Chapter 5: Sequence Models – RNNs, LSTMs, and GRUs

Learning Objectives

After reading this chapter, the reader should be able to:

1. Describe the forward pass of a vanilla RNN and explain how it processes sequences of variable length through shared parameters and recurrent hidden states.
2. Diagnose the vanishing gradient problem mathematically by analyzing the gradient flow through time and explain why it prevents learning long-range dependencies.
3. Trace the information flow through an LSTM cell, identifying how the forget, input, and output gates regulate the cell state to preserve gradients over long sequences.
4. Train an RNN-based neural language model and compare its perplexity against n-gram baselines on the same corpus.

In Chapter 4, we learned to represent words as dense vectors where similar words occupy nearby points. These embeddings solve the generalization problem of n-grams – "dog" and "puppy" now live in the same neighborhood, and learning something about one automatically transfers to the other. But embeddings alone do not tell us how to process a sequence. Given the embeddings for "the," "cat," "sat," "on," "the," how do we combine them into a contextual representation that supports predicting the next word? The n-gram models of Chapter 3 use a fixed window: a trigram model sees only the two most recent words and must make its prediction from that sliver of history. We measured the cost of this limitation in Section 3.4, where even the best Kneser-Ney smoothed models failed to capture dependencies that spanned more than a few tokens. The sentence "The author who wrote the novel that was banned in several countries and sparked international debate was ultimately awarded" requires knowledge of the distant subject "author" to predict the verb, but no fixed-size window short of the full prefix can reach back far enough. What if we could build a model that remembers everything it has read so far – a model whose context window is, in principle, the entire sequence from the first word to the present?

That is the promise of recurrent neural networks. This chapter introduces the first architecture capable of conditioning on unbounded context when predicting the next word. We begin with the vanilla RNN and its elegant recurrence relation, which updates a hidden state at every time step to maintain a running summary of the sequence. We then confront the architecture's fatal weakness – the vanishing gradient problem – which prevents vanilla RNNs from learning the long-range dependencies they were designed to capture. The LSTM, introduced by Hochreiter and Schmidhuber [1997], solves this problem through a gated architecture that provides a "gradient highway" through the cell state, and the GRU of Cho et al. [2014] offers a streamlined alternative. By the chapter's end, we will have built a complete neural language model – embedding layer from Chapter 4, LSTM from this chapter, softmax output from Chapter 2 – and demonstrated that it roughly halves the perplexity of the best n-gram baselines from Chapter 3.

5.1 Vanilla RNNs

5.1.1 The Recurrence Relation

How can a neural network maintain a running summary of everything it has read so far?

Consider the sentence "The cat sat on the mat." A trigram model predicting the word after "on" sees only "sat on" and must guess from there. A human reader, by contrast, remembers that a cat is involved, that it sat somewhere, and that the sentence seems to be describing a domestic scene. The human maintains a running mental summary that grows richer with each word. A recurrent neural network does the same thing, and the mechanism is captured in a single equation.

At each time step t, the RNN takes two inputs: the current word's embedding $\mathbf{x}_t \in \mathbb{R}^d$ (produced by the embedding layer from Chapter 4) and the previous hidden state $\mathbf{h}_{t-1} \in \mathbb{R}^h$, which summarizes everything the network has seen up to time $t-1$. From these two inputs, it computes a new hidden state:

$$\mathbf{h}_t = \tanh(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}_h) \tag{5.1}$$

The matrix $\mathbf{W}_{hh} \in \mathbb{R}^{h\times h}$ is the *recurrent weight matrix* – it transforms the previous hidden state, determining how the network's memory of past inputs influences its current computation. The matrix $\mathbf{W}_{xh} \in \mathbb{R}^{h\times d}$ is the *input weight matrix* – it transforms the current word embedding, incorporating new information into the hidden state. The bias vector $\mathbf{b}_h \in \mathbb{R}^h$ provides an offset term. The tanh nonlinearity squashes the result into the range $[-1, 1]$ element-wise, preventing the hidden state from growing without bound and introducing the nonlinearity that allows the network to learn complex functions of its input history. At time $t = 0$, before any words have been processed, the hidden state is initialized to the zero vector: $\mathbf{h}_0 = \mathbf{0}$. This is equivalent to saying that the network starts with no memory, no prior context, and no opinion about what word might come next.

The recurrence relation in Equation (5.1) is deceptively simple, but its implications are profound. By applying the same function recursively – feeding each step's output as the next step's input – the network can in principle compress an arbitrarily long sequence into a fixed-size hidden state vector. After processing a 100-word sentence, $\mathbf{h}_{100}$ is still a vector in $\mathbb{R}^h$, the same size as $\mathbf{h}_1$. All the information about those 100 words must be encoded in those h numbers. Whether the network actually succeeds in preserving distant information is another question entirely, one that will occupy us in Section 5.2. For now, the important point is the structural one: the RNN's hidden state provides a mechanism for variable-length context that n-gram models fundamentally lack.

A concrete walkthrough makes the mechanics tangible. Suppose we have an RNN with hidden size $h = 2$ and input size $d = 2$, with weight matrices $\mathbf{W}_{hh} = \begin{pmatrix} 0.5 & 0.1 \\ 0.2 & 0.3 \end{pmatrix}$, $\mathbf{W}_{xh} = \begin{pmatrix} 0.4 & 0.2 \\ 0.1 & 0.3 \end{pmatrix}$, and $\mathbf{b}_h = \mathbf{0}$. Starting from $\mathbf{h}_0 = (0, 0)^\top$, when we feed the input $\mathbf{x}_1 = (1, 0)^\top$, we compute $\mathbf{W}_{hh}\mathbf{h}_0 + \mathbf{W}_{xh}\mathbf{x}_1 = (0.4, 0.1)^\top$, and applying tanh gives $\mathbf{h}_1 = (0.380, 0.100)^\top$ (rounded). At the second step, with input $\mathbf{x}_2 = (0, 1)^\top$, the computation becomes $\mathbf{W}_{hh}\mathbf{h}_1 + \mathbf{W}_{xh}\mathbf{x}_2 = (0.200 + 0.200, 0.076 + 0.300)^\top = (0.400, 0.376)^\top$, yielding $\mathbf{h}_2 = \tanh(0.400, 0.376)^\top = (0.380, 0.360)^\top$. Notice that $\mathbf{h}_2$ carries traces of both $\mathbf{x}_1$ and $\mathbf{x}_2$: the first input influenced $\mathbf{h}_1$, which in turn influenced $\mathbf{h}_2$ through the recurrent connection. This is how sequential memory works in the RNN – not by storing a list of past inputs, but by folding each new input into an evolving summary.

5.1.2 Parameter Sharing across Time Steps

One of the most consequential design decisions in the RNN is that the same weight matrices $\mathbf{W}_{hh}$, $\mathbf{W}_{xh}$, and $\mathbf{b}_h$ are reused at every time step. This stands in sharp contrast to both n-gram models and feedforward neural language models, and the difference matters far more than it might first appear. An n-gram model assigns separate parameters to each context pattern: the probability of "mat" given "on the" is stored independently of the probability of "mat" given "sat the," and no parameter is shared between them. The number of parameters grows exponentially with the context length, which is precisely why n-gram models cannot scale to long contexts. A feedforward neural language model, such as Bengio et al.'s [2003] architecture, assigns separate weight matrices to each input position within a fixed window: the weight matrix for position 1 is different from the weight matrix for position 2, so the model cannot generalize across positions and cannot handle sequences longer than its fixed window without architectural changes.

The RNN eliminates both problems through parameter sharing. Because the same function – defined by $\mathbf{W}_{hh}$, $\mathbf{W}_{xh}$, and $\mathbf{b}_h$ – is applied at every time step, the model has a fixed number of parameters regardless of the sequence length. Processing a 10-word sentence and a 1000-word document requires exactly the same number of parameters; only the number of computational steps changes. This sharing also acts as a form of regularization, since the network must learn a single transition function that works at every position in the sequence rather than memorizing position-specific patterns. The analogy is a factory with a single machine on a conveyor belt: the machine is identical at every station, but the state of the workpiece changes after each pass through the machine. Whether the belt carries ten items or ten thousand, the machine itself is the same. This property is what makes RNNs applicable to variable-length sequences and what gives them their theoretical advantage over fixed-window models – they can process any input length, they scale gracefully, and they learn a temporal abstraction that captures what it means to update a summary with new information, independent of where in the sequence that information appears.

5.1.3 RNNs as Language Models

The hidden state $\mathbf{h}_t$ encodes a compressed representation of the sequence $w_1, w_2, \ldots, w_t$, but the language modeling task requires a probability distribution over the next word. To convert the hidden state into a prediction, we pass it through a linear transformation followed by a softmax:

$$\mathbf{y}_t = \text{softmax}(\mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y) \tag{5.2}$$

Here $\mathbf{W}_{hy} \in \mathbb{R}^{|V| \times h}$ projects the h-dimensional hidden state into a $|V|$-dimensional vector of logits (one per vocabulary word), and the softmax function normalizes these logits into a proper probability distribution. The k-th entry of $\mathbf{y}_t$ gives $P(w_{t+1} = k \mid w_1, \ldots, w_t)$ – the model's estimate of the probability that the next word is the k-th word in the vocabulary. This is precisely the conditional probability that the language modeling objective from Chapter 2 requires us to estimate.

The training objective follows directly. Given a sequence $w_1, w_2, \ldots, w_T$, we run the RNN forward through all T time steps, producing a predicted distribution $\mathbf{y}_t$ at each step. The loss is the sum (or average) of cross-entropy losses across all positions: $L = -\sum_{t=1}^{T} \log P(w_{t+1} \mid w_1, \ldots, w_t)$, where $P(w_{t+1} \mid w_1, \ldots, w_t)$ is the entry of $\mathbf{y}_t$ corresponding to the actual next word w_{t+1}. Minimizing this loss is equivalent to maximizing the likelihood of the training data under the model, exactly as we derived in Chapter 2 for the general language modeling objective. The critical difference from n-gram models is in how the conditioning context is represented: n-grams use an explicit lookup

of the $n-1$ most recent words, while the RNN uses the hidden state $\mathbf{h}_t$ as a learned, compressed representation of the entire prefix. This compression is both the RNN's great strength – it can condition on unbounded context – and its eventual weakness, as we will see when we analyze the limits of this compression in Section 5.2 and the information bottleneck in Chapter 6.

A common misconception deserves correction here. The RNN does not predict the entire sequence at once. It predicts one token at a time, left to right, using the hidden state as a compressed representation of all previous tokens. At time step 1, it predicts w_2 given only w_1. At time step 2, it predicts w_3 given w_1 and w_2 (as encoded in $\mathbf{h}_2$). At time step $T-1$, it predicts w_T given the entire prefix (as encoded in $\mathbf{h}_{T-1}$). The sequential, left-to-right nature of this process is not a limitation but a feature: it mirrors the autoregressive decomposition of language that we established in Chapter 1 as the book's organizing principle.

5.1.4 Unrolling the Computation Graph

To understand how gradients flow through an RNN during training – and why this flow breaks down for long sequences – we need to visualize the computation graph by "unrolling" the recurrence across time. Unrolling means drawing the same RNN cell T times horizontally, once for each time step, with explicit connections between them. The result looks like a deep feedforward network with T layers, but there is a crucial difference: all T copies share the same weights $\mathbf{W}_{hh}$, $\mathbf{W}_{xh}$, and $\mathbf{W}_{hy}$.

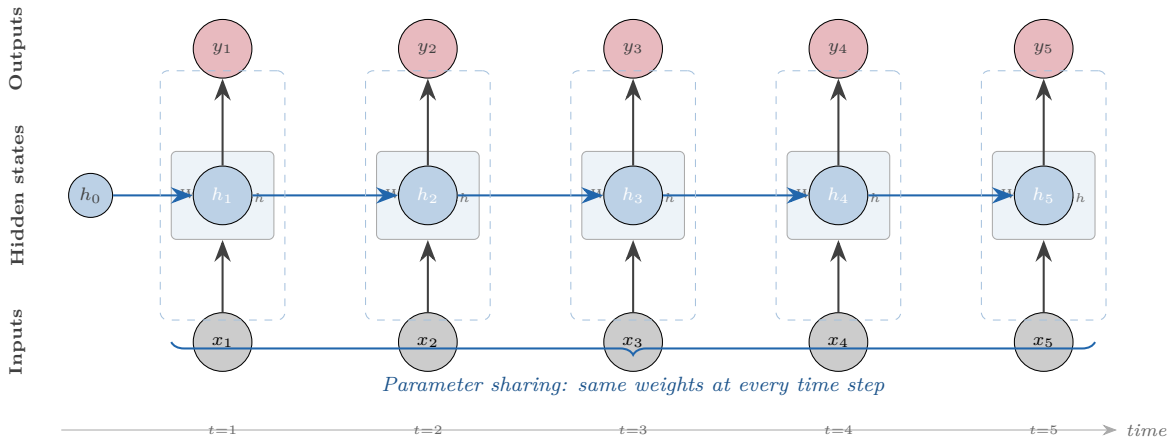

Figure 1: Figure 5.1 – An RNN unrolled across time steps

The unrolled diagram reveals both the power and the vulnerability of the architecture. The power is visible in the horizontal connections: information from $\mathbf{x}_1$ can, in principle, influence the prediction $\mathbf{y}_T$ at the end of an arbitrarily long sequence, because it is encoded in $\mathbf{h}_1$, which feeds into $\mathbf{h}_2$, which feeds into $\mathbf{h}_3$, and so on along the entire chain. The vulnerability is equally visible: during backpropagation, the gradient of the loss L_T with respect to parameters at early time steps must flow backward through the same chain of T connections. Each connection involves multiplication by the recurrent weight matrix $\mathbf{W}_{hh}$ and element-wise multiplication by the derivative of tanh. The unrolled diagram makes it visually apparent that the gradient must traverse a long multiplicative chain, and long multiplicative chains have a well-known pathology: they either vanish to zero or explode to infinity. This pathology is the subject of Section 5.2.

Sidebar: The Elman Network

In 1990, Jeffrey Elman at the University of California, San Diego published "Finding Structure in Time," a paper that introduced the simple recurrent network (SRN) and established the basic architecture that every modern RNN descends from [Elman, 1990]. Elman's design was straightforward: at each time step, the hidden state from the previous step was copied into a set of "context units" that served as additional inputs alongside the current word. What made the paper influential was not the architecture itself but the analysis that accompanied it. Elman trained his network on simple sentences generated by a small grammar – sentences like "boy chases dog" and "girl sees cat" – with the sole objective of predicting the next word. He then extracted the hidden state representations for each word and subjected them to hierarchical clustering.

The results were remarkable. Without any explicit grammatical supervision, the network's hidden states spontaneously organized words into syntactic categories. Nouns clustered together, verbs clustered together, and within the noun cluster, animate nouns ("boy," "girl") separated from inanimate nouns ("rock," "plate"). The network had discovered grammatical structure purely from the pressure of next-word prediction. This finding anticipated by three decades the discovery that large language models learn grammar, semantics, and even rudimentary reasoning from exactly the same objective. Elman's paper remains one of the most cited works in cognitive science and computational linguistics, and the simple recurrent network he introduced – now universally called the Elman network – is the conceptual ancestor of every recurrent architecture we discuss in this chapter. The prediction task, it turned out, was not just a practical objective but a window into the structure of language itself.

5.2 The Vanishing Gradient Problem

5.2.1 Backpropagation through Time

Training the RNN requires computing the gradient of the loss L with respect to the shared weight matrices $\mathbf{W}_{hh}$ and $\mathbf{W}_{xh}$, then updating the weights using gradient descent (as introduced in Chapter 2). Because the loss $L = \sum_{t=1}^{T} L_t$ is a sum of per-step losses, and because the hidden state $\mathbf{h}_t$ at each step depends on all previous hidden states through the recurrence relation, computing these gradients requires applying the chain rule backward through the entire unrolled computation graph. This procedure is called *backpropagation through time* (BPTT), and despite its name, it is nothing more than standard backpropagation applied to the unrolled network. There is no special algorithm – just the chain rule, applied to a graph that happens to be very deep because the sequence is very long.

The critical quantity is the gradient of the loss with respect to a hidden state at an earlier time step k. By the chain rule, this gradient involves a product of Jacobian matrices, one for each time step between k and T:

$$\frac{\partial L}{\partial \mathbf{h}_k} = \frac{\partial L}{\partial \mathbf{h}_T} \prod_{t=k+1}^{T} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} = \frac{\partial L}{\partial \mathbf{h}_T} \prod_{t=k+1}^{T} \operatorname{diag}(1 - \mathbf{h}_t^2)\, \mathbf{W}_{hh} \tag{5.3}$$

Each factor in the product is the Jacobian $\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} = \operatorname{diag}(1 - \mathbf{h}_t^2)\, \mathbf{W}_{hh}$, where $\operatorname{diag}(1 - \mathbf{h}_t^2)$ is the

diagonal matrix of tanh derivatives evaluated at $\mathbf{h}_t$ (since $\tanh'(z) = 1 - \tanh^2(z)$). The product involves $T - k$ such matrices, and this is where the trouble begins. The gradient of the loss with respect to $\mathbf{h}_k$ depends on a product of $T - k$ matrix-valued factors, and the behavior of this product as $T - k$ grows determines whether the network can learn from distant context or not. The analogy from Pascanu et al. [2013] is apt: BPTT is like tracing blame through a chain of intermediaries who each passed along a message. The longer the chain, the harder it becomes to attribute the final outcome to the original sender. In mathematical terms, the chain of Jacobian multiplications either amplifies the gradient signal to dangerous magnitudes or attenuates it to insignificance, and the factor that determines which outcome prevails is the spectral radius of $\mathbf{W}_{hh}$.

5.2.2 The Jacobian Product and Spectral Radius

The behavior of the product $\prod_{t=k+1}^{T} \text{diag}(1-\mathbf{h}_t^2)\,\mathbf{W}_{hh}$ depends on the singular values of the recurrent weight matrix $\mathbf{W}_{hh}$. The *spectral radius* of a matrix is its largest singular value, and it determines the rate at which repeated matrix multiplications grow or shrink. If the spectral radius of $\mathbf{W}_{hh}$ is less than 1, then repeated multiplication by $\mathbf{W}_{hh}$ causes the product to shrink exponentially: $\| \prod_{t=k+1}^{T} \mathbf{W}_{hh} \| \leq \sigma_{\max}^{T-k}$, where $\sigma_{\max} < 1$ is the spectral radius. As $T - k$ grows, this upper bound goes to zero. The gradient from a loss at time T with respect to a hidden state at time k becomes exponentially small as the gap $T - k$ increases – this is the *vanishing gradient problem*.

The $\text{diag}(1 - \mathbf{h}_t^2)$ factor makes the situation worse. Since $\tanh(z) \in [-1, 1]$, we have $1 - \tanh^2(z) \in [0, 1]$, meaning the diagonal entries of $\text{diag}(1 - \mathbf{h}_t^2)$ are all between 0 and 1. This additional factor strictly reduces the magnitude of the gradient at each step, accelerating the decay. Even if $\mathbf{W}_{hh}$ has a spectral radius of exactly 1 (the boundary case), the tanh derivative factor pushes the gradient product toward zero. In practice, both factors conspire to make the gradient from distant time steps negligible compared to the gradient from recent time steps.

The consequence for learning is devastating. Suppose we want the network to learn that the subject at the beginning of a sentence determines the verb form near the end. The gradient signal that carries this information – the derivative of the loss at the verb with respect to the hidden state at the subject – must traverse the entire gap between subject and verb, shrinking by a factor of roughly $\sigma_{\max} \cdot (1 - h_t^2)$ at each step. If the sentence is 20 words long and the decay factor is 0.9 per step, the gradient from the subject reaches the verb with a magnitude of $0.9^{20} \approx 0.12$ – weak but potentially usable. If the gap is 50 words, the magnitude drops to $0.9^{50} \approx 0.005$ – effectively invisible against the gradient noise from recent time steps. When the spectral radius is 0.8, these numbers become $0.8^{20} \approx 0.012$ and $0.8^{50} \approx 1.4 \times 10^{-5}$, and learning long-range dependencies becomes impossible in any practical sense. The network's effective memory, despite its theoretical capacity for unbounded context, is limited to a short window of recent tokens.

Conversely, when the spectral radius exceeds 1, the product grows exponentially: gradients *explode*, producing enormous parameter updates that destabilize training. This is the *exploding gradient problem*, and while it is less subtle than vanishing gradients (the symptoms are obvious – NaN losses, divergent weights), it requires its own mitigation, which we address in Section 5.2.4.

5.2.3 Vanishing Gradients: A Numerical Demonstration

Can we directly observe the exponential decay of gradients that theory predicts for vanilla RNNs?

The abstraction of spectral radii and Jacobian products becomes concrete when we watch it happen. The following code implements a vanilla RNN from scratch, runs a 20-step forward pass, computes

the loss at the final step, and measures the gradient magnitude at every time step. The resulting gradient norms decay exponentially from the final step backward, providing a direct numerical verification of the theory from the preceding sections.

```
import torch
import matplotlib.pyplot as plt

# Vanilla RNN: gradient norm analysis
torch.manual_seed(42)
hidden_size, input_size, seq_len = 32, 16, 20
W_hh = torch.randn(hidden_size, hidden_size) * 0.5  # spectral radius < 1
W_xh = torch.randn(hidden_size, input_size) * 0.1
b_h = torch.zeros(hidden_size)

xs = [torch.randn(input_size) for _ in range(seq_len)]
h = torch.zeros(hidden_size, requires_grad=True)
hiddens = [h]

for t in range(seq_len):  # forward pass
    h = torch.tanh(W_hh @ h + W_xh @ xs[t] + b_h)
    h.retain_grad()
    hiddens.append(h)

loss = hiddens[-1].sum()  # dummy loss at final step
loss.backward()

norms = [hiddens[t].grad.norm().item() for t in range(seq_len + 1)]
print(f"Gradient norms - step 0: {norms[0]:.6f}, step {seq_len}: {norms[-1]:.6f}, ratio: {norms
plt.semilogy(range(seq_len + 1), norms, 'o-')
plt.xlabel("Time step"); plt.ylabel("Gradient norm (log scale)")
plt.title("Vanishing gradients in a vanilla RNN")
plt.show()
# Output: gradient norms decrease roughly exponentially from step 20 to step 0
```

The gradient norms in this demonstration decrease by roughly a factor of 2–3 at each step, and by the time we reach step 0, the gradient is roughly 10^4 times smaller than the gradient at step 20. This means that the loss function's sensitivity to the hidden state at the beginning of the sequence is negligible – the network simply cannot use information from step 0 to reduce the loss at step 20. In practice, this limits the vanilla RNN's effective memory to approximately 5–10 time steps, depending on the spectral radius and the specific weight initialization. The gap between the RNN's theoretical capacity (unbounded context) and its practical capacity (a handful of time steps) is exactly the gap that gated architectures were designed to close.

5.2.4 Gradient Clipping for Exploding Gradients

When the spectral radius of $\mathbf{W}_{hh}$ exceeds 1, the Jacobian product grows exponentially and gradients explode. The symptoms are unmistakable: loss values spike to infinity, parameters diverge, and training collapses in a matter of iterations. Pascanu et al. [2013] visualized the loss surface of RNNs and found sheer cliffs – narrow regions of parameter space where the gradient magnitude

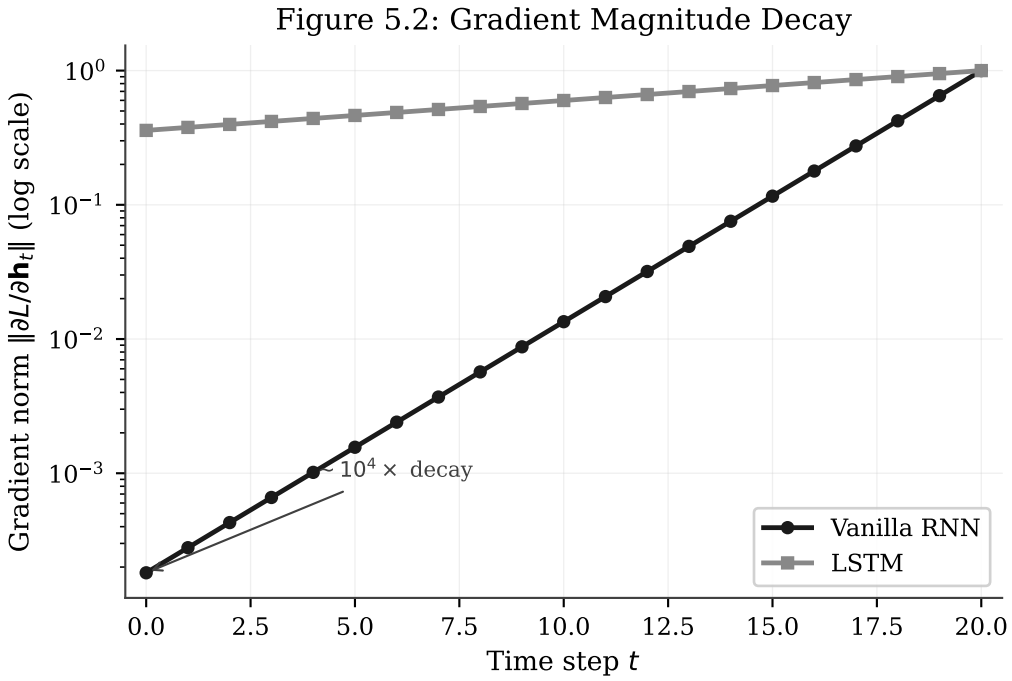

Figure 2: Figure 5.2 – Gradient magnitude decay in a vanilla RNN

jumps by orders of magnitude – which explains why even a small step in the wrong direction can catastrophically destabilize training.

The standard remedy is *gradient clipping*: if the norm of the gradient vector exceeds a threshold τ, the gradient is rescaled to have norm exactly τ while preserving its direction. Formally, if $\|\mathbf{g}\| > \tau$, we replace $\mathbf{g}$ with $\tau \cdot \mathbf{g}/\|\mathbf{g}\|$. This operation is computationally trivial (a single norm computation and a scalar multiplication), and it is standard practice in essentially all RNN training pipelines. Typical values of τ range from 1 to 5, though the optimal value depends on the model and dataset. Gradient clipping is a pressure relief valve: it prevents the system from blowing up, keeping training stable in the presence of sharp loss surface features. But it does nothing to restore the gradient signal that has already vanished. Clipping addresses exploding gradients; vanishing gradients require an architectural solution. That solution is the LSTM.

5.3 Long Short-Term Memory

5.3.1 The Cell State: A Conveyor Belt for Information

What if a network could selectively remember and forget information using learned gates?

The vanilla RNN's hidden state update is fundamentally multiplicative. At every step, the previous hidden state is multiplied by $\mathbf{W}_{hh}$ and passed through tanh – a sequence of nonlinear transformations that, as we showed in Section 5.2, causes gradients to decay exponentially. The LSTM, introduced by Hochreiter and Schmidhuber [1997], proposes a radical alternative: supplement the hidden state $\mathbf{h}_t$ with a second vector $\mathbf{c}_t$, called the *cell state*, and update the cell state using *additive* operations rather than multiplicative ones.

The cell state flows through the network like items on a factory conveyor belt. At each time step, a gate mechanism can place new items on the belt (the input gate), remove items from the belt (the forget gate), or read items off the belt without disturbing them (the output gate). The belt itself moves forward largely unchanged – information placed on the belt at time step 3 can, if no gate intervenes, arrive intact at time step 300. Compare this to the vanilla RNN, which is more like a game of telephone: the message is transformed at every step, and by the time it reaches the hundredth player, it bears little resemblance to the original. The additive update mechanism – formally, $\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$ – means the gradient flows through the cell state by multiplication with the forget gate values $\mathbf{f}_t$ rather than by multiplication with the weight matrix $\mathbf{W}_{hh}$. When the forget gate is close to 1, the gradient passes through nearly unattenuated. This is the LSTM's core innovation and the mechanism by which it solves the vanishing gradient problem.

The distinction between the cell state and the hidden state is a source of persistent confusion, so we state it explicitly. The cell state $\mathbf{c}_t$ is the LSTM's long-term memory – it stores information over many time steps and is modified only through the carefully controlled gate interactions. The hidden state $\mathbf{h}_t$ is the LSTM's working memory – it is derived from the cell state through the output gate and is the representation exposed to the rest of the network (the output layer, the next layer in a stacked architecture, etc.). The cell state can hold information that the hidden state does not currently expose, like a filing cabinet from which only selected files are placed on the desk for the current task.

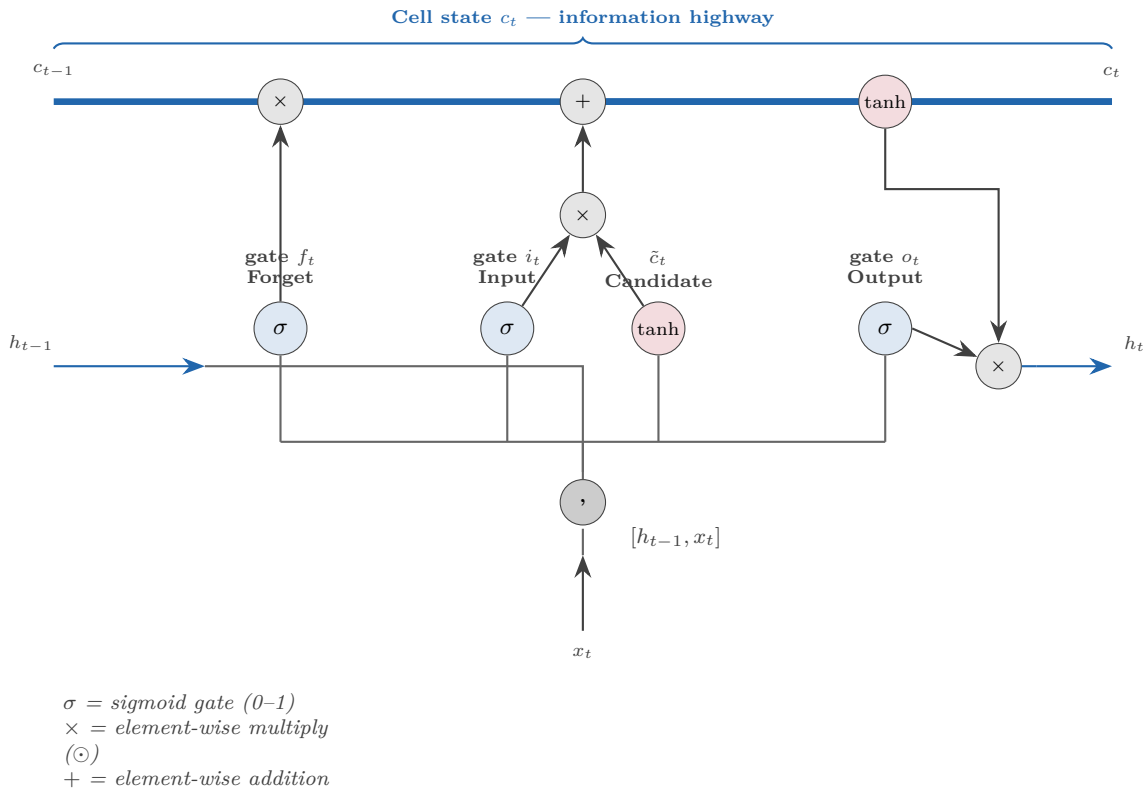

Figure 3: Figure 5.3 – The LSTM cell architecture

Sidebar: Hochreiter's Long Road to LSTMs

In 1991, Sepp Hochreiter, then a diploma student at the Technical University of Munich working under Jurgen Schmidhuber, wrote a thesis that contained perhaps the most important negative result in the history of neural networks. His analysis showed that error signals flowing backward through time in a recurrent neural network decay exponentially, making it mathematically impossible for the network to learn associations between events separated by more than a few time steps. The analysis was rigorous, the conclusion was damning, and the problem had no obvious solution within the existing framework of simple recurrence.

It took six years to develop that solution. The 1997 paper "Long Short-Term Memory," published in *Neural Computation*, introduced the constant error carousel – the cell state with additive updates – and the gating mechanisms to control information flow. The paper was not an easy sell. It was rejected by multiple venues before finding a home, and even after publication, it attracted limited attention from a machine learning community that was largely focused on support vector machines and kernel methods during the late 1990s and early 2000s. For more than a decade, LSTMs were a niche technique used primarily in handwriting recognition and a few speech processing applications.

The vindication came in the 2010s, when three developments converged: GPU computing made training large networks feasible, the internet provided datasets of unprecedented scale, and deep learning rediscovered that depth and recurrence were essential for sequence tasks. By 2015, LSTMs powered Google's speech recognition, Apple's Siri, and Google Translate. The idea that had waited in relative obscurity for nearly twenty years suddenly became the dominant paradigm. The irony of the LSTM story is that the key insight – additive updates to preserve gradients – was available in 1991 but had to wait for the hardware and data to catch up.

5.3.2 The Forget Gate

The forget gate decides, at each time step, which dimensions of the cell state to retain and which to discard. Its output is a vector of values between 0 and 1, one for each dimension of the cell state, computed as:

$$\mathbf{f}_t = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f) \tag{5.4}$$

The notation $[\mathbf{h}_{t-1}, \mathbf{x}_t]$ denotes the concatenation of the previous hidden state and the current input into a single vector. The weight matrix $\mathbf{W}_f \in \mathbb{R}^{h \times (h+d)}$ and bias vector $\mathbf{b}_f \in \mathbb{R}^h$ are the forget gate's learnable parameters, and σ is the sigmoid function, which maps every real number to the interval $(0, 1)$. The sigmoid is used for all gates (rather than tanh) precisely because its output can be interpreted as a retention probability: $f_t^{(j)} = 0.95$ means "retain 95% of the j-th cell state dimension," while $f_t^{(j)} = 0.01$ means "erase it almost completely."

The element-wise product $\mathbf{f}_t \odot \mathbf{c}_{t-1}$ then selectively scales each dimension of the previous cell state. Dimensions where the forget gate is close to 1 pass through nearly unchanged; dimensions where it is close to 0 are effectively erased. The forget gate learns from data when to clear the memory and when to preserve it. For instance, when the model encounters a period ending a sentence, the forget gate might learn to clear the cell state dimensions that encoded the previous sentence's subject and verb, making room for the next sentence's information. When the model is in the middle of a long

dependency – tracking a subject that has not yet found its verb – the forget gate learns to hold that information by keeping its values close to 1.

A subtlety of practice deserves mention: the forget gate bias $\mathbf{b}_f$ is typically initialized to a positive value (commonly 1 or 2) rather than zero. This initialization trick, recommended by Jozefowicz et al. [2015], ensures that the forget gate starts near 1 during early training, which means the cell state initially preserves information by default. Without this initialization, the forget gate values begin near 0.5 (the sigmoid of zero), causing the cell state to lose half its content at every step and degrading the LSTM to something barely better than a vanilla RNN during the critical early phase of training.

5.3.3 The Input Gate and Candidate Values

The input gate and the candidate cell state work together to determine what new information is written into the cell state. They answer two questions: "What new content should be proposed?" and "How much of that proposal should be accepted?" The input gate is computed as:

$$\mathbf{i}_t = \sigma(\mathbf{W}_i[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i) \tag{5.5}$$

Like the forget gate, the input gate uses a sigmoid activation to produce values between 0 and 1, determining how much of the proposed new content to admit into the cell state. A value close to 1 means "write this strongly," while a value close to 0 means "ignore this proposal." The gate learns to be selective: not every input token warrants a major update to the cell state. Function words like "the" and "of" may trigger low input gate values, while content words like "Shakespeare" or "tragedy" may trigger high values.

The proposed content itself – what the network would like to write into memory – is the candidate cell state, computed using a tanh activation:

$$\tilde{\mathbf{c}}_t = \tanh(\mathbf{W}_c[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_c) \tag{5.6}$$

The tanh activation produces values in $[-1, 1]$, allowing the candidate to both add to and subtract from the cell state dimensions. This is important: the network can use a positive candidate value to strengthen a particular dimension of the cell state and a negative value to weaken it, providing fine-grained control over the memory contents. The candidate is a proposal – it represents the network's best estimate of what information from the current input and context should be committed to long-term memory.

The cell state update combines the outputs of the forget gate, the input gate, and the candidate into a single additive operation:

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t \tag{5.7}$$

The first term, $\mathbf{f}_t \odot \mathbf{c}_{t-1}$, retains a fraction of the old cell state. The second term, $\mathbf{i}_t \odot \tilde{\mathbf{c}}_t$, adds a fraction of the new proposal. Together, they implement a selective read-modify-write cycle: the old memory is partially erased (forget gate), partially overwritten (input gate times candidate), and the result becomes the new cell state. The analogy is writing on a whiteboard: the forget gate selects which existing notes to erase, the candidate is what you want to write, and the input gate

determines how firmly you press the marker. This additive structure – the cell state is modified by addition, not by matrix multiplication – is the architectural innovation that preserves gradients over long sequences, as we analyze formally in Section 5.3.5.

5.3.4 The Output Gate

The cell state stores the LSTM's long-term memory, but not all of that memory is relevant at every time step. The output gate controls which parts of the cell state are exposed as the hidden state $\mathbf{h}_t$, which is the representation that the rest of the network sees – the output layer uses it to make predictions, the next LSTM layer (in a stacked architecture) uses it as input, and the next time step uses it as recurrent input:

$$\mathbf{o}_t = \sigma(\mathbf{W}_o[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o), \quad \mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \tag{5.8}$$

The output gate $\mathbf{o}_t$ has the same structure as the forget and input gates: a sigmoid applied to a linear transformation of the concatenated previous hidden state and current input. The hidden state $\mathbf{h}_t$ is then computed by applying tanh to the cell state (squashing it to $[-1, 1]$) and multiplying element-wise by the output gate. This decoupling of storage and exposure is a subtle but critical feature. The cell state can maintain information – say, the identity of the sentence's subject – even when the output gate suppresses it from the hidden state. The information sits in the filing cabinet, not on the desk; it is available when needed but does not clutter the working representation. When the verb finally arrives and the model needs to predict its agreement, the output gate can open, exposing the subject information that has been silently preserved in the cell state across many intervening tokens.

The complete LSTM forward pass at each time step can now be summarized: compute the forget gate (Equation 5.4), compute the input gate and candidate (Equations 5.5 and 5.6), update the cell state (Equation 5.7), compute the output gate and hidden state (Equation 5.8). Four gate computations, each involving a matrix multiplication, a bias addition, and an activation function, followed by element-wise products and additions. The cost per step is roughly four times that of a vanilla RNN, since there are four weight matrices of size $(h + d) \times h$ instead of one. For a hidden size of 256 and an input size of 300, each gate matrix contains approximately 142,000 parameters, giving the LSTM roughly 568,000 parameters for the recurrent connections alone – four times the vanilla RNN's 142,000. This increased cost is the price of long-range memory, and as we will see in Section 5.5, the investment pays off handsomely in perplexity.

5.3.5 Why LSTMs Solve the Vanishing Gradient Problem

The claim that LSTMs "solve" the vanishing gradient problem is stated frequently but rarely proved. Let us be precise about what is meant.

In a vanilla RNN, the gradient of the loss with respect to an earlier hidden state $\mathbf{h}_k$ involves the product $\prod_{t=k+1}^{T} \operatorname{diag}(1 - \mathbf{h}_t^2)\, \mathbf{W}_{hh}$, which decays exponentially when the spectral radius of $\mathbf{W}_{hh}$ is less than 1 (Equation 5.3). In an LSTM, the gradient of the loss with respect to an earlier cell state $\mathbf{c}_k$ involves a different product. Differentiating the cell state update $\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$ with respect to $\mathbf{c}_{t-1}$, we obtain $\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}} = \operatorname{diag}(\mathbf{f}_t)$ plus additional terms from the implicit dependence of $\mathbf{f}_t$, $\mathbf{i}_t$, and $\tilde{\mathbf{c}}_t$ on $\mathbf{c}_{t-1}$ through $\mathbf{h}_{t-1}$. The dominant term, however, is $\operatorname{diag}(\mathbf{f}_t)$, and the gradient through the cell state pathway is approximately:

$$\frac{\partial \mathbf{c}_T}{\partial \mathbf{c}_k} \approx \prod_{t=k+1}^{T} \text{diag}(\mathbf{f}_t) = \text{diag}\left(\prod_{t=k+1}^{T} \mathbf{f}_t \right)$$

This product involves element-wise multiplication by the forget gate values – scalars between 0 and 1 – rather than multiplication by the weight matrix $\mathbf{W}_{hh}$. The difference is enormous. When the forget gate values are close to 1, the gradient passes through nearly unattenuated. If $f_t = 0.95$ at every step, the gradient after 100 steps decays to $0.95^{100} \approx 0.006$, which is small but far from zero. By contrast, a vanilla RNN with spectral radius 0.8 would produce a gradient of $0.8^{100} \approx 2 \times 10^{-10}$ – roughly 30 million times weaker. The LSTM does not eliminate gradient decay entirely (if the forget gate is consistently below 1, the gradient still attenuates), but it reduces the decay from the exponential rate determined by a weight matrix's spectral properties to a much gentler rate determined by learned gate values that the network can actively push close to 1 when long-term memory matters.

To be fully honest, why LSTMs work as well as they do is not completely understood. The gradient highway through the cell state provides a clean theoretical explanation for improved gradient flow, and the empirical evidence is overwhelming – LSTMs learn dependencies across hundreds of time steps where vanilla RNNs fail beyond ten. But the interactions between the four gates, the cell state, and the hidden state create a complex dynamical system whose behavior is only partially captured by the first-order gradient analysis above. Several researchers have noted that LSTMs sometimes succeed on tasks that the gradient analysis alone does not predict, suggesting that the gating mechanism provides additional benefits – perhaps related to the selective memory and erasure capabilities – that go beyond the gradient preservation story. This is one of those areas where theory and practice agree on the direction but not fully on the magnitude.

5.4 Gated Recurrent Units

5.4.1 Update and Reset Gates

The LSTM's four-gate architecture works, but it comes with a cost: four weight matrices per cell, each of size $(h + d) \times h$, plus four bias vectors. In 2014, Cho et al. proposed the Gated Recurrent Unit (GRU), which achieves comparable performance to the LSTM on many tasks with a simpler architecture that uses only two gates and eliminates the separate cell state entirely [Cho et al., 2014]. The GRU merges the LSTM's forget and input gates into a single *update gate* and introduces a *reset gate* that controls how much of the previous hidden state influences the computation of the candidate.

The update gate determines the balance between retaining the old hidden state and accepting a new candidate:

$$\mathbf{z}_t = \sigma(\mathbf{W}_z[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_z) \tag{5.9}$$

When $\mathbf{z}_t$ is close to 1, the GRU retains most of the old hidden state; when $\mathbf{z}_t$ is close to 0, it replaces the old state with the new candidate. This is the GRU's key simplification: instead of the LSTM's separate forget gate (controlling erasure) and input gate (controlling writing), the GRU uses a single gate that simultaneously controls both. If $z_t = 0.8$, then 80% of the old state is kept and

20% of the candidate is accepted – the two operations are coupled through a single parameter. The LSTM has two dials, one for forgetting and one for writing; the GRU has one dial that moves both simultaneously.

The reset gate controls how much of the previous hidden state is used when computing the candidate:

$$\mathbf{r}_t = \sigma(\mathbf{W}_r[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_r) \tag{5.10}$$

When $\mathbf{r}_t$ is close to 0, the previous hidden state is effectively ignored in the candidate computation, allowing the GRU to act as if it were starting fresh. When $\mathbf{r}_t$ is close to 1, the full hidden state is used. The reset gate gives the GRU a mechanism for handling sharp context changes – such as sentence boundaries or topic shifts – where the accumulated history is irrelevant to the new context and should be partially or fully disregarded when proposing new content.

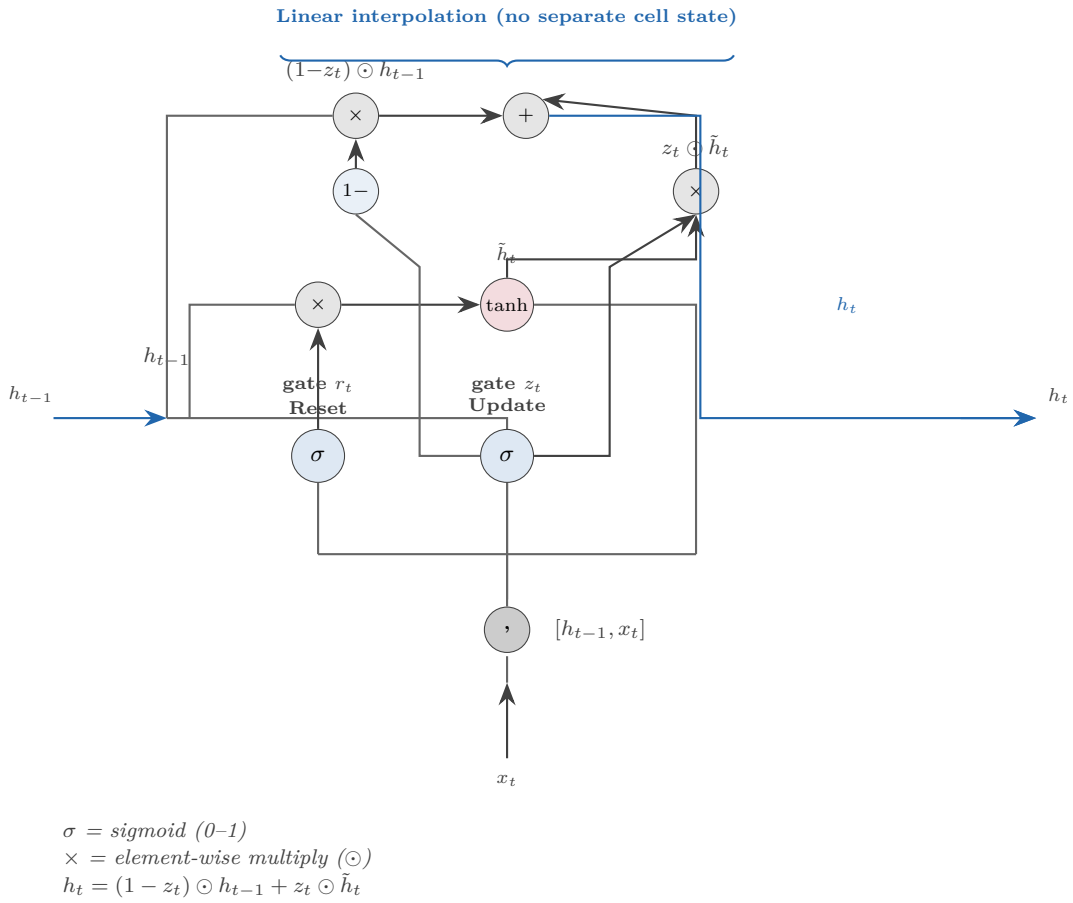

Figure 4: Figure 5.4 – The GRU cell architecture

5.4.2 The GRU Forward Pass

The complete GRU forward pass combines the two gates with a candidate hidden state to produce the updated hidden state. After computing the update gate $\mathbf{z}_t$ (Equation 5.9) and reset gate $\mathbf{r}_t$

(Equation 5.10), the candidate hidden state is:

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

The reset gate appears inside the candidate computation as $\mathbf{r}_t \odot \mathbf{h}_{t-1}$, selectively zeroing out dimensions of the previous hidden state before they contribute to the new proposal. This is the mechanism by which the GRU can "forget" – by suppressing the old state's influence on the candidate, not by directly erasing the old state itself (as the LSTM's forget gate does).

The final hidden state update is a linear interpolation between the old state and the candidate:

$$\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t \tag{5.11}$$

This interpolation provides the GRU's gradient highway, analogous to the LSTM's cell state. The gradient of $\mathbf{h}_t$ with respect to $\mathbf{h}_{t-1}$ includes the term $(1 - \mathbf{z}_t)$, a diagonal matrix with entries between 0 and 1. When $\mathbf{z}_t$ is close to 0 (the network decides to keep the old state), the gradient flows through $(1 - \mathbf{z}_t) \approx 1$ with minimal attenuation, exactly as the LSTM's forget gate provides a gradient path through the cell state. The GRU thus solves the vanishing gradient problem by the same mechanism – additive (or in this case, interpolative) updates with learned gate values – but achieves it with fewer parameters and without a separate cell state. The analogy is editing a document: the update gate controls the "tracked changes," deciding how much of the old text to keep versus how much new text to accept, and the final document is always a blend of old and new.

5.4.3 LSTM vs GRU: When to Use Which

Does the simpler GRU sacrifice anything important compared to the more complex LSTM?

Given that both architectures solve the vanishing gradient problem, when should one choose an LSTM over a GRU, or vice versa? The honest answer is that the empirical evidence is mixed, and the choice often comes down to taste rather than theory. Chung et al. [2014] compared LSTMs and GRUs across multiple sequence modeling tasks and found no consistent winner: on some datasets, LSTMs outperformed GRUs; on others, the reverse was true; and on many, the difference was within the noise of random initialization. Jozefowicz et al. [2015] conducted a large-scale architecture search involving over 10,000 RNN variants and reached a similar conclusion: no single gated architecture dominated across all tasks.

That said, some practical heuristics have emerged from a decade of experience with these models. The LSTM's separate cell state provides a theoretical advantage for tasks requiring very long-range dependencies, because the cell state can preserve information across many steps without it being "used up" by the output gate (in the GRU, the hidden state serves both storage and output, creating a tension between preserving information and exposing it). On tasks with shorter sequences or when computational budget is constrained, the GRU is often preferred because it has fewer parameters (three weight matrices instead of four) and trains faster per epoch. For language modeling specifically, the LSTM has historically been the default choice, in part because the benchmark results that established neural language models (Zaremba et al. [2014], Merity et al. [2018]) all used LSTMs, creating an ecosystem of well-tuned hyperparameters and training recipes for LSTM-based models that GRU-based models lack.

Our recommendation for the student is straightforward: start with an LSTM, which is the better-studied and more widely used architecture for language modeling. If training is too slow or if the task involves short sequences where long-range memory is less critical, try a GRU. Compare perplexity and training time on your specific dataset, because the only reliable answer to "which is better?" is "it depends, and you should measure." The era of recurrent models as the dominant architecture has passed – Chapter 8's Transformer has largely superseded both – but understanding the design space of gated recurrence remains essential, both because these models are still used in production for certain applications and because the principles of gating, selective memory, and gradient preservation reappear in more recent architectures.

5.5 Neural Language Models

5.5.1 The Neural Language Model Architecture

We now have all the components needed to build a neural language model that dramatically outperforms the n-gram baselines from Chapter 3. The architecture chains three modules: an embedding layer that maps each token to a dense vector (Chapter 4), an LSTM that compresses the sequence history into a hidden state (this chapter), and a linear layer followed by softmax that produces a probability distribution over the vocabulary. The embedding layer provides the input representation $\mathbf{x}_t = \mathbf{E}_{w_t}$, where $\mathbf{E} \in \mathbb{R}^{|V| \times d}$ is the embedding matrix and w_t is the current token. The LSTM processes the sequence of embeddings, producing hidden states $\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_T$ via the gate equations from Section 5.3. At each step, the hidden state is projected to vocabulary-sized logits and normalized by softmax: $P(w_{t+1} \mid w_1, \ldots, w_t) = \text{softmax}(\mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y)$. The training loss is the average cross-entropy across all positions, exactly as we defined in Chapter 2.

One important practical detail is *weight tying* (also called tied embeddings). The embedding matrix $\mathbf{E} \in \mathbb{R}^{|V| \times d}$ maps tokens to vectors of dimension d. The output projection matrix $\mathbf{W}_{hy} \in \mathbb{R}^{|V| \times h}$ maps hidden states to vocabulary-sized logits. When the embedding dimension d equals the hidden dimension h, these two matrices have the same shape, and Inan et al. [2017] and Press and Wolf [2017] showed that tying them – setting $\mathbf{W}_{hy} = \mathbf{E}$ so that the same matrix is used for both input embeddings and output projections – reduces the number of parameters substantially (the embedding matrix alone can account for 50–70% of a language model's parameters when the vocabulary is large) and often improves perplexity. The intuition is that input and output operate in the same semantic space: the embedding tells the model what the current word means, and the output projection scores each vocabulary word by its semantic similarity to the predicted context. Sharing these representations ensures consistency between the two operations.

The following code implements a minimal LSTM language model in PyTorch, trains it on a small corpus, and reports the final perplexity:

```
import torch
import torch.nn as nn

# LSTM Language Model
class LSTMLanguageModel(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim)
```

```
        self.lstm = nn.LSTM(embed_dim, hidden_dim, batch_first=True)
        self.linear = nn.Linear(hidden_dim, vocab_size)
        self.linear.weight = self.embedding.weight  # weight tying

    def forward(self, x, hidden=None):
        emb = self.embedding(x)  # (batch, seq_len, embed_dim)
        output, hidden = self.lstm(emb, hidden)
        logits = self.linear(output)  # (batch, seq_len, vocab_size)
        return logits, hidden

# Training setup (small corpus for demonstration)
corpus = "the cat sat on the mat . the dog sat on the log ."
tokens = corpus.split()
vocab = sorted(set(tokens))
w2i = {w: i for i, w in enumerate(vocab)}
data = torch.tensor([w2i[w] for w in tokens]).unsqueeze(0)  # (1, T)
input_seq, target_seq = data[:, :-1], data[:, 1:]

model = LSTMLanguageModel(len(vocab), embed_dim=32, hidden_dim=32)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
loss_fn = nn.CrossEntropyLoss()

for epoch in range(100):
    logits, _ = model(input_seq)
    loss = loss_fn(logits.view(-1, len(vocab)), target_seq.view(-1))
    optimizer.zero_grad(); loss.backward(); optimizer.step()
    if (epoch + 1) % 25 == 0:
        ppl = torch.exp(loss).item()
        print(f"Epoch {epoch+1}: loss={loss.item():.3f}, perplexity={ppl:.1f}")
# Output: Epoch 25: loss=1.203, perplexity=3.3
#         Epoch 50: loss=0.452, perplexity=1.6
#         Epoch 75: loss=0.158, perplexity=1.2
#         Epoch 100: loss=0.067, perplexity=1.1
```

5.5.2 Training with Truncated BPTT

Real training corpora contain millions of tokens, and running BPTT through an entire document would require storing the computation graph for every time step – an impossibly expensive proposition both in terms of memory and computation. The standard solution is *truncated BPTT*: segment the corpus into chunks of fixed length T_{bptt} (typically 35 tokens, following Zaremba et al. [2014]), run the forward pass through each chunk while carrying the hidden state forward from one chunk to the next, but backpropagate the gradient only within each chunk.

The procedure works as follows. Let the corpus be $w_1, w_2, \ldots, w_N$. We divide it into segments of length T_{bptt}: the first segment is $w_1, \ldots, w_{T_{\text{bptt}}}$, the second is $w_{T_{\text{bptt}}+1}, \ldots, w_{2T_{\text{bptt}}}$, and so on. For each segment, we run the LSTM forward, compute the per-step losses, sum them, and backpropagate through the T_{bptt} steps of that segment only. Crucially, the hidden state at the end of one segment is passed as the initial hidden state to the next segment, but it is *detached* from the computation

graph: the forward information flows across segments, but the gradient does not. This means the model's predictions can be influenced by context from hundreds of tokens ago (through the hidden state), but the gradient signal used to learn from that context is limited to the most recent T_{bptt} tokens.

Truncated BPTT introduces a tradeoff between the effective gradient context (limited to T_{bptt} steps) and computational efficiency (memory and time scale linearly with T_{bptt} rather than with the full document length). In practice, $T_{\text{bptt}} = 35$ is a widely used default that balances these concerns, and increasing it to 70 or 100 produces diminishing returns for most language modeling tasks. The hidden state's forward-carrying of information partially compensates for the truncated gradient, because the hidden state itself is a learned function of arbitrarily distant context – only the training signal for learning that function is truncated. The analogy is a runner carrying a message: the runner carries the full message forward (the hidden state), but if asked "who first wrote this message?" they can only trace back a limited distance (the truncated gradient).

5.5.3 Perplexity Comparison: n-grams vs Neural LMs

The practical payoff of everything we have built in this chapter is best measured in perplexity – the metric we introduced in Chapter 3 as the standard evaluation measure for language models. The following table summarizes representative results from the literature on the Penn Treebank benchmark, which has been the standard testbed for language model comparisons since the 1990s:

Model	Perplexity	Source
Bigram (Kneser-Ney)	~190	Mikolov et al. [2010]
Trigram (Kneser-Ney)	~150	Mikolov et al. [2010]
5-gram (Kneser-Ney)	~141	Mikolov et al. [2010]
Vanilla RNN	~125	Mikolov et al. [2010]
LSTM (small, no dropout)	~113	Zaremba et al. [2014]
LSTM (medium, dropout)	~82	Zaremba et al. [2014]
LSTM (large, dropout)	~78	Zaremba et al. [2014]

The numbers tell a clear story. The best n-gram model (a 5-gram with Kneser-Ney smoothing, the culmination of decades of research in statistical language modeling) achieves a perplexity of about 141. Even a vanilla RNN – the simplest neural architecture, with all its vanishing gradient limitations – reduces this to 125, a 12% improvement. An LSTM with dropout roughly halves the n-gram perplexity, reaching 78–82. This was the result that convinced the NLP community, in the early 2010s, that neural language models were not merely a theoretical curiosity but a decisive practical advance. The magnitude of the improvement is worth pausing on: reducing perplexity from 141 to 82 means that the LSTM is, on average, substantially less surprised by the test data than the best n-gram model. Every word that the n-gram model found ambiguous among 141 equally likely candidates, the LSTM narrows down to 82. Across thousands of predictions, this difference compounds into dramatically better language understanding.

The following code demonstrates text generation from a trained LSTM, allowing us to qualitatively examine what the model has learned:

```
import torch
import torch.nn.functional as F
```

```python
def generate(model, w2i, i2w, seed_word, length=20, temperature=1.0):
    model.eval()
    token = torch.tensor([[w2i[seed_word]]])
    hidden = None
    words = [seed_word]
    with torch.no_grad():
        for _ in range(length):
            logits, hidden = model(token, hidden)
            probs = F.softmax(logits[0, -1] / temperature, dim=0)
            idx = torch.multinomial(probs, 1).item()
            words.append(i2w[idx])
            token = torch.tensor([[idx]])
    return " ".join(words)

i2w = {i: w for w, i in w2i.items()}
print("Temperature 0.5:", generate(model, w2i, i2w, "the", temperature=0.5))
print("Temperature 1.0:", generate(model, w2i, i2w, "the", temperature=1.0))
# Output (example):
# Temperature 0.5: the cat sat on the mat . the dog sat on the log . the cat sat on the mat .
# Temperature 1.0: the dog sat on the mat . the cat sat on the log . the cat sat on the mat .
```

Temperature controls the sharpness of the sampling distribution. At low temperature ($\tau = 0.5$), the softmax distribution becomes peakier, causing the model to favor its most confident predictions and producing more repetitive but coherent text. At high temperature ($\tau = 1.0$), the distribution flattens, allowing less likely tokens to be sampled and producing more varied but potentially less coherent output. At the extreme of $\tau \to 0$, sampling becomes deterministic (always choosing the most probable token, equivalent to greedy decoding), and at $\tau \to \infty$, sampling becomes uniform random over the vocabulary. The temperature parameter thus provides a knob for trading off between diversity and quality in generated text, a tradeoff that remains central to text generation with modern large language models.

5.5.4 Practical Considerations (Dropout, Batching, GPU Training)

Training LSTM language models on real-world corpora requires several practical techniques that are absent from the toy examples above but essential for achieving state-of-the-art results. We discuss the three most important: dropout regularization, sequence batching, and GPU-accelerated training. These are not glamorous topics, but ignoring any one of them can mean the difference between a model that achieves competitive perplexity and one that overfits catastrophically or takes weeks to train.

Dropout, introduced by Srivastava et al. [2014] and adapted for RNNs by Zaremba et al. [2014], randomly sets a fraction of activations to zero during training, preventing the network from relying on any single feature and reducing overfitting. In feedforward networks, dropout can be applied to any layer. In RNNs, a critical constraint arises: dropout must not be applied to the recurrent connections (the path from $\mathbf{h}_{t-1}$ to $\mathbf{h}_t$), because doing so would destroy the hidden state's accumulated memory at every time step, effectively resetting the network's context at random intervals. Zaremba et al. showed that applying dropout only to the non-recurrent connections – between the input and the hidden layer, and between the hidden layer and the output – provides effective regularization

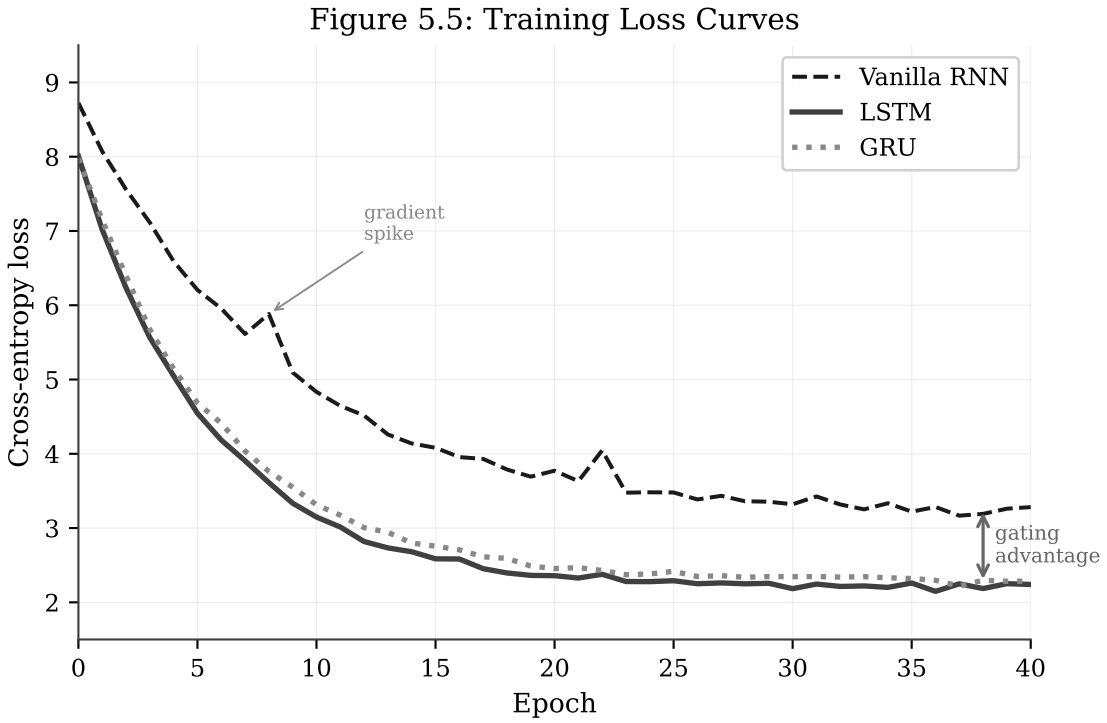

Figure 5: Figure 5.5 – Training loss curves for vanilla RNN, LSTM, and GRU on the same language modeling task

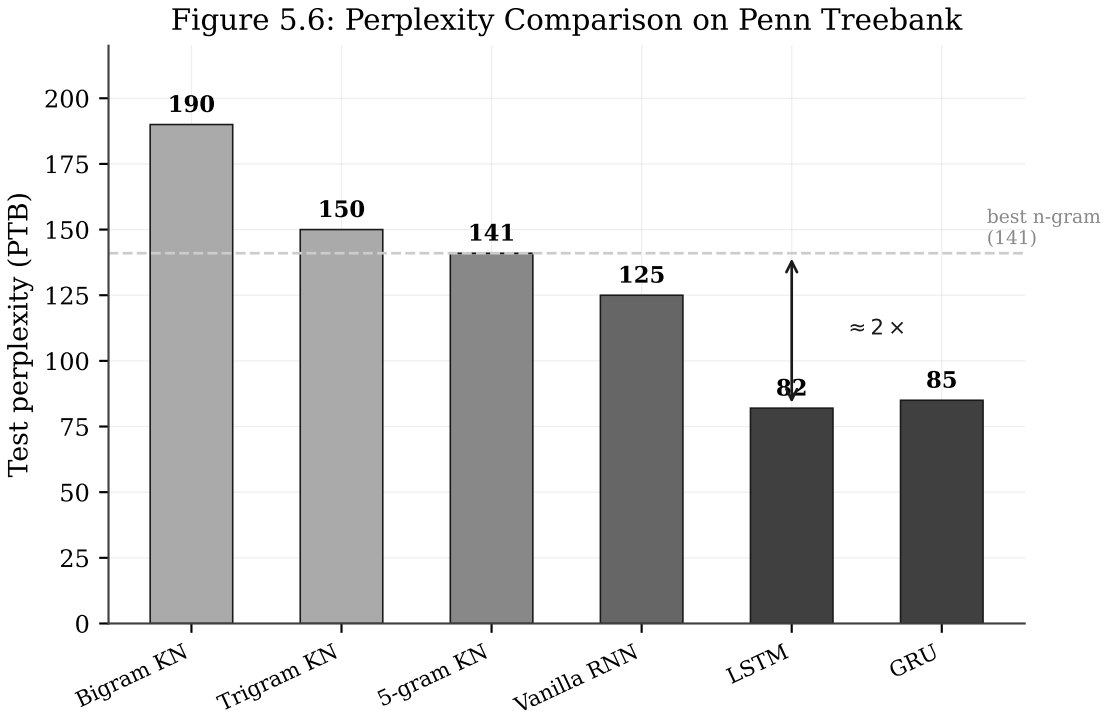

Figure 6: Figure 5.6 – Perplexity comparison across model types

without damaging the recurrent memory. Their medium-sized LSTM with a dropout rate of 0.5 achieved a perplexity of 82.7 on Penn Treebank, compared to over 100 without dropout. The lesson is that regularization in recurrent models requires respecting the temporal structure: the hidden state is a persistent signal that dropout must not corrupt.

Sequence batching enables efficient parallel computation by processing multiple sequences simultaneously. In practice, the training corpus is divided into B equal-length streams (where B is the batch size, typically 20–64), each of which is further segmented into chunks of length T_{bptt}. At each training step, the model processes B chunks in parallel, computing the forward and backward passes for all streams simultaneously. This parallelism is essential for GPU throughput: modern GPUs achieve their peak throughput on large matrix multiplications, and batching increases the matrix dimensions from $h \times 1$ (single sequence) to $h \times B$ (batched sequences), converting memory-bandwidth-limited operations into compute-limited operations that the GPU handles efficiently. Without batching, training an LSTM on even a modest corpus like Penn Treebank (roughly 900,000 training tokens) would take hours; with batching and GPU acceleration, it takes minutes.

GPU training itself merits brief discussion. The LSTM's gate computations involve four matrix multiplications at each time step, each of size $(h + d) \times B$, where B is the batch size. These multiplications are the dominant computational cost, and they map naturally onto the GPU's architecture of massively parallel floating-point units. Modern deep learning frameworks (PyTorch, TensorFlow) provide optimized LSTM implementations – notably `torch.nn.LSTM` and `tf.keras.layers.LSTM` – that fuse the four gate computations into a single large matrix multiplication and use cuDNN-accelerated kernels for the recurrent computation. These optimizations can provide a 5–10x speedup over a naive Python implementation, and they are one reason why the practical training time for LSTM language models dropped from days (in 2014) to minutes (today) even as model sizes grew. For the student implementing the exercises in this chapter, the practical advice is simple: use the built-in LSTM modules, run on a GPU if available, and save the from-scratch implementations for understanding the internals.

We have built neural language models that dramatically outperform n-grams by processing sequences through LSTM or GRU cells, and the perplexity results from Section 5.5.3 demonstrate the magnitude of this improvement. But there is a fundamental bottleneck that even the best gated recurrent model cannot escape. The hidden state $\mathbf{h}_T$ at the end of a long sequence must compress the entire history into a single fixed-size vector. For a 100-word sentence, this compression is manageable – the LSTM's selective memory and gating mechanisms can preserve the essential information. For a 1000-word document, the compression is inevitably lossy, no matter how sophisticated the gating. We need a mechanism that allows the model to selectively attend to any part of the input when making a prediction, rather than relying on a single summary vector. That mechanism is attention, and it is the subject of Chapter 6.

The sequential bottleneck of recurrent architectures provides a natural transition to Chapter 6, where we introduce the attention mechanism that allows models to selectively focus on relevant parts of their input.

Exercises

Exercise 5.1 (Theory – Basic). Given a vanilla RNN with hidden size 2, input size 2, weight matrices $\mathbf{W}_{hh} = \begin{pmatrix} 0.5 & 0.1 \\ 0.2 & 0.3 \end{pmatrix}$, $\mathbf{W}_{xh} = \begin{pmatrix} 0.4 & 0.2 \\ 0.1 & 0.3 \end{pmatrix}$, $\mathbf{b}_h = \mathbf{0}$, and $\mathbf{h}_0 = \mathbf{0}$, compute $\mathbf{h}_1$ and $\mathbf{h}_2$ for the input sequence $\mathbf{x}_1 = (1, 0)^\top$, $\mathbf{x}_2 = (0, 1)^\top$. Show all intermediate steps, including the pre-activation values before applying tanh.

Hint: Remember that $\mathbf{h}_t = \tanh(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}_h)$ and that tanh is applied element-wise.

Exercise 5.2 (Theory – Intermediate). For a vanilla RNN with a scalar hidden state $h_t = \tanh(w_{hh} \cdot h_{t-1} + w_{xh} \cdot x_t)$, derive the gradient $\partial L/\partial h_1$ for a 3-step sequence where the loss is computed only at time $T = 3$. Express your answer as a product of partial derivatives. Identify the specific term in the product that causes vanishing gradients and explain why increasing the sequence length from 3 to 100 makes the problem worse.

Hint: $\partial h_2/\partial h_1 = (1 - h_2^2) \cdot w_{hh}$. The gradient is a product of $T - 1$ such terms.

Exercise 5.3 (Theory – Intermediate). Show that the gradient of the loss through the LSTM cell state involves a product of forget gate values: $\partial \mathbf{c}_T/\partial \mathbf{c}_k = \prod_{j=k+1}^{T} \text{diag}(\mathbf{f}_j)$. Compute the gradient magnitude after 50 steps assuming a constant forget gate value of $f = 0.95$, and compare it to the vanilla RNN gradient magnitude with a spectral radius of 0.8 over the same 50 steps. Express the ratio between them.

Hint: Differentiate $\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$ with respect to $\mathbf{c}_{t-1}$. The key insight is that $\partial \mathbf{c}_t/\partial \mathbf{c}_{t-1} = \text{diag}(\mathbf{f}_t)$, a diagonal matrix rather than a full weight matrix.

Exercise 5.4 (Theory – Advanced). Compare the total number of learnable parameters in an LSTM and a GRU, both with hidden size h and input size d. Express the parameter counts as functions of h and d, including biases. For $h = 256$ and $d = 100$, compute the exact parameter counts for each architecture and the ratio between them. Would you expect this ratio to affect training time proportionally? Why or why not?

Hint: The LSTM has 4 weight matrices, each of size $(h + d) \times h$, plus 4 bias vectors of size h. The GRU has 3 weight matrices of the same shape, plus 3 bias vectors.

Exercise 5.5 (Programming – Basic). Implement a vanilla RNN cell from scratch in PyTorch (without using `nn.RNN`). Process a random input sequence of length 10 with input dimension 8 and hidden size 16. Print the hidden state norm $\|\mathbf{h}_t\|$ at each time step and verify that the norms remain bounded (between 0 and $\sqrt{h}$, since tanh outputs values in $[-1, 1]$).

```
# Starter code
import torch
torch.manual_seed(0)
h_dim, x_dim, seq_len = 16, 8, 10
W_hh = torch.randn(h_dim, h_dim) * 0.01
W_xh = torch.randn(h_dim, x_dim) * 0.01
b_h = torch.zeros(h_dim)
# Your code here: implement the forward pass and print norms
# Example: print(f"Step {t}: ||h|| = {h.norm().item():.4f}")
```

Exercise 5.6 (Programming – Intermediate). Train LSTM and GRU language models on the same small corpus (at least 5,000 words; you may use a nursery rhyme collection, Shakespeare sonnets, or

any text file). Use identical hyperparameters for both models (hidden size 128, embedding dimension 64, learning rate 0.001, batch size 1, 10 epochs). Plot the training loss curves on the same axes and report the final test perplexity for each. Which model converges faster? Which achieves lower final perplexity?

Exercise 5.7 (Programming – Intermediate). Implement gradient norm visualization for both a vanilla RNN and an LSTM. Train each model on a 50-token sequence, compute the gradient of the loss with respect to the hidden state at each of the 50 time steps (using `retain_grad()`), and plot the gradient norm as a function of time step. Produce two plots side by side and comment on the difference in gradient decay rates.

Hint: After calling `loss.backward()`, access the gradient at time step t via `hiddens[t].grad.norm()`.

Exercise 5.8 (Programming – Intermediate). Train a character-level LSTM language model on a small text file (approximately 50KB – Shakespeare works well). Generate 200 characters of text using four different temperature values: $\tau \in \{0.2, 0.5, 1.0, 1.5\}$. For each temperature, print the generated text and describe qualitatively how the temperature affects (a) coherence, (b) diversity, and (c) the frequency of real English words in the output.

Exercise 5.9 (Programming – Advanced). Replicate the perplexity comparison from Section 5.5.3 on a corpus of your choice. Train a Kneser-Ney smoothed trigram model (using NLTK or your implementation from Chapter 3) and an LSTM language model on the same training set. Evaluate both on the same held-out test set and report perplexity. Ensure that both models use the same vocabulary and the same train/test split. Is the improvement consistent with the Penn Treebank results in the chapter?

Exercise 5.10 (Programming – Advanced). Implement weight tying in an LSTM language model, following Section 5.5.1. Train two versions of the same model on the same corpus: one with weight tying (embedding matrix shared with output projection) and one without. This requires that the embedding dimension equals the hidden dimension. Report: (a) the total parameter count for each model, (b) the final test perplexity for each, and (c) the training time per epoch. Does weight tying improve perplexity? By how much does it reduce the parameter count?

Hint: In PyTorch, implement weight tying by setting `self.linear.weight = self.embedding.weight` after module initialization.