Neural Networks

Robert Minneker

2026-01-29

Sources

Content derived from: J&M Ch. 6

Part 1: Foundations of Neural Networks

Neural networks emerged from decades of research on connectionist computation

- Early foundations: McCulloch & Pitts (1943), Hebb (1949), Turing (1948)
- Turing's insight: Complex behaviors emerge from many simple interacting units
- The connectionist paradigm: computation through distributed, parallel processing

1943 McCulloch-Pitts Neuron → **1958** Rosenblatt's Perceptron → **1986** Backpropagation → **2012** Deep Learning Revolution → **2017** Transformers

The perceptron was the first learnable neural architecture

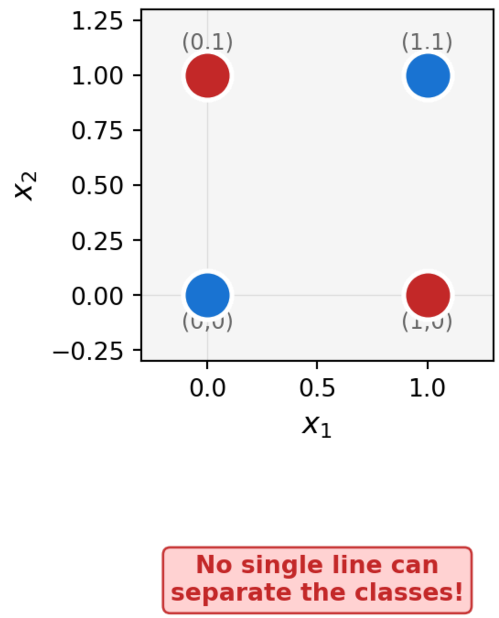
- Rosenblatt (1958) formalized learning from labeled examples
- Update rule adjusts weights based on prediction errors:

$$\mathbf{w} \leftarrow \mathbf{w} + \eta(y - \hat{y})\mathbf{x}$$

- Enabled learning of **linear decision boundaries** in input space

Perceptron Can Learn	Perceptron Cannot Learn
AND, OR, NOT gates	XOR gate
Linearly separable problems	Non-linearly separable

Why XOR breaks the perceptron



The XOR problem: no single line can separate the classes. Red points (output=1) sit on opposite corners.

Solution: Add a hidden layer to create a non-linear decision boundary.

Backpropagation enabled training of multi-layer networks

- Rumelhart, Hinton, and Williams (1986) introduced error backpropagation
- Uses the chain rule to compute gradients through layers:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}} \cdot \frac{\partial \mathbf{a}}{\partial \theta}$$

- Allowed training networks with **hidden layers**, overcoming perceptron limitations

Modern architectures build on these foundational principles

- Hierarchical representation learning extracts increasingly abstract features
- Non-linear function approximation enables modeling complex relationships
- Scalability through parallelization and specialized hardware (GPUs, TPUs)

Applications span:

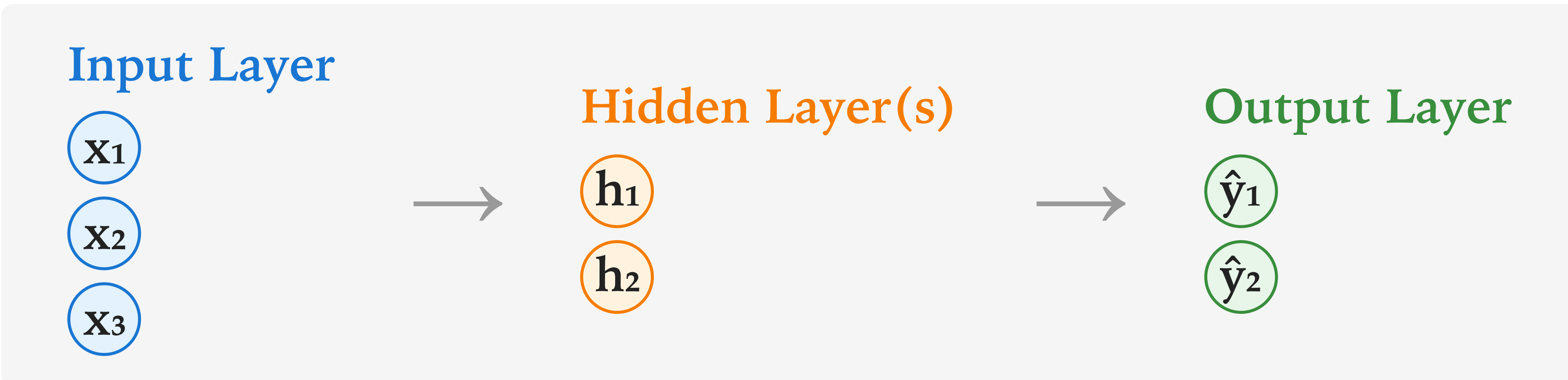
A neural network is a computational graph of interconnected processing units

- Each **neuron** computes a weighted sum of inputs plus bias, then applies an activation:

$$h = f(\mathbf{w}^\top \mathbf{x} + b)$$

- $\mathbf{w}$: weight vector (learned parameters)
- $\mathbf{x}$: input vector
- b: bias term
- f: nonlinear activation function

Networks are organized into layers with distinct roles



- **Input layer**: Receives raw features (e.g., word embeddings)
- **Hidden layers**: Learn intermediate representations
- **Output layer**: Produces predictions (often via softmax)

Feedforward networks process information in one direction only

- Data flows from input → hidden → output with **no cycles**
- Each layer's output becomes the next layer's input
- The entire computation is a composition of functions:

$$\hat{y} = f^{[L]}(f^{[L-1]}(\cdots f^{[1]}(\mathbf{x}) \cdots))$$

Recurrent networks can model sequential dependencies through cycles

- Allow information to persist across time steps
- Hidden state $\mathbf{h}_t$ encodes history of previous inputs
- Essential for language modeling, speech recognition, time series

Feedforward

Fixed-size input → Fixed-size output

Image classification, sentiment analysis

Recurrent

Sequence → Sequence or output

Language modeling, machine translation

Neural network computation is fundamentally linear algebra

- Inputs, parameters, and activations are vectors and matrices
- Core operation: matrix-vector multiplication plus bias

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

- Dimensions must align: if $\mathbf{x} \in \mathbb{R}^d$ and $\mathbf{W} \in \mathbb{R}^{m \times d}$, then $\mathbf{z} \in \mathbb{R}^m$

Dot products measure vector similarity

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\| \cos(\theta)$$

Similar (θ ≈ 0°)

→ →

`a · b > 0`

Vectors point same direction

Orthogonal (θ = 90°)

→ ↑

`a · b = 0`

Vectors are unrelated

Opposite (θ ≈ 180°)

→ ←

`a · b < 0`

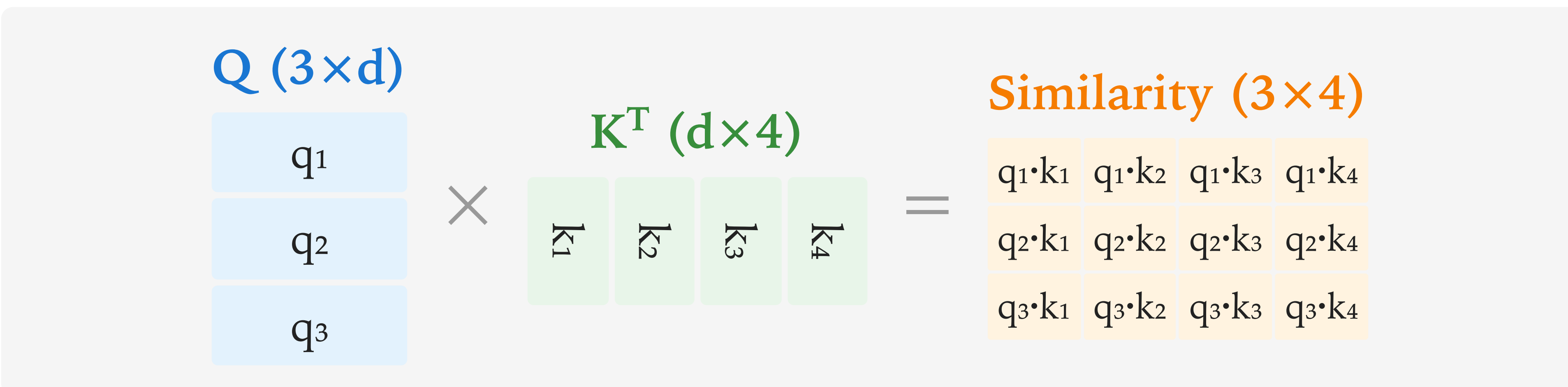
Vectors point opposite

Neural networks use dot products to measure relevance between vectors

Matrix multiplication computes all pairwise dot products

- If $\mathbf{Q} \in \mathbb{R}^{n \times d}$ and $\mathbf{K} \in \mathbb{R}^{m \times d}$, then:

$$(\mathbf{Q}\mathbf{K}^\top)_{ij} = \mathbf{q}_i \cdot \mathbf{k}_j$$



One matrix multiply computes all 12 similarities in parallel!

Activation functions introduce essential nonlinearity

After computing **z**, we apply an activation function elementwise:

$$\mathbf{h} = \phi(\mathbf{z})$$

ReLU

`max(0, z)`

Simple, fast, sparse

Sigmoid

$1/(1+e^{-z})$

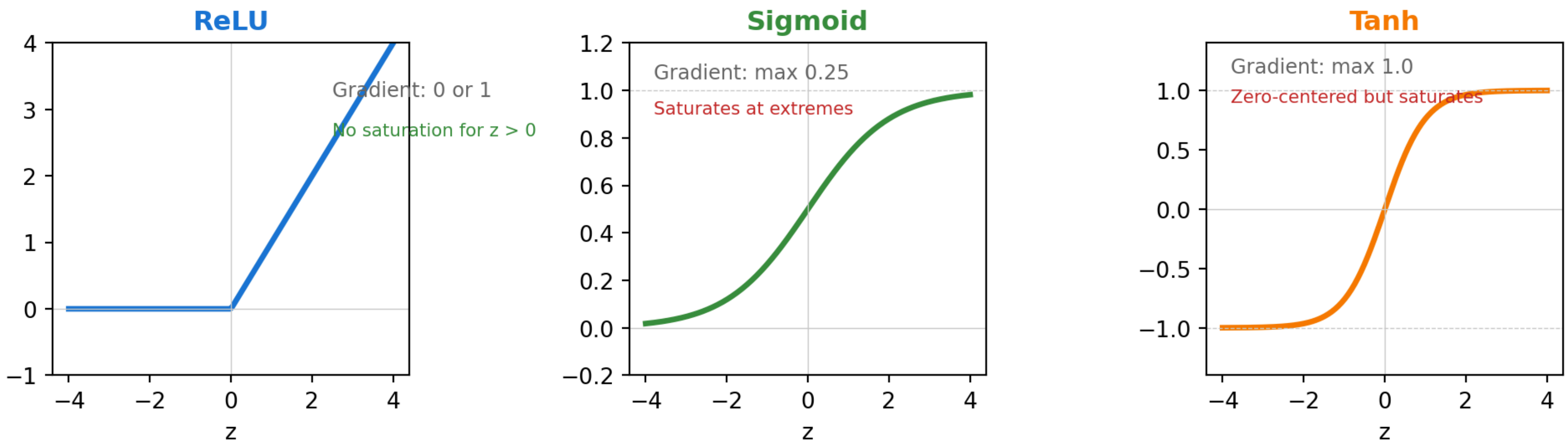
Output in (0,1)

Tanh

$(e^{z}-e^{-z})/(e^{z}+e^{-z})$

Output in (-1,1)

Activation function shapes determine gradient flow



Activation functions and their gradients. ReLU dominates modern networks due to its constant gradient for positive inputs.

Why ReLU dominates: Constant gradient (1) for positive inputs prevents vanishing gradients in deep networks.

The softmax function converts scores to probabilities

- Transforms a vector **z** into a probability distribution:

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{d} \exp(z_j)}$$

- Guarantees: $\sum_i \text{softmax}(z_i) = 1$ and $0 < \text{softmax}(z_i) < 1$

Part 2: Feedforward Neural Networks

Each layer transforms its input through weights, bias, and activation

For layer i:

$$\mathbf{x}^{[i]} = f^{[i]}\left(\mathbf{W}^{[i]}\mathbf{x}^{[i-1]} + \mathbf{b}^{[i]}\right)$$

- $\mathbf{W}^{[i]}$: learnable weight matrix
- $\mathbf{b}^{[i]}$: learnable bias vector
- $f^{[i]}$: activation function

$$\mathbf{x}^{(i-1)} \longrightarrow \mathbf{W}^{(i)}\mathbf{x} + \mathbf{b}^{(i)} \longrightarrow \mathbf{f}^{(i)}(\cdot) \longrightarrow \mathbf{x}^{(i)}$$

Activation functions serve different purposes in different layers

Location	Common Choice	Purpose
Hidden layers	ReLU, GELU	Introduce nonlinearity, sparse activation
Binary output	Sigmoid	Probability in [0,1]
Multi-class output	Softmax	Probability distribution
Regression output	None (linear)	Unbounded real values

The universal approximation theorem guarantees theoretical expressivity

Theorem: A feedforward network with one hidden layer and sufficient neurons can approximate any continuous function on a compact domain to arbitrary precision.

$$\forall \epsilon > 0, \exists \hat{f} : \sup_{x \in K} |f(x) - \hat{f}(x)| < \epsilon$$

But this doesn't mean shallow networks are always practical:

- May require exponentially many neurons
- Says nothing about learnability or generalization

Depth enables efficient representation of compositional structure

Shallow Network

$O(2^n)$

neurons for some functions

Deep Network

$O(n)$

neurons for same functions

- Telgarsky (2016): Some functions require exponentially more units in shallow vs. deep networks
- Depth captures hierarchical/compositional structure naturally
- Language has recursive, hierarchical structure → depth helps

Part 3: Learning and Training Algorithms

Supervised learning optimizes parameters using labeled examples

- Training data: pairs $(\mathbf{x}, y)$ of inputs and target outputs
- Model predicts: $\hat{y} = f_\theta(\mathbf{x})$
- Objective: minimize average loss over training set

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \ell\big(f_\theta(\mathbf{x}_i), y_i\big)$$

Loss functions measure prediction quality

Cross-Entropy (Classification)

$\ell = -\Sigma_k y_k \log \hat{y}_k$

Penalizes low probability on correct class

MSE (Regression)

$\ell = (y - \hat{y})^2$

Penalizes squared deviation from target

Cross-entropy loss for classification:

$$\ell_{\mathrm{CE}}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k$$

NLP applications of supervised learning span many tasks

Task	**Input x**	**Output y**
POS tagging	Word + context	POS tag (noun, verb, etc.)
Sentiment	Sentence/document	Sentiment class
NER	Word + context	Entity type or O
Text classification	Document	Topic/category

Backpropagation computes gradients efficiently via the chain rule

- We need $\frac{\partial L}{\partial W^{[l]}}$ for each layer to update weights
- Backprop propagates error signals backward through the network

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

where $\delta^{[l]} = \frac{\partial L}{\partial z^{[l]}}$ is the error signal at layer l.

Error signals propagate backward through the network

Forward Pass

Input x $\rightarrow$ Hidden h^1 $\rightarrow$ Hidden h^2 $\rightarrow$ **Output $\hat{y}$**

Backward Pass

$\partial L/\partial \hat{y}$ $\leftarrow$ δ^2 $\leftarrow$ δ^1 $\leftarrow$ **$\partial L/\partial x$**

The recursive error signal computation:

$$\delta^{[l]} = \left(W^{[l+1]}\right)^T \delta^{[l+1]} \odot \sigma'(z^{[l]})$$

Gradients drive parameter updates via gradient descent

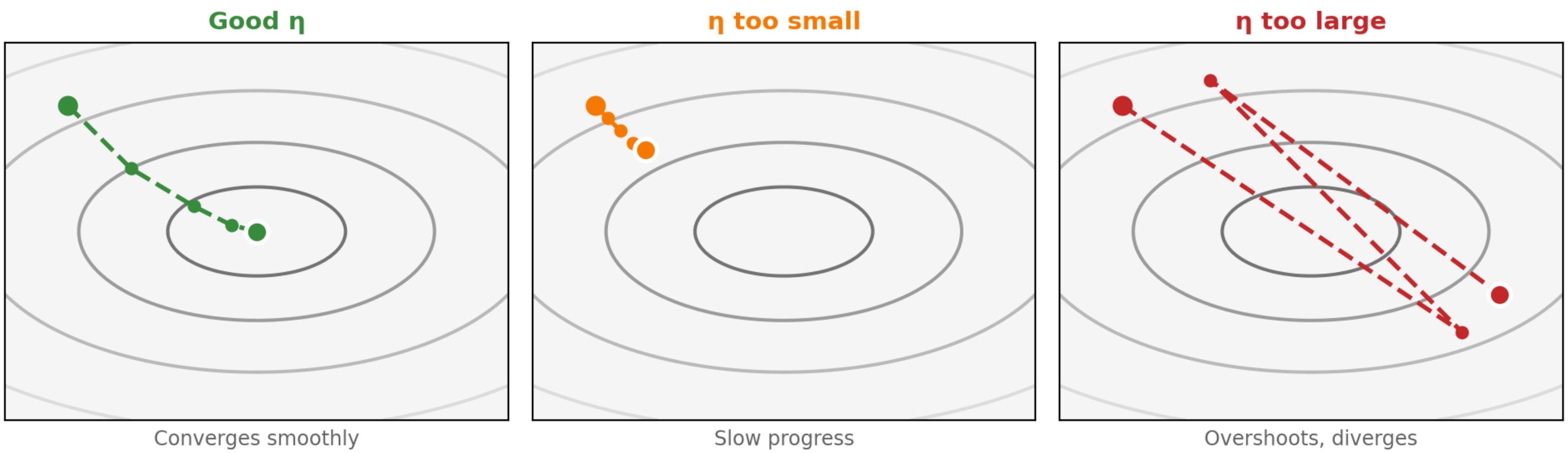
Weight update rule:

$$W^{[l]} \leftarrow W^{[l]} - \eta \frac{\partial L}{\partial W^{[l]}}$$

$$b^{[l]} \leftarrow b^{[l]} - \eta \frac{\partial L}{\partial b^{[l]}}$$

- η: learning rate (critical hyperparameter)
- Too high → unstable, may diverge; Too low → slow convergence

Gradient descent navigates the loss landscape



Gradient descent behavior with different learning rates. The learning rate η controls convergence speed and stability.

Gradient descent follows the steepest downhill direction; step size η determines how far we move each update.

Backpropagation is not an optimizer—it computes gradients for optimizers

Common misconception: Backprop = training algorithm

Reality:

- Backprop: efficient gradient computation method
- SGD/Adam/etc.: optimization algorithms that use gradients
- Together they enable training, but they're distinct concepts

Loss L $\longrightarrow$ **Backpropagation** $\longrightarrow$ **Gradients ∇L** $\longrightarrow$ **Optimizer (SGD, Adam)** $\longrightarrow$ **New Weights**

SGD processes mini-batches for efficient, noisy updates

Stochastic Gradient Descent:

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

- $\mathcal{B}_t$: random mini-batch at step t
- Noisy gradients can help escape local minima
- Much faster than full-batch gradient descent

Adam combines momentum and adaptive learning rates

Adam update equations:

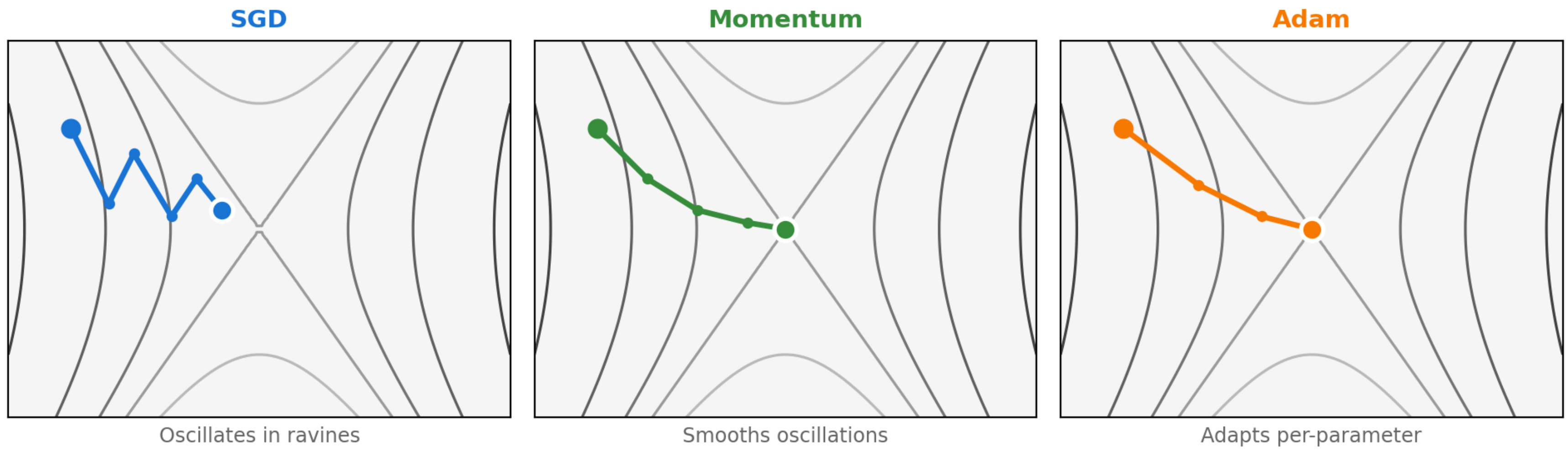
$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla_\theta L(\theta_t)$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla_\theta L(\theta_t))^2$$

$$\theta_{t+1} = \theta_t - \eta \frac{m_t}{\sqrt{v_t} + \epsilon}$$

- m_t: momentum (smoothed gradient)
- v_t: adaptive scaling (per-parameter)
- Default: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$

Optimizers behave differently on complex loss surfaces



Optimizer behavior on a saddle-like loss surface. SGD oscillates, Momentum smooths, Adam adapts step sizes per-parameter.

Adam combines momentum's smoothing with per-parameter learning rate scaling—faster and more robust convergence.

Regularization prevents overfitting by constraining model capacity

L2 Regularization

$L' = L + \lambda\|\theta\|^2$

Penalizes large weights

Dropout

$h_i = r_i \cdot h_i$

$r_i \sim \text{Bernoulli}(p)$

Early Stopping

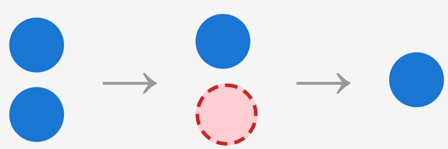
Stop when val↑

Monitor validation loss

Dropout forces the network to learn redundant representations

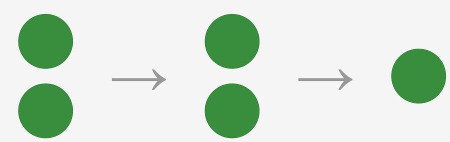
- During training: randomly set fraction p of activations to 0
- During inference: use all neurons, scale by $(1 - p)$
- Effect: no single neuron can become too important

Training (with dropout)



Some neurons "dropped"

Inference (no dropout)



All neurons active

Dropout signficantly reduces overfitting

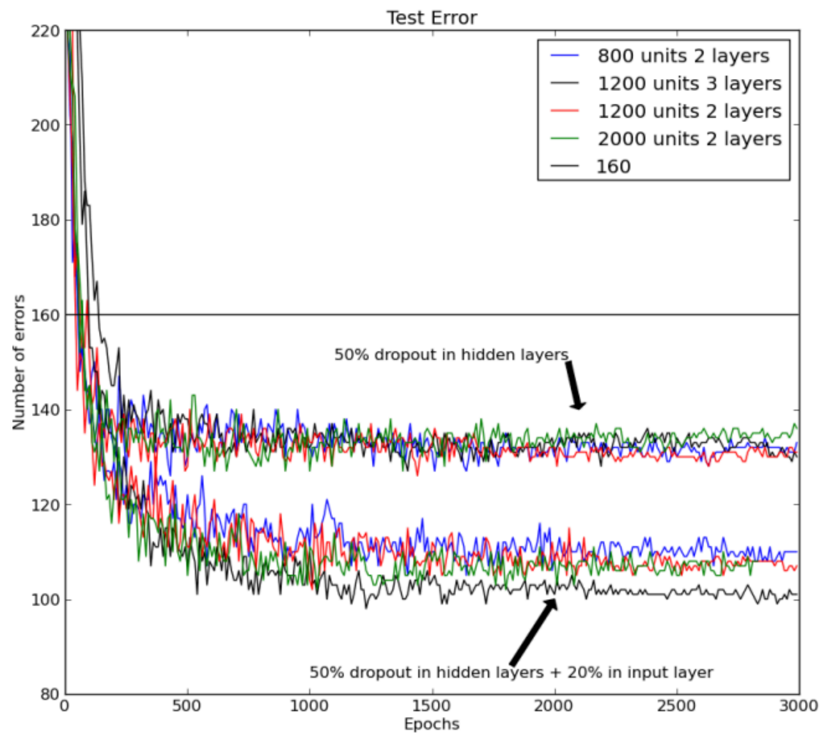


Fig. 1: The error rate on the MNIST test set for a variety of neural network architectures trained with backpropagation using 50% dropout for all hidden layers. The lower set of lines also use 20% dropout for the input layer. The best previously published result for this task using backpropagation without pre-training or weight-sharing or enhancements of the training set is shown as a horizontal line.

Improving neural networks by preventing co-adaptation of feature detectors (Hinton et al., 2012)

Part 4: Advanced Architectures and Extensions

Vanishing gradients make training deep networks difficult

- Gradients shrink exponentially when propagated through many layers
- For sigmoid: $\sigma'(z) \leq 0.25$, so gradient decays by at least $4\times$ per layer

$$\delta^{[l]} = (W^{[l+1]})^T \delta^{[l+1]} \odot \sigma'(z^{[l]})$$

- With 10 layers: gradient shrinks by $\approx 4^{10} \approx 10^6$

Exploding gradients cause training instability

- The opposite problem: gradients grow exponentially
- Occurs when $\|W^{[l]}\| > 1$ and compounds across layers
- Symptoms: NaN losses, weights becoming infinite

Solutions:

- Gradient clipping: $g \leftarrow g \cdot \frac{\tau}{\|g\|}$ if $\|g\| > \tau$
- Careful weight initialization
- Batch normalization

Proper initialization maintains gradient flow

Xavier/Glorot initialization:

$$\mathrm{Var}[W_{ij}] = \frac{2}{n_{in} + n_{out}}$$

He initialization (for ReLU):

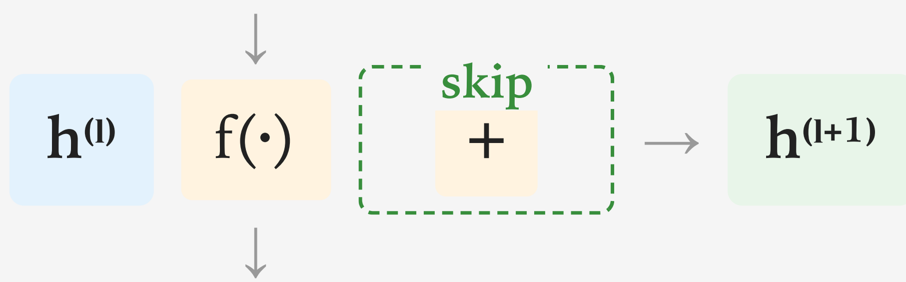
$$\mathrm{Var}[W_{ij}] = \frac{2}{n_{in}}$$

- Goal: keep activation and gradient variance stable across layers
- Critical for training networks with 10+ layers

Residual connections enable training of very deep networks

- **Key idea:** Add input directly to output via "skip connection"

$$\mathbf{h}^{[l+1]} = f(\mathbf{h}^{[l]}) + \mathbf{h}^{[l]}$$



Why it helps: Gradient flows directly through the skip connection:

$$\frac{\partial \mathbf{h}^{[l+1]}}{\partial \mathbf{h}^{[l]}} = \frac{\partial f}{\partial \mathbf{h}^{[l]}} + \mathbf{I}$$

Residual connections: empirical evidence

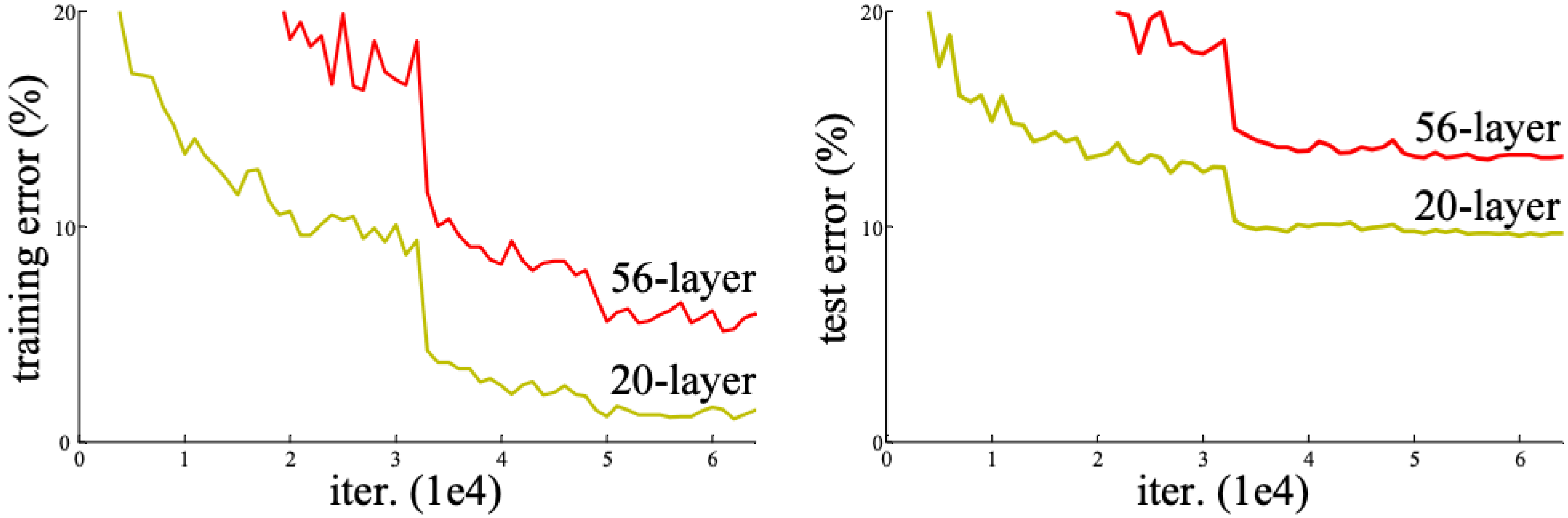


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

Training error on CIFAR-10: without residuals, a 56-layer "plain" network performs worse than a 20-layer network—even on training data. With residual connections (right), deeper networks consistently outperform shallower ones.

Figure from He et al. (2015), "Deep Residual Learning for Image Recognition"

Layer normalization stabilizes training of deep networks

$$\text{LayerNorm}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \mu}{\sigma + \epsilon} + \beta$$

- μ, σ: mean and std computed **across features** (per example)
- γ, β: learnable scale and shift parameters

Batch Normalization

Normalize across **batch** dimension

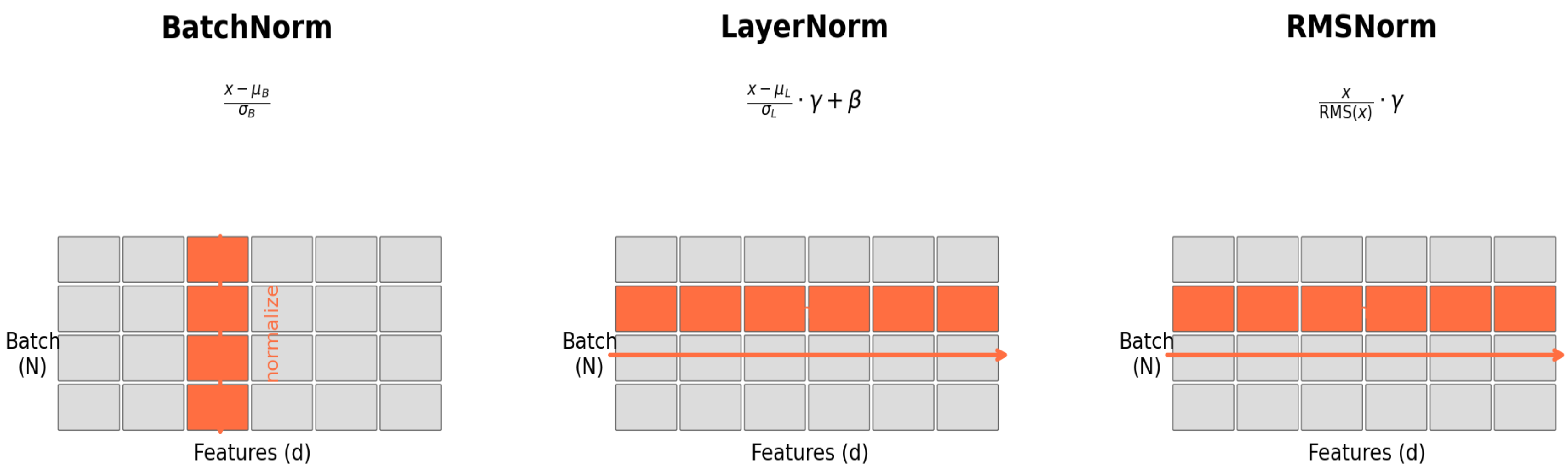
- Depends on batch statistics
- Different behavior train vs test
- Problematic for variable-length sequences

Layer Normalization

Normalize across **feature** dimension

- Independent of batch size
- Same behavior train and test
- Works with any sequence length ✓

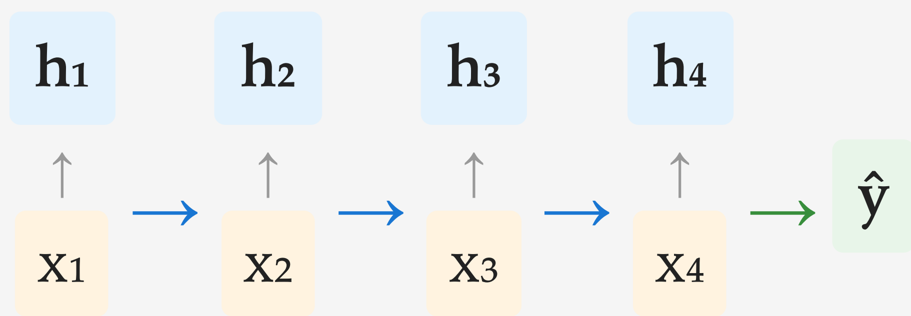
Comparing normalization techniques: BatchNorm vs LayerNorm vs RMSNorm



	BatchNorm	LayerNorm	RMSNorm
Normalizes	Batch dim	Feature dim	Feature dim
Train/Test	Different	Same	Same
Used in	CNNs	Transformers	LLaMA, Gemma

RNNs model sequences by maintaining state across time steps

$$\mathbf{h}_t = f(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h)$$



- Hidden state $\mathbf{h}_t$ encodes information from $x_1, \ldots, x_t$
- Same parameters ($\mathbf{W}_{xh}, \mathbf{W}_{hh}$) used at every time step

The RNN sequential bottleneck limits parallelization

- $\mathbf{h}_t$ depends on $\mathbf{h}_{t-1}$ creates a **dependency chain**
- Cannot compute $\mathbf{h}_5$ until $\mathbf{h}_1, \mathbf{h}_2, \mathbf{h}_3, \mathbf{h}_4$ are finished

$\mathbf{h_1}$ WAIT $\longrightarrow$ $\mathbf{h_2}$ WAIT $\longrightarrow$ $\mathbf{h_3}$ WAIT $\longrightarrow$ $\mathbf{h_4}$ WAIT $\longrightarrow$ $\mathbf{h_5}$

O(T) sequential steps for sequence length T

Implications:

- Cannot fully utilize parallel hardware (GPUs have thousands of cores)
- Training time scales linearly with sequence length
- **Key question:** What if we could process all tokens at once?

Autoregressive vs bidirectional processing

Autoregressive (left-to-right): Each position only sees the past

$$P(x_1, x_2, \ldots, x_n) = \prod_{t=1}^{n} P(x_t | x_1, \ldots, x_{t-1})$$

Bidirectional: Each position sees the entire sequence

Autoregressive (GPT)

The | cat | **sat** | ? | ?

Position 3 sees only positions 1-2

Bidirectional (BERT)

The | cat | **[MASK]** | on | mat

Position 3 sees all positions

Standard RNNs struggle with long-range dependencies

"The **cat**, which was sitting on the mat in the living room where my grandmother used to read her favorite novels during the winter evenings, **was** hungry."

The verb "was" must agree with "cat" despite 20+ intervening words.

- Gradient signal degrades over long distances
- LSTM and GRU use **gating mechanisms** to preserve information
- Transformers use **attention** to directly connect distant positions

Context as a weighted combination of representations

- What if we could directly access all previous representations?
- Key insight: compute a **weighted sum** based on relevance

$$\text{output}_t = \sum_{j=1}^{t} \alpha_{tj} \cdot \mathbf{v}_j$$

- α_{tj}: attention weight (how relevant is position j to position t?)
- $\mathbf{v}_j$: value vector at position j

$\mathbf{v}_1$ $\mathbf{v}_2$ $\mathbf{v}_3$ $\longrightarrow$ $\mathbf{output}_4$

$\alpha_{41} = 0.1$ $\alpha_{42} = 0.5$ $\alpha_{43} = 0.3$ $\Sigma\, \alpha_i \cdot v_i$

Attention weights reveal what the model focuses on

Attention from "it" in: "The cat sat on the mat because it was tired"

	The	cat	sat	on	the	mat	because	it	was
it →	0.05	0.45	0.10	0.03	0.02	0.15	0.05	0.10	0.05

What attention reveals:

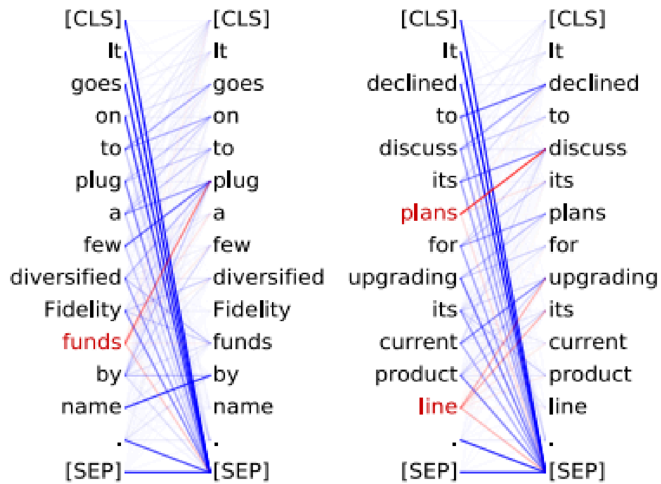
The pronoun "it" attends most strongly to "cat" (0.45)—the model has learned coreference!

Darker = higher attention weight

BERT attention learns coreference resolution

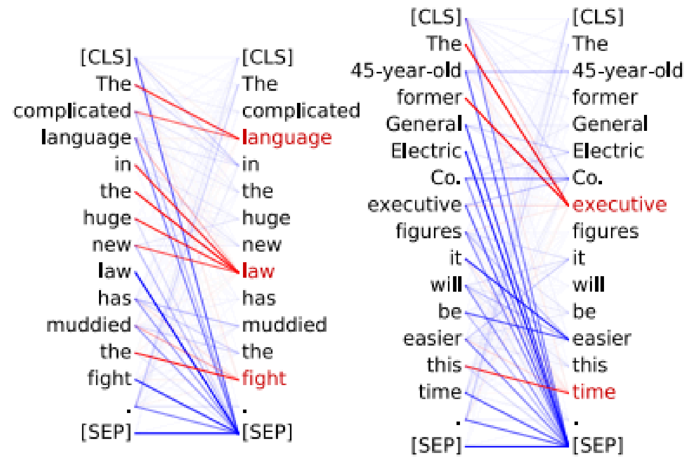
Head 8-10

- **Direct objects** attend to their verbs
- 86.8% accuracy at the `dobj` relation



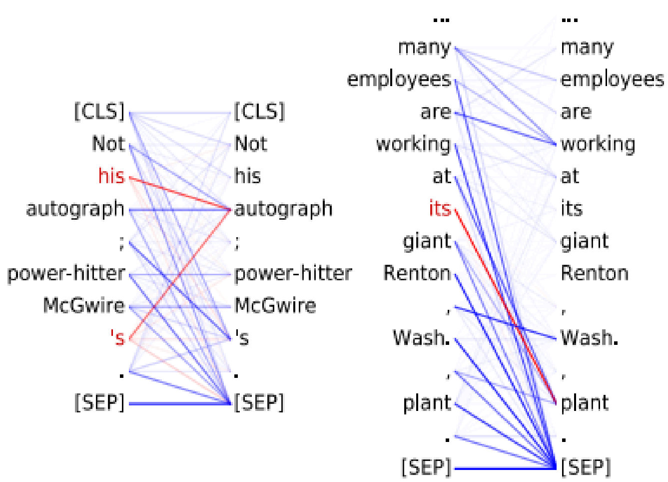
Head 8-11

- **Noun modifiers** (e.g., determiners) attend to their noun
- 94.3% accuracy at the `det` relation



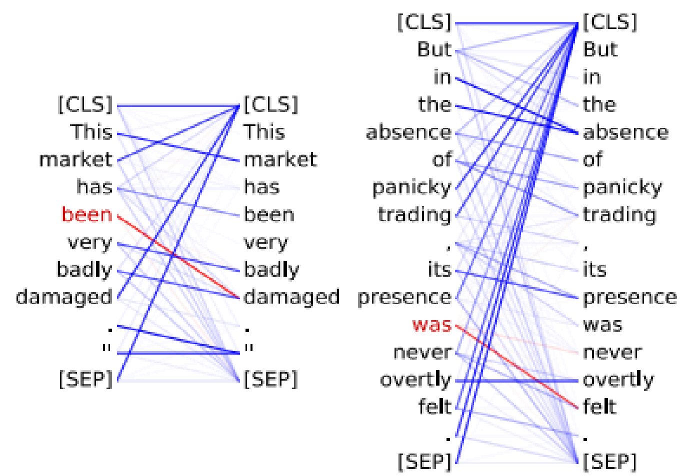
Head 7-6

- **Possessive pronouns** and apostrophes attend to the head of the corresponding NP
- 80.5% accuracy at the `poss` relation



Head 4-10

- **Passive auxiliary verbs** attend to the verb they modify
- 82.5% accuracy at the `auxpass` relation



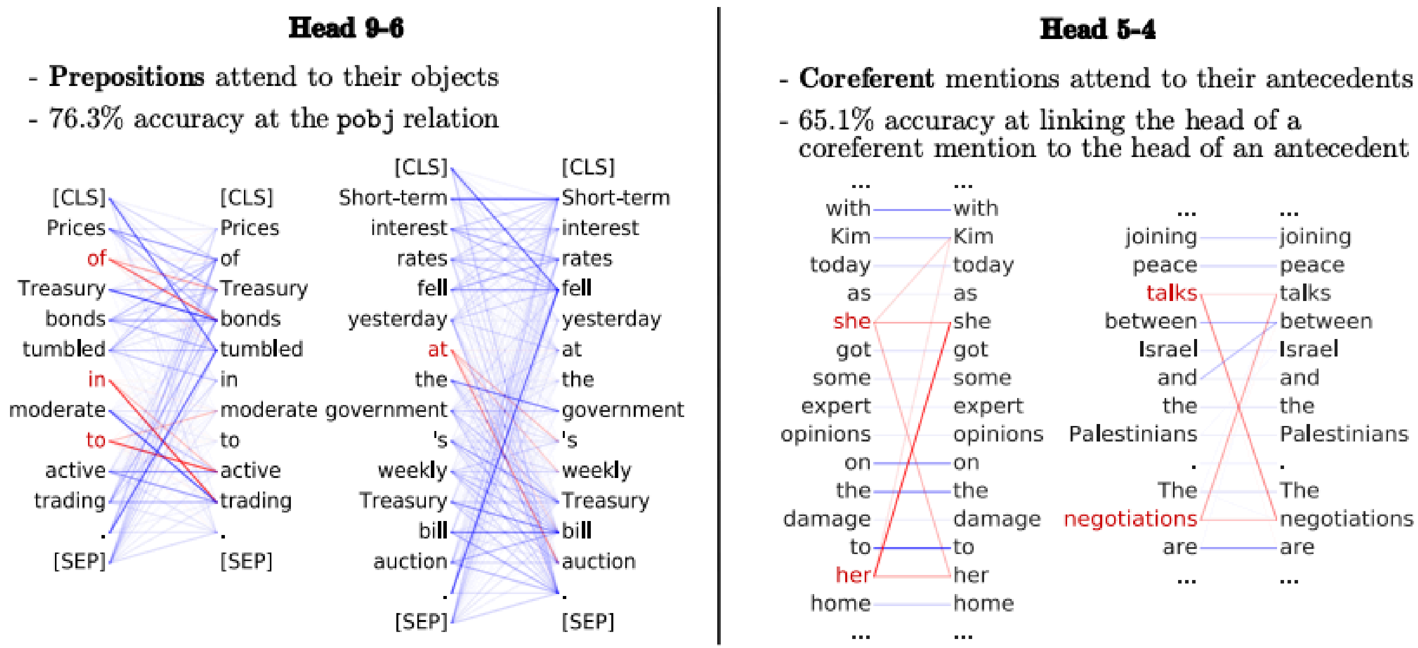


Figure 5: BERT attention heads that correspond to linguistic phenomena. In the example attention maps, the darkness of a line indicates the strength of the attention weight. All attention to/from red words is colored red; these colors are there to highlight certain parts of the attention heads' behaviors. For Head 9-6, we don't show attention to [SEP] for clarity. Despite not being explicitly trained on these tasks, BERT's attention heads perform remarkably well, illustrating how syntax-sensitive behavior can emerge from self-supervised training alone.

Attention patterns from BERT showing coreference resolution. The model learns to link pronouns to their antecedents — here "it" attends strongly to "cat" and "dog" to resolve what each pronoun refers to.

Figure from Clark et al. (2019), "What Does BERT Look At? An Analysis of BERT's Attention"

Position information must be explicitly encoded

- Without recurrence, order information is lost
- “Dog bit man” and “man bit dog” would be identical!

Without Position Info

{dog, bit, man} = {man, bit, dog}

Bag of words—order lost!

With Position Encoding

(dog, 1), (bit, 2), (man, 3)

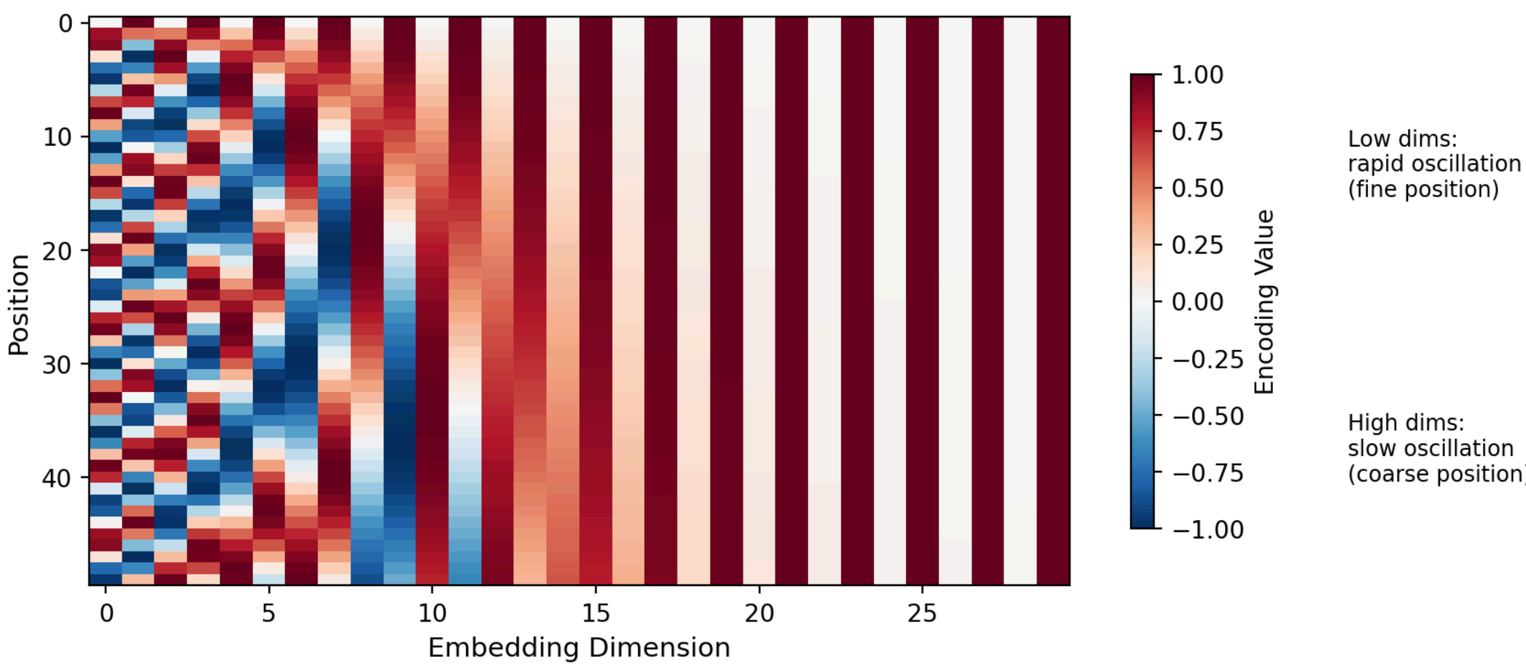
Order preserved via position

Key question: How do we inject position information into parallel architectures?

Sinusoidal position encodings (Vaswani et al., 2017)

Add a position-dependent vector to each token embedding:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right) \quad PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$



Sinusoidal position encodings: each position gets a unique pattern. Low dimensions oscillate rapidly (fine position), high dimensions oscillate slowly (coarse position).

Key properties: Each position gets a unique encoding; relative positions computable via linear transformation; generalizes to longer sequences.

Part 5: Applications

Neural networks power core NLP tasks

Task	Architecture	Output
Text classification	Feedforward / CNN / Transformer	Class probabilities (softmax)
Language modeling	RNN / Transformer	Next token probabilities
Sequence labeling	BiLSTM / Transformer	Tag per token
Machine translation	Encoder-decoder	Target sequence

Text classification assigns labels to documents

Document 'Great movie!' → **Embedding** Layer → **Encoder** (CNN/LSTM/Transformer) →

Pooling → Dense → Softmax → **P(pos)=0.92** P(neg)=0.08

Applications: Sentiment analysis, spam detection, topic classification

Language modeling predicts the next token in a sequence

$$P(w_t|w_1,\ldots,w_{t-1}) = \mathrm{softmax}(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{b})$$

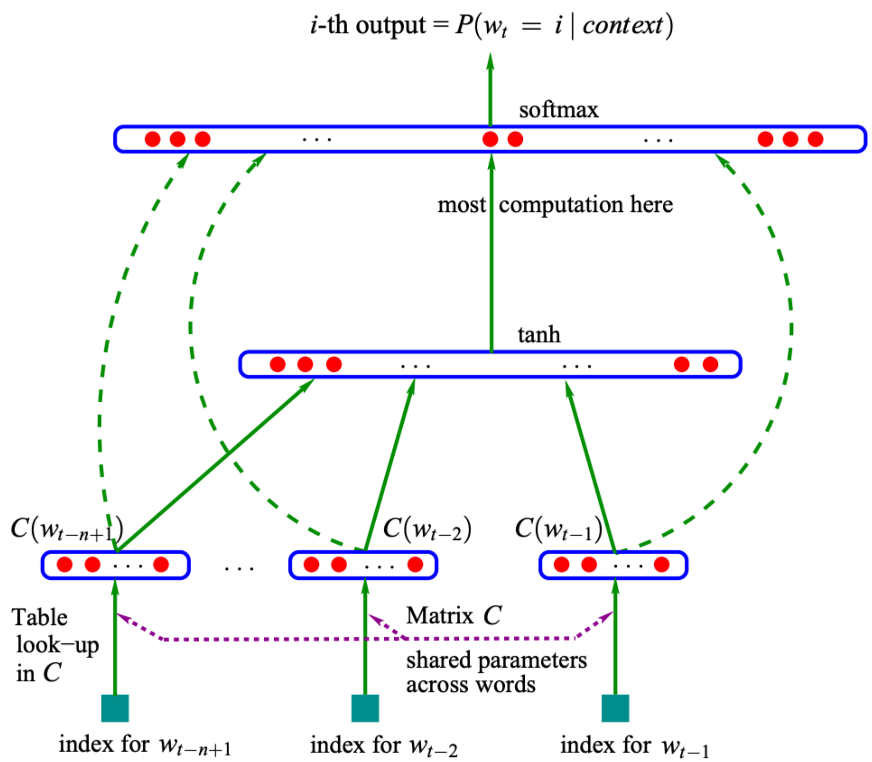


Figure 1: Neural architecture: $f(i, w_{t-1}, \cdots, w_{t-n+1}) = g(i, C(w_{t-1}), \cdots, C(w_{t-n+1}))$ where g is the neural network and $C(i)$ is the i-th word feature vector.

Neural networks excel across diverse domains

Computer Vision

CNNs for image classification, object detection, segmentation

Reinforcement Learning

Deep Q-networks, policy gradients for game playing, robotics

Bioinformatics

Protein structure prediction (AlphaFold), genomics

Recommendations

Neural collaborative filtering, content-based systems

- Same fundamental principles (backprop, gradient descent) apply
- Architecture choices encode domain-specific inductive biases

Part 6: Further Reading and Historical Notes

Key references for deeper understanding

Textbooks:

- Goodfellow, Bengio, & Courville (2016), *Deep Learning* — comprehensive theory and practice
- Jurafsky & Martin, *Speech and Language Processing* Ch. 6-8

Key milestones in neural network history

Foundations

1943 — McCulloch-Pitts neuron
1958 — Perceptron (Rosenblatt)
1969 — Perceptrons book → AI Winter

Revival

1986 — Backpropagation
1990 — Elman RNNs
1997 — LSTM (Hochreiter & Schmidhuber)

Modern Era

2012 — AlexNet → Deep learning revolution
2017 — Transformer architecture
2018 — BERT, GPT

Summary: Neural Networks - Key Takeaways

- **Architecture**: Networks of neurons organized in layers; feedforward vs. recurrent
- **Learning**: Supervised learning minimizes loss via gradient descent
- **Backpropagation**: Efficient gradient computation using the chain rule
- **Optimization**: SGD, Adam; regularization (dropout, L2) prevents overfitting
- **Challenges**: Vanishing/exploding gradients addressed by careful design
- **Applications**: Text classification, language modeling, and beyond

Questions?

Coming up next: Transformers and attention mechanisms

Resources:

- Goodfellow et al. *Deep Learning* (free online)
- 3Blue1Brown neural network videos (visual intuition)
- PyTorch tutorials (hands-on practice)