

CS 489: Introduction to Natural Language Processing

Lecture 6: Neural Network Foundations

Instructor: Freda Shi

September 29, 2026



Outline

1. Recap: neural classifiers (Lecture 4) and N -gram LMs (Lecture 5)
2. New: backpropagation in detail (computation graphs and gradients)
3. New: RNNs for sequences and backpropagation through time
4. New: vanishing gradients, LSTMs, and GRUs
5. Practical considerations for recurrent models

Recap: Where We Are

Already covered:

- **Lectures 1–3:** Language models and information theory, tokenization, and word embeddings.
- **Lecture 4:** Supervised classification, MLPs, nonlinear activations, softmax, cross-entropy, and the idea of backpropagation with SGD.
- **Lecture 5:** N -gram LMs, the Markov assumption, counting and smoothing, and perplexity.

New today: Derive **backpropagation** in detail, then use it to train **sequence models** (RNNs, LSTMs, GRUs).

Motivation: Why Go Neural?

Recall the limitations of N -gram language models (Lecture 5):

- **Bounded context:** Only look at the last $N-1$ words.
- **Limited generalization:** Similar words do not automatically share their N -gram counts.
- **Sparsity:** Possible N -grams grow exponentially with N .

Motivation: Why Go Neural?

Recall the limitations of N -gram language models (Lecture 5):

- **Bounded context:** Only look at the last $N-1$ words.
- **Limited generalization:** Similar words do not automatically share their N -gram counts.
- **Sparsity:** Possible N -grams grow exponentially with N .

Neural approaches can help:

- **Continuous representations:** The word embeddings from Lecture 3 let similar words share statistical strength.
- **Parameter sharing:** RNNs apply the same weight matrices across positions.
- **Longer context** (in principle): RNNs can condition on the entire history, though learning long-range dependencies is difficult.

Recap: Neural Text Classifiers (Lecture 4)

Previously: Average word embeddings for a fixed-size sentence representation, then apply an MLP and softmax:

$$\begin{aligned}\mathbf{x} &= \frac{1}{k} \sum_{i=1}^k \text{emb}(w_i), \\ \mathbf{z} &= g(\mathbf{W}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}), \\ \mathbf{p} &= \text{softmax}(\mathbf{W}^{(1)}\mathbf{z} + \mathbf{b}^{(1)}).\end{aligned}$$

Also covered: Nonlinear activations, cross-entropy loss, and the high-level idea of backpropagation with SGD.

Recap: Neural Text Classifiers (Lecture 4)

Previously: Average word embeddings for a fixed-size sentence representation, then apply an MLP and softmax:

$$\begin{aligned}\mathbf{x} &= \frac{1}{k} \sum_{i=1}^k \text{emb}(w_i), \\ \mathbf{z} &= g(\mathbf{W}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}), \\ \mathbf{p} &= \text{softmax}(\mathbf{W}^{(1)}\mathbf{z} + \mathbf{b}^{(1)}).\end{aligned}$$

Also covered: Nonlinear activations, cross-entropy loss, and the high-level idea of backpropagation with SGD.

Today: Work out the gradients, then replace order-insensitive pooling with a recurrent sequence model.

Recap: Training Neural Networks (Lecture 4)

Goal: Find parameters $\theta = \{\mathbf{W}^{(\ell)}, \mathbf{b}^{(\ell)}\}_\ell$ that minimize a loss function $\mathcal{L}$.

Recipe:

1. **Forward pass:** Compute the prediction $\hat{\mathbf{y}}$ and the loss $\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$.
2. **Backward pass:** Compute the gradient $\nabla_{\theta}\mathcal{L}$ using the chain rule.
3. **Update:** $\theta \leftarrow \theta - \eta \nabla_{\theta}\mathcal{L}$ (SGD).
4. Repeat.

The backward pass is called **backpropagation** (Rumelhart et al., 1986).

New today: Work through those gradients on a computation graph, then backpropagate through an unrolled RNN.

Computation Graphs

A **computation graph** represents the forward computation as a directed acyclic graph.

Example: Binary cross-entropy ($y \in \{0, 1\}$).

$$\mathcal{L} = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})]$$

where $\hat{y} = \sigma(s)$ and $s = \mathbf{w}^\top \mathbf{x} + b$.

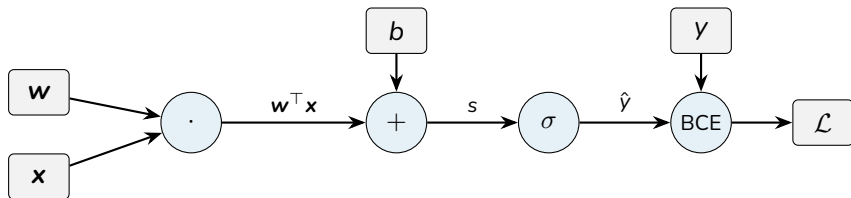
Computation Graphs

A **computation graph** represents the forward computation as a directed acyclic graph.

Example: Binary cross-entropy ($y \in \{0, 1\}$).

$$\mathcal{L} = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})]$$

where $\hat{y} = \sigma(s)$ and $s = \mathbf{w}^\top \mathbf{x} + b$.



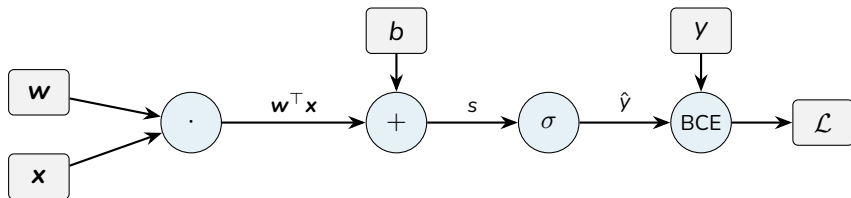
Computation Graphs

A **computation graph** represents the forward computation as a directed acyclic graph.

Example: Binary cross-entropy ($y \in \{0, 1\}$).

$$\mathcal{L} = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})]$$

where $\hat{y} = \sigma(s)$ and $s = \mathbf{w}^\top \mathbf{x} + b$.



Backprop traverses this graph *in reverse*, computing $\frac{\partial \mathcal{L}}{\partial \text{node}}$ at each step using the chain rule.

Backpropagation: Chain Rule

Chain rule: If $\mathcal{L} = f(g(h(\theta)))$, then:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial h} \cdot \frac{\partial h}{\partial \theta}$$

Backpropagation: Chain Rule

Chain rule: If $\mathcal{L} = f(g(h(\theta)))$, then:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial h} \cdot \frac{\partial h}{\partial \theta}$$

Concrete example: 2-layer network for binary classification.

$$\mathbf{a}^{(1)} = \mathbf{W}^{(0)} \mathbf{x} + \mathbf{b}^{(0)} \quad \text{(pre-activation)}$$

$$\mathbf{z}^{(1)} = \sigma(\mathbf{a}^{(1)}) \quad \text{(activation)}$$

$$\mathbf{a}^{(2)} = \mathbf{w}^{(1)\top} \mathbf{z}^{(1)} + b^{(1)}$$

$$\hat{y} = \sigma(a^{(2)})$$

$$\mathcal{L} = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})] \quad \text{(binary cross-entropy)}$$

Backpropagation: Chain Rule

Chain rule: If $\mathcal{L} = f(g(h(\theta)))$, then:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial h} \cdot \frac{\partial h}{\partial \theta}$$

Concrete example: 2-layer network for binary classification.

$$\mathbf{a}^{(1)} = \mathbf{W}^{(0)} \mathbf{x} + \mathbf{b}^{(0)} \quad \text{(pre-activation)}$$

$$\mathbf{z}^{(1)} = \sigma(\mathbf{a}^{(1)}) \quad \text{(activation)}$$

$$\mathbf{a}^{(2)} = \mathbf{w}^{(1)\top} \mathbf{z}^{(1)} + b^{(1)}$$

$$\hat{y} = \sigma(a^{(2)})$$

$$\mathcal{L} = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})] \quad \text{(binary cross-entropy)}$$

Backprop computes: $\frac{\partial \mathcal{L}}{\partial \hat{y}} \rightarrow \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(2)}} \rightarrow \frac{\partial \mathcal{L}}{\partial \mathbf{w}^{(1)}} \rightarrow \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(1)}} \rightarrow \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(1)}} \rightarrow \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(0)}}$

Backpropagation: Worked Gradients

Starting from $\mathcal{L} = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})]$, $\hat{y} = \sigma(a^{(2)})$:

Step 1: Output gradient.

$$\frac{\partial \mathcal{L}}{\partial a^{(2)}} = \hat{y} - y$$

(A clean result from combining cross-entropy + sigmoid.)

Backpropagation: Worked Gradients

Starting from $\mathcal{L} = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})]$, $\hat{y} = \sigma(a^{(2)})$:

Step 1: Output gradient.

$$\frac{\partial \mathcal{L}}{\partial a^{(2)}} = \hat{y} - y$$

(A clean result from combining cross-entropy + sigmoid.)

Step 2: Gradient w.r.t. second-layer weights.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}^{(1)}} = \frac{\partial \mathcal{L}}{\partial a^{(2)}} \cdot \mathbf{z}^{(1)} = (\hat{y} - y) \mathbf{z}^{(1)}$$

Backpropagation: Worked Gradients (continued)

Step 3: Propagate to hidden layer.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(2)}} \cdot \mathbf{w}^{(1)} = (\hat{y} - y) \mathbf{w}^{(1)}$$

Backpropagation: Worked Gradients (continued)

Step 3: Propagate to hidden layer.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(2)}} \cdot \mathbf{w}^{(1)} = (\hat{y} - y) \mathbf{w}^{(1)}$$

Step 4: Through activation and to first-layer weights.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(0)}} = \underbrace{\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(1)}} \odot \sigma'(\mathbf{a}^{(1)})}_{\delta^{(1)}} \cdot \mathbf{x}^\top$$

Backpropagation: Key Takeaways

1. It is just the chain rule.

- Each node in the computation graph computes a local gradient.
- Multiply local gradients along the path from loss to parameter.

Backpropagation: Key Takeaways

1. It is just the chain rule.

- Each node in the computation graph computes a local gradient.
- Multiply local gradients along the path from loss to parameter.

2. **Efficiency:** Backprop computes *all* gradients in $O(\text{forward pass})$ time.

- Without backprop: would need one forward pass *per parameter* (finite differences).
- With backprop: one forward + one backward = all gradients.

Backpropagation: Key Takeaways

1. It is just the chain rule.

- Each node in the computation graph computes a local gradient.
- Multiply local gradients along the path from loss to parameter.

2. **Efficiency:** Backprop computes *all* gradients in $O(\text{forward pass})$ time.

- Without backprop: would need one forward pass *per parameter* (finite differences).
- With backprop: one forward + one backward = all gradients.

3. **Automatic differentiation:**

- Modern frameworks (PyTorch, JAX) build the computation graph automatically.
- You define the forward pass; gradients are computed for free.
- `loss.backward()` does exactly what we described.

From Fixed-Size Classifiers to Sequence Models

Consider processing a sentence token by token.

Feedforward (MLP) approach:

- Average embeddings → lose word order.
- Concatenate embeddings → fixed-length input required.
- Cannot handle variable-length sequences gracefully.

From Fixed-Size Classifiers to Sequence Models

Consider processing a sentence token by token.

Feedforward (MLP) approach:

- Average embeddings $\rightarrow$ lose word order.
- Concatenate embeddings $\rightarrow$ fixed-length input required.
- Cannot handle variable-length sequences gracefully.

What we want:

- Process **one token at a time**, in order.
- Maintain a **running summary** (hidden state) of what we have seen so far.
- The summary at position t depends on **all previous tokens** $x_1, \dots, x_t$.

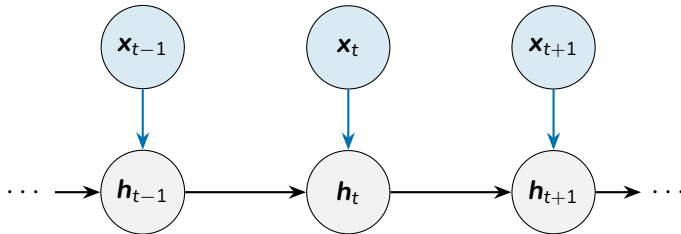
$\Rightarrow$ **Recurrent Neural Networks** (Elman, 1990).

Recurrent Neural Networks (RNNs)

Idea: Apply the **same** transformation at each time step, feeding in the current input and the previous hidden state.

$$h_t = g(W[x_t; h_{t-1}] + b)$$

where $[x_t; h_{t-1}]$ denotes concatenation, and g is typically $\tanh$.

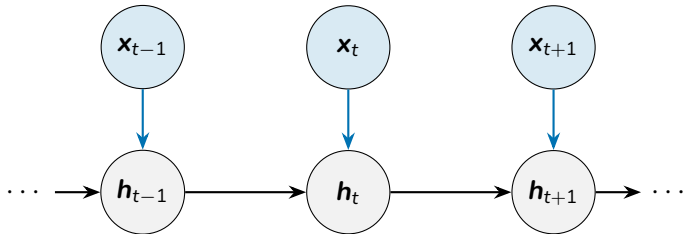


Recurrent Neural Networks (RNNs)

Idea: Apply the **same** transformation at each time step, feeding in the current input and the previous hidden state.

$$h_t = g(W[x_t; h_{t-1}] + b)$$

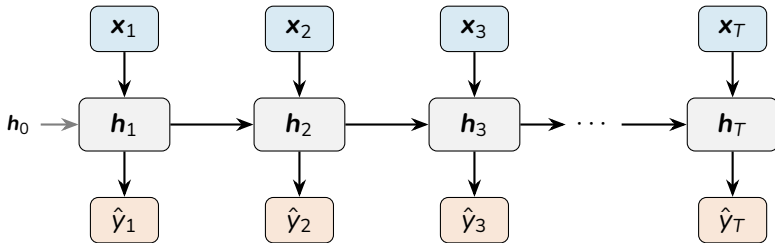
where $[x_t; h_{t-1}]$ denotes concatenation, and g is typically $\tanh$.



Key: W and b are **shared across all time steps** (parameter sharing).

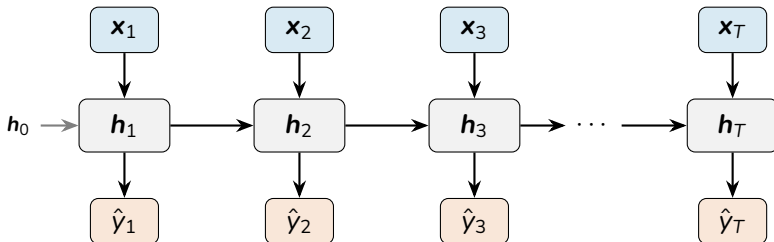
RNN: Unrolled View

The recurrence $\mathbf{h}_t = g(\mathbf{W}[\mathbf{x}_t; \mathbf{h}_{t-1}] + \mathbf{b})$ can be **unrolled** over time:



RNN: Unrolled View

The recurrence $\mathbf{h}_t = g(\mathbf{W}[\mathbf{x}_t; \mathbf{h}_{t-1}] + \mathbf{b})$ can be **unrolled** over time:



- The unrolled RNN looks like a **very deep feedforward network** with **shared weights**.
- At each step, we can optionally produce an output $\hat{\mathbf{y}}_t$ (e.g., for language modeling: predict next token).
- Or use only $\mathbf{h}_T$ for a final prediction (e.g., for classification).

RNNs for Language Modeling

An RNN language model predicts a distribution over the next token at each step:

$$P(\cdot \mid x_1, \dots, x_t) = \text{softmax}(\mathbf{W}_o \mathbf{h}_t + \mathbf{b}_o)$$

Advantages over N -gram models:

- **Unbounded context:** $\mathbf{h}_t$ can encode the entire history in principle.
- **Shared representations:** Similar words produce similar hidden states.
- **Recurrent weights:** $O(d^2)$ parameters, independent of sequence length.

RNNs for Language Modeling

An RNN language model predicts a distribution over the next token at each step:

$$P(\cdot \mid x_1, \dots, x_t) = \text{softmax}(\mathbf{W}_o \mathbf{h}_t + \mathbf{b}_o)$$

Advantages over N -gram models:

- **Unbounded context:** $\mathbf{h}_t$ can encode the entire history in principle.
- **Shared representations:** Similar words produce similar hidden states.
- **Recurrent weights:** $O(d^2)$ parameters, independent of sequence length.

Training: **Teacher forcing:** at each step, feed the *true* previous token (not the model's own prediction) as input.

Loss: Sum of cross-entropy losses at each time step:

$$\mathcal{L} = - \sum_{t=1}^T \log P(x_{t+1} \mid x_1, \dots, x_t)$$

Backpropagation Through Time (BPTT)

Since $\mathbf{W}$ is shared across all time steps, its gradient accumulates contributions from **every** time step:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \sum_{t=1}^T \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}}$$

Backpropagation Through Time (BPTT)

Since $\mathbf{W}$ is shared across all time steps, its gradient accumulates contributions from **every** time step:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \sum_{t=1}^T \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}}$$

And the gradient flowing to $\mathbf{h}_t$ depends on **future** time steps:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} = \frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t} + \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{t+1}} \cdot \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t}$$

where $\mathcal{L}_t$ is the loss at time step t .

Intuition: Unrolling the RNN creates a *very deep* network. Backprop through this is called **Backpropagation Through Time (BPTT)**.

The Vanishing Gradient Problem

Consider the gradient from step k back to step t ($t < k$):

$$\frac{\partial \mathbf{h}_k}{\partial \mathbf{h}_t} = \prod_{i=t+1}^k \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}} = \prod_{i=t+1}^k \text{diag}(g'(\mathbf{a}_i)) \mathbf{W}_h$$

where $\mathbf{W}_h$ is the hidden-to-hidden weight submatrix and $\mathbf{a}_i$ are pre-activations.

The Vanishing Gradient Problem

Consider the gradient from step k back to step t ($t < k$):

$$\frac{\partial \mathbf{h}_k}{\partial \mathbf{h}_t} = \prod_{i=t+1}^k \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}} = \prod_{i=t+1}^k \text{diag}(g'(\mathbf{a}_i)) \mathbf{W}_h$$

where $\mathbf{W}_h$ is the hidden-to-hidden weight submatrix and $\mathbf{a}_i$ are pre-activations.

Problem: This is a **product of $k - t$ matrices**.

- If the largest singular value of $\text{diag}(g'(\mathbf{a}_i))\mathbf{W}_h < 1$: gradients **vanish** exponentially.
- If the largest singular value > 1 : gradients **explode** exponentially.

For **tanh**: $|g'(x)| \leq 1$, so $\|\text{diag}(g'(\mathbf{a}_i))\| \leq 1$.

Unless $\mathbf{W}_h$ has large singular values, the product **shrinks to zero**.

Vanishing Gradients: Why It Matters

Consequence: The RNN cannot learn **long-range dependencies**.

Example: “The **cat**, which had been sitting on the mat in the corner of the room all afternoon, **was** sleeping.”

- The verb “was” must agree with “cat” (singular).
- But the gradient signal from “was” to “cat” must travel through many time steps.
- With vanishing gradients, the model **cannot learn this dependency**.

Vanishing Gradients: Why It Matters

Consequence: The RNN cannot learn **long-range dependencies**.

Example: “The **cat**, which had been sitting on the mat in the corner of the room all afternoon, **was** sleeping.”

- The verb “was” must agree with “cat” (singular).
- But the gradient signal from “was” to “cat” must travel through many time steps.
- With vanishing gradients, the model **cannot learn this dependency**.

Empirical evidence (Khandelwal et al., ACL 2018):

- At time step t , what matters to $\mathbf{h}_t$ is mostly $\mathbf{x}_{t'}$ where t' is **close** to t .
- Even LSTMs effectively use context of ~ 200 tokens.

⇒ This motivates the development of **gated architectures**.

Gradient Explosion: A Simpler Problem

Exploding gradients are easier to fix than vanishing gradients.

Gradient clipping (Pascanu et al., 2013):

After computing the gradient $\mathbf{g} = \nabla_{\theta} \mathcal{L}$, if $\|\mathbf{g}\|_2 > C$:

$$\mathbf{g} \leftarrow C \cdot \frac{\mathbf{g}}{\|\mathbf{g}\|_2}$$

- C is a hyperparameter (e.g., $C = 1.0$ or 5.0).
- Rescales the gradient to have norm at most C .
- Does *not* change the gradient direction, only its magnitude.
- Simple, effective, and widely used (even with LSTMs and Transformers).

Gradient Explosion: A Simpler Problem

Exploding gradients are easier to fix than vanishing gradients.

Gradient clipping (Pascanu et al., 2013):

After computing the gradient $\mathbf{g} = \nabla_{\theta} \mathcal{L}$, if $\|\mathbf{g}\|_2 > C$:

$$\mathbf{g} \leftarrow C \cdot \frac{\mathbf{g}}{\|\mathbf{g}\|_2}$$

- C is a hyperparameter (e.g., $C = 1.0$ or 5.0).
- Rescales the gradient to have norm at most C .
- Does *not* change the gradient direction, only its magnitude.
- Simple, effective, and widely used (even with LSTMs and Transformers).

Vanishing gradients are harder: the gradient is not wrong in direction, it is just **too small to provide a useful learning signal**.

⇒ We need an **architectural** solution.

Long Short-Term Memory (LSTM)

Key idea (Hochreiter & Schmidhuber, 1997):

Add a **cell state** c_t that acts as a memory highway: information can flow through it with **minimal transformation**.

Long Short-Term Memory (LSTM)

Key idea (Hochreiter & Schmidhuber, 1997):

Add a **cell state** $\mathbf{c}_t$ that acts as a memory highway: information can flow through it with **minimal transformation**.

Gates control what information to:

- **Forget** from the cell state (forget gate $\mathbf{f}_t$)
- **Add** to the cell state (input gate $\mathbf{i}_t$)
- **Output** from the cell state (output gate $\mathbf{o}_t$)

Gates are vectors of values in $(0, 1)$, computed via **sigmoid**:

- ≈ 0 : “block everything”
- ≈ 1 : “let everything through”

The gates are **learned**: the network learns *when* to remember and when to forget.

LSTM: The Equations

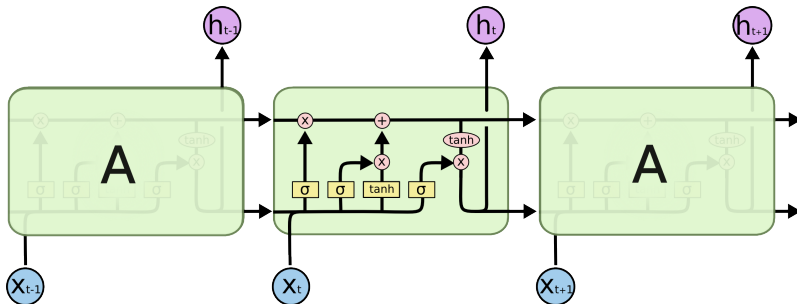


Figure credit: [Chris Olah](#)

Forget gate:

$$f_t = \sigma(W_f[x_t; h_{t-1}] + b_f)$$

Input gate:

$$i_t = \sigma(W_i[x_t; h_{t-1}] + b_i)$$

Candidate cell:

$$\tilde{c}_t = \tanh(W_c[x_t; h_{t-1}] + b_c)$$

Cell state update:

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

LSTM: The Equations (continued)

Output gate:

$$\mathbf{o}_t = \sigma(\mathbf{W}_o[\mathbf{x}_t; \mathbf{h}_{t-1}] + \mathbf{b}_o)$$

Hidden state:

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

Notation:

- $[\mathbf{x}_t; \mathbf{h}_{t-1}]$: concatenation of input and previous hidden state.
- $\odot$: elementwise multiplication.
- $\tanh(\mathbf{c}_t)$ squashes the memory read-out to $(-1, 1)$; the output gate selects which parts to expose.

LSTM: Why It Helps with Vanishing Gradients

The key is the **cell state update**:

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$$

LSTM: Why It Helps with Vanishing Gradients

The key is the **cell state update**:

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$$

Direct path (ignoring paths through the gates):

$$\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}} = \text{diag}(\mathbf{f}_t)$$

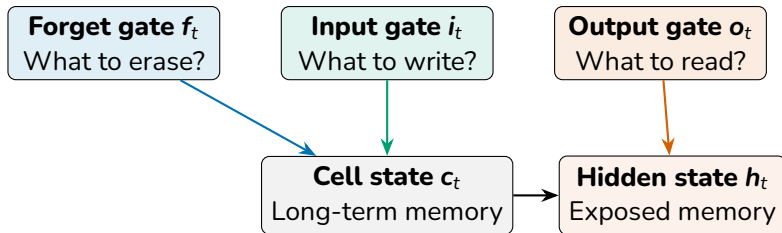
- If $\mathbf{f}_t \approx 1$: gradient flows through unchanged, with no vanishing.
- If $\mathbf{f}_t \approx 0$: the cell *chooses* to forget, and the gradient stops (intentionally).
- No repeated multiplication by a weight matrix $\mathbf{W}$.

Analogy: The cell state is a conveyor belt: information flows through **additive** interactions, as in a residual connection ($\mathbf{x} + f(\mathbf{x})$).

In practice: LSTMs can learn dependencies of length ~ 100 – 200 tokens (vs. ~ 10 for vanilla RNNs).

LSTM: Intuitive Summary

Think of the LSTM cell as a **gated memory unit**:



- **Forget gate:** “Should I keep remembering the subject of the sentence?”
- **Input gate:** “Is this new word important enough to store?”
- **Output gate:** “What part of my memory is relevant right now?”

The LSTM **learns** when to use each gate from data.

Gated Recurrent Units (GRUs)

A **simpler** alternative to LSTMs (Cho et al., 2014):

- Only **two gates** (update and reset) instead of three.
- **No separate cell state:** it merges $\mathbf{c}_t$ and $\mathbf{h}_t$.
- Fewer parameters, generally works comparably well.

Update gate:

$$\mathbf{z}_t = \sigma(\mathbf{W}_z[\mathbf{x}_t; \mathbf{h}_{t-1}] + \mathbf{b}_z)$$

Reset gate:

$$\mathbf{r}_t = \sigma(\mathbf{W}_r[\mathbf{x}_t; \mathbf{h}_{t-1}] + \mathbf{b}_r)$$

Hidden state update:

$$\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tanh(\mathbf{W}[\mathbf{x}_t; \mathbf{r}_t \odot \mathbf{h}_{t-1}] + \mathbf{b})$$

GRU: Intuition

$$h_t = \underbrace{(1 - z_t) \odot h_{t-1}}_{\text{keep old state}} + \underbrace{z_t \odot \tilde{h}_t}_{\text{add new info}}$$

Update gate z_t : Interpolates between old and new hidden state.

- $z_t \approx 0$: keep the old state (“nothing new”).
- $z_t \approx 1$: replace with new candidate (“important update”).

GRU: Intuition

$$h_t = \underbrace{(1 - z_t) \odot h_{t-1}}_{\text{keep old state}} + \underbrace{z_t \odot \tilde{h}_t}_{\text{add new info}}$$

Update gate z_t : Interpolates between old and new hidden state.

- $z_t \approx 0$: keep the old state (“nothing new”).
- $z_t \approx 1$: replace with new candidate (“important update”).

Reset gate r_t : Controls how much of h_{t-1} to use when computing the candidate.

- $r_t \approx 0$: ignore previous state (act like a feedforward network).
- $r_t \approx 1$: fully use previous state (act like a vanilla RNN).

Gradient flow: Like the LSTM, when $z_t \approx 0$, gradients flow through the identity $(1 - z_t) \approx 1$, mitigating vanishing gradients.

LSTM vs. GRU

	LSTM	GRU
Gates	3 (forget, input, output)	2 (update, reset)
States	$\mathbf{h}_t$ and $\mathbf{c}_t$ (separate)	$\mathbf{h}_t$ only
Parameters	$4 \times d \times (d + d_x)$	$3 \times d \times (d + d_x)$
Memory path	$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \dots$	$\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \dots$

LSTM vs. GRU

	LSTM	GRU
Gates	3 (forget, input, output)	2 (update, reset)
States	$\mathbf{h}_t$ and $\mathbf{c}_t$ (separate)	$\mathbf{h}_t$ only
Parameters	$4 \times d \times (d + d_x)$	$3 \times d \times (d + d_x)$
Memory path	$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \dots$	$\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \dots$

Which is better?

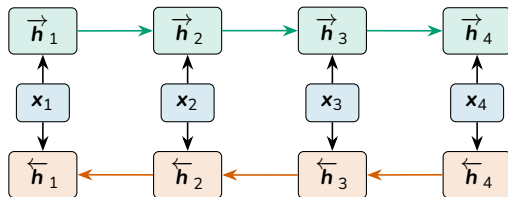
- No consistent winner across all tasks (Chung et al., 2014).
- GRU: faster to train, fewer parameters, can work better on smaller datasets.
- LSTM: often slightly better on longer sequences and language modeling.
- Both are significantly better than vanilla RNNs.

In modern NLP: Both LSTM and GRU have been largely **superseded by Transformers** (Lecture 7), but they remain useful building blocks and appear in many deployed systems.

Bidirectional RNNs

A standard RNN processes tokens left-to-right. But for many tasks (classification, tagging), we want to use context from **both directions**.

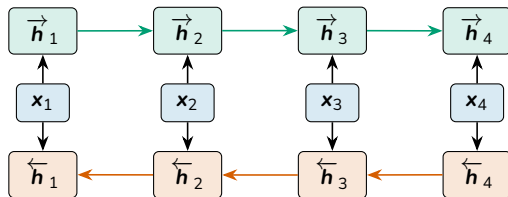
Bidirectional RNN (BiRNN):



Bidirectional RNNs

A standard RNN processes tokens left-to-right. But for many tasks (classification, tagging), we want to use context from **both directions**.

Bidirectional RNN (BiRNN):



Final representation at position t : $h_t = [\vec{h}_t; \overleftarrow{h}_t]$ (concatenation).

- BiRNNs typically produce **more powerful features** than unidirectional RNNs.
- **Cannot** be used for autoregressive language modeling (would “cheat” by seeing future tokens).

Multi-Layer (Stacked) RNNs

We can stack multiple RNN layers:

- Layer 1: $\mathbf{h}_t^{(1)} = \text{RNN}_1(\mathbf{x}_t, \mathbf{h}_{t-1}^{(1)})$
- Layer 2: $\mathbf{h}_t^{(2)} = \text{RNN}_2(\mathbf{h}_t^{(1)}, \mathbf{h}_{t-1}^{(2)})$
- ...

Multi-Layer (Stacked) RNNs

We can stack multiple RNN layers:

- Layer 1: $h_t^{(1)} = \text{RNN}_1(\mathbf{x}_t, h_{t-1}^{(1)})$
- Layer 2: $h_t^{(2)} = \text{RNN}_2(h_t^{(1)}, h_{t-1}^{(2)})$
- ...

Why?

- Lower layers capture **local** patterns (e.g., morphology, POS).
- Higher layers capture **global** patterns (e.g., semantics, syntax).

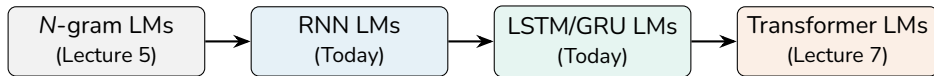
In practice:

- 2–4 layers are common for RNNs.
- Deeper stacks benefit from **residual connections** and **dropout between layers**.
- ELMo (Peters et al., 2018): 2-layer BiLSTM, a landmark pre-neural-Transformer model.

Practical Training Tips for RNNs

1. **Gradient clipping:** Always clip gradients (e.g., $C = 1.0$ or 5.0).
2. **Truncated BPTT:** Instead of backpropagating through the *entire* sequence, truncate to a fixed window (e.g., 35 or 70 steps). Balances memory and learning.
3. **Dropout:** Apply between layers (not within the recurrence). Variational dropout (Gal & Ghahramani, 2016) uses the *same* mask across time steps.
4. **Initialization:** Initialize forget gate biases to $+1$ (Jozefowicz et al., 2015) so the LSTM defaults to *remembering*.
5. **Learning rate:** For SGD with momentum, start near 1.0; for Adam, start near 10^{-3} . Decay on a plateau.

The Story So Far: From N -Grams to LSTMs



	N-gram	Vanilla RNN	LSTM/GRU
Context	Fixed ($N-1$)	Unbounded	Unbounded
Effective context	$N-1$	~ 10 tokens	~ 200 tokens
Param. sharing	None	Across time	Across time
Long-range deps.	No	Poor	Moderate
Training	Count + smooth	BPTT	BPTT

Summary

- **MLPs** are the basic building block: linear transformation + nonlinear activation, stacked in layers.
- **Nonlinear activations** (ReLU, sigmoid, tanh) are essential. Without them, deep networks collapse to a single linear model.
- **Backpropagation** efficiently computes all gradients via the chain rule on the computation graph.
- **RNNs** process sequences by maintaining a hidden state, applying the same weights at each time step.
- The **vanishing gradient problem** prevents vanilla RNNs from learning long-range dependencies.
- **LSTMs** solve this with a gated cell state that allows additive gradient flow.
- **GRUs** are a simpler alternative with comparable performance.
- Practical tricks: gradient clipping, truncated BPTT, proper initialization.

Readings

Optional / further reading:

- Jurafsky & Martin, *SLP 3*, [Chapter 6: Neural Networks](#).
- Jurafsky & Martin, *SLP 3*, [Chapter 14: RNNs and LSTMs](#).
- Olah, C. (2015). [Understanding LSTM Networks](#). (Excellent visual explanation.)
- Hochreiter, S. & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8), 1735–1780.
- Pascanu, R., Mikolov, T., & Bengio, Y. (2013). [On the difficulty of training recurrent neural networks](#). *ICML*.

Next Lecture

Transformers from Scratch

- Attention (Q/K/V), multi-head attention, positional encoding.
- Layer norm, residual connections, feed-forward networks.
- Full encoder-decoder walkthrough.