Step 4: Calculate standard deviation

$$\sigma = \sqrt{5.2}$$
$$= \boxed{2.28\%}$$

Final Report:
Model Performance: $85.0\% \pm 2.28\%$
Confidence Interval (approximately):

- Lower bound: $85\% - 2.28\% = 82.72\%$
- Upper bound: $85\% + 2.28\% = 87.28\%$

Interpretation:

- The model achieves 85% accuracy on average
- Performance is quite consistent (low std = 2.28%)
- We can be confident the model will score between 83-87% on new data
- If std was high (e.g., 10%), we'd be less confident

Memory Hook — Remember This!

K-Fold Cross Validation Quick Guide:
When to use:

- Small datasets (get more reliable estimate)
- Hyperparameter tuning (find best learning rate, dropout rate, etc.)
- Model comparison (which architecture is better?)
- When you need confidence intervals

Typical workflow:

1. Split: 80% for K-fold, 20% for final test (never touch!)
2. Use K-fold on the 80% to:
    - Choose hyperparameters
    - Compare architectures
    - Get performance estimate
3. After selecting best model, train on ALL 80%
4. Evaluate ONCE on the 20% test set

Choosing K:

- Small data ($<$1000 samples): K=10
- Medium data (1000-10000): K=5
- Large data ($>$10000): K=3 (or just use single validation split)

Drawback: K times more expensive! (K=5 means 5x training time)

9 Advanced Optimization Methods

Why It Matters

Standard gradient descent (SGD) is slow and gets stuck easily. Advanced optimizers like Adam, RMSprop, and Adagrad make training MUCH faster and more stable by:

- Adapting learning rate per parameter
- Using momentum to escape local minima
- Handling sparse gradients better

These are used in almost ALL modern deep learning! Understanding them is crucial.

9.1 Problems with Vanilla SGD

Why We Need Better Optimizers

Problems with Standard SGD:

1. **Same learning rate for ALL parameters**
   - Some parameters need big steps, others need small steps
   - One learning rate doesn't fit all!
2. **Gets stuck in local minima or saddle points**
   - No momentum to escape
3. **Very sensitive to learning rate choice**
   - Too high: diverges
   - Too low: takes forever
4. **Poor on sparse data**
   - Parameters with rare features don't update enough

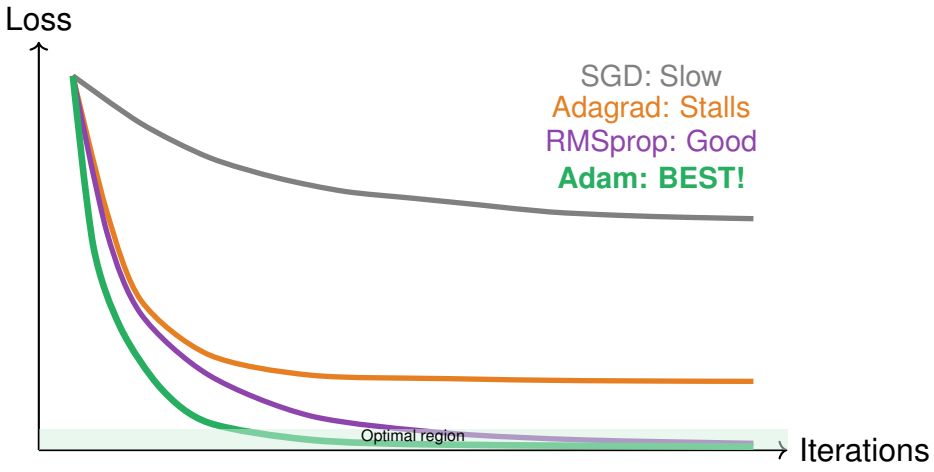


Figure 8: Typical convergence behavior: Adam usually converges fastest and most stably

9.2 Adagrad: Adaptive Learning Rates

Adagrad - Simple Explanation

Main Idea: Give each parameter its OWN learning rate based on how often it's updated.
How it works:

- Parameters that update frequently → get SMALLER learning rate
- Parameters that update rarely → get LARGER learning rate

Good for: Sparse data (e.g., text, where most words don't appear)
Problem: Learning rate keeps shrinking → eventually stops learning

Adagrad Update Rules

Accumulate squared gradients:

$$\text{cache} \leftarrow \text{cache} + (\nabla w)^2$$

Update weight:

$$\boxed{w \leftarrow w - \frac{\eta}{\sqrt{\text{cache}} + \epsilon} \cdot \nabla w}$$

Where:

- cache = Running sum of ALL past squared gradients
- ϵ = Small constant (10^{-8}) to prevent division by zero
- As cache grows, effective learning rate = $\frac{\eta}{\sqrt{\text{cache}}}$ shrinks

Problem: Cache keeps growing forever → learning rate goes to zero → stops learning!

Analogy — Think of It Like This

Adagrad is like taking smaller steps the more you walk:

- First step: Full stride
- After 100 steps: Half stride (getting tired)
- After 1000 steps: Tiny steps (exhausted)
- After 10000 steps: Barely moving (stopped!)

Good for short journeys, but not for long training!

9.3 RMSprop: Fixing Adagrad

RMSprop - The Fix

Main Idea: Instead of accumulating ALL past gradients, use a MOVING AVERAGE.
Key improvement: Old gradients are "forgotten" → learning rate doesn't shrink forever
Created by: Geoffrey Hinton (Turing Award winner)
Usage: Very popular, especially for RNNs

RMSprop Update Rules

Update cache (moving average):

$$\boxed{\text{cache} \leftarrow \rho \cdot \text{cache} + (1-\rho) \cdot (\nabla w)^2}$$

Update weight:

$$\boxed{w \leftarrow w - \frac{\eta}{\sqrt{\text{cache}} + \epsilon} \cdot \nabla w}$$

Where:

- ρ = Decay rate (typically 0.9 or 0.99)
- $(1-\rho)$ = How much weight to give new gradient
- Moving average "forgets" old gradients exponentially

Example with $\rho = 0.9$:

- Current gradient gets weight: $1 - 0.9 = 0.1$ (10%)
- Previous cache gets weight: 0.9 (90%)
- Very old gradients have weight: $0.9^{100} \approx 0$ (forgotten!)

Analogy — Think of It Like This

RMSprop vs Adagrad:
Adagrad (Bad):

- Remembers EVERY step you've ever taken
- Gets more and more tired
- Eventually can't walk anymore

RMSprop (Good):

- Only remembers recent steps
- Old steps fade away
- Maintains steady energy level

It's like having a "short memory" - you don't get exhausted by ancient history!

9.4 Adam: The King of Optimizers

Adam - Adaptive Moment Estimation

Main Idea: Combine the BEST of both worlds:

- Momentum (like a ball rolling downhill - builds speed)
- Adaptive learning rates (like RMSprop - per-parameter rates)

Result: Fast, stable, works well almost everywhere!
Most popular optimizer in deep learning today (2024-2025)

Adam Update Rules - Complete

Adam tracks TWO moving averages:
Step 1: Update first moment (momentum)

$$m \leftarrow \beta_1 \cdot m + (1 - \beta_1) \cdot \nabla w$$

Step 2: Update second moment (squared gradients)

$$v \leftarrow \beta_2 \cdot v + (1 - \beta_2) \cdot (\nabla w)^2$$

Step 3: Bias correction (important for early steps!)

$$\hat{m} = \frac{m}{1 - \beta_1^t}, \quad \hat{v} = \frac{v}{1 - \beta_2^t}$$

Where t = current iteration number
Step 4: Update weight

$$\boxed{w \leftarrow w - \frac{\eta}{\sqrt{\hat{v}} + \epsilon} \cdot \hat{m}}$$

Default hyperparameters (almost always work!):

- $\beta_1 = 0.9$ (momentum decay)
- $\beta_2 = 0.999$ (RMSprop decay)
- $\epsilon = 10^{-8}$
- $\eta = 0.001$ or 0.0001 (learning rate)

Example 6: Adam Optimizer (Complete Step)

Given Data:

- Current weight: $w = 2.0$
- Current gradient: $\nabla w = 0.5$
- Learning rate: $\eta = 0.1$
- Parameters: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$
- Initial state: $m = 0$, $v = 0$
- Current iteration: $t = 1$ (first step)

Solution:
Step 1: Update first moment (momentum)

$$\begin{aligned} m &= \beta_1 \cdot m + (1 - \beta_1) \cdot \nabla w \\ &= 0.9 \times 0 + (1 - 0.9) \times 0.5 \\ &= 0 + 0.1 \times 0.5 \\ &= \boxed{0.05} \end{aligned}$$

Step 2: Update second moment (squared gradients)

$$\begin{aligned} v &= \beta_2 \cdot v + (1-\beta_2) \cdot (\nabla w)^2 \\ &= 0.999 \times 0 + (1-0.999) \times (0.5)^2 \\ &= 0 + 0.001 \times 0.25 \\ &= \boxed{0.00025} \end{aligned}$$

Step 3: Bias correction (critical for early iterations!)
For m:

$$\begin{aligned} \hat{m} &= \frac{m}{1-\beta_1^t} = \frac{0.05}{1-0.9^1} \\ &= \frac{0.05}{1-0.9} = \frac{0.05}{0.1} \\ &= \boxed{0.5} \end{aligned}$$

For v:

$$\begin{aligned} \hat{v} &= \frac{v}{1-\beta_2^t} = \frac{0.00025}{1-0.999^1} \\ &= \frac{0.00025}{1-0.999} = \frac{0.00025}{0.001} \\ &= \boxed{0.25} \end{aligned}$$

Why bias correction? Without it, m and v start biased toward zero (since initialized at 0)!
Step 4: Update weight

$$\begin{aligned} w_{\text{new}} &= w - \frac{\eta}{\sqrt{\hat{v}} + \epsilon} \cdot \hat{m} \\ &= 2.0 - \frac{0.1}{\sqrt{0.25} + 10^{-8}} \times 0.5 \\ &= 2.0 - \frac{0.1}{0.5 + 0.00000001} \times 0.5 \\ &= 2.0 - \frac{0.1}{0.5} \times 0.5 \\ &= 2.0 - 0.2 \times 0.5 \\ &= 2.0 - 0.1 \\ &= \boxed{\mathbf{1.9}} \end{aligned}$$

Interpretation:

- Weight changed from 2.0 $\rightarrow$ 1.9
- The bias correction scaled up m from 0.05 to 0.5 (10x!)
- Without bias correction, update would be too small initially
- This is why Adam converges faster than RMSprop early in training

Optimizer	Pros	Cons	Best For
SGD	Simple, well-understood, proven	Slow, sensitive to LR	When you have time to tune
Adagrad	Good for sparse data	Learning rate decays to 0	Text/NLP with rare words
RMSprop	Doesn't stop learning	Still sensitive to LR	RNNs, time series
Adam	Fast, robust, works everywhere	Can overfit more	DEFAULT CHOICE!

Table 2: Optimizer comparison - When to use what

Memory Hook — Remember This!

Quick Optimizer Selection Guide:
90% of the time: Use **Adam** with default parameters!

- Learning rate: Start with 0.001
- If training is unstable: reduce to 0.0001
- If training is too slow: increase to 0.01

Special cases:

- **Text/NLP with sparse features:** Adagrad
- **RNNs:** RMSprop or Adam
- **Need absolute best performance (and have time):** SGD with momentum + careful tuning
- **Not sure:** Adam (it almost always works!)

Pro tip: Adam with batch normalization = very stable training!

Exam Question from MID-1

MID-1 Exam Question (3 marks):
"With sparse, high-dimensional, and noisy molecular data, should you choose Gradient Descent, SGD, Adam, or RMSProp? Explain how your choice affects convergence and stability."

Model Answer

Model Answer:
Best Choice: Adam Optimizer
Why Adam for this scenario:
1. Handles Sparse Data:

- Molecular features are sparse (many zeros in descriptor vectors)
- Adam adapts learning rate per parameter
- Rare molecular features get larger updates when they appear
- Parameters update even with infrequent gradients

2. Robust to Noise:

- Momentum (first moment m) smooths out noisy gradients

- Moving average filters high-frequency noise
- More stable than plain SGD which jumps around with noisy gradients

3. High-Dimensional Data:

- Adaptive per-parameter learning rates crucial for 1000+ dimensional molecular descriptors
- Different features have vastly different scales $\rightarrow$ need different step sizes
- Adam automatically handles this without manual tuning

Effect on Convergence:

- Converges 5-10x faster than SGD
- Reaches lower training error in fewer epochs
- Less sensitive to initial learning rate choice
- Bias correction prevents slow start

Effect on Stability:

- Momentum smooths training trajectory
- Adaptive rates prevent divergence from outliers
- More consistent performance across different random seeds
- Less likely to get stuck in bad local minima

Recommended Settings:

$$\text{Adam:} \quad \eta = 0.001,\ \beta_1 = 0.9,\ \beta_2 = 0.999$$

Alternative: RMSprop would work, but Adam's momentum provides better convergence in practice.

10 Putting It All Together: A Complete Strategy

Why It Matters

You've learned many techniques - but how do you use them together? This section gives you a PRACTICAL RECIPE for training robust models.

10.1 The Standard Deep Learning Recipe

Recommended Training Strategy

1. Data Preparation:

- Split: 60% train, 20% validation, 20% test
- Apply data augmentation to training set
- Normalize/standardize all features

2. Model Architecture:

- Layer structure: Dense $\rightarrow$ Batch Norm $\rightarrow$ ReLU $\rightarrow$ Dropout $\rightarrow$ Dense $\rightarrow$ ...
- Use Batch Normalization after each dense layer
- Apply Dropout (p=0.5) after activation functions

3. Regularization:

- L2 regularization: $\lambda = 0.01$ or 0.001
- Dropout: $p = 0.5$ for hidden layers
- Early stopping: patience = 10

4. Optimization:

- Optimizer: Adam (default parameters)
- Learning rate: Start with 0.001
- Batch size: 32 or 64

5. Training Loop:

```
for epoch in range(max_epochs):
    # Training
    for batch in training_data:
        forward_pass()
        compute_loss()  # Loss + L2 penalty
        backward_pass()
        optimizer.step()

    # Validation
    val_error = evaluate(validation_data)

    # Early stopping check
    if val_error < best_val_error:
        save_model()
```

```
    else:
        patience_counter += 1

    if patience_counter >= patience:
        break
```

6. Final Evaluation:

- Load best model (from early stopping)
- Evaluate ONCE on test set
- Report mean ± std if using k-fold

10.2 Troubleshooting Guide

Problem	Symptoms	Solutions to Try
Overfitting	Train error $\ll$ Test error	Increase dropout ($\rightarrow$ 0.6), add L2, early stopping, more data
Underfitting	Both errors high	Add layers/neurons, train longer, reduce regularization
Slow training	Takes forever	Use Adam, increase learning rate, add batch norm
Unstable training	Loss jumps around	Reduce learning rate, add batch norm, reduce batch size
Exploding gradients	Loss $\rightarrow$ NaN	Reduce LR, gradient clipping, check data
Vanishing gradients	No learning in deep networks	Use ReLU not sigmoid, add batch norm, residual connections

11 Summary and Key Takeaways

The Most Important Points

1. The Goal: Build models that GENERALIZE (work on new data), not just memorize

2. Two Main Problems:

- **Underfitting**: Model too simple → increase complexity
- **Overfitting**: Model too complex → use techniques below!

3. Regularization Techniques (prevent overfitting):

- **L2 (Weight Decay)**: Penalize large weights, $w \leftarrow w(1 - \eta\lambda)$
- **L1**: Sparse models, some weights → 0
- **Dropout**: Randomly drop neurons (p=0.5), scale by 1/(1-p)
- **Batch Normalization**: Normalize layer inputs, stabilizes training
- **Early Stopping**: Stop when validation error increases
- **Data Augmentation**: Create more training data

4. Proper Evaluation:

- **K-Fold Cross Validation**: Get mean ± std estimate
- **Never touch test set** until final evaluation!

5. Optimizers:

- **Adam**: Best default choice (fast, robust)
- **RMSprop**: Good for RNNs
- **Adagrad**: Sparse data

6. Best Practices:

- Start with: Adam + Dropout + Batch Norm + Early Stopping
- Monitor BOTH training and validation error
- If overfitting: more regularization
- If underfitting: more capacity

Self-Test — Check Your Understanding

Quick Self-Test:

1. Your model has training error = 2%, test error = 25%. What's the problem?
2. If dropout probability p=0.3, what scaling factor is applied?
3. Why do we use bias correction in Adam?
4. When should you use L1 instead of L2 regularization?
5. What's wrong with using test set for early stopping?

Answers:

1. Overfitting! Gap = 23%. Use dropout, L2, early stopping, or more data.
2. $\frac{1}{1-p} = \frac{1}{0.7} \approx 1.43$
3. Because m and v are initialized at 0, they're biased toward 0 initially. Correction

fixes this.
4. When you have many features but only some are important (want sparse model).
5. You'll overfit to the test set! Use validation set for early stopping, test set only for final evaluation.

Exam Question from MID-1

MID-1 Exam Question 3 (8 marks) - Matching:
"Match each Column A challenge with the most appropriate Column B ANN technique."

Column A - Challenge	Column B - Solution
A. Vanishing gradients in deep layers	ReLU / Leaky ReLU activation
B. Modeling hierarchical non-additive interactions	Multiple hidden layers with nonlinear activations
C. Predicting multiple targets with imbalanced classes	Weighted categorical cross-entropy loss
D. Sparse, noisy, high-dimensional data convergence	Adam optimizer
E. Overfitting rare active compounds	Dropout, weight decay, early stopping
F. Stabilizing learning across varying input scales	Batch normalization / layer normalization
G. Reliable predictions with confidence	Dropout (uncertainty estimation)
H. Efficient training under limited resources	Mini-batch training or adaptive batch sizing

Model Answer

Complete Solutions with Explanations:

A → ReLU/Leaky ReLU:

- Sigmoid/tanh saturate (gradient → 0) in deep networks
- ReLU: gradient = 1 for positive values (no saturation)
- Leaky ReLU: small gradient even for negative values
- Enables training of 50+ layer networks

B → Multiple Hidden Layers:

- Deep networks learn hierarchical representations
- Early layers: simple features
- Middle layers: combinations of features
- Deep layers: complex non-additive interactions
- Nonlinear activations enable learning complex functions

C → Weighted Cross-Entropy:

- Standard loss gives equal weight to all classes
- Imbalanced data: model ignores rare class
- Weighted loss: penalize rare class errors more heavily
- Multi-task: separate loss per target, weighted sum

D → Adam Optimizer:

- Sparse data: infrequent gradients
- Noise: requires robust averaging
- High-dimensional: needs adaptive per-parameter rates
- Adam handles all three with momentum + adaptive learning rates

E → Dropout + Weight Decay + Early Stopping:

- Rare compounds: easy to memorize
- Dropout: forces redundant learning
- Weight decay (L2): prevents large weights on specific features
- Early stopping: prevents late-stage memorization

F → Batch Normalization:

- Molecular descriptors: widely varying scales (0.01 to 1000)
- BN normalizes to mean=0, std=1 per layer
- Stabilizes gradients, allows higher learning rates
- Reduces internal covariate shift

G → Dropout (for Uncertainty):

- Dropout = ensemble of models
- At test time: run multiple forward passes with dropout
- Variance in predictions = uncertainty estimate
- High variance = model unsure (don't trust prediction)

H → Mini-batch Training:

- Full-batch: too memory intensive
- Single sample (SGD): too noisy, slow
- Mini-batches (32-256): balance speed and memory
- Adaptive batch size: start small, increase over time

12 Final Words: Your Action Plan

What to Do Next

To Master This Material:

1. **Practice the numerical examples** (Examples 1-6)
   - Do them by hand first
   - Then implement in code
2. **Implement from scratch** (learning exercise):
   - Dropout layer
   - L2 regularization
   - Early stopping
   - K-fold cross validation
3. **Use in real projects:**
   - Start with standard recipe (Adam + Dropout + BN + Early Stopping)
   - Monitor training vs validation error curves
   - Adjust based on what you see
4. **For exams:**
   - Memorize key formulas (in the summary boxes)
   - Understand WHY each technique works
   - Practice exam questions at end of each section
   - Can explain trade-offs (L1 vs L2, Adam vs RMSprop)

Remember: These techniques are used in EVERY production deep learning system. Master them, and you're 80% of the way to being a deep learning practitioner!