Predicting the Next Word

From Shannon to ChatGPT

Test Compilation - Chapter 5

Contents

5 RNNs and LSTMs: Sequential Context for Next-Word Prediction 1

5.1 From Static Embeddings to Sequential Context . . . 1
5.1.1 The Aggregation Problem in Language Modeling . . . 1
5.1.2 Sequential Processing as Solution . . . 2
5.1.3 Introduction of Running Example . . . 3
5.2 Vanilla RNN Architecture . . . 6
5.2.1 The RNN Cell . . . 6
5.2.2 Unrolling and Parameter Sharing . . . 6
5.2.3 The Short Memory Problem . . . 9
5.3 LSTM: Learning What to Remember and Forget . . . 10
5.3.1 The Gating Intuition . . . 10
5.3.2 LSTM Cell Architecture . . . 11
5.3.3 Information Flow Through Gates . . . 11
5.3.4 Why Gates Help: Gradient Highways . . . 12
5.4 GRU and Architecture Comparisons . . . 19
5.4.1 GRU Architecture . . . 19
5.4.2 LSTM vs GRU Trade-offs . . . 19
5.4.3 When to Use Which . . . 20
5.5 Training RNNs for Language Modeling . . . 23
5.5.1 The Language Modeling Objective . . . 23
5.5.2 Backpropagation Through Time . . . 23
5.5.3 Practical Training Considerations . . . 24
5.6 Context Representation in RNNs . . . 26

Chapter 5

RNNs and LSTMs: Sequential Context for Next-Word Prediction

In this chapter, we advance next-word prediction by:

- Processing word sequences incrementally to build dynamic context representations
- Introducing hidden states that evolve as each word is encountered
- Solving the vanishing gradient problem through gated memory mechanisms
- Enabling models to capture long-range dependencies between distant words

5.1 From Static Embeddings to Sequential Context

The word embeddings developed in Chapter **??** represent a significant advance over discrete one-hot encodings: each word $w \in \mathcal{V}$ maps to a dense vector $\mathbf{e}_w \in \mathbb{R}^d$ where geometric proximity reflects semantic similarity. However, these embeddings remain fundamentally static—the vector for "bank" is identical whether it appears in "the river bank" or "the savings bank". The embedding captures general distributional properties learned across the entire corpus but cannot adapt to the specific context in which a word appears. For next-word prediction, this limitation is severe: the probability $P(w_{t+1} \mid w_1, \ldots, w_t)$ depends critically on the preceding context, yet static embeddings provide no mechanism to incorporate positional or sequential information. The n-gram models of Chapter **??** addressed context through fixed-size tuples, but these suffer from exponential parameter growth and the inability to generalize across similar contexts. What we need is a representation that evolves as we process the sequence word by word, accumulating information about the entire history into a fixed-size vector suitable for prediction. This chapter introduces recurrent neural networks (RNNs), which maintain a hidden state $\mathbf{h}_t$ that updates after each word, encoding the relevant information from $w_1, \ldots, w_t$ in a form suitable for predicting w_{t+1}.

5.1.1 The Aggregation Problem in Language Modeling

Consider the challenge of predicting the next word given a prefix of arbitrary length. For the sequence "The trophy that the athletes from the national team had won during the championship couldn't fit in the display case because it ...", predicting whether the next word is "was" or "were" requires understanding that the subject is "trophy" (singular), not "athletes" or "team". This grammatical agreement spans sixteen words—far beyond any practical n-gram window. How can we construct a context representation $\mathbf{c}_t$ that captures the relevant information from $w_1, \ldots, w_t$ regardless of sequence length? One naive approach averages the word embeddings: $\mathbf{c}_t = \frac{1}{t} \sum_{i=1}^{t} \mathbf{e}_{w_i}$. This bag-of-words representation ignores word order entirely, treating "dog bites man" and "man bites dog" identically. A concatenation approach $\mathbf{c}_t = [\mathbf{e}_{w_1}; \ldots; \mathbf{e}_{w_t}]$ preserves order but

produces variable-length representations unsuitable for fixed-size neural network layers, and grows linearly with sequence length. Neither approach is satisfactory: we require a fixed-size representation that nonetheless captures sequential structure. The key insight is that we should process words incrementally, updating our context representation after each word rather than computing it from scratch. This leads to the recurrence relation $\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{e}_{w_t})$, where the hidden state $\mathbf{h}_t$ summarizes the sequence $w_1, \ldots, w_t$ and f is a learned function that determines how to incorporate new information while preserving relevant history.

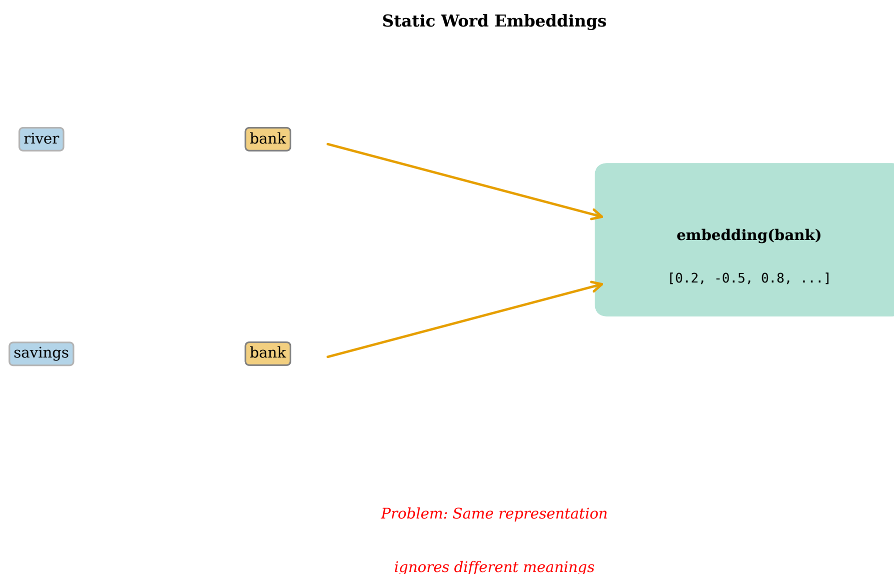


Figure 5.1: Static embeddings versus dynamic context representations. The left panel shows that static word embeddings assign identical vectors to "bank" regardless of context: both "river bank" and "savings bank" yield the same embedding, making disambiguation impossible at prediction time. The right panel illustrates the dynamic approach where recurrent processing builds context-dependent representations: the hidden state after "river" differs from the hidden state after "savings", enabling the model to disambiguate polysemous words based on their sequential context.

5.1.2 Sequential Processing as Solution

Sequential processing provides an elegant solution to the aggregation problem. Rather than computing context from all words simultaneously, we process the sequence left-to-right, maintaining a hidden state $\mathbf{h}_t \in \mathbb{R}^{d_h}$ that serves as a fixed-size summary of the words seen so far. At each time step t, the model receives the current word embedding $\mathbf{e}_{w_t}$ and the previous hidden state $\mathbf{h}_{t-1}$, then computes an updated hidden state $\mathbf{h}_t$ through a learned transformation. The hidden state can be thought of as the model's "memory" or "working representation" of the sequence: it must encode whatever information from the past is relevant for future predictions. The prediction at each step is then $P(w_{t+1} \mid \mathbf{h}_t)$, where the hidden state $\mathbf{h}_t$ implicitly encodes the entire history $w_1, \ldots, w_t$. This framework has several advantages. First, the representation size is constant: $\mathbf{h}_t \in \mathbb{R}^{d_h}$ regardless of whether $t = 5$ or $t = 500$. Second, the same parameters are used at every time step, enabling the model to generalize across positions and handle sequences longer than those seen during training. Third, the sequential nature respects the causal structure of language: we predict future words based only on past context, matching the left-to-right reading process. The challenge lies in learning a transition function that preserves relevant information across many time steps while forgetting irrelevant details—a balance that simple recurrent networks struggle to achieve but that gated architectures address effectively.

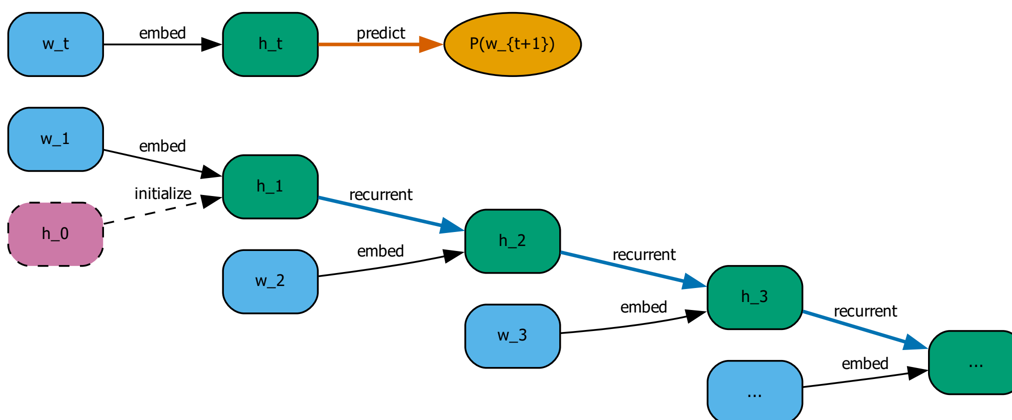


Figure 5.2: Sequential processing with hidden states. Words enter the model one at a time from left to right, each producing an updated hidden state. The initial hidden state $\mathbf{h}_0$ is typically set to zeros or learned as a parameter. After processing word w_t, the hidden state $\mathbf{h}_t$ encodes information about the entire prefix $w_1, \ldots, w_t$. The final hidden state feeds into a prediction layer that outputs $P(w_{t+1} \mid \mathbf{h}_t)$ as a distribution over the vocabulary. This architecture processes variable-length sequences while maintaining fixed-size internal representations.

5.1.3 Introduction of Running Example

Throughout this chapter, we examine a running example that illustrates the challenge of long-range dependencies: “The trophy that the athletes from the national team had won during the championship couldn’t fit in the display case because it was too large.” The critical dependency is between “trophy” at position 2 and “was” at position 18—a span of sixteen words. Predicting “was” (singular) rather than “were” (plural) requires the model to remember that the grammatical subject is “trophy”, not the intervening nouns “athletes”, “team”, or “case”. The nested relative clause “that the athletes from the national team had won during the championship” creates this challenging structure: many words intervene between the subject and its verb, and several plural nouns could mislead a model with limited memory. This example is not contrived—such long-range dependencies are pervasive in natural language, arising from relative clauses, coordination, embedding, and discourse structure. A successful language model must maintain relevant information (the singular subject) through the intervening material while appropriately ignoring irrelevant details. We will trace how vanilla RNNs fail on this example due to vanishing gradients, and how LSTM’s gating mechanism succeeds by selectively preserving the subject information through the clause structure.

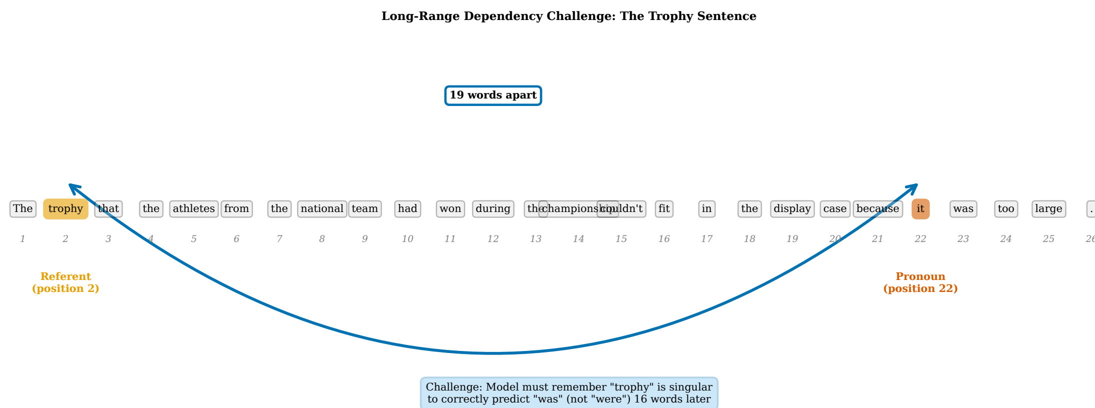


Figure 5.3: Running example sentence with long-range subject-verb dependency. The sentence "The trophy that the athletes from the national team had won during the championship couldn't fit in the display case because it was too large" requires remembering "trophy" (singular, position 2) to correctly predict "was" (position 18). The intervening relative clause contains multiple plural nouns (athletes, team) that could mislead the model. The curved arrow indicates the grammatical dependency spanning 16 words—far beyond typical n-gram windows. This dependency is typical of complex English sentences and requires effective long-range memory mechanisms.

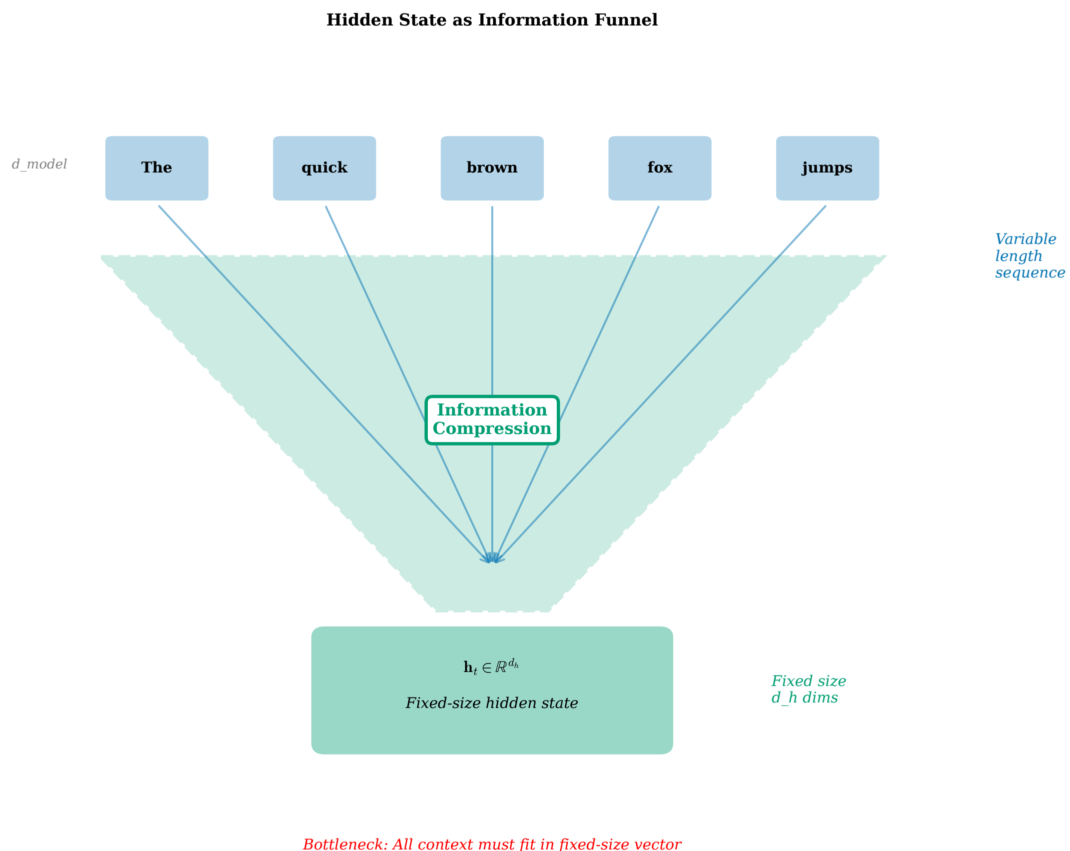


Figure 5.4: Hidden state as compressed sequence representation. The funnel visualization shows how a variable-length sequence of word embeddings is progressively compressed into a fixed-size hidden state vector. Each word contributes information to the hidden state, which must encode the relevant aspects of the entire history in d_h dimensions regardless of sequence length. This compression is both the strength and the limitation of recurrent architectures: it enables processing of arbitrary-length sequences but requires the model to learn which information to preserve and which to discard.

5.2 Vanilla RNN Architecture

The vanilla recurrent neural network (RNN) implements sequential processing through a simple recurrence relation. At each time step t, the hidden state $\mathbf{h}_t$ is computed as a function of the previous hidden state $\mathbf{h}_{t-1}$ and the current input $\mathbf{e}_{w_t}$. The update equation combines these inputs through learned weight matrices and a nonlinear activation function. Despite its simplicity, this architecture introduced the fundamental concept of recurrent computation for sequence modeling: the same parameters are applied at every time step, enabling the network to generalize across positions and handle variable-length inputs. The vanilla RNN was the dominant sequence model from the late 1980s through the early 1990s, applied successfully to tasks with short-range dependencies such as phoneme recognition and simple language modeling. However, researchers quickly discovered that training vanilla RNNs on tasks requiring long-range dependencies proved extremely difficult. The vanishing gradient problem, which we examine in Section 5.2.3, causes the gradient signal to decay exponentially as it propagates backward through time, making it nearly impossible to learn dependencies spanning more than five to ten time steps. Understanding the vanilla RNN architecture is essential both for historical context and because it clearly illustrates why gated architectures like LSTM are necessary for effective sequence modeling.

5.2.1 The RNN Cell

The basic RNN cell computes the hidden state update through a linear transformation followed by a nonlinear activation, implementing the core recurrence that enables sequential processing. Let $\mathbf{e}_t = \mathbf{e}_{w_t} \in \mathbb{R}^d$ denote the embedding of word w_t, and let $\mathbf{h}_{t-1} \in \mathbb{R}^{d_h}$ denote the previous hidden state, which encodes the model's memory of all words processed so far. The vanilla RNN update equation combines these two sources of information through learned weight matrices: $\mathbf{h}_t = \tanh(\mathbf{W}_{xh}\mathbf{e}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h)$, where $\mathbf{W}_{xh} \in \mathbb{R}^{d_h \times d}$ maps input embeddings to hidden space, $\mathbf{W}_{hh} \in \mathbb{R}^{d_h \times d_h}$ maps previous hidden state to current hidden space, $\mathbf{b}_h \in \mathbb{R}^{d_h}$ is a bias vector, and tanh is applied element-wise. The hyperbolic tangent function $\tanh(x) = (e^x - e^{-x})/(e^x + e^{-x})$ squashes the output to the range $[-1, 1]$, introducing nonlinearity while keeping activations bounded. This bounded activation is important for training stability: without it, hidden state values could grow unboundedly through the recurrence, causing numerical overflow. For prediction, the hidden state is projected to vocabulary size through an output layer: $\mathbf{y}_t = \text{softmax}(\mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y)$, where $\mathbf{W}_{hy} \in \mathbb{R}^{|\mathscr{V}| \times d_h}$ and $\mathbf{y}_t \in \mathbb{R}^{|\mathscr{V}|}$ gives the predicted probability distribution over next words. The total parameter count for a vanilla RNN is $d \cdot d_h + d_h^2 + d_h + |\mathscr{V}| \cdot d_h + |\mathscr{V}|$: linear in vocabulary size and quadratic in hidden dimension. For typical values $d = 300$, $d_h = 256$, $|\mathscr{V}| = 50{,}000$, this gives approximately 13 million parameters, dominated by the output projection. The recurrence is initialized with $\mathbf{h}_0 = \mathbf{0}$ (zeros) or a learned initial state, and the same parameters are used at every time step, enabling the model to generalize across positions.

5.2.2 Unrolling and Parameter Sharing

To understand how RNNs process sequences and how gradients flow during training, we unroll the recurrence relation through time. For a sequence of T words, unrolling produces a computational graph with T copies of the RNN cell, connected in series through the hidden states. At step 1, $\mathbf{h}_1 = \tanh(\mathbf{W}_{xh}\mathbf{e}_1 + \mathbf{W}_{hh}\mathbf{h}_0 + \mathbf{b}_h)$; at step 2, $\mathbf{h}_2 = \tanh(\mathbf{W}_{xh}\mathbf{e}_2 + \mathbf{W}_{hh}\mathbf{h}_1 + \mathbf{b}_h)$; and so on through step T. Critically, the same weight matrices $\mathbf{W}_{xh}$, $\mathbf{W}_{hh}$, and $\mathbf{W}_{hy}$ are used at every time step—this parameter sharing is the defining characteristic of recurrent architectures. Parameter sharing provides several benefits. First, it reduces the number of parameters dramatically: instead of T separate sets of weights, we have a single shared set, enabling the model to handle variable-length sequences without parameter explosion. Second, it enables generalization across positions: the model learns position-invariant features that apply regardless of where a word appears in the sequence. Third, it allows processing sequences longer than those seen during training, since the same cell applies recursively. The unrolled view reveals that RNNs are simply deep feedforward networks with shared weights across layers, where "depth" corresponds to sequence length. This perspective is crucial for understanding gradient flow: during backpropagation, gradients must traverse the entire unrolled graph, passing through T multiplicative operations involving $\mathbf{W}_{hh}$. The repeated multiplication by the same matrix is the source of both exploding and vanishing gradients.

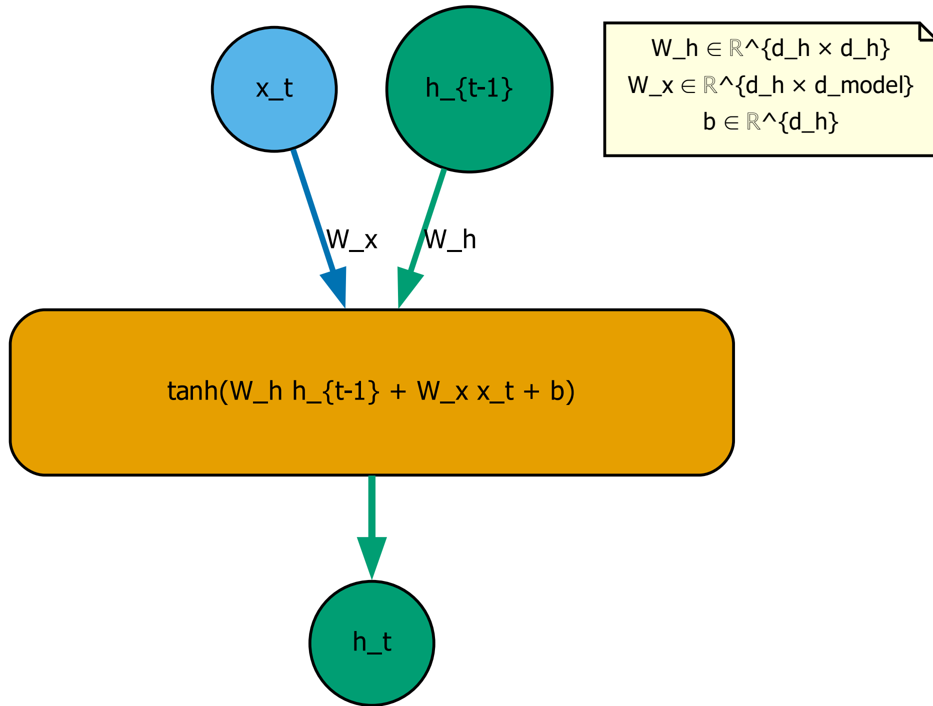


Figure 5.5: Vanilla RNN cell architecture. The cell receives two inputs: the current word embedding $\mathbf{e}_t$ and the previous hidden state $\mathbf{h}_{t-1}$. These are linearly combined through weight matrices $\mathbf{W}_{xh}$ and $\mathbf{W}_{hh}$, with a bias $\mathbf{b}_h$ added. The tanh nonlinearity produces the new hidden state $\mathbf{h}_t$, which serves both as output and as input to the next time step. The output layer applies a linear projection $\mathbf{W}_{hy}$ followed by softmax to produce a probability distribution over vocabulary words for next-word prediction.

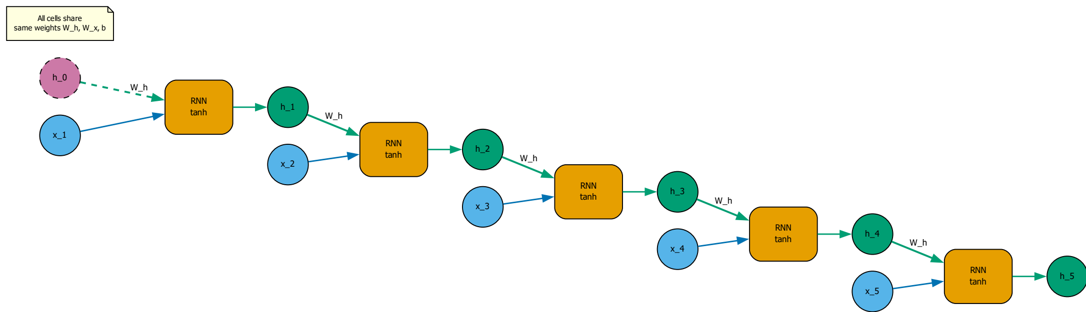


Figure 5.6: Unrolled RNN through five time steps. The recurrence relation is expanded into a computational graph where each time step corresponds to one copy of the RNN cell. Hidden states flow left-to-right through $\mathbf{h}_0 \to \mathbf{h}_1 \to \mathbf{h}_2 \to \mathbf{h}_3 \to \mathbf{h}_4 \to \mathbf{h}_5$, with word embeddings $\mathbf{e}_1$ through $\mathbf{e}_5$ entering at each step. The key feature is parameter sharing: the weight matrices $\mathbf{W}_{xh}$ and $\mathbf{W}_{hh}$ (shown with dashed lines) are identical across all time steps. During backpropagation, gradients flow right-to-left through this entire graph, multiplying by $\mathbf{W}_{hh}^\top$ at each step.

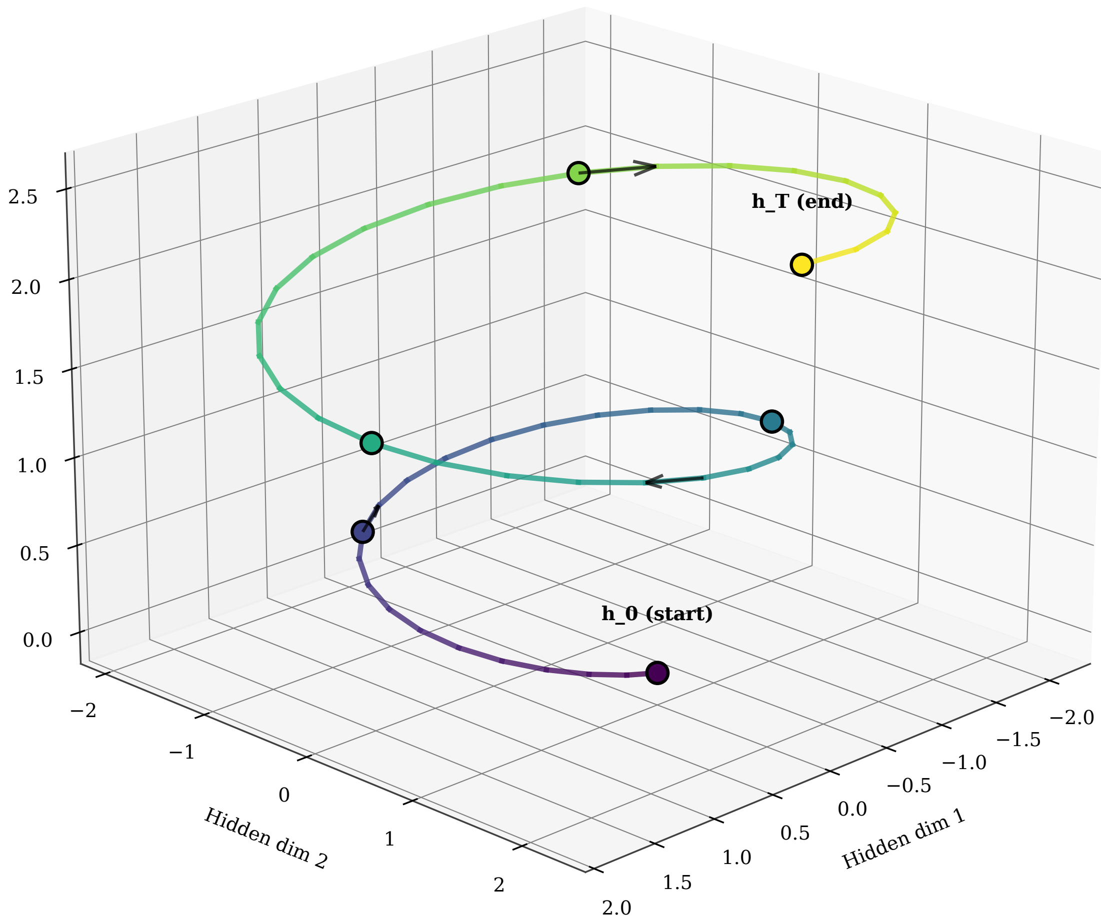


Figure 5.7: Hidden state trajectory in 3D embedding space. As the RNN processes a sequence, the hidden state traces a path through the high-dimensional hidden space (projected here to 3D via PCA). Each point represents the hidden state after processing a word, with color indicating position in the sequence from blue (early) to red (late). The trajectory shows how the hidden state evolves as context accumulates: early words establish the initial direction, while later words cause the trajectory to curve and shift. The final hidden state encodes the cumulative information from the entire sequence.

5.2.3 The Short Memory Problem

Despite their theoretical ability to capture dependencies of arbitrary length, vanilla RNNs in practice struggle to learn dependencies beyond five to ten time steps. This limitation arises from the vanishing gradient problem: during backpropagation through time (BPTT), gradients must pass through many multiplicative operations involving the recurrent weight matrix $\mathbf{W}_{hh}$, and these repeated multiplications cause gradients to either vanish (become negligibly small) or explode (become unmanageably large). Consider the gradient of the loss at time T with respect to the hidden state at time t: by the chain rule, this involves the product $\prod_{k=t+1}^{T} \frac{\partial \mathbf{h}_k}{\partial \mathbf{h}_{k-1}}$, which includes factors of the form $\mathrm{diag}(\tanh'(\cdot))\mathbf{W}_{hh}$. If the largest singular value of $\mathbf{W}_{hh}$ is less than 1, this product shrinks exponentially with $(T-t)$; if greater than 1, it grows exponentially. For our running example with "trophy" at position 2 and "was" at position 18, the gradient must survive 16 multiplicative steps. With typical weight initialization, the gradient contribution from position 2 is 10^{-4} or smaller by position 18—effectively zero for learning purposes. The model cannot learn the association between "trophy" and "was" because the gradient signal is too weak to drive parameter updates. We can observe this failure empirically: when processing our running example, the vanilla RNN's hidden state "forgets" the subject "trophy" after a few words, and by position 17 contains almost no information distinguishing singular from plural subjects. The prediction at position 18 is essentially random between "was" and "were". This short memory problem motivated the development of gated architectures that can selectively preserve information across long sequences.

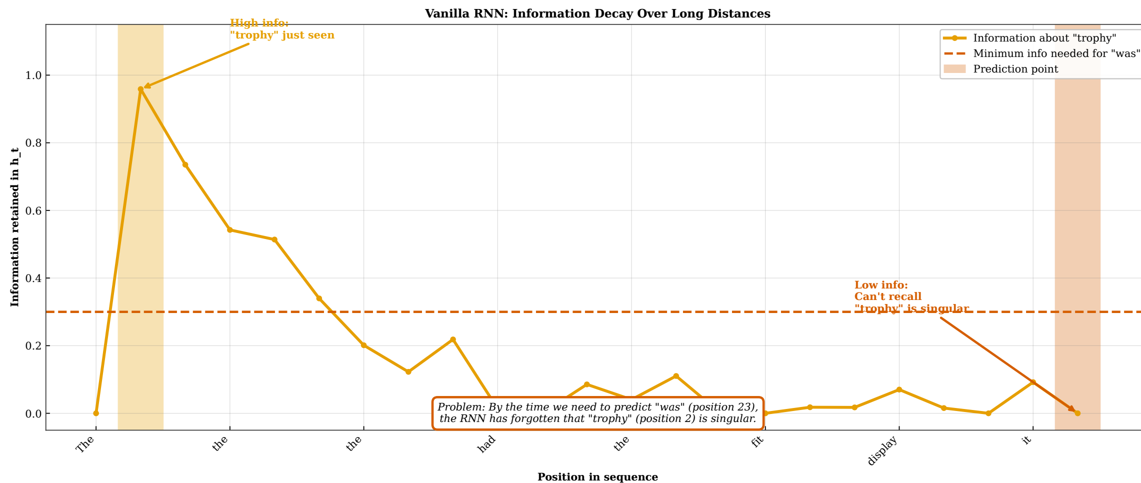


Figure 5.8: Information decay in vanilla RNN on the running example. The plot tracks how much information about "trophy" (position 2) is preserved in the hidden state as the sequence is processed. The y-axis shows a measure of "trophy information" derived from probing the hidden state's ability to distinguish singular versus plural subjects. In the vanilla RNN, this information decays exponentially: by position 10 it has dropped below 10%, and by position 18 (where "was" must be predicted) it is effectively zero. The shaded region indicates where the model cannot reliably predict the correct verb form. This failure illustrates the short memory problem that motivates LSTM architecture.

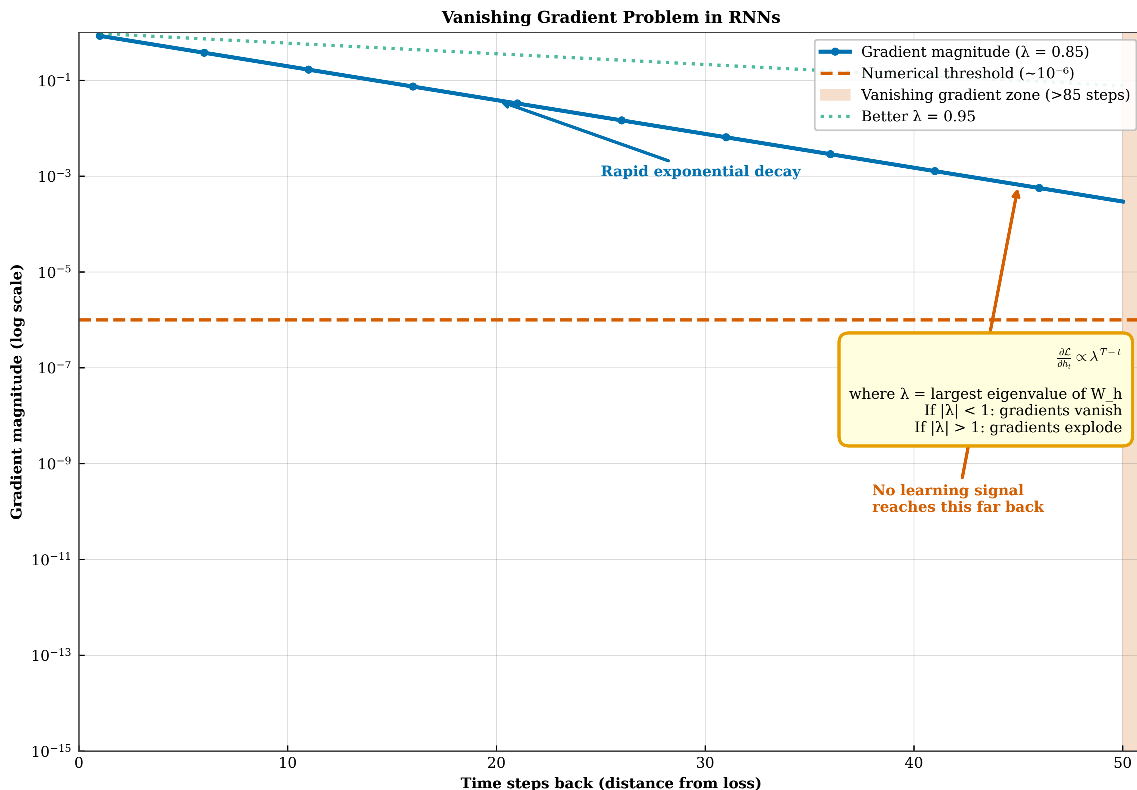


Figure 5.9: Gradient magnitude decay during backpropagation through time. The y-axis (log scale) shows the magnitude of the gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}$ as it propagates backward from the loss at position T. For vanilla RNNs, the gradient decays exponentially with temporal distance, losing 90% of its magnitude every 5-7 steps. By 15 steps back, the gradient is below 10^{-3}—too small to drive meaningful learning. The horizontal dashed line indicates the effective training threshold: gradients below this level contribute negligibly to parameter updates. Long-range dependencies require gradients to survive far beyond this threshold, which vanilla RNNs cannot achieve.

5.3 LSTM: Learning What to Remember and Forget

Long Short-Term Memory (LSTM) networks, introduced by Hochreiter and Schmidhuber (1997) [Hochreiter and Schmidhuber, 1997], address the vanishing gradient problem through a carefully designed gating mechanism. The key innovation is the cell state $\mathbf{c}_t$, a separate memory channel that runs parallel to the hidden state $\mathbf{h}_t$ and uses additive updates rather than multiplicative overwrites. Information flows along the cell state with minimal interference, protected by learnable gates that control what information enters, persists, and exits. The gates are neural network layers with sigmoid activation that produce values between 0 (completely blocked) and 1 (completely passed). By learning to open and close these gates appropriately, the LSTM can preserve relevant information across hundreds of time steps while forgetting irrelevant details. The LSTM architecture has three gates: the forget gate decides what to discard from the previous cell state, the input gate decides what new information to add, and the output gate decides what to expose in the hidden state. These gates are conditioned on the current input and previous hidden state, enabling context-dependent memory management. For our running example, the LSTM can learn to keep the forget gate high (near 1) for subject-related information while processing the relative clause, preserving "trophy" until it becomes relevant for predicting "was". This selective persistence is precisely what vanilla RNNs lack.

5.3.1 The Gating Intuition

Before examining LSTM equations, consider the intuition behind gates. A gate is a vector of values between 0 and 1, computed by a sigmoid function $\sigma(\cdot)$. When multiplied element-wise with another vector, the gate

selectively scales each dimension: values near 1 pass information through unchanged, while values near 0 block information. The crucial insight is that gates are learned, not fixed: the network discovers through training which information to preserve and which to discard based on task demands. Consider reading a sentence to answer a question about the subject. A well-trained gate would learn to preserve subject information (nouns in subject position) while forgetting modifiers and parentheticals that are irrelevant to subject-verb agreement. Different gates serve different functions: the forget gate controls persistence of old information, the input gate controls incorporation of new information, and the output gate controls visibility of stored information. The combination of these three gates provides fine-grained control over the memory cell's contents and their influence on predictions. The gating mechanism also enables gradient flow: when the forget gate is near 1, gradients pass through the cell state nearly unchanged, avoiding the repeated multiplication by weight matrices that causes vanishing gradients in vanilla RNNs. This gradient highway is the mathematical reason LSTMs can learn long-range dependencies that vanilla RNNs cannot.

5.3.2 LSTM Cell Architecture

The LSTM cell maintains both a hidden state $\mathbf{h}_t \in \mathbb{R}^{d_h}$ and a cell state $\mathbf{c}_t \in \mathbb{R}^{d_h}$, with the cell state serving as the primary long-term memory channel that can preserve information across many time steps. Given input $\mathbf{e}_t$ and previous states $\mathbf{h}_{t-1}$, $\mathbf{c}_{t-1}$, the update proceeds through computing three gates and a candidate cell state, each implemented as a simple neural network layer with sigmoid or tanh activation. The forget gate $\mathbf{f}_t = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1};\mathbf{e}_t] + \mathbf{b}_f)$ determines what fraction of each cell state dimension to retain from the previous step, with values near 1 indicating preservation and values near 0 indicating deletion. The input gate $\mathbf{i}_t = \sigma(\mathbf{W}_i[\mathbf{h}_{t-1};\mathbf{e}_t] + \mathbf{b}_i)$ determines what fraction of new information to incorporate into the cell state, controlling the flow of new content. The candidate $\tilde{\mathbf{c}}_t = \tanh(\mathbf{W}_c[\mathbf{h}_{t-1};\mathbf{e}_t] + \mathbf{b}_c)$ represents the potential new cell content computed from current inputs, and the output gate $\mathbf{o}_t = \sigma(\mathbf{W}_o[\mathbf{h}_{t-1};\mathbf{e}_t] + \mathbf{b}_o)$ determines what fraction of the cell state to expose in the hidden state output. In all these equations, $[\mathbf{h}_{t-1};\mathbf{e}_t]$ denotes concatenation of the previous hidden state and current input embedding, $\mathbf{W}_f, \mathbf{W}_i, \mathbf{W}_c, \mathbf{W}_o \in \mathbb{R}^{d_h \times (d_h+d)}$ are learnable weight matrices, and $\mathbf{b}_f, \mathbf{b}_i, \mathbf{b}_c, \mathbf{b}_o \in \mathbb{R}^{d_h}$ are learnable bias vectors. Each gate receives the same concatenated inputs but learns different weights through training, enabling context-dependent decisions about what information to remember, update, and output at each position.

The cell state and hidden state are then updated through the core LSTM equations: $\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$ for the cell state and $\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$ for the hidden state, where $\odot$ denotes element-wise (Hadamard) multiplication. The cell state update equation is the architectural innovation that gives LSTM its power: it is additive rather than purely multiplicative, combining retained old content $(\mathbf{f}_t \odot \mathbf{c}_{t-1})$ with gated new content $(\mathbf{i}_t \odot \tilde{\mathbf{c}}_t)$ through summation rather than transformation. When $\mathbf{f}_t \approx 1$ and $\mathbf{i}_t \approx 0$, the cell state passes through nearly unchanged: $\mathbf{c}_t \approx \mathbf{c}_{t-1}$, enabling information to persist across arbitrarily long sequences without degradation. This additive structure creates a gradient highway that allows gradients to flow backward through time without the exponential decay caused by repeated matrix multiplication in vanilla RNNs, because the Jacobian of the cell state update with respect to the previous cell state is close to the identity matrix when the forget gate is high. The hidden state $\mathbf{h}_t$ is computed from the cell state through a nonlinearity and output gate, serving as the LSTM's output at each time step for prediction while the cell state maintains the long-term memory for future use.

5.3.3 Information Flow Through Gates

Understanding how information flows through LSTM gates is essential for interpreting what the network learns. Consider processing our running example sentence word by word. At "The" (position 1), all gates activate moderately: the forget gate is high because there is little prior context to preserve, the input gate opens to accept the determiner, and the output gate prepares to emit a hidden state. At "trophy" (position 2), the input gate activates strongly for dimensions encoding noun information and singular number, writing this subject information to the cell state. The forget gate remains high, preserving the determiner context. As we process the relative clause "that the athletes from the national team had won during the championship", the forget gate for subject-related dimensions stays near 1, preserving the "trophy" information despite the intervening

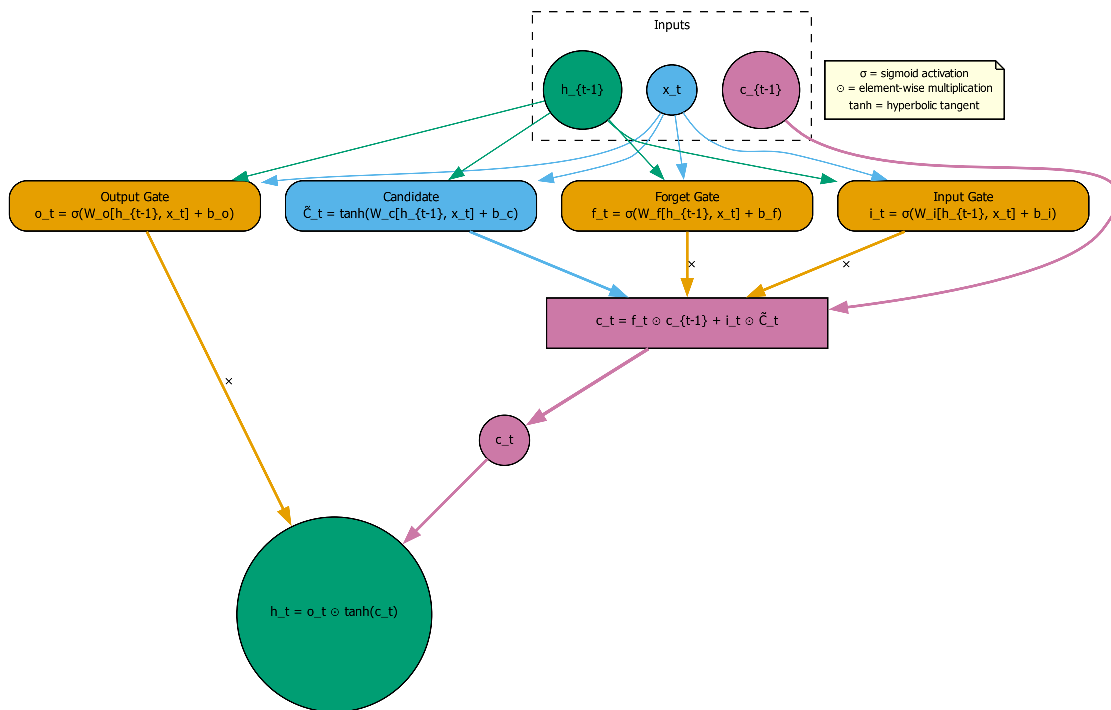


Figure 5.10: Complete LSTM cell architecture. The cell receives input embedding $\mathbf{e}_t$ and previous states $\mathbf{h}_{t-1}$, $\mathbf{c}_{t-1}$. Four parallel computations produce the forget gate $\mathbf{f}_t$, input gate $\mathbf{i}_t$, candidate $\tilde{\mathbf{c}}_t$, and output gate $\mathbf{o}_t$. The cell state update combines forgotten old content ($\mathbf{f}_t \odot \mathbf{c}_{t-1}$) with gated new content ($\mathbf{i}_t \odot \tilde{\mathbf{c}}_t$). The hidden state is the gated cell output ($\mathbf{o}_t \odot \tanh(\mathbf{c}_t)$). Element-wise multiplications ($\odot$) are shown as circles; the addition ($+$) for cell state update is the key to gradient preservation.

material. Simultaneously, the input gate selectively adds clause-internal information (like "athletes", "team") to different cell dimensions. The output gate modulates what information influences word predictions within the clause. By position 17 ("it"), the cell state still contains the subject information from position 2 because the forget gate protected it. When predicting position 18, this preserved information enables correct prediction of "was" (singular) rather than "were" (plural). The key is that the gates learn to identify what information is relevant for long-range dependencies and protect it appropriately—a capability that emerges through training on examples requiring such dependencies.

5.3.4 Why Gates Help: Gradient Highways

The gating mechanism provides more than intuitive information control—it fundamentally changes gradient dynamics during training, enabling effective learning of long-range dependencies that would be impossible for vanilla RNNs. Consider the gradient $\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}}$ from the cell state update equation, which determines how error signals propagate backward through the cell state pathway during backpropagation through time. By the chain rule, this gradient depends on how the current cell state changes with respect to the previous cell state: $\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}} = \text{diag}(\mathbf{f}_t) + (\text{terms involving gate derivatives})$, where $\text{diag}(\mathbf{f}_t)$ is a diagonal matrix with the forget gate values on the diagonal. When the forget gate $\mathbf{f}_t \approx 1$, this Jacobian is close to the identity matrix, meaning gradients flowing backward through the cell state multiply by matrices close to identity and thereby preserve their magnitude across time steps. This behavior stands in stark contrast to vanilla RNNs, where the Jacobian $\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}$ involves $\text{diag}(\tanh'(\cdot))\mathbf{W}_{hh}$, with singular values that are typically far from 1 and cause exponential growth or decay of gradients. The cell state thus acts as a gradient highway: when the forget gate is open (near 1), gradients pass through with minimal attenuation, enabling learning signals to reach early time steps with

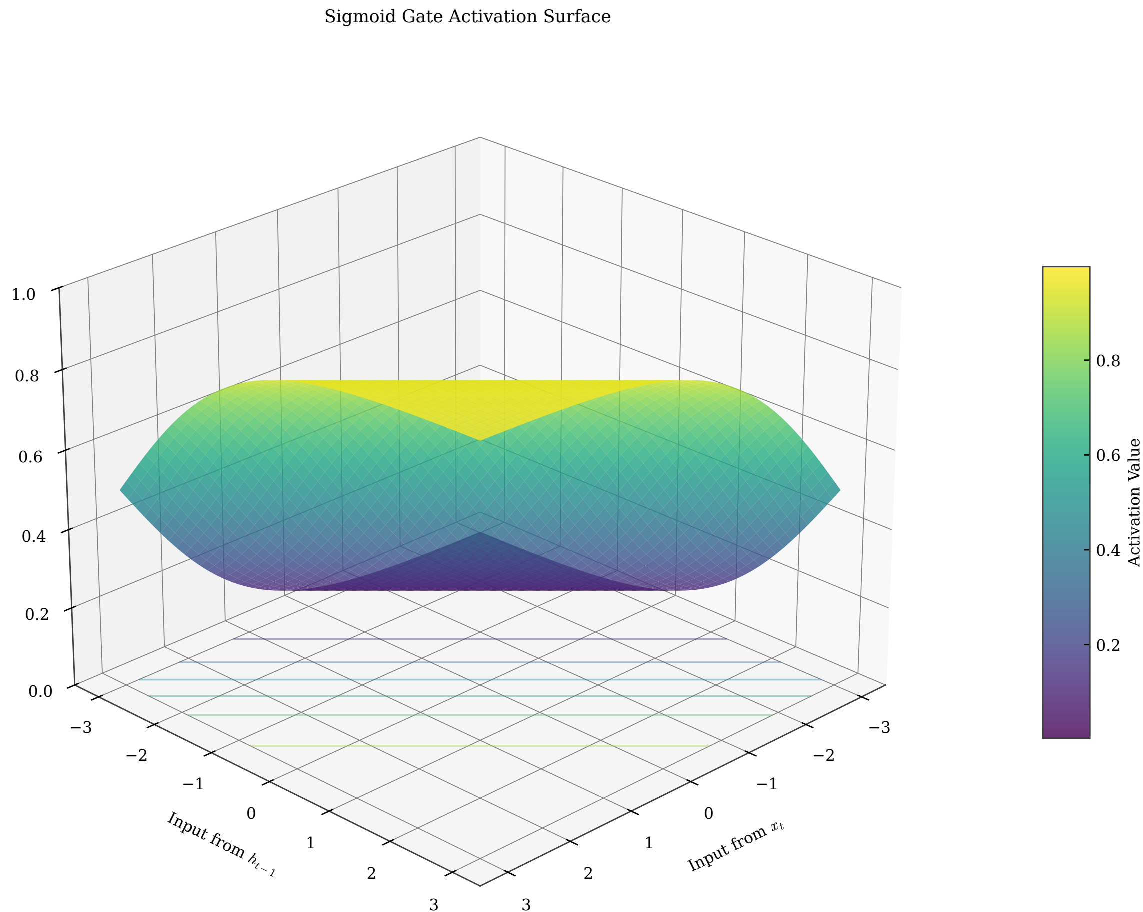


Figure 5.11: Gate activation surface in 3D. The sigmoid activation function produces gate values between 0 and 1 based on weighted inputs from $\mathbf{e}_t$ and $\mathbf{h}_{t-1}$. The surface shows how gate activation varies smoothly across the input space: regions with high weighted sum (upper right) produce gate values near 1 (information passes), while regions with low weighted sum (lower left) produce values near 0 (information blocked). The learned weights determine where in input space each gate opens or closes, enabling context-dependent memory control. The transition region (middle) allows for partial gating.

sufficient magnitude to drive parameter updates. The model can learn to keep forget gates high precisely when long-range gradient flow is needed for the task at hand, enabling effective credit assignment across many time steps without explicit supervision about what to remember. This self-regulating property—where the same gates that control information flow during the forward pass also control gradient flow during the backward pass—is the key to LSTM's trainability and the reason it can learn dependencies spanning hundreds of time steps, far beyond the five-to-ten step limit of vanilla RNNs. For our running example, gradients from the loss at position 18 can reach position 2 with sufficient magnitude to update the parameters responsible for preserving subject information, enabling the model to learn the subject-verb agreement pattern through standard gradient descent.

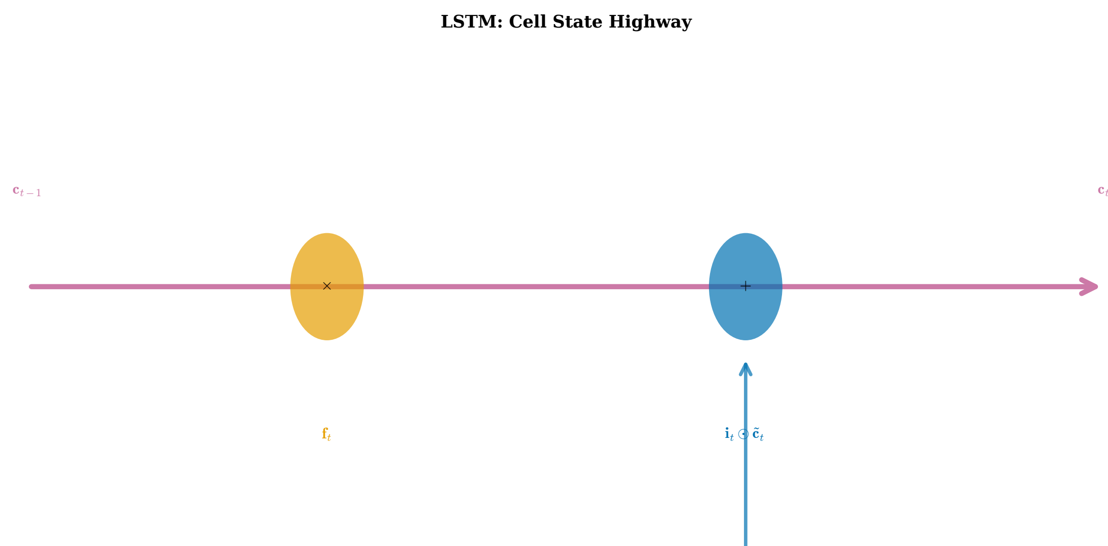


Figure 5.12: Cell state as memory highway. The top diagram shows the LSTM cell state pathway: information can flow unchanged when the forget gate is near 1 and input gate is near 0, creating a "highway" for information preservation. The additive update $\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$ means cell content persists without repeated matrix multiplication. The bottom diagram contrasts with vanilla RNN, where hidden state updates involve multiplicative transformation through the weight matrix at every step, causing information to be transformed (and potentially lost) continuously. This architectural difference explains LSTM's superior long-range memory.

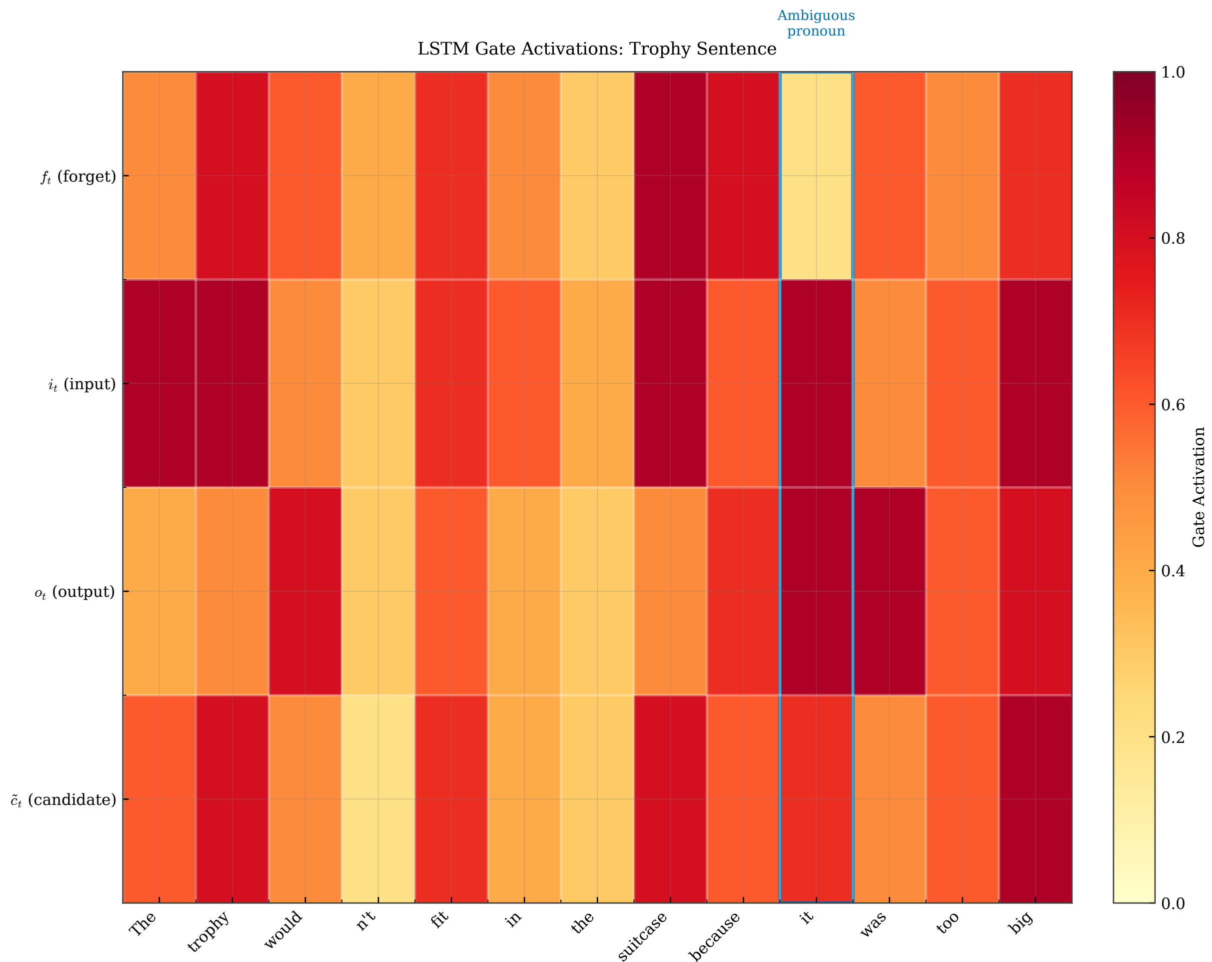


Figure 5.13: Gate activations for the running example in LSTM. The heatmap shows gate values (0 = dark, 1 = light) across sequence positions for a trained LSTM. The forget gate row shows consistently high values, indicating information preservation through the relative clause. The input gate row shows high activation at content words (“trophy”, “athletes”, “won”) and low activation at function words. The output gate row shows higher activation before major predictions. The cell state row shows accumulated information growing through the sequence. At position 18 (predicting “was”), the preserved “trophy” information enables correct singular verb prediction.

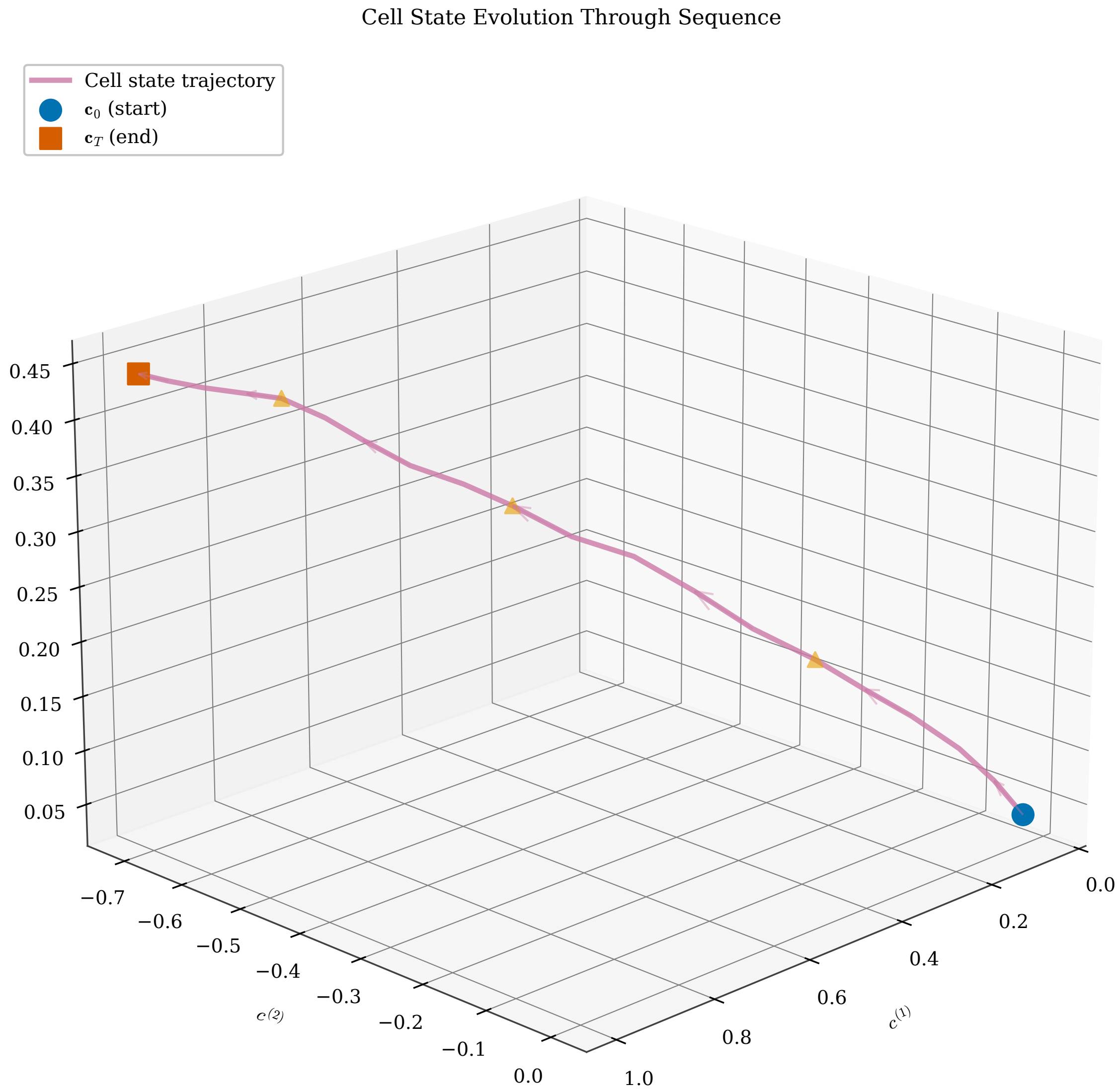


Figure 5.14: Cell state trajectory in 3D space. Unlike the hidden state which transforms at every step, the cell state (shown in pink) follows a smoother, more stable trajectory through sequence processing. The cell state changes primarily at positions where the input gate opens to add new information, remaining relatively constant through intervening material. This stability enables information preservation across long sequences. Compare with Figure 5.7: the cell state path shows less curvature and more consistent direction, reflecting its role as long-term memory.

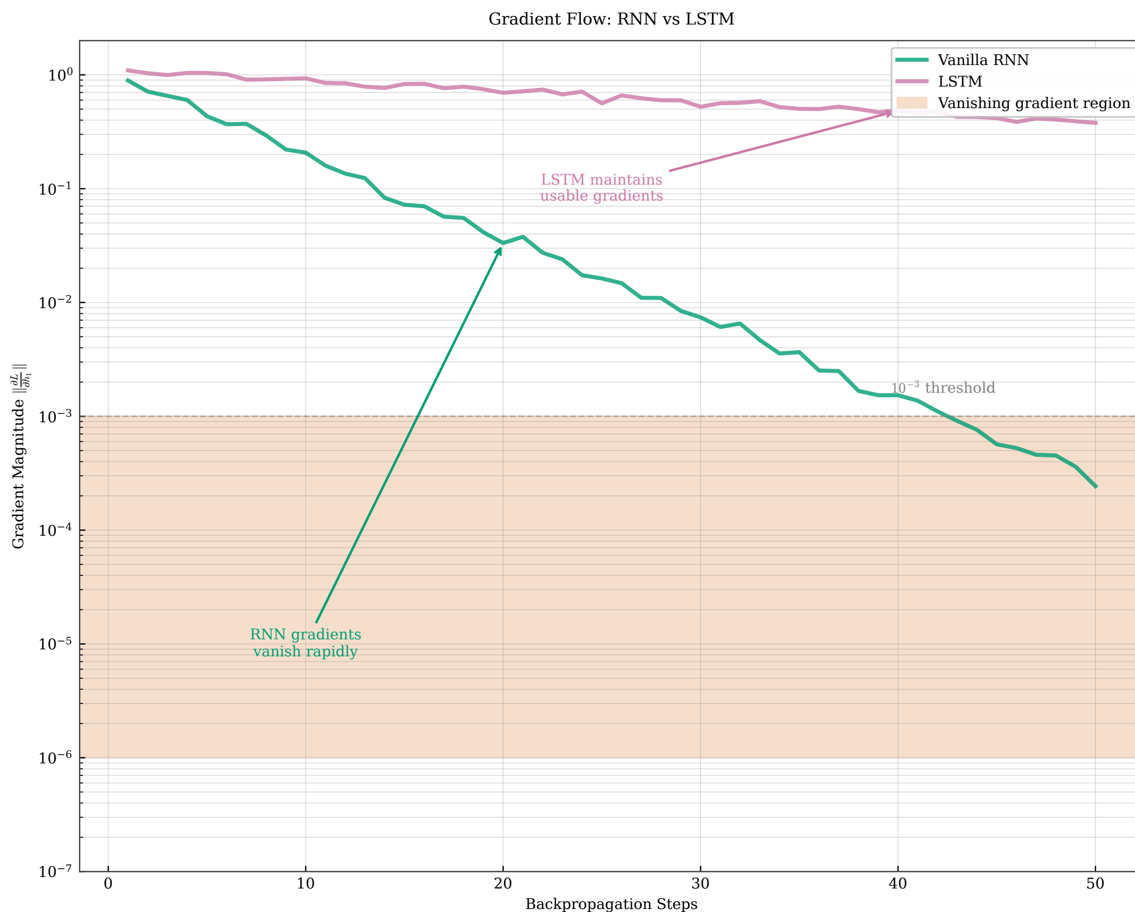


Figure 5.15: Gradient magnitude comparison: vanilla RNN versus LSTM. Both curves show gradient magnitude (log scale) as a function of temporal distance during backpropagation. The vanilla RNN (red) shows exponential decay, with gradients becoming negligible after 10-15 steps. The LSTM (green) maintains gradient magnitude across the full sequence, with the curve staying above the effective training threshold (dashed line) even at 50 steps. The shaded region indicates where vanilla RNN fails to learn due to vanishing gradients. LSTM's gradient highway enables learning dependencies of arbitrary length.

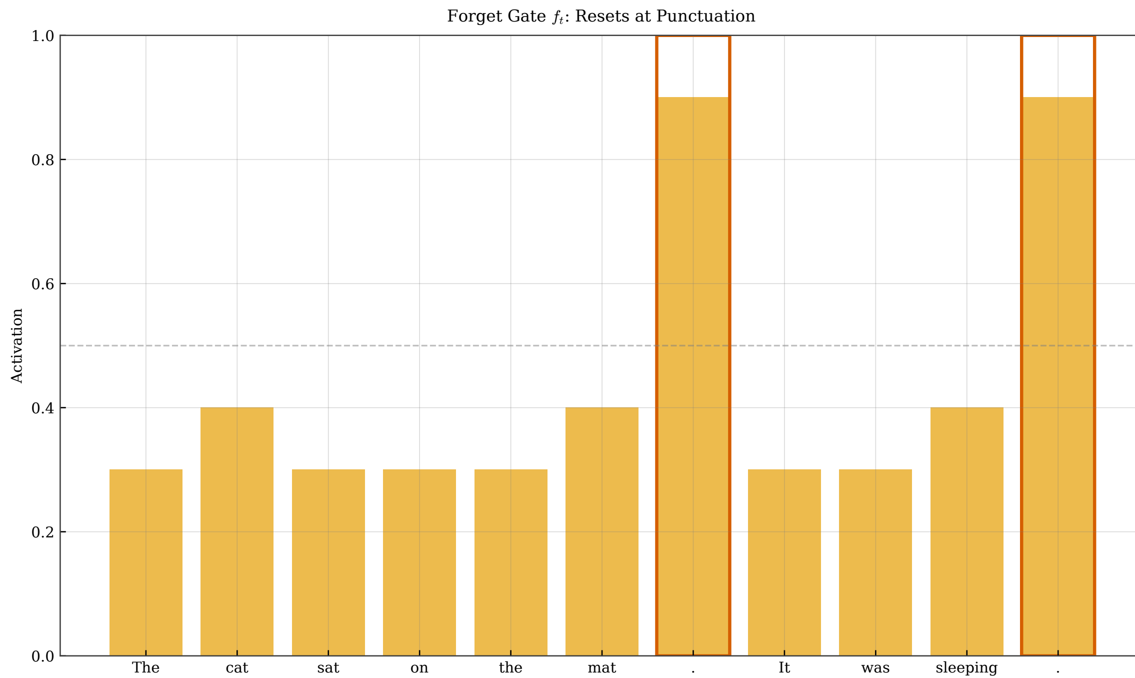


Figure 5.16: Learned gate patterns for different linguistic phenomena. Analysis of trained LSTM gates reveals interpretable patterns. The forget gate shows high activation at sentence boundaries and clause boundaries, resetting context for new units. The input gate shows high activation for content words (nouns, verbs) and low activation for function words (the, of, a). The output gate shows high activation before positions requiring prediction (verbs, objects) and lower activation elsewhere. These patterns emerge from training without explicit linguistic supervision, demonstrating that LSTMs learn linguistically meaningful representations.

5.4 GRU and Architecture Comparisons

While LSTM remains the most widely used gated recurrent architecture, the Gated Recurrent Unit (GRU), introduced by Cho et al. (2014) [Cho et al., 2014], offers a simpler alternative with comparable performance on many tasks. The GRU architecture combines the forget and input gates into a single update gate and merges the cell state and hidden state into a unified representation, reducing the number of parameters and computations per time step while retaining the essential gating mechanism that enables long-range memory. This simplification emerged from empirical observations that the full LSTM architecture, while powerful, may be over-parameterized for certain tasks, and that a more parsimonious design could achieve similar performance with reduced computational cost. This section examines the GRU architecture in detail and compares the three recurrent architectures—vanilla RNN, GRU, and LSTM—in terms of parameter efficiency, computational cost, and performance on language modeling tasks. The choice between GRU and LSTM is often empirical, depending on the specific task, dataset size, and computational constraints, though LSTM's explicit separation of cell state and hidden state can provide advantages for very long sequences where fine-grained memory control is beneficial. Both gated architectures represent major improvements over vanilla RNNs for capturing long-range dependencies, and understanding their similarities and differences informs architecture selection for practical applications in language modeling and other sequential tasks.

5.4.1 GRU Architecture

The GRU simplifies LSTM by using two gates instead of three and eliminating the separate cell state, resulting in a more compact architecture that is often easier to train and deploy. The update gate $\mathbf{z}_t = \sigma(\mathbf{W}_z[\mathbf{h}_{t-1};\mathbf{e}_t] + \mathbf{b}_z)$ controls how much of the previous hidden state to retain versus how much to update with new content, effectively combining the functions of LSTM's forget and input gates into a single mechanism. The reset gate $\mathbf{r}_t = \sigma(\mathbf{W}_r[\mathbf{h}_{t-1};\mathbf{e}_t] + \mathbf{b}_r)$ controls how much of the previous hidden state influences the candidate update, allowing the model to selectively ignore past information when computing new content. The candidate hidden state is computed as $\tilde{\mathbf{h}}_t = \tanh(\mathbf{W}_h[\mathbf{r}_t \odot \mathbf{h}_{t-1};\mathbf{e}_t] + \mathbf{b}_h)$, where the reset gate modulates the contribution of the previous hidden state before concatenation with the current input. Finally, the hidden state update $\mathbf{h}_t = (1-\mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t$ interpolates between keeping the old hidden state (when $\mathbf{z}_t \approx 0$) and fully adopting the candidate (when $\mathbf{z}_t \approx 1$). This interpolation mechanism provides the same gradient highway property as LSTM's forget gate: when $\mathbf{z}_t \approx 0$, the hidden state passes through unchanged, allowing gradients to flow backward without attenuation. The reset gate enables the model to "start fresh" when encountering a new context by setting $\mathbf{r}_t \approx 0$, which causes the candidate computation to ignore the previous hidden state. The GRU has approximately 25% fewer parameters than an LSTM with the same hidden dimension, since it computes three transformations (update, reset, candidate) versus LSTM's four (forget, input, candidate, output), and this efficiency advantage translates to faster training and inference times.

5.4.2 LSTM vs GRU Trade-offs

The choice between LSTM and GRU involves trade-offs along several dimensions, and understanding these trade-offs enables informed architecture selection for specific applications. In terms of parameters, GRU requires $3 \times (d_h \times (d_h + d) + d_h)$ parameters for its gates and candidate, while LSTM requires $4 \times (d_h \times (d_h + d) + d_h)$ for its four transformations. For typical values $d_h = 256$ and $d = 128$, LSTM has approximately 787K parameters versus GRU's 590K—a 25% reduction that can be significant for resource-constrained deployment. This parameter efficiency means GRU is less prone to overfitting on smaller datasets and faster to train, making it attractive when training data is limited or computational resources are constrained. From a computational perspective, GRU performs fewer matrix multiplications per time step, yielding 20-30% faster training and inference times while maintaining comparable accuracy on most benchmarks. However, LSTM's separate cell state provides a more direct gradient pathway through the additive cell state updates, which can be advantageous for very long sequences spanning hundreds of steps where gradient preservation is critical. The explicit output gate in LSTM also provides finer control over what information influences predictions versus what persists in memory, potentially enabling more nuanced memory management in complex tasks. Empirically, perfor-

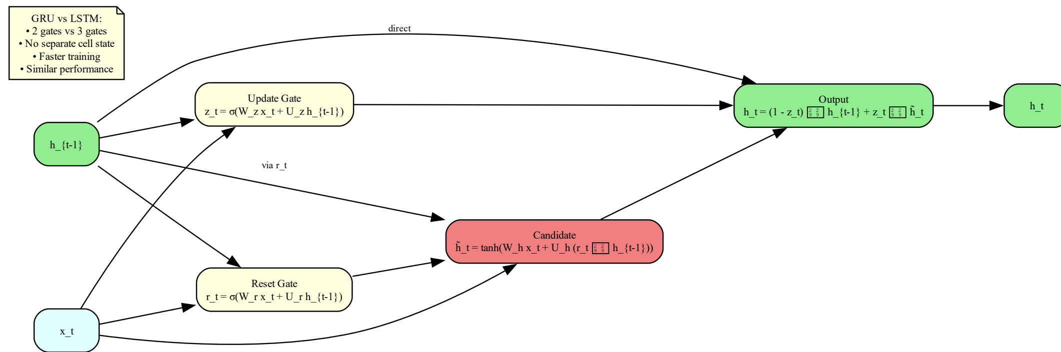


Figure 5.17: GRU cell architecture compared with LSTM. The GRU (left) has two gates: the update gate $\mathbf{z}_t$ that combines LSTM's forget and input functions, and the reset gate $\mathbf{r}_t$ that controls candidate computation. There is no separate cell state—the hidden state $\mathbf{h}_t$ serves both as output and as memory. The LSTM (right, simplified) maintains separate cell state $\mathbf{c}_t$ and hidden state $\mathbf{h}_t$ with three gates. The annotation highlights that GRU uses two gates versus LSTM's three, resulting in 25% fewer parameters for the same hidden dimension.

mance differences between LSTM and GRU are often within statistical noise on standard language modeling benchmarks such as Penn Treebank and WikiText. Both architectures achieve similar perplexity when hyperparameters are properly tuned, and the choice is often made based on computational budget (favoring GRU) or prior experience with the architecture. Some practitioners prefer LSTM for its interpretability: the separate cell state and explicit gates provide clearer semantics for understanding what the model has learned, facilitating analysis of learned representations. For our running example sentence, both LSTM and GRU successfully predict "was" by preserving subject information across the relative clause, though they accomplish this through slightly different gating mechanisms.

5.4.3 When to Use Which

Choosing among vanilla RNN, GRU, and LSTM depends on task requirements and constraints. Vanilla RNN should generally be avoided for language modeling due to its inability to capture dependencies beyond a few time steps; it may still be appropriate for tasks with very short sequences (fewer than 10 tokens) where computational efficiency is paramount. GRU is preferred when computational resources are limited, training data is modest (reducing overfitting risk with fewer parameters), or when the maximum dependency length is moderate (fewer than 50 tokens). LSTM is preferred when very long dependencies are expected (hundreds of tokens), when the separate cell state semantics aid interpretability, or when following established practices in a domain where LSTM has been extensively validated. In practice, the best approach is often to try both GRU and LSTM with proper hyperparameter tuning, as the optimal choice is dataset-dependent. Modern sequence models like Transformers (Chapter **??**) have largely superseded both for tasks where computational resources allow, but LSTM and GRU remain important for edge deployment, real-time applications, and understanding the historical development of neural language models.

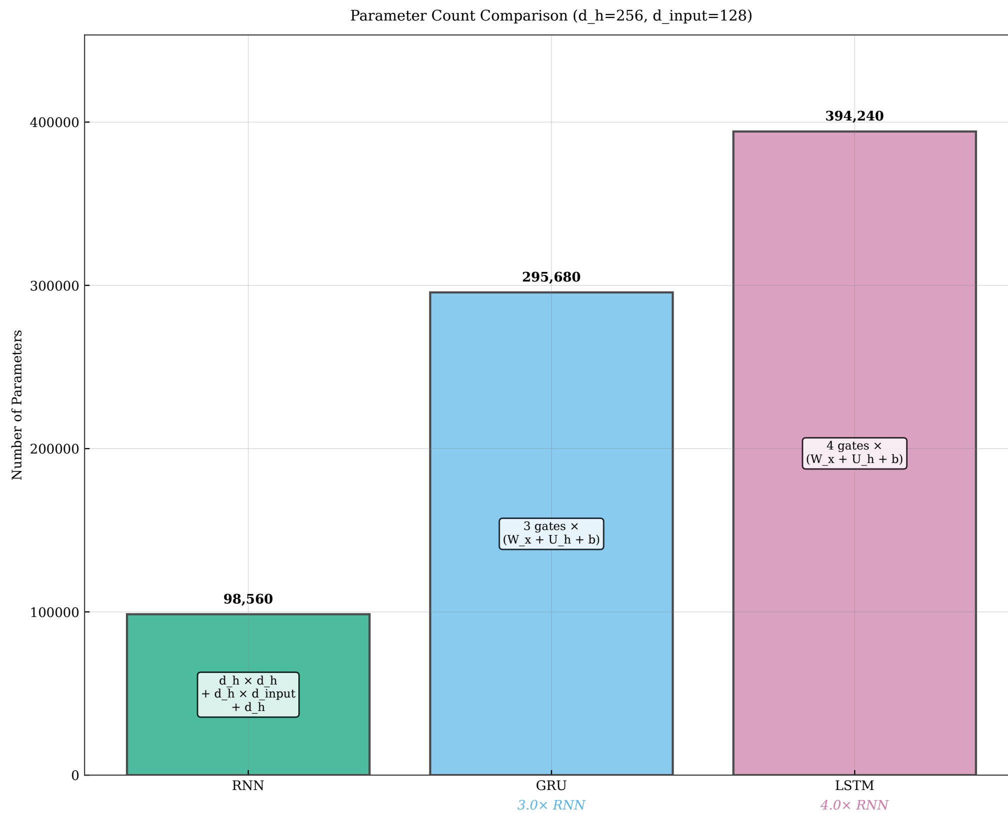


Figure 5.18: Parameter count comparison for recurrent architectures. For hidden dimension $d_h = 256$ and input dimension $d = 128$, the bar chart shows total parameters excluding the embedding and output layers. Vanilla RNN has fewest parameters (131K), GRU has moderate (590K), and LSTM has most (787K). The percentages indicate parameter count relative to LSTM: vanilla RNN is 17%, GRU is 75%. Despite having fewer parameters, GRU and LSTM dramatically outperform vanilla RNN on tasks requiring long-range memory, demonstrating that architectural innovations matter more than raw parameter count for sequence modeling.

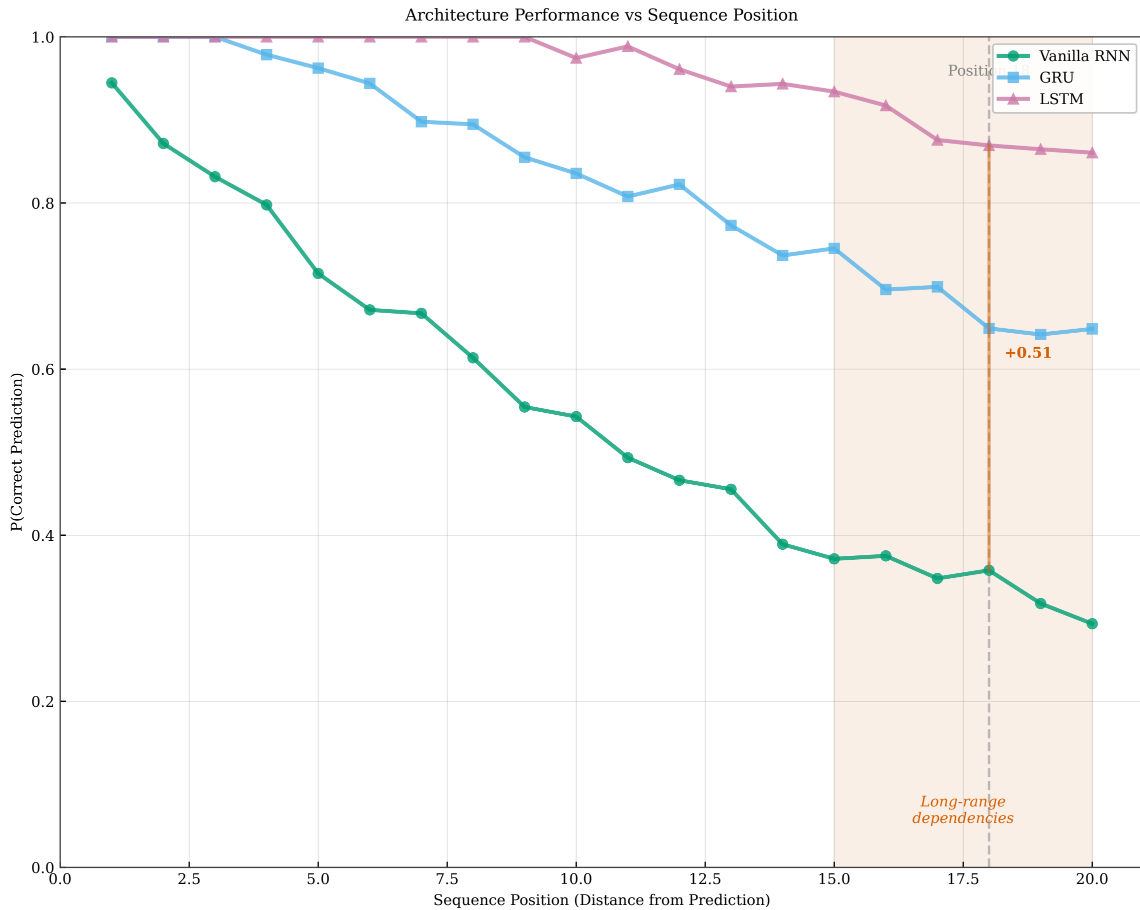


Figure 5.19: Performance comparison on running example. The plot shows prediction confidence $P(\text{was} \mid \cdot)$ at each position for vanilla RNN (red), GRU (cyan), and LSTM (orange). All architectures perform similarly for short-range predictions (positions 1-5). As sequence length increases, vanilla RNN performance degrades rapidly, producing near-random predictions (50%) by position 18. Both GRU and LSTM maintain high confidence in the correct prediction "was" throughout the sequence, with LSTM showing marginally higher confidence. The vertical dashed line at position 18 highlights the critical prediction where long-range dependency matters.

5.5 Training RNNs for Language Modeling

Training recurrent networks for language modeling requires adapting the standard supervised learning framework to the unique characteristics of sequential data, where each prediction depends on an evolving hidden state that encodes context from all preceding words. The training objective is next-word prediction: given a sequence of words $w_1, \ldots, w_T$, the model should maximize $\prod_{t=1}^{T} P(w_t \mid w_1, \ldots, w_{t-1})$, or equivalently minimize the negative log-likelihood (cross-entropy loss). This objective aligns perfectly with the language modeling task: at each position, the model receives a training signal indicating how well it predicted the actual next word, and this signal drives parameter updates that improve future predictions. Gradient computation requires backpropagation through time (BPTT), which unrolls the recurrence relation and computes gradients through the entire computational graph spanning multiple time steps. For long sequences containing hundreds or thousands of tokens, full BPTT becomes computationally expensive and memory-intensive, motivating truncated BPTT that limits the backward pass to a fixed window while maintaining the forward context. This section examines the training objective in detail, the mechanics of BPTT gradient computation, and practical considerations including gradient clipping, learning rate scheduling, and efficient sequence batching. Understanding these training details is essential for successfully applying recurrent networks to language modeling tasks and for interpreting the behavior and limitations of trained models in practice.

5.5.1 The Language Modeling Objective

The language modeling objective directly aligns with next-word prediction, making it a natural fit for training recurrent architectures. Given a corpus $\mathscr{D} = [w_1, \ldots, w_T]$, we seek to maximize the joint probability $P(w_1, \ldots, w_T)$, which by the chain rule of probability factorizes as $\prod_{t=1}^{T} P(w_t \mid w_1, \ldots, w_{t-1})$. In practice, we minimize the average negative log-likelihood, also known as cross-entropy loss: $\mathscr{L} = -\frac{1}{T}\sum_{t=1}^{T} \log P(w_t \mid \mathbf{h}_{t-1})$, where $\mathbf{h}_{t-1}$ is the hidden state encoding the context $w_1, \ldots, w_{t-1}$, and the probability is computed through the output layer as $P(w_t \mid \mathbf{h}_{t-1}) = \text{softmax}(\mathbf{W}_{hy}\mathbf{h}_{t-1} + \mathbf{b}_y)[w_t]$. This loss measures the cross-entropy between the model's predicted distribution over vocabulary and the one-hot target distribution indicating the actual next word. The perplexity, a more interpretable metric commonly reported in language modeling research, is the exponentiated average loss: $\text{PPL} = \exp(\mathscr{L}) = \exp\left(-\frac{1}{T}\sum_{t=1}^{T} \log P(w_t \mid \mathbf{h}_{t-1})\right)$. Perplexity can be interpreted as the effective number of equally likely words the model considers at each position: a perplexity of 100 means the model's uncertainty is equivalent to choosing uniformly among 100 words, while lower perplexity indicates more confident and accurate predictions. State-of-the-art LSTM language models achieve perplexities of 50-70 on standard benchmarks like Penn Treebank, representing substantial improvements over n-gram baselines. The training procedure employs stochastic gradient descent (SGD) or variants like Adam, computing gradients of the loss with respect to all parameters and updating parameters in the direction that decreases loss, with gradient computation requiring the BPTT algorithm we examine next.

5.5.2 Backpropagation Through Time

Backpropagation through time (BPTT) applies the chain rule to compute gradients through the unrolled recurrent network, treating the unrolled sequence as a deep feedforward network with shared weights. Consider the gradient of the loss at position t with respect to the hidden-to-hidden weights $\mathbf{W}_{hh}$, which are used at every time step. This gradient depends on how $\mathbf{W}_{hh}$ affects $\mathbf{h}_t$, which depends on $\mathbf{h}_{t-1}$, which depends on $\mathbf{h}_{t-2}$, and so on back to $\mathbf{h}_1$—creating a chain of dependencies spanning the entire sequence. The full gradient accumulates contributions from all time steps: $\frac{\partial \mathscr{L}}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^{T}\sum_{k=1}^{t} \frac{\partial \mathscr{L}_t}{\partial \mathbf{h}_t}\left(\prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}\right)\frac{\partial \mathbf{h}_k}{\partial \mathbf{W}_{hh}}$, where the product term $\prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}$ propagates gradients backward through time from position t to position k. For vanilla RNNs, this product causes vanishing or exploding gradients because it involves repeated multiplication by the Jacobian matrix; for LSTMs, the cell state provides an alternative pathway where the products are closer to identity matrices. The memory cost of BPTT is $O(T \cdot d_h)$ to store all intermediate hidden states and activations needed for the backward pass. For very long sequences with $T > 1000$ tokens, this memory requirement becomes prohibitive, often exceeding available GPU memory. Computation time is $O(T)$ for both forward and backward

passes, with the constant factor including matrix multiplications at each step. The strictly sequential nature of RNNs means that neither forward nor backward computation can be parallelized across time steps, limiting training throughput compared to architectures like Transformers that process all positions in parallel.

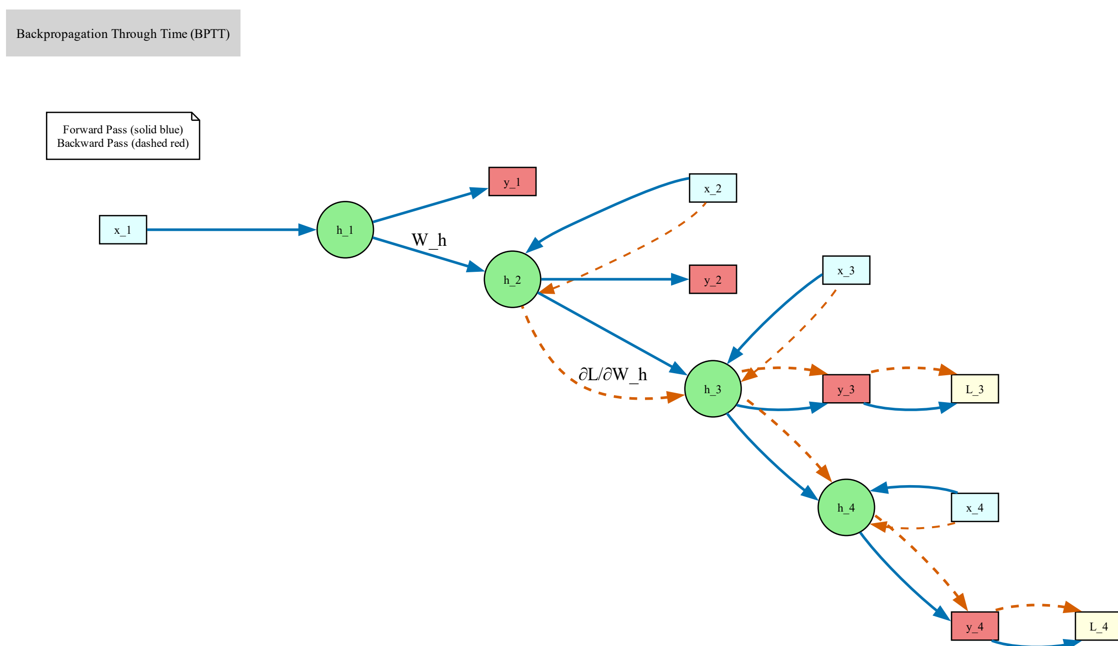


Figure 5.20: Backpropagation through time visualization. The diagram shows the unrolled RNN with forward pass (blue arrows) and backward pass (red arrows). During the forward pass, hidden states are computed left-to-right and stored in memory. During the backward pass, gradients flow right-to-left through the stored hidden states. At each position, the loss gradient $\frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t}$ enters from the output layer, then propagates backward through the chain of hidden states. Gradients accumulate at shared parameters (weights, biases) across all time steps, requiring summation over the entire sequence.

5.5.3 Practical Training Considerations

Several practical techniques are essential for successfully training recurrent language models. Truncated BPTT limits the backward pass to K steps rather than the full sequence, reducing memory from $O(T)$ to $O(K)$ and enabling training on arbitrarily long sequences. The forward pass still processes the entire sequence to build correct hidden states, but gradients are only computed for the most recent K steps. Typical values are $K \in [35, 100]$; smaller values risk missing long-range dependencies, while larger values increase memory and computation. Hidden states are carried forward across truncation boundaries, so the model still sees full context during forward passes. Gradient clipping prevents exploding gradients by scaling the gradient vector when its norm exceeds a threshold: if $\|\nabla\mathcal{L}\| > \tau$, set $\nabla\mathcal{L} \leftarrow \tau \cdot \nabla\mathcal{L}/\|\nabla\mathcal{L}\|$. Typical thresholds are $\tau \in [1, 5]$. This ensures stable training without limiting the model's ability to learn long-range dependencies (which require gradients to be preserved, not prevented from growing). Learning rate scheduling is critical: starting with a high learning rate enables fast initial progress, then decaying the rate allows fine-tuning. Common schedules include step decay (reduce by factor after fixed epochs) and cosine annealing. Regularization through dropout (applied to non-recurrent connections) and weight decay prevents overfitting on smaller corpora.

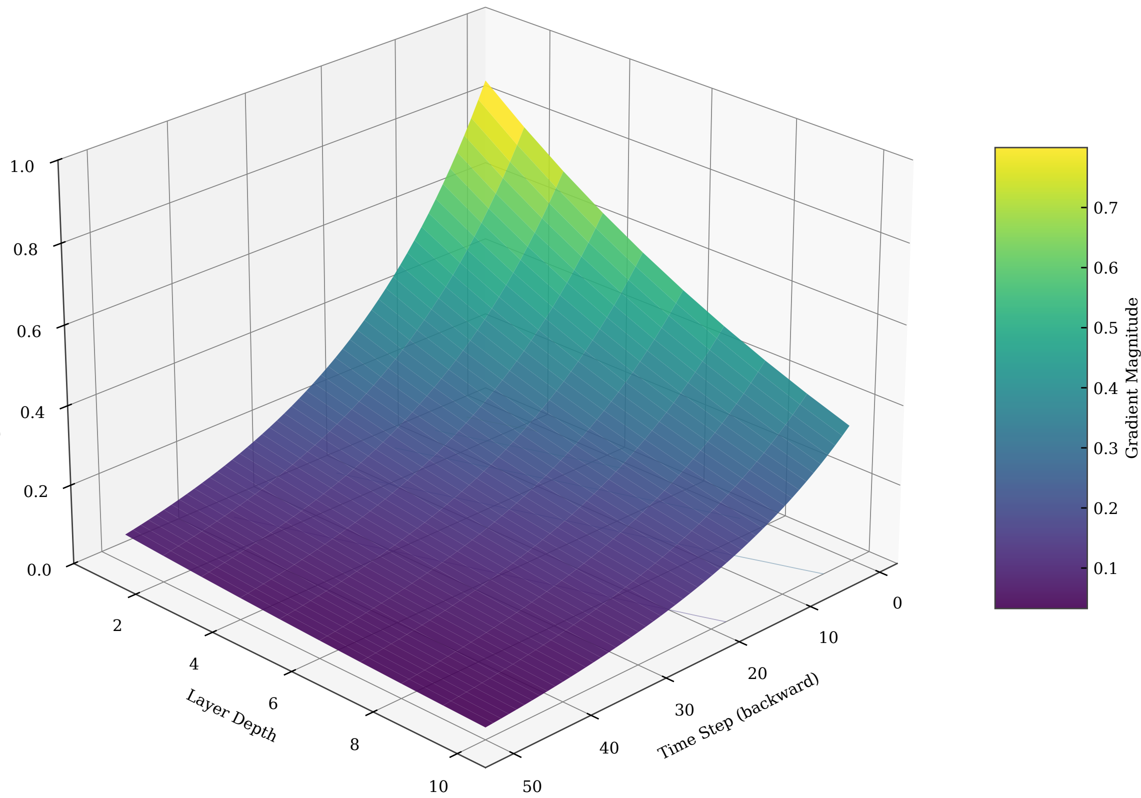


Figure 5.21: Gradient magnitude surface over time and layer depth. For a stacked recurrent network, the 3D surface shows how gradient magnitude varies across time steps (x-axis) and layers (y-axis). Gradients are strongest near the output (recent time, top layer) and weaken toward the input (early time, bottom layer). The surface for vanilla RNN (not shown) would decay much more steeply. This LSTM gradient surface shows that significant gradient signal reaches early time steps and bottom layers, enabling learning of long-range dependencies across the full network depth.

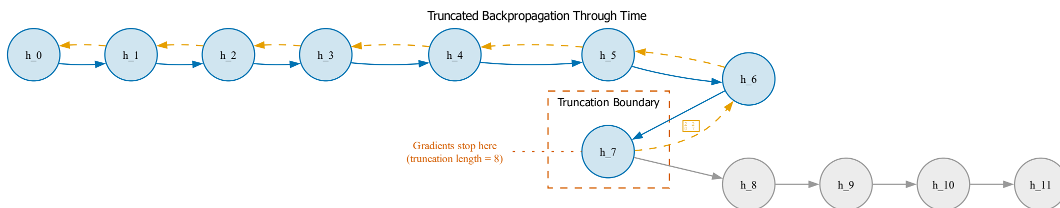


Figure 5.22: Truncated backpropagation through time. For a long sequence, the forward pass (blue) processes all time steps, accumulating hidden state information. The backward pass (red) is truncated to a window of K steps: gradients are computed within this window, then detached. The next forward segment starts from the hidden state at the truncation boundary, preserving context continuity. This approach trades off gradient fidelity (missing contributions from beyond K steps) for computational efficiency. The dashed vertical line shows where gradients stop, with the annotation indicating the trade-off between efficiency and long-range learning.

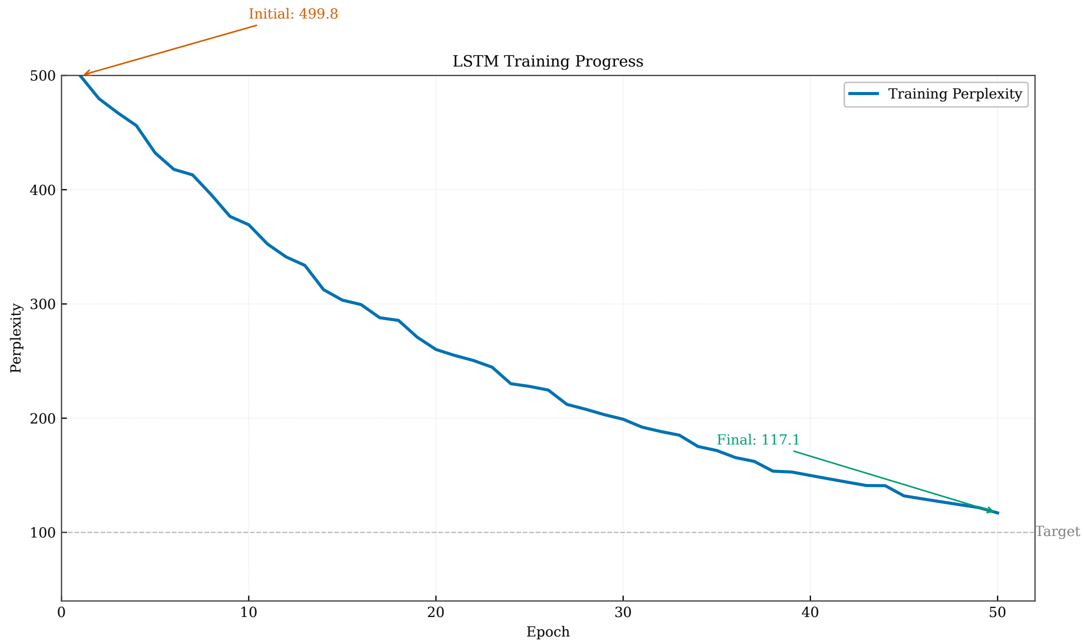


Figure 5.23: Training loss curve for LSTM language model. Perplexity on the training set (blue) decreases from approximately 450 (near random for a 50K vocabulary) to below 70 over 50 epochs. The curve shows rapid initial improvement as the model learns basic word frequencies and short-range patterns, then slower improvement as it captures longer-range dependencies. The validation curve (orange) typically runs slightly higher, with the gap indicating overfitting degree. Training is stopped when validation perplexity stops improving (early stopping) to prevent overfitting.

5.6 Context Representation in RNNs

Having developed recurrent architectures for sequential processing, we now examine how RNNs represent context compared to previous approaches. Each chapter in this textbook advances context representation: n-grams use discrete tuples, embeddings use static vectors, and RNNs use dynamic hidden states. This section synthesizes these progressions and identifies the limitations that motivate attention mechanisms in Chapter **??**. The hidden state $\mathbf{h}_t$ serves as the context representation in RNNs—a learned, fixed-size summary of the entire sequence history $w_1, \ldots, w_t$. Unlike n-gram contexts that are limited to fixed windows, the hidden state can theoretically encode information from arbitrarily far back in the sequence. Unlike static embeddings that ignore context entirely, the hidden state adapts based on the specific sequence of words that preceded the current position. However, the hidden state representation has fundamental limitations: it must compress variable-length history into a fixed-size vector, creating an information bottleneck for very long sequences. Additionally, the sequential processing means that information must pass through many intermediate states to travel from early positions to later ones, even when a direct connection would be more appropriate. These limitations point toward the attention mechanism, which allows the model to directly access any position in the input sequence rather than relying on a single compressed representation.

How This Chapter Represents Context:

- **Context representation:** The hidden state $\mathbf{h}_t \in \mathbb{R}^{d_h}$ encodes variable-length history in a fixed-size vector
- **Context encoding:** Sequential processing builds context incrementally through learned recurrence relations
- **Key advance:** Dynamic representations that evolve with each word, capturing long-range dependencies through gating
- **Limitation:** Compression bottleneck forces all history through a fixed-size vector; Chapter **??** introduces attention for direct access to all positions

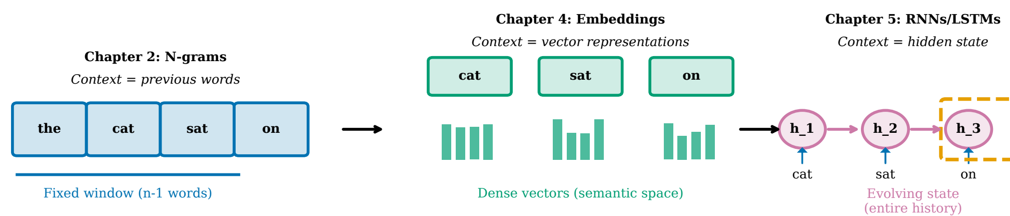


Figure 5.24: Evolution of context representation across chapters. Chapter 2 (n-grams) represents context as a discrete tuple of previous words, limited to a fixed window and unable to generalize across similar contexts. Chapter 4 (embeddings) represents each word as a continuous vector but ignores sequential context entirely. Chapter 5 (RNNs) represents context as a dynamic hidden state that evolves through the sequence, encoding variable-length history in a fixed-size vector. Each representation advances the expressiveness and generalization capability for next-word prediction.

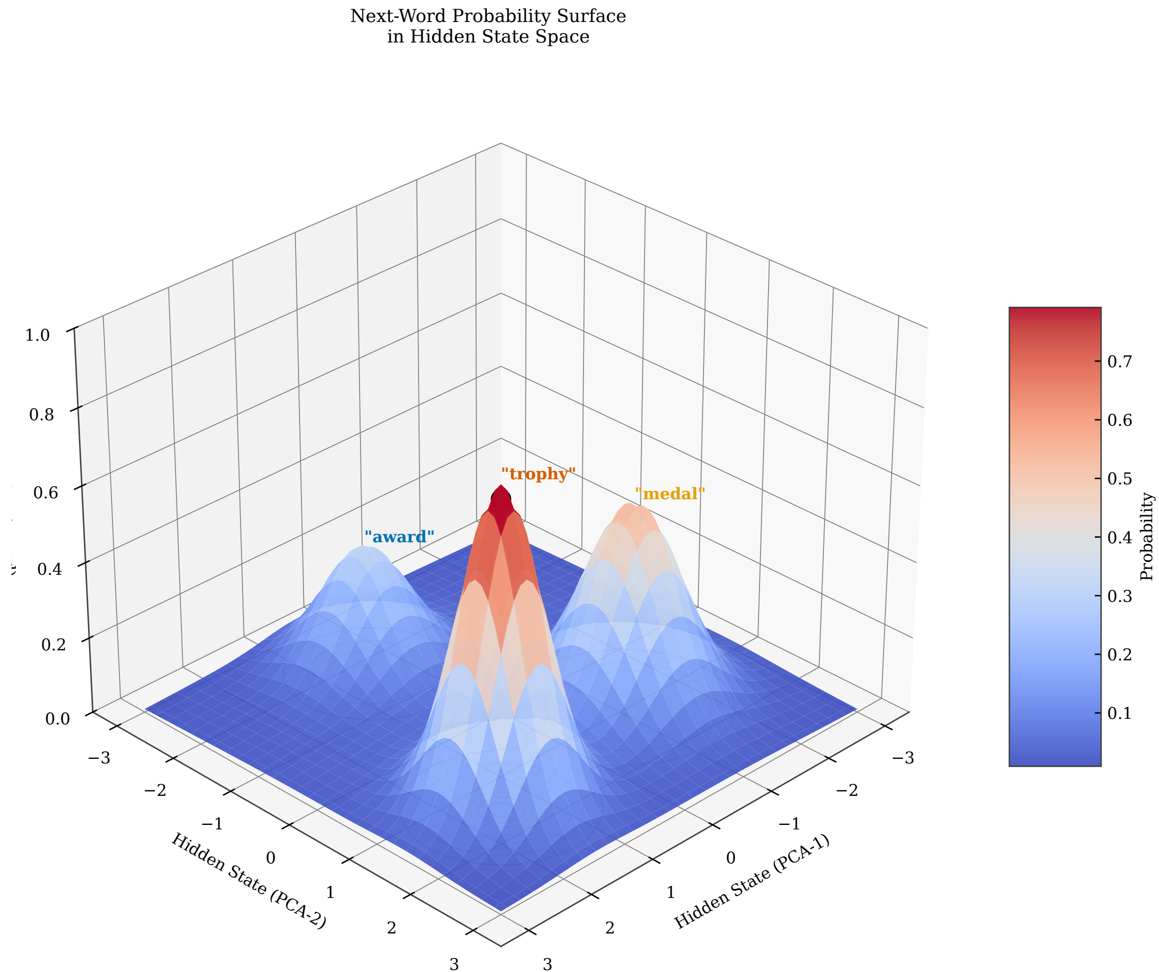


Figure 5.25: Prediction probability surface over hidden state space. The 3D surface shows how $P(w_{t+1} \mid \mathbf{h}_t)$ varies across a 2D projection (via PCA) of the hidden state space. Peaks correspond to hidden state regions that strongly predict specific next words (labeled). The surface topology reveals that the hidden state encodes predictive structure: similar hidden states lead to similar prediction distributions, and distinct semantic contexts occupy distinct regions. This visualization demonstrates that the hidden state successfully organizes contextual information for next-word prediction.

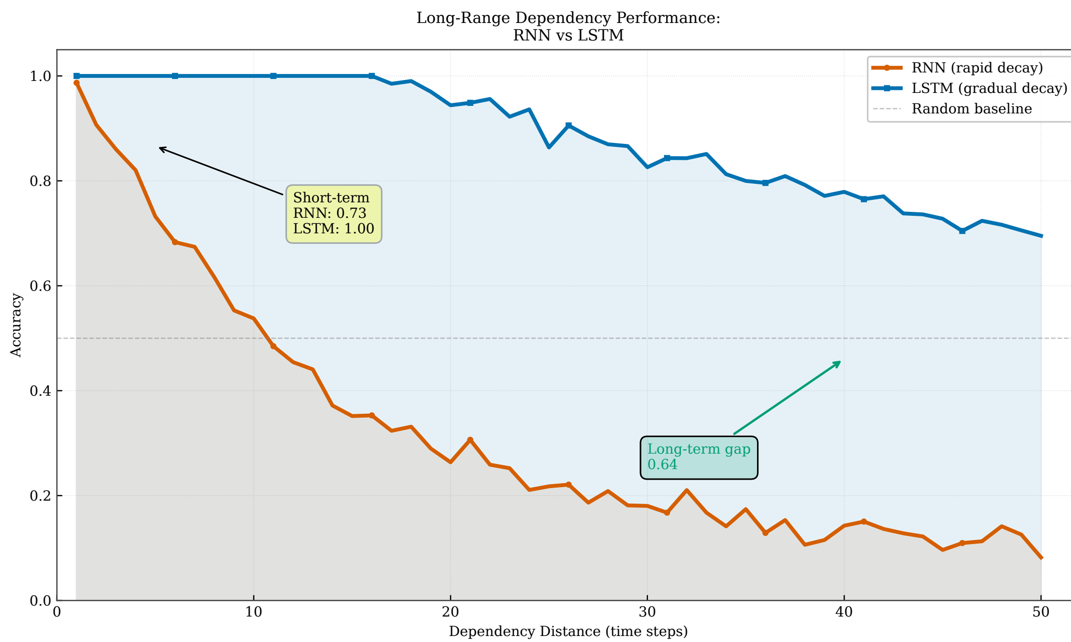


Figure 5.26: Effect of dependency distance on prediction accuracy. The plot shows accuracy of predicting a dependent word as a function of the number of words between the dependent elements. Vanilla RNN (red) shows rapid accuracy decay, dropping below 60% by distance 10 and approaching chance (50%) by distance 20. LSTM (green) maintains higher accuracy across all distances, staying above 70% even at distance 30. This difference quantifies the long-range memory advantage of gated architectures over vanilla RNNs.

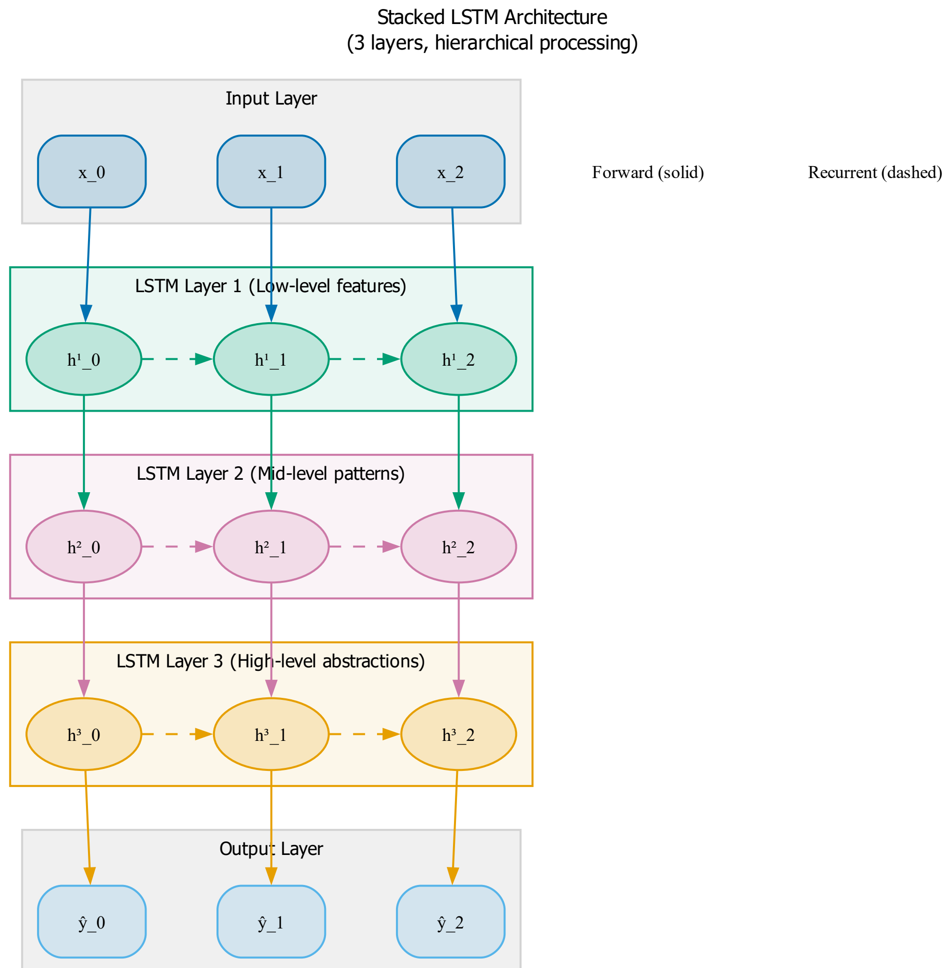
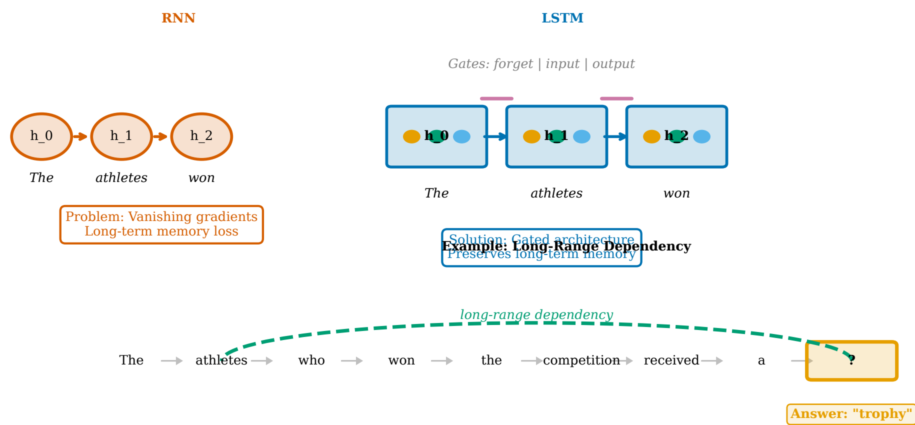


Figure 5.27: Stacked LSTM architecture for hierarchical representation. Multiple LSTM layers are stacked vertically, with the hidden state output of each layer serving as input to the next. The bottom layer processes raw word embeddings and captures low-level features (syntax, local patterns). Middle layers capture phrase-level and clause-level structure. Top layers capture discourse-level and long-range semantic patterns. This hierarchical organization, learned through training, enables richer context representation than single-layer architectures. Dropout is applied between layers to prevent overfitting.

Chapter 5 Summary: RNNs and LSTMs for Next-Word Prediction



Key Takeaways:

1. RNNs process sequences with recurrent hidden states
2. LSTMs solve vanishing gradients via gated architecture
3. Gates control information flow: forget, input, output
4. Cell state preserves long-term context
5. Enables prediction from distant dependencies

Figure 5.28: Visual summary of Chapter 5 key concepts. The diagram synthesizes the chapter's main ideas: sequential processing with hidden states (left), the LSTM gating mechanism that enables long-range memory (center), and the running example demonstrating successful long-range dependency handling (right). The key insight is that gated architectures solve the vanishing gradient problem through additive cell state updates, enabling models to learn dependencies spanning dozens or hundreds of time steps. This capability is essential for effective language modeling but still relies on compressing history into a fixed-size vector.

We can now predict better because:

- **Variable-length context:** Hidden states encode entire sequence history, not just fixed windows, enabling context-dependent predictions regardless of sequence length
- **Long-range dependencies:** LSTM gates preserve relevant information across many time steps by creating gradient highways through the cell state
- **Dynamic representations:** The same word receives different hidden state representations based on its sequential context, enabling disambiguation of polysemy
- **Foundation for sequence modeling:** Recurrent architectures provide the conceptual foundation for understanding modern sequence models, including Transformers

Next: Chapter **??** introduces the attention mechanism, which overcomes the sequential bottleneck by allowing the model to directly access any position in the input sequence. Rather than compressing all history into a single hidden state, attention computes weighted combinations of all past representations, enabling efficient parallel computation and more effective long-range dependency modeling. This leads to the Transformer architecture, which has become the foundation of modern large language models.

Exercises

1. **Forward pass calculation.** Given a 3-word sequence with embeddings $\mathbf{e}_1 = [1,0]$, $\mathbf{e}_2 = [0,1]$, $\mathbf{e}_3 = [1,1]$, initial hidden state $\mathbf{h}_0 = [0,0]$, weights $\mathbf{W}_{xh} = \begin{pmatrix} 0.5 & 0.3 \\ 0.2 & 0.4 \end{pmatrix}$, $\mathbf{W}_{hh} = \begin{pmatrix} 0.1 & 0.2 \\ 0.3 & 0.1 \end{pmatrix}$, and bias $\mathbf{b}_h = [0,0]$, compute $\mathbf{h}_1$, $\mathbf{h}_2$, and $\mathbf{h}_3$ for a vanilla RNN.
2. **Hidden dimension trade-offs.** Explain the trade-off between hidden state dimension d_h and model capacity. How does doubling d_h affect (a) the number of parameters, (b) computational cost per time step, and (c) the model's ability to memorize training data versus generalize?
3. **LSTM gate activation patterns.** For the sentence "The cat sat on the mat", predict which of the three LSTM gates (forget, input, output) would have highest activation when processing the word "sat". Justify your prediction based on the linguistic function of each gate.
4. **Forget gate interpretation.** In the sentence "The trophy that the athletes won was large", what forget gate value (high or low) should the model learn for the "trophy" information while processing the relative clause "that the athletes won"? Explain how this enables correct prediction of "was".
5. **Cell state update calculation.** Given forget gate $\mathbf{f}_t = [0.9, 0.3]$, input gate $\mathbf{i}_t = [0.2, 0.8]$, previous cell state $\mathbf{c}_{t-1} = [1.0, 0.5]$, and candidate $\tilde{\mathbf{c}}_t = [0.0, 1.0]$, compute the new cell state $\mathbf{c}_t$. Explain what this update accomplishes in terms of preserving old information versus incorporating new information.
6. **Parameter count comparison.** For hidden dimension $d_h = 256$ and input dimension $d = 128$, calculate the exact number of parameters (weights and biases) for (a) vanilla RNN, (b) GRU, and (c) LSTM. Show your work and express each as a percentage of LSTM's parameter count.
7. **BPTT gradient flow.** Explain why the gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{h}_1}$ (gradient at the first hidden state) is typically much smaller in magnitude than $\frac{\partial \mathcal{L}}{\partial \mathbf{h}_T}$ (gradient at the final hidden state) for a vanilla RNN. How does LSTM's cell state address this problem?
8. **Truncated BPTT trade-offs.** A language model is trained with truncated BPTT using window size $K = 20$. What is lost by not backpropagating beyond 20 steps? Describe a specific example of a dependency that would be missed by this truncation.

9. **Long-range dependency analysis.** Identify three English constructions besides subject-verb agreement that create long-range dependencies (where words separated by 10 or more tokens must agree or relate). For each, explain what information must be preserved and how LSTM's gating mechanism could accomplish this.

10. **Perplexity interpretation.** If an n-gram model achieves perplexity 150 and an LSTM achieves perplexity 80 on the same test set, quantify the improvement. In terms of "effective vocabulary size" at each prediction, how much more certain is the LSTM about its predictions?

11. **Bidirectional RNNs.** ($\star$) In a bidirectional RNN, two RNNs process the sequence in opposite directions and their hidden states are concatenated. How does this change the context representation $\mathbf{h}_t$? What types of predictions could benefit from bidirectional context that unidirectional (left-to-right) RNNs cannot capture?

12. **Stacked architectures.** ($\star$) Describe how stacking multiple LSTM layers creates hierarchical representations. What might each layer in a 3-layer stacked LSTM learn to represent (give specific examples for language modeling)? How does depth trade off against width (hidden dimension) for fixed parameter budget?

Bibliography

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

Index

backpropagation through time, *see* BPTT
bag-of-words, 1
bidirectional RNN, 33
BPTT, 23, 33

cell state, 10, 33
context representation, 33
cross-entropy loss, 23, 33

exploding gradient, 33

forget gate, 10, 33

Gated Recurrent Unit, *see* GRU
gating mechanism, 10, 33
gradient clipping, 23, 24, 33
gradient highway, 11, 33
GRU, 19, 33

hidden state, 1, 33

input gate, 10, 33

language modeling, 33
Long Short-Term Memory, *see* LSTM
LSTM, 10, 33

n-gram, 1

output gate, 10, 33

parameter sharing, 6, 33
perplexity, 23, 33

recurrent neural network, 1, 6, *see* RNN
reset gate, 19, 33
RNN, 33
RNN cell, 6
running example, 3

sequential processing, 2, 33
short memory problem, 9

truncated BPTT, 23, 33

unrolling, 6, 33
update gate, 19, 33

vanishing gradient, 6, 9, 33

word embeddings, 1