140.800: How to AI (for Public Health)

Week 2: From Theory to Practice - Optimization, Neural Networks, and Text Processing

Yiqun T. Chen
Email: yiqunc@jhu.edu
Schedule office hours via email

Departments of Biostatistics and Computer Science
Data Science & AI Initiative and Malone Center for Engineering in Health

The Universal ML Framework: $Y = f(X) + \epsilon$

Quick Recap:

- Y: Outcomes we want to predict (diagnosis, treatment response)
- X: Features/predictors (symptoms, test results, demographics)
- f: The function we're trying to learn
- ϵ: Random noise and unmeasured factors

Key Insight: Machine learning is about finding the best approximation to f

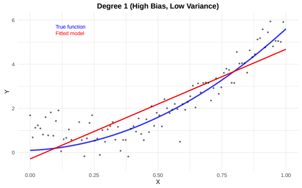
Today's Focus: How do we actually find f in practice?

- Optimization: How to search for the best f
- Neural networks: Flexible function approximators
- Text processing: Handling non-numerical data

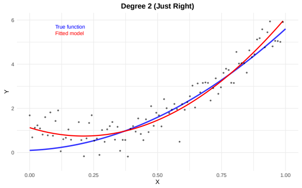
Bias-Variance Tradeoff Recap

Remember our polynomial example:

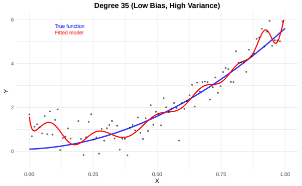
Degree 1 (High Bias)



Degree 2 (Just Right)



Degree 35 (High Variance)



The Central Challenge: How complex should our model be?

Formal Definition: Bias-Variance Decomposition

For any learning algorithm, the expected prediction error decomposes as:

$$\mathbb{E}[(Y - \hat{f}(X))^2] = \text{Bias}^2[\hat{f}(X)] + \text{Var}[\hat{f}(X)] + \sigma^2$$

Where:

- $\text{Bias}[\hat{f}(X)] = \mathbb{E}[\hat{f}(X)] - f(X)$
- $\text{Var}[\hat{f}(X)] = \mathbb{E}[(\hat{f}(X) - \mathbb{E}[\hat{f}(X)])^2]$
- σ^2 is irreducible error (noise in the data)

Biomedicine Example:

- **High Bias**: Simple rule "age $> 65 \rightarrow$ high risk" (systematic errors)
- **High Variance**: Complex model that changes dramatically with new patients
- **Goal**: Find the sweet spot that minimizes total error

Train/Validation/Test Split Strategy

The Gold Standard Approach:

Training Set (60-70%): Learn model parameters

Validation Set (15-20%): Select model complexity/hyperparameters

Test Set (15-20%): Final unbiased performance evaluation

Why Three Sets?

- Training: Optimizes parameters for that specific data
- Validation: Prevents overfitting during model selection
- Test: Gives honest estimate of real-world performance

Cross-Validation: Making Better Use of Data

Problem: Small datasets → unreliable validation estimates
Solution: K-fold cross-validation

1. Divide data into K folds (typically 5 or 10)
2. Train on K-1 folds, validate on 1 fold
3. Repeat K times, each fold as validation once
4. Average performance across all folds

Biomedicine Advantage:

- Better use of limited patient data
- More robust performance estimates
- Reduces impact of "lucky" or "unlucky" splits

Leave-One-Out (LOO): Special case where K = sample size

- Maximum use of training data
- Computationally expensive for large datasets

Modern Data Challenges: Beyond Random Splits

Traditional Assumption: Data is independent and identically distributed (i.i.d.)

Reality Check: Three major challenges invalidate random splits

1. **Temporal Dependencies**: Future data differs from past data
2. **Distributional Shift**: Population characteristics change over time
3. **Similarity Constraints**: Related samples should not span train/test

Why This Matters: Random splits give overly optimistic performance estimates

Modern Data Challenges: Detailed Examples

1. **Temporal Dependencies:**
   - Train on 2020-2022 data, test on 2023 data
   - Accounts for changes in practice patterns, technology updates
   - Example: Medical guidelines evolve, treatment protocols change
2. **Distributional Shift:**
   - **Covariate shift**: Demographics change (aging population, migration)
   - **Label shift**: Disease prevalence changes (pandemics, seasonal effects)
   - Example: COVID-19 dramatically shifted disease patterns
3. **Similarity Constraints:**
   - Split by institution (hospital-to-hospital generalization)
   - Split by patient ID (prevent data leakage from same individual)
   - Split by related cases (family studies, genetic similarities)

Types of Features in Biomedical Data

Categorical Features:

- **Nominal**: Gender, race, diagnosis codes (no natural order)
- **Ordinal**: Severity scores, education levels (ordered categories)

Continuous Features:

- Lab values, vital signs, age, BMI
- May need scaling/normalization

Non-Numerical Features:

- **Text**: Clinical notes, pathology reports
- **Images**: X-rays, MRIs, pathology slides
- **Sequences**: Time series, DNA sequences

Key Challenge: Computers only understand numbers!

- Need to encode everything into numerical representation
- Encoding choice affects model performance

From Manual to Automatic Feature Learning

Traditional Text Processing Pipeline:

1. **Tokenization**: "Patient has diabetes" $\rightarrow$ [Patient, has, diabetes]
2. **Normalization**: Lowercase, remove punctuation
3. **Stop word removal**: Remove "the", "and", "is"
4. **Stemming/Lemmatization**: "running" $\rightarrow$ "run"

Traditional ML: Domain expert designs features manually
Modern Deep Learning: Let gradient descent find optimal features

Key Insight: We will revisit how modern approaches learn representations automatically

Why the Shift to Deep Learning?

Scale and Performance:

- Modern datasets too large/complex for manual feature engineering
- Deep models consistently outperform hand-crafted features
- Same architectures work across domains (vision, language, audio)

The Learning Problem: Back to $Y = f(X) + \epsilon$

Empirical Risk Minimization (ERM): Given training data $(x_1, y_1), ..., (x_n, y_n)$, find f_θ that minimizes:

$$\mathcal{L}(\theta) = \frac{1}{n} \sum_{i=1}^{n} \ell(f_\theta(x_i), y_i)$$

Key Insight: Loss function $\ell(\cdot, \cdot)$ is our way to obtain $f(X)$

- Tells us how "wrong" our predictions are
- Guides the learning algorithm toward better solutions
- Different losses $\rightarrow$ different learned functions

Requirements for Loss Functions:

- (Almost) differentiable for gradient-based optimization
- Should align with what we actually care about

The Two Most Important Loss Functions

1. Mean Squared Error (MSE) - For Regression:

$$\ell_{\mathsf{MSE}}(y, \hat{y}) = (y - \hat{y})^2$$

Properties:

- Penalizes large errors
- Differentiable everywhere
- Used when Y is (almost) continuous (blood pressure, age, etc.)

2. Cross-Entropy Loss - For Classification:

$$\ell_{\mathsf{CE}}(y, \hat{y}) = -\sum_{c=1}^{C} y_c \log(\hat{y}_c)$$

Properties:

- $y_c \in \{0, 1, \ldots, C\}$ (true class), $\hat{y}_c \in [0, 1]$ (predicted probability for class c)
- Penalizes confident wrong predictions

These two losses power most of modern machine learning!

Worked Example: Linear Regression

Problem: Find best line $y = ax + b$ for data points
Step 1: Define loss function

$$\mathcal{L}(a, b) = \frac{1}{n}\sum_{i=1}^{n}(y_i - (ax_i + b))^2$$

Step 2: Compute gradients

$$\frac{\partial \mathcal{L}}{\partial a} = -\frac{2}{n}\sum_{i=1}^{n} x_i(y_i - ax_i - b)$$

$$\frac{\partial \mathcal{L}}{\partial b} = -\frac{2}{n}\sum_{i=1}^{n}(y_i - ax_i - b)$$

Step 3: Update parameters

$$a_{t+1} = a_t - \eta\frac{\partial \mathcal{L}}{\partial a}, \quad b_{t+1} = b_t - \eta\frac{\partial \mathcal{L}}{\partial b}$$

Gradient Descent: The Core Algorithm

The fundamental optimization algorithm:

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

Where:

- θ: model parameters (weights)
- η: learning rate (step size)
- $\nabla_\theta \mathcal{L}$: gradient of loss with respect to parameters

Intuition:

- Gradient points in direction of steepest increase
- We want to minimize loss $\rightarrow$ go in opposite direction
- Step size controlled by learning rate η

Key Insight: This same algorithm scales from simple linear regression to billion-parameter neural networks!

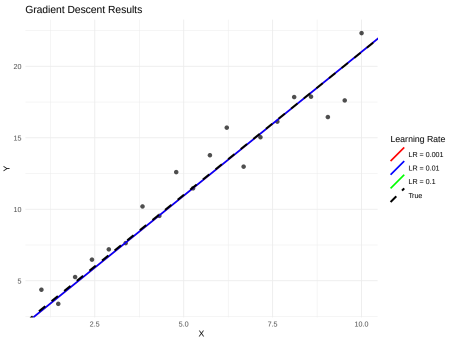
Numerical Example: First 5 Iterations

Data: True line is $y = 2x + 1$, learning rate $\eta = 0.01$

Iteration	a (slope)	b (intercept)	Loss
0	0.000	0.000	225.000
1	1.615	0.244	175.167
2	1.985	0.305	11.196
3	2.069	0.325	2.542
4	2.088	0.334	2.081
5	2.091	0.342	2.052

Observation: Rapid convergence from random initialization (0,0) toward true values (2,1)
Key Insight: Loss decreases dramatically in first few steps!

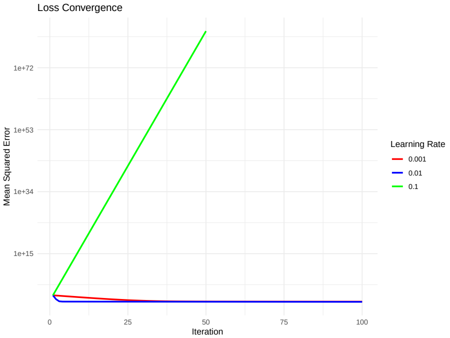
Gradient Descent in Action



Key Observations:

- Different learning rates affect convergence speed
- Too small → slow convergence
- Too large → may overshoot and diverge
- "Just right" → efficient convergence to optimal solution

Learning Rate Effects



Learning Rate Selection:

- Start with common values: 0.01, 0.001, 0.1
- Monitor loss convergence during training
- Use learning rate schedules (decrease over time)
- Modern optimizers adapt learning rates automatically

Batch Gradient Descent

The Standard Approach: Process all training data at once

$$\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n} \ell(f_\theta(x_i), y_i)$$

Advantages:

Disadvantages:

Batch Gradient Descent

The Standard Approach: Process all training data at once

$$\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n} \ell(f_\theta(x_i), y_i)$$

Advantages:

- Stable gradient estimates (true gradient)
- Guaranteed convergence to local minimum
- Reproducible results

Disadvantages:

- Computationally expensive for large datasets
- Memory requirements scale with dataset size
- Slow convergence (especially early in training)

When to use: Small to medium datasets (<10k samples)

Stochastic & Mini-batch Gradient Descent

Stochastic Gradient Descent (SGD):

$$\mathcal{L}(\theta) = \ell(f_\theta(x_i), y_i) \quad \text{(single sample)}$$

- Uses one sample at a time
- Fast updates, but noisy gradients
- Can escape local minima due to noise

Mini-batch Gradient Descent: The practical choice

$$\mathcal{L}(\theta) = \frac{1}{B} \sum_{i \in \text{batch}} \ell(f_\theta(x_i), y_i) \quad \text{(batch size B)}$$

- Uses small batches (32, 64, 128, 256)
- Good balance of speed and stability
- Enables efficient GPU parallelization

Modern Optimizers: Beyond Basic SGD

Why Basic SGD Has Problems:

- Same learning rate for all parameters
- Can get stuck in poor local minima
- Sensitive to learning rate choice

Adam Optimizer (Most Popular):

- Adaptive learning rates per parameter
- Combines momentum with adaptive scaling
- Works well "out of the box" for most problems

PyTorch Usage:

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
# Also available:  SGD, AdamW, RMSprop, etc.
```

Key Insight: Adam is often the default choice because it "just works" for most neural network training scenarios.

Training Concepts: Key Terminology

Batch Size: Number of samples per update

- Common sizes: 32, 64, 128, 256
- Smaller = more updates, more noise

Epoch: One complete pass through training data

- Example: 1000 samples, batch size 100 $\rightarrow$ 10 batches per epoch

Shuffling: Randomize sample order between epochs

- Prevents memorizing data order
- Standard practice for better generalization

From Linear to Non-Linear Models

Linear Model Limitations:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_p x_p$$

- Can only model linear relationships
- No feature interactions without manual engineering
- Limited expressiveness for complex patterns

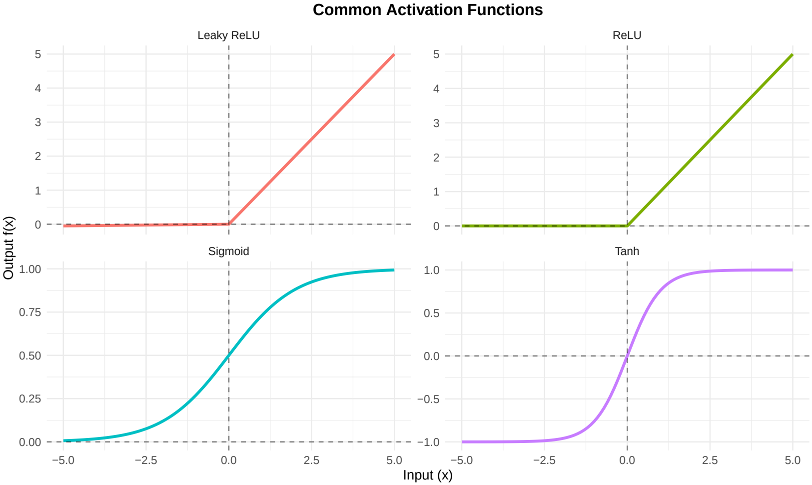
Neural Network Solution: Add hidden layers with non-linear activation functions:

$$\mathbf{h}_1 = \sigma(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1)$$

$$y = \mathbf{W}_2 \mathbf{h}_1 + b_2$$

- σ is activation function - introduces non-linearity
- Multiple layers can learn complex feature interactions
- Universal approximation: can approximate any continuous function

Activation Functions: The Key to Non-linearity



Central Question: Why do we need non-linear activation functions?

Why Non-linearity Matters

The Mathematical Reality:

- Without activation functions, multiple layers collapse to single linear transformation
- Example: $f(g(x)) = W_2(W_1x + b_1) + b_2 = (W_2W_1)x + (W_2b_1 + b_2)$

Activation Function Properties:

- **ReLU**: Most popular - simple, efficient, avoids vanishing gradients
- **Sigmoid**: Good for binary classification outputs (0-1 range)
- **Tanh**: Centered around zero, good for hidden layers

Key Insight: Non-linearity enables the network to learn complex patterns that no linear model can capture

Worked Example: 2-Layer Neural Network

Input: $x_1 = 0.5, x_2 = -0.3$
Layer 1: $\mathbf{h}_1 = \text{ReLU}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$

$$\mathbf{W}_1 = \begin{pmatrix} 0.2 & -0.5 \\ 0.8 & 0.1 \end{pmatrix}, \quad \mathbf{b}_1 = \begin{pmatrix} 0.3 \\ -0.1 \end{pmatrix}$$

$$\mathbf{z}_1 = \begin{pmatrix} 0.2 & -0.5 \\ 0.8 & 0.1 \end{pmatrix} \begin{pmatrix} 0.5 \\ -0.3 \end{pmatrix} + \begin{pmatrix} 0.3 \\ -0.1 \end{pmatrix} = \begin{pmatrix} 0.55 \\ 0.27 \end{pmatrix}$$

$$\mathbf{h}_1 = \text{ReLU}(\mathbf{z}_1) = \begin{pmatrix} 0.55 \\ 0.27 \end{pmatrix}$$

Layer 2: $y = \text{Sigmoid}(\mathbf{W}_2\mathbf{h}_1 + b_2)$

$$y = \text{Sigmoid}(1.2 \times 0.55 + (-0.7) \times 0.27 + 0.1) = \text{Sigmoid}(0.571) = 0.639$$

Compare to Linear: $y_{\text{linear}} = 0.5 \times 0.5 + (-0.2) \times (-0.3) + 0.1 = 0.41$
Key Insight: Non-linear activation allows the network to learn complex patterns that linear models cannot capture!

Computing Derivatives: Deep Learning ≈ Computing Derivatives

The Challenge: How do we compute gradients efficiently in deep networks?

Chain Rule to the Rescue: For a 2-layer network:
$y = \sigma_2(\mathbf{W}_2\sigma_1(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + b_2)$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_1} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial \mathbf{h}_1}\frac{\partial \mathbf{h}_1}{\partial \mathbf{z}_1}\frac{\partial \mathbf{z}_1}{\partial \mathbf{W}_1}$$

Key Insight: Chain rule enables efficient gradient computation through complex networks

Backpropagation Algorithm

The Three-Step Process:

1. **Forward Pass:**
   - Compute predictions layer by layer: $\mathbf{x} \to \mathbf{h}_1 \to \mathbf{h}_2 \to y$
   - Calculate loss: $\mathcal{L}(y, y_{true})$
2. **Backward Pass:**
   - Compute gradients using chain rule (right to left)
   - Start from loss, propagate back to all parameters
3. **Parameter Update:**
   - Apply gradient descent: $\mathbf{W} \leftarrow \mathbf{W} - \eta \nabla_{\mathbf{W}} \mathcal{L}$

This enables training networks with millions of parameters!

From Text to Numbers

The Challenge: Computers only understand numbers, but biomedicine generates lots of text

Clinical Text Examples:

- Progress notes, discharge summaries
- Radiology reports, pathology reports
- Drug prescriptions, adverse event reports
- Patient surveys and questionnaires

Text Processing Pipeline:

1. **Tokenization:** Break text into words/subwords
2. **Normalization:** Handle case, punctuation, abbreviations
3. **Vectorization:** Convert to numerical representation
4. **Classification:** Apply machine learning

Bag of Words: A Simple Example

Let's work through a concrete example with 4 sentences:
Documents:

- D1: "The patient has a fever"
- D2: "The patient needs a treatment"
- D3: "A fever requires the treatment"
- D4: "The treatment helps the patient"

Step 1: Create Vocabulary

- Unique words: [the, patient, has, a, fever, needs, treatment, requires, helps]
- Vocabulary size: 9 words
- Notice: Many common words repeated: "the" (5x), "patient" (3x), "a" (3x)

Step 2: Build BOW Matrix (next slide)

BOW Matrix for Our Example

BOW Matrix (Documents $\times$ Vocabulary):

	the	patient	has	a	fever	needs	treatment	requires	helps
D1	1	1	1	1	1	0	0	0	0
D2	1	1	0	1	0	1	1	0	0
D3	1	0	0	1	1	0	1	1	0
D4	2	1	0	0	0	0	1	0	1

Observations:

- Each document is now a vector of word counts
- Common words dominate: "the" appears 5 times total, "patient" 3 times
- We can now compute similarity between documents
- Problem: Common words like "the" overwhelm meaningful words

TF-IDF: Beyond Simple Word Counts

Problem with BoW: Common words dominate ("the", "a", "patient")
TF-IDF Solution: Weight words by importance

$$\text{TF-IDF}(t, d) = \text{TF}(t, d) \times \log \frac{N}{\text{DF}(t)}$$

Where:

- **TF**(t, d): Term frequency in document d
- **DF**(t): Number of documents containing term t
- **N**: Total number of documents

TF-IDF Intuition: Why It Works

Let's apply TF-IDF to our example:
Word Frequency Analysis:

- "the" appears in 4/4 documents → very common word
- "patient" appears in 3/4 documents → common word
- "treatment" appears in 3/4 documents → common word
- "has", "helps", "requires" appear in 1/4 documents each → rare words

TF-IDF Weighting Results:

- Very low weight: "the" (appears in all docs)
- Low weight: "patient", "treatment" (appear in many docs)
- High weight: "has", "helps", "requires" (rare, discriminative)

Key Insight: TF-IDF automatically identifies the most informative words for distinguishing between documents!

TF-IDF Matrix: Actual Calculated Weights

TF-IDF Matrix for Our Example:

	the	patient	has	a	fever	needs	treatment	requires	helps
D1	0.00	0.10	0.30	0.00	0.30	0.00	0.00	0.00	0.00
D2	0.00	0.10	0.00	0.00	0.00	0.30	0.10	0.00	0.00
D3	0.00	0.00	0.00	0.00	0.30	0.00	0.10	0.30	0.00
D4	0.00	0.10	0.00	0.00	0.00	0.00	0.10	0.00	0.30

Key Observations:

- "the" gets weight 0.00 (appears in all documents - not discriminative)
- Unique words get weight 0.30: "has", "fever", "needs", "requires", "helps"
- Common words get lower weights: "patient", "treatment" (0.10)
- TF-IDF automatically downweights common words and emphasizes rare ones

The Word Order Problem in BOW

The Classic "Dog Bites Man" Example:

- "Dog bites man" → Common occurrence (not newsworthy)
- "Man bites dog" → Unusual event (front-page news!)

BOW Representation: Identical vectors!

Word	dog	bites	man
Count	1	1	1

The Problem:

- Completely different meanings and newsworthiness
- BOW treats them identically - subject/object roles lost
- Word order determines **who does what to whom**

N-grams: Capturing Some Context

Problem: BoW loses word order
Solution: N-grams capture local context

- **Unigrams:** individual words
- **Bigrams:** pairs of consecutive words
- **Trigrams:** triplets of consecutive words

Medical Example: "Patient has no chest pain"

- Unigrams: [patient, has, no, chest, pain]
- Bigrams: [patient has, has no, no chest, chest pain]
- Key insight: "no chest" helps detect negation

Interactive Demo: Try different n-gram combinations on medical text classification!

More BOW Failures: Negation

Negation Flips Meaning:

- "I liked the movie" → Positive sentiment
- "I didn't like the movie" → Negative sentiment

BOW Problem: Same words, similar counts; scope of "not" is lost

N-grams: Help only locally ("didn't like") but explode feature space

Why Embeddings Work Better:

- Contextual models (like BERT) bind "not" to "like" via sequence context
- Bidirectional attention captures negation scope
- Learn that "didn't like" ≈ "disliked" in vector space

More BOW Failures: Paraphrase and Synonyms

Semantic Similarity with Different Words:

- "He purchased a vehicle"
- "He bought a car"

Same meaning, different words!

BOW Problem: Low word overlap $\rightarrow$ vectors far apart

Why Embeddings Work Better:

- Distributed representations place synonyms near each other
- "purchased" $\approx$ "bought", "vehicle" $\approx$ "car" in vector space
- Sentence encoders keep semantically similar sentences close
- Learn meaning from context, not just word identity

More BOW Failures: Long-Distance Dependencies

Dependencies Across Clauses:

- "The book that you recommended was fantastic"
- "book" and "was" are grammatically linked but separated by words

BOW Problem: Can't model dependency between "book" and "was"
N-grams Problem: Can't stretch reliably across long distances

Why Embeddings Work Better:

- Self-attention (in Transformers) links distant tokens directly
- Each word can "attend" to any other word in the sentence
- Models learn grammatical relationships regardless of distance

More BOW Failures: Word Sense Disambiguation

Same Word, Different Meanings:

- "I went to the bank to deposit money" (financial institution)
- "We sat by the river bank" (riverside)

BOW Problem: One column per token; no sense differentiation

Why Embeddings Work Better:

- Contextual vectors (like BERT) give different embeddings for different senses
- "bank" + "deposit money" $\rightarrow$ financial meaning
- "bank" + "river" $\rightarrow$ geographical meaning
- Context determines representation dynamically

From Sparse to Dense Representations

Problem with BoW and TF-IDF:

- Sparse, high-dimensional vectors (vocabulary size = 10,000+)
- No semantic relationships: "doctor" and "physician" are unrelated
- Bag of words loses all word order information

Solution: Dense Word Embeddings

- Map each word to a dense vector (typically 100-300 dimensions)
- Words with similar meanings have similar vectors
- Capture semantic relationships: king - man + woman $\approx$ queen

Key Insight: "You shall know a word by the company it keeps"

Word2Vec: Learning Word Representations

Skip-gram Architecture: Single hidden layer neural network

Mathematical Objective: Maximize log probability of context words

$$J(\theta) = \frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0}\log p(w_{t+j}|w_t)$$

Where:

- T = total words in corpus
- c = context window size
- w_t = target word at position t
- w_{t+j} = context word at position $t+j$

Softmax Probability:

$$p(w_o|w_c) = \frac{\exp(u_o^T v_c)}{\sum_{i=1}^{|V|}\exp(u_i^T v_c)}$$

Where v_c = center word vector, u_o = context word vector

Word2Vec: Concrete Training Example

Training Sentence: "The patient has diabetes and requires treatment"

Skip-gram Training Pairs (window size = 2):

Target → Context
patient → [The, has]
has → [The, patient, diabetes]
diabetes → [patient, has, and]
and → [has, diabetes, requires]
requires → [diabetes, and, treatment]

Learning Process:

1. Initialize random 300-dim vectors for each word
2. For each training pair, predict context probability
3. Use gradient descent to adjust vectors to increase probability
4. Similar words end up with similar vectors through shared contexts

Word Embedding Properties: Similarity and Bias

Semantic Similarity (Cosine Distance):

Word	Most Similar Words
doctor	physician (0.82), surgeon (0.79), clinician (0.76)
diabetes	hypertension (0.71), cardiovascular (0.68)
treatment	therapy (0.85), medication (0.73)

The Famous Analogy: Vector Arithmetic

$$\text{king} - \text{man} + \text{woman} \approx \text{queen}$$

Why This Works:

- king $-$ man $\approx$ "royalty" concept
- woman$+$ "royalty" $\approx$ female royalty $=$ queen
- Linear relationships in embedding space capture semantic relationships

Word Embedding Bias: A Critical Issue

Embeddings Inherit Training Data Biases:

Gender Bias Examples:

- "Programmer" closer to "he" than "she"
- "Nurse" closer to "she" than "he"
- "Doctor" historically closer to male pronouns

Racial and Cultural Biases:

- Names associated with race affect sentiment scores
- Historical medical literature biases get encoded
- Geographic and socioeconomic biases persist

Critical for Biomedical AI:

- Can perpetuate healthcare disparities
- May misclassify based on patient demographics
- Requires careful auditing and debiasing techniques
- Active area of AI ethics research

Why Sequence Matters: A Critical Example

Famous Example:

- "John loves Mary"
- "Mary loves John"

BoW vectors are identical:

Word	john	loves	mary
Count	1	1	1

The Problem: Completely different relationships, but BOW treats them as identical!
Solution: Sequential processing captures **who does what to whom**.

Sequential Processing: How Order Saves the Day

Let's trace through: "The drug kills cancer cells effectively"

Sequential Processing Steps:

1. Read "The" → Article, something specific coming
2. Read "drug" → Subject identified: pharmaceutical agent
3. Read "kills" → Action: drug is the agent doing the killing
4. Read "cancer" → Target specification: what's being killed
5. Read "cells" → Target refinement: cancer cells specifically
6. Read "effectively" → Evaluation: the killing is successful

Key Insight: Sequential processing captures **who does what to whom**

- Agent: drug (good guy)
- Action: kills
- Target: cancer cells (bad guys)
- Result: Therapeutic success!

Sequential Model Training: The Setup

Core Training Objective: Predict next word given previous context

Training Example:

"Patient has diabetes and ___________"

Model Task:

- **Input:** "Patient has diabetes and"
- **Goal:** Predict probability distribution over next word
- **Possible completions:** "needs" (0.3), "requires" (0.2), "shows" (0.15), ...

Self-Supervised Learning: We can create millions of training examples from any text corpus!

Training Process: Step by Step

Training Sentence: "Patient has diabetes and requires insulin treatment"

Training Steps:

1. **Step 1:** "Patient" → predict "has"
2. **Step 2:** "Patient has" → predict "diabetes"
3. **Step 3:** "Patient has diabetes" → predict "and"
4. **Step 4:** "Patient has diabetes and" → predict "requires"
5. **Step 5:** "Patient has diabetes and requires" → predict "insulin"

Key Insight: One sentence provides multiple training examples!
Learning Process: Gradient descent updates model to minimize prediction errors

What Sequential Models Learn

Through Next-Word Prediction, Models Learn:

1. **Grammar and Syntax:**
   - "Patient has" (not "Patient have")
   - Verb agreement, word order, sentence structure
2. **Medical Domain Knowledge:**
   - "diabetes and hypertension" (common comorbidities)
   - "insulin injection" (treatment relationships)
3. **Context-Dependent Meanings:**
   - "acute" means different things in "acute pain" vs "acute care"
   - Model learns these contextual nuances automatically
4. **Long-Range Dependencies:**
   - "Patient with diabetes... [50 words later] ...needs glucose monitoring"

Current Approach Limitations

Text Processing Issues:

- **Sparsity:** Most features are zero
- **High dimensionality:** Vocabulary can be huge
- **Limited context:** N-grams only capture local patterns
- **Synonyms:** "MI" vs "heart attack" treated differently
- **Word order:** "patient improved" vs "patient not improved"

Biomedicine-Specific Text Challenges:

- **Context-dependent meanings**: "Positive" (good outcome vs test result)
- **Complex temporal relationships**: Treatment sequences, disease progression
- **Domain expertise required**: Clinical validation and interpretation
- **Abbreviations and negation**: Require specialized handling

The Path to Modern AI

This Week's Foundation:

- Optimization is central - gradient descent powers everything
- Neural networks are universal - can learn complex patterns
- Text needs special handling - converting language to numbers
- End-to-end learning - automatic feature discovery

Key Insight: Modern LLMs use the same core principles (gradient descent, backprop) but at massive scale with better architectures

Evolution to Modern Systems

What Changed:

- **Scale:** Billions of parameters vs thousands
- **Architecture:** Transformers vs simple MLPs
- **Training data:** Internet-scale vs small labeled sets
- **Compute:** Thousands of GPUs vs single machines

What Stayed the Same:

- Gradient descent optimization
- Backpropagation algorithm
- Numerical text representation
- Loss function minimization