```
-->
```

Chapter 6: Deep Learning for Natural Language Processing — Recurrent Networks and Transformers

Covered Material	Value
Chapter	Chapter 8 (Jurafsky & Martin)
Related Week	Week 6
Related Slides / Files	week6.md, 8.md

Index

Brief Content Summary

This chapter provides a comprehensive treatment of deep learning architectures for Natural Language Processing, progressing from Recurrent Neural Networks (RNNs) to the Transformer architecture. RNNs introduce temporal structure through recurrent connections that allow hidden states to carry information across time steps, enabling language modeling without fixed context windows. The chapter derives the forward inference equations for simple RNNs and RNN-based language models (RNNLMs), including the cross-entropy training objective. Long Short-Term Memory (LSTM) networks are presented as solutions to the vanishing gradient problem, employing gating mechanisms to control information flow. The encoder-decoder architecture extends RNNs to sequence-to-sequence tasks like machine translation. The attention mechanism addresses the encoder's representational bottleneck by computing dynamic context vectors as weighted sums of encoder hidden states. Finally, Transformers replace recurrence entirely with self-attention, using Query-Key-Value projections and scaled dot-product attention to achieve parallelizable sequence modeling.

Abstract

This chapter develops the theoretical foundations of deep learning for sequential language processing. The exposition begins with Recurrent Neural Networks (RNNs), which process sequences element-by-element while maintaining hidden state representations that encode prior context. The forward inference equations are derived: $h_t = g(Uh_{t-1} + Wx_t)$ for the hidden state and $\hat{y}_t = \mathrm{softmax}(Vh_t)$ for the output distribution. RNN language models are trained via cross-entropy loss against the true next word: $L_{CE} = -\log \hat{y}_t[w_{t+1}]$. The chapter addresses the vanishing gradient problem that limits RNNs' ability to capture long-range dependencies, presenting Long Short-Term Memory (LSTM) networks with their forget, add, and output gates. The encoder-decoder architecture is introduced for sequence-to-sequence tasks, where an encoder maps input sequences to context representations consumed by an autoregressive decoder. The attention mechanism computes dynamic context vectors $c_i = \sum_j \alpha_{ij} h_j^e$ via softmax-normalized relevance scores. The chapter culminates with the Transformer architecture, which replaces recurrence with self-attention: $\mathrm{SelfAttention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$. Multi-head attention, layer normalization, and positional encodings complete the Transformer framework that underlies modern large language models.

Keywords

Recurrent Neural Networks; RNN Language Models; Backpropagation Through Time; Vanishing Gradients; Long Short-Term Memory; LSTM Gates; Encoder-Decoder; Sequence-to-Sequence; Attention Mechanism; Self-Attention; Transformers; Query-Key-Value; Multi-Head Attention; Positional Encodings; Cross-Entropy Loss

1 Introduction

This section motivates the transition from feedforward architectures to recurrent and attention-based models for sequential language processing.

1.1 The Temporal Nature of Language

Language is inherently temporal. Spoken language unfolds as a sequence of acoustic events; written language is comprehended and produced as a sequential stream of tokens. This temporal structure is reflected in common metaphors: we speak of the *flow of conversation*, *news feeds*, and *streaming* content—all emphasizing language as a sequence unfolding in time.

Some algorithms explicitly model this temporal structure. The Viterbi algorithm for HMM-based sequence labeling, introduced in Chapter 5, processes input word-by-word, carrying forward information at each step. However, many machine learning approaches—including the feedforward classifiers of earlier chapters—assume simultaneous access to all input features, treating documents as unordered bags of words.

1.2 From Feedforward to Recurrent Architectures

Feedforward neural networks can model sequences through sliding windows: concatenating embeddings for n consecutive words and predicting the next word. This approach has two fundamental limitations:

1. **Fixed context size.** The network observes only $n-1$ preceding words, regardless of sentence length. Information beyond the window boundary is invisible.

2. **No parameter sharing across positions.** Each position in the window has dedicated parameters, preventing the network from learning position-invariant patterns.

Recurrent Neural Networks (RNNs) address both limitations through a fundamentally different architecture. Rather than processing fixed windows, RNNs maintain a **hidden state** h_t that is updated at each time step as a function of the current input and the previous hidden state:

$$h_t = f(h_{t-1}, x_t)$$

This recurrent connection allows information to flow forward through arbitrarily long sequences. The hidden state h_t can, in principle, encode information from all preceding inputs $x_1, x_2, \ldots, x_t$—without imposing a fixed context window.

1.3 Chapter Overview

The chapter proceeds as follows:

- **Section 2** introduces the simple RNN architecture and its weight matrices.
- **Section 3** applies RNNs to language modeling with cross-entropy training.
- **Section 4** extends RNNs to sequence labeling, classification, and generation.
- **Section 5** presents advanced architectures: stacked RNNs, bidirectional RNNs, and LSTMs.
- **Section 6** develops the encoder-decoder model for sequence-to-sequence tasks.
- **Section 7** introduces attention mechanisms.
- **Section 8** presents the Transformer architecture.

2 Recurrent Neural Networks

This section introduces the architecture and mathematics of simple recurrent neural networks.

2.1 The Simple Recurrent Network

A **Recurrent Neural Network (RNN)** is any network containing a cycle within its connections—the value of some unit depends, directly or indirectly, on its own earlier outputs. The constrained architecture known as an **Elman Network** or **Simple Recurrent Network (SRN)** has proven particularly effective for language processing.

The key architectural innovation is the **recurrent connection** from the hidden layer at time $t - 1$ to the hidden layer at time t. This connection provides a form of **memory**: the hidden layer from the previous time step encodes earlier processing and informs decisions at later points.

Critically, this approach imposes no fixed-length limit on prior context. The context embodied in the previous hidden layer can, in principle, include information extending back to the beginning of the sequence.

2.2 RNN Components and Weight Matrices

An RNN processes sequences one element at a time. At each time step t:

1. The network receives an input vector x_t (e.g., a word embedding)
2. The network produces a hidden state h_t
3. The network generates an output y_t

The computation involves three weight matrices:

Matrix	Dimensions	Function
$\mathbf{W}$	$d_h \times d_{in}$	Maps input to hidden layer
$\mathbf{U}$	$d_h \times d_h$	Maps previous hidden state to current hidden layer
$\mathbf{V}$	$d_{out} \times d_h$	Maps hidden layer to output

where d_{in} is the input dimensionality (embedding size), d_h is the hidden layer dimensionality, and d_{out} is the output dimensionality (vocabulary size for language modeling).

The matrix $\mathbf{U}$ is the distinguishing feature of RNNs—it connects the hidden layer from the previous time step to the current hidden layer computation.

2.3 Forward Inference

Forward inference computes the output sequence $y_1, y_2, \ldots, y_T$ given an input sequence $x_1, x_2, \ldots, x_T$. At each time step t:

Hidden state computation:

$$h_t = g(Uh_{t-1} + Wx_t)$$

Output computation:

$$y_t = f(Vh_t)$$

where g is a nonlinear activation function (typically tanh or ReLU) and f is the output function (typically softmax for classification).

Initialization. The hidden state h_0 is initialized to the zero vector before processing the first input.

Expanded equations for language modeling:

$$e_t = Ex_t$$

$$h_t = g(Uh_{t-1} + We_t)$$

$$\hat{y}_t = \mathrm{softmax}(Vh_t)$$

where E is the embedding matrix and x_t is a one-hot vector selecting the t-th word's embedding.

2.4 Unrolling in Time

The recurrent structure can be visualized by **unrolling** the network across time steps. Each layer is replicated for each time step, with:

- Different hidden state values $h_1, h_2, \ldots, h_T$
- Different output values $y_1, y_2, \ldots, y_T$
- **Shared** weight matrices $\mathbf{W}$, $\mathbf{U}$, $\mathbf{V}$ across all time steps

This unrolling reveals that an RNN is equivalent to a deep feedforward network whose depth equals the sequence length. Training proceeds via **Backpropagation Through Time (BPTT)**: gradients are propagated backward through the unrolled graph, accumulating contributions from each time step.

The unrolled view clarifies two key properties:

1. The hidden state h_t influences both the current output y_t and all future hidden states $h_{t+1}, h_{t+2}, \ldots$
2. Error signals must propagate backward through all time steps, enabling the network to learn long-range dependencies (in principle)

---

3 RNN Language Models

This section develops RNNs for the language modeling task with detailed mathematical derivations.

3.1 Language Modeling with RNNs

Recall that language models assign probabilities to sequences by decomposing into conditional next-word probabilities:

$$P(w_{1:n}) = \prod_{i=1}^{n} P(w_i | w_{1:i-1})$$

N-gram models approximate this by conditioning on only $n-1$ previous words. Feedforward language models condition on a fixed window. **RNN language models (RNNLMs)** condition on the entire preceding context, encoded in the hidden state.

At each time step t, the RNNLM:

1. Encodes the current word as an embedding: $e_t = Ex_t$
2. Updates the hidden state: $h_t = g(Uh_{t-1} + We_t)$
3. Produces a probability distribution over the vocabulary: $\hat{y}_t = \text{softmax}(Vh_t)$

The probability of word k being the next word is:

$$P(w_{t+1} = k | w_1, \ldots, w_t) = \hat{y}_t[k]$$

where $\hat{y}_t[k]$ denotes the k-th component of the softmax output.

The probability of an entire sequence is the product of per-position probabilities:

$$P(w_{1:n}) = \prod_{i=1}^{n} \hat{y}_i[w_i]$$

3.2 Training with Cross-Entropy Loss

RNNLMs are trained using **self-supervision**: the next word in the training corpus serves as the target at each position. No manual labeling is required.

The **cross-entropy loss** measures the divergence between the predicted distribution $\hat{y}_t$ and the true distribution y_t (a one-hot vector for the actual next word):

$$L_{CE} = -\sum_{w \in V} y_t[w] \log \hat{y}_t[w]$$

Since y_t is one-hot (only the true next word has probability 1), this simplifies to:

$$L_{CE}(\hat{y}_t, y_t) = -\log \hat{y}_t[w_{t+1}]$$

The loss is the negative log-probability assigned to the correct next word. Lower loss means higher probability on the correct word.

Total sequence loss averages over all positions:

$$L = \frac{1}{T} \sum_{t=1}^{T} L_{CE}(\hat{y}_t, y_t) = \frac{1}{T} \sum_{t=1}^{T} -\log \hat{y}_t[w_{t+1}]$$

Connection to perplexity. The perplexity of the model on a sequence is:

$$\text{PPL} = \exp(L) = \exp\left(\frac{1}{T} \sum_{t=1}^{T} -\log \hat{y}_t[w_{t+1}]\right)$$

3.3 Weight Tying

The embedding matrix $\mathbf{E}$ and the output matrix $\mathbf{V}$ serve similar purposes:

- $\mathbf{E}$ maps words to dense vectors (embeddings)
- $\mathbf{V}$ maps hidden states to vocabulary-sized scores

When the embedding dimension equals the hidden dimension ($d_e = d_h = d$), we have:

- $\mathbf{E} \in \mathbb{R}^{d \times |V|}$
- $\mathbf{V} \in \mathbb{R}^{|V| \times d}$

These are transposes of each other. **Weight tying** uses a single embedding matrix for both purposes:

$$e_t = Ex_t$$

$$h_t = g(Uh_{t-1} + We_t)$$

$$\hat{y}_t = \text{softmax}(E^T h_t)$$

The matrix E^T (sometimes called the **unembedding matrix**) replaces V. This reduces parameters and often improves perplexity by enforcing consistency between input and output word representations.

3.4 Worked Example: Forward Pass in an RNNLM

Consider an RNNLM processing the input "So long" to predict the sequence "long and".

Vocabulary and embeddings:

Word	Index	Embedding
and	0	$[0.087, 0.940]^T$
for	1	$[0.698, 0.711]^T$
long	2	$[0.474, 0.897]^T$
so	3	$[0.698, 0.978]^T$
thanks	4	$[0.122, 0.175]^T$

Input encoding (one-hot):

$$x_0 = [0, 0, 0, 1, 0]^T \quad \text{(so)}$$

$$x_1 = [0, 0, 1, 0, 0]^T \quad \text{(long)}$$

Embedding lookup:

$$e_0 = Ex_0 = \begin{bmatrix} 0.698 \\ 0.978 \end{bmatrix}$$

$$e_1 = Ex_1 = \begin{bmatrix} 0.474 \\ 0.897 \end{bmatrix}$$

Weight matrices (randomly initialized):

$$\mathbf{W} = \begin{bmatrix} 0.375 & 0.951 \\ 0.732 & 0.599 \end{bmatrix}, \quad \mathbf{U} = \begin{bmatrix} 0.156 & 0.156 \\ 0.058 & 0.866 \end{bmatrix}$$

$$\mathbf{V} = \begin{bmatrix} 0.601 & 0.683 \\ 0.021 & 0.970 \\ 0.832 & 0.212 \\ 0.182 & 0.183 \\ 0.304 & 0.525 \end{bmatrix}$$

Time step 0: Process "so", predict next word.

Hidden state (with $h_{-1} = \mathbf{0}$ and ReLU activation):

$$h_0 = g(Uh_{-1} + We_0) = g\left(\mathbf{0} + \begin{bmatrix} 0.375 & 0.951 \\ 0.732 & 0.599 \end{bmatrix} \begin{bmatrix} 0.698 \\ 0.978 \end{bmatrix}\right)$$

$$= g\left(\begin{bmatrix} 0.375 \times 0.698 + 0.951 \times 0.978 \\ 0.732 \times 0.698 + 0.599 \times 0.978 \end{bmatrix}\right) = g\left(\begin{bmatrix} 1.192 \\ 1.097 \end{bmatrix}\right) = \begin{bmatrix} 1.192 \\ 1.097 \end{bmatrix}$$

Output distribution:

$$\hat{y}_0 = \text{softmax}(Vh_0) = \text{softmax}\left(\begin{bmatrix} 1.466 \\ 1.089 \\ 1.224 \\ 0.418 \\ 0.938 \end{bmatrix}\right) = \begin{bmatrix} 0.293 \\ 0.201 \\ 0.230 \\ 0.103 \\ 0.173 \end{bmatrix}$$

Prediction: $\arg\max = 0$ → "and" (incorrect; true next word is "long")

Time step 1: Process "long", predict next word.

Hidden state (now $h_0 \neq \mathbf{0}$):

$$
\begin{aligned}
h_1 &= g(Uh_0 + We_1) \\
&= g\left(\begin{bmatrix}0.156 & 0.156\\0.058 & 0.866\end{bmatrix}\begin{bmatrix}1.192\\1.097\end{bmatrix} + \begin{bmatrix}0.375 & 0.951\\0.732 & 0.599\end{bmatrix}\begin{bmatrix}0.474\\0.897\end{bmatrix}\right) \\
&= g\left(\begin{bmatrix}0.357\\1.019\end{bmatrix} + \begin{bmatrix}1.031\\0.884\end{bmatrix}\right) = \begin{bmatrix}1.388\\1.903\end{bmatrix}
\end{aligned}
$$

Output distribution:

$$\hat{y}_1 = \text{softmax}(Vh_1) = \begin{bmatrix}0.329\\0.254\\0.185\\0.071\\0.161\end{bmatrix}$$

Prediction: $\arg\max = 0$ → "and" (correct!)

Loss computation:

True next words: $w_1 = \text{long}$ (index 2), $w_2 = \text{and}$ (index 0)

$$
\begin{aligned}
L = \frac{1}{2}\left(-\log \hat{y}_0[2] - \log \hat{y}_1[0]\right) = \frac{1}{2}\left(-\log(0.230) - \log(0.329)\right) \\
= \frac{1}{2}(1.470 + 1.112) = 1.291
\end{aligned}
$$

Perplexity: $\text{PPL} = e^{1.291} \approx 3.64$

4 RNNs for NLP Tasks

This section extends RNNs beyond language modeling to sequence labeling, classification, and generation.

4.1 Sequence Labeling

In **sequence labeling**, each input token receives a label from a fixed tagset. Part-of-speech tagging and named entity recognition are canonical examples.

Architecture. The RNN processes the input sequence, producing hidden states $h_1, h_2, \ldots, h_n$. At each position, a softmax over the tagset produces label probabilities:

$$\hat{y}_t = \text{softmax}(Vh_t)$$

The predicted tag is $\arg\max_k \hat{y}_t[k]$.

Training. Cross-entropy loss is computed at each position against the gold tag:

$$L = \frac{1}{n}\sum_{t=1}^{n} -\log \hat{y}_t[y_t^*]$$

where y_t^* is the true tag at position t.

Difference from language modeling. In language modeling, we predict the *next* word; in sequence labeling, we predict a label for the *current* word. The output dimensionality is the tagset size (e.g., 17 for UD POS tags) rather than the vocabulary size.

4.2 Sequence Classification

In **sequence classification**, an entire sequence receives a single label. Sentiment analysis and topic classification are examples.

Architecture. The RNN processes the full sequence, but only the *final* hidden state h_n is used —it serves as a compressed representation of the entire input. This representation feeds into a feedforward network with softmax:

$$\hat{y} = \text{softmax}(W_{ff} \cdot h_n + b)$$

Alternative: Pooling. Instead of using only h_n, we can pool across all hidden states:

Mean pooling:

$$h_{mean} = \frac{1}{n}\sum_{i=1}^{n} h_i$$

Max pooling: The element-wise maximum across all hidden states:

$$h_{max}[k] = \max_{i\in\{1,\ldots,n\}} h_i[k]$$

Training. The loss is computed only at the final classification:

$$L = -\log \hat{y}[y^*]$$

Error signals backpropagate through the feedforward classifier, through the final hidden state, and through all preceding hidden states via BPTT. This is **end-to-end training**: the entire network is trained jointly for the downstream task.

4.3 Autoregressive Text Generation

RNN language models enable **autoregressive generation** (also called **causal LM generation**): generating text one token at a time, where each generated token is conditioned on all previously generated tokens.

Generation procedure:

1. Initialize with a start token $\langle s \rangle$
2. Compute $\hat{y}_1 = \text{softmax}(V h_1)$
3. **Sample** a word w_1 from the distribution $\hat{y}_1$
4. Use w_1's embedding as input for the next time step
5. Repeat until end token $\langle / s \rangle$ is generated or maximum length reached

Sampling strategies:

- **Greedy decoding:** Always select $\arg\max \hat{y}_t$
- **Random sampling:** Sample from the full distribution
- **Top-k sampling:** Sample from the k most probable words
- **Temperature scaling:** Divide logits by temperature τ before softmax; $\tau < 1$ sharpens the distribution, $\tau > 1$ flattens it

Conditioning on context. For tasks like translation or summarization, generation is *conditioned* on source text. Instead of starting with $\langle s \rangle$, we first encode the source and use the encoder's final hidden state to initialize generation

5 Advanced RNN Architectures

This section presents architectural extensions that improve RNN performance: stacking, bidirectionality, and gating mechanisms.

5.1 Stacked RNNs

Stacked RNNs (also called deep RNNs) consist of multiple RNN layers, where the output sequence of one layer serves as the input sequence for the next:

$$h_t^{(1)} = g(U^{(1)}h_{t-1}^{(1)} + W^{(1)}x_t)$$

$$h_t^{(2)} = g(U^{(2)}h_{t-1}^{(2)} + W^{(2)}h_t^{(1)})$$

$$\vdots$$

$$h_t^{(L)} = g(U^{(L)}h_{t-1}^{(L)} + W^{(L)}h_t^{(L-1)})$$

The final layer's hidden states $h_t^{(L)}$ produce the output.

Motivation. Stacking enables hierarchical representation learning. Lower layers may capture local syntactic patterns; higher layers may capture longer-range semantic dependencies. This parallels the visual system, where early layers detect edges and later layers detect complex shapes.

Trade-offs. Deeper networks require more computation and are harder to train. The optimal depth is task- and data-dependent, typically 2–4 layers for many NLP tasks.

5.2 Bidirectional RNNs

Standard RNNs process sequences left-to-right, so h_t encodes only the *left* context $x_1, \ldots, x_t$. For tasks where we have access to the full sequence (unlike autoregressive generation), we can also leverage *right* context.

A **bidirectional RNN** runs two separate RNNs:

Forward RNN:

$$\overrightarrow{h}_t = \text{RNN}_{forward}(x_1, \ldots, x_t)$$

Backward RNN (processes sequence in reverse):

$$\overleftarrow{h}_t = \text{RNN}_{backward}(x_t, \ldots, x_n)$$

Concatenated representation:

$$h_t = [\overrightarrow{h}_t; \overleftarrow{h}_t] = \overrightarrow{h}_t \oplus \overleftarrow{h}_t$$

where $\oplus$ denotes vector concatenation.

The bidirectional hidden state h_t has dimension $2d_h$ and captures both left and right context. This is particularly valuable for sequence labeling, where the tag for a word depends on both preceding and following context.

For sequence classification, the final forward hidden state $\overrightarrow{h}_n$ and initial backward hidden state $\overleftarrow{h}_1$ are concatenated to represent the full sequence.

5.3 The Vanishing Gradient Problem

In practice, simple RNNs struggle to learn long-range dependencies. Consider:

> *The flights the airline was canceling were full.*

Predicting "were" (plural) requires remembering "flights" across the intervening clause "the airline was canceling." The singular "airline" provides misleading local context.

Two related problems impede learning such dependencies:

Vanishing gradients. During BPTT, gradients are multiplied by weight matrices at each time step. If the largest eigenvalue of U is less than 1, gradients shrink exponentially:

$$\frac{\partial L}{\partial h_1} = \frac{\partial L}{\partial h_T} \cdot \prod_{t=2}^{T} \frac{\partial h_t}{\partial h_{t-1}}$$

After many multiplications, gradients approach zero, preventing updates to early-layer weights.

Conflicting objectives. The hidden state must simultaneously:

1. Represent information useful for the *current* prediction
2. Carry forward information needed for *future* predictions

These objectives may conflict, and the network has no explicit mechanism to manage this trade-off.

5.4 Long Short-Term Memory Networks

Long Short-Term Memory (LSTM) networks address these problems through an explicit **context vector** c_t (also called the cell state) and **gating mechanisms** that control information flow.

LSTM components:

Component	Symbol	Purpose
Cell state	c_t	Long-term memory
Hidden state	h_t	Short-term output
Forget gate	f_t	Controls what to remove from cell
Input (add) gate	i_t	Controls what to add to cell
Output gate	o_t	Controls what to expose as output

Gate structure. Each gate computes a sigmoid-weighted mask:

$$\text{gate} = \sigma(U_{gate} \cdot h_{t-1} + W_{gate} \cdot x_t)$$

The sigmoid outputs values in $(0, 1)$, which are then used for element-wise multiplication to "open" or "close" information flow.

LSTM equations:

Forget gate — decide what to remove from the cell state:

$$f_t = \sigma(U_f h_{t-1} + W_f x_t)$$

Candidate cell content — compute potential new information:

$$g_t = \tanh(U_g h_{t-1} + W_g x_t)$$

Input gate — decide what new information to add:

$$i_t = \sigma(U_i h_{t-1} + W_i x_t)$$

Cell state update — combine old (gated) content with new (gated) content:

$$c_t = (c_{t-1} \odot f_t) + (g_t \odot i_t)$$

where $\odot$ denotes element-wise (Hadamard) multiplication.

Output gate — decide what to expose as hidden state:

$$o_t = \sigma(U_o h_{t-1} + W_o x_t)$$

Hidden state — gated, transformed cell content:

$$h_t = o_t \odot \tanh(c_t)$$

Gradient flow. The cell state c_t provides a "highway" for gradients: the additive update $c_t = c_{t-1} \odot f_t + \ldots$ allows gradients to flow backward without repeated multiplication by weight matrices, mitigating vanishing gradients.

Intuition. The LSTM learns *when* to update its memory (input gate), *what* to forget (forget gate), and *what* to output (output gate). These decisions are context-dependent, allowing the network to maintain relevant information across long sequences

6 The Encoder-Decoder Architecture

This section develops the encoder-decoder model for sequence-to-sequence tasks.

6.1 Sequence-to-Sequence Models

Many NLP tasks require mapping an input sequence to an output sequence of *different length*:

Task	Input	Output
Machine translation	English sentence	French sentence
Summarization	Document	Summary
Question answering	Question + context	Answer

Unlike sequence labeling (where input and output have the same length with aligned elements), these tasks require generating output sequences whose length and content are determined dynamically.

Encoder-decoder (or **sequence-to-sequence**) networks address this through two components:

1. **Encoder:** Processes the input sequence and produces a context representation
2. **Decoder:** Generates the output sequence conditioned on the context

6.2 The Encoder

The encoder is an RNN (typically an LSTM or stacked bidirectional LSTM) that processes the input sequence $x_1, x_2, \ldots, x_n$ and produces:

1. A sequence of hidden states $h^e_1, h^e_2, \ldots, h^e_n$
2. A **context vector** c that summarizes the entire input

In the simplest version, the context is the final encoder hidden state:

$$c = h^e_n$$

This context vector must encode *everything* the decoder needs to know about the input sequence.

6.3 The Decoder

The decoder is an autoregressive RNN that generates the output sequence $y_1, y_2, \ldots, y_m$ one token at a time, conditioned on:

1. The context vector c
2. Previously generated tokens $y_1, \ldots, y_{t-1}$

Decoder equations:

$$h^d_0 = c$$

$$h^d_i = g(\hat{y}_{i-1}, h^d_{i-1}, c)$$

$$\hat{y}_i = \text{softmax}(Vh^d_i)$$

The context c is provided to every decoder step, not just the first. This ensures the context's influence doesn't diminish as generation proceeds.

Decoding process:

1. Initialize decoder hidden state with context: $h^d_0 = c$
2. Start with separator token $\langle s \rangle$
3. At each step, sample from $\hat{y}_t$ and use the sampled token as input for the next step
4. Continue until $\langle / s \rangle$ is generated or maximum length reached

Probability decomposition:

$$P(y|x) = \prod_{i=1}^{m} P(y_i|y_1, \ldots, y_{i-1}, x)$$

Each factor is computed as $\hat{y}_i[y_i]$—the softmax probability of the correct token at position i.

6.4 Training Encoder-Decoder Models

Training uses **parallel corpora**: pairs of (source, target) sequences. For machine translation, these are sentence pairs in two languages.

Teacher forcing. During training, the decoder receives the *gold* target tokens as input at each step, rather than its own (potentially incorrect) predictions. This speeds training by providing consistent, correct context.

Training procedure:

1. Encode source sequence to obtain context c
2. For each target position i:
    - Compute $\hat{y}_i = \mathrm{softmax}(Vh_i^d)$
    - Compute loss: $L_i = -\log \hat{y}_i[y_i^*]$
    - Feed *gold* token y_i^* as input for next step (teacher forcing)
3. Total loss: $L = \frac{1}{m} \sum_{i=1}^{m} L_i$
4. Backpropagate through decoder *and* encoder

End-to-end training. Gradients flow from the decoder loss through the context vector to the encoder weights. Both components are trained jointly to minimize translation (or summarization, etc.) loss

7 Attention Mechanisms

This section introduces attention as a solution to the encoder-decoder bottleneck problem.

7.1 The Bottleneck Problem

In the basic encoder-decoder model, the context vector $c = h_n^e$ is the *only* information the decoder receives about the source sequence. This final hidden state must encode the meaning of the entire input—a representational **bottleneck**.

Problems with the bottleneck:

1. **Information loss.** Early parts of long sequences may be poorly represented in h_n^e
2. **Fixed representation.** The same context is used for all decoder steps, even though different output positions may need different source information
3. **Gradient flow.** Gradients must flow through the bottleneck, potentially limiting learning

For machine translation, consider translating "The green witch arrived" to Spanish "Llegó la bruja verde." When generating "verde" (green), the decoder needs information about "green"—but this may be poorly preserved after processing "witch arrived."

7.2 Attention as Weighted Context

The **attention mechanism** allows the decoder to access *all* encoder hidden states, not just the final one. At each decoder step i, attention computes a custom context vector c_i as a weighted combination of encoder states:

$$c_i = \sum_{j=1}^{n} \alpha_{ij} h_j^e$$

The weights α_{ij} are **attention weights**: they determine how much the decoder at step i should "attend to" each encoder position j.

Key insight: The context is now *dynamic*—different for each decoder step. When generating "verde," the attention can weight "green" heavily; when generating "bruja," it can weight "witch."

7.3 Dot-Product Attention

How do we compute the attention weights α_{ij}? The idea is to measure **relevance**: how relevant is each encoder hidden state h_j^e to the current decoder state h_{i-1}^d?

Dot-product attention uses the dot product as a similarity measure:

$$\text{score}(h^d_{i-1}, h^e_j) = h^d_{i-1} \cdot h^e_j = \sum_{k=1}^{d} h^d_{i-1}[k] \cdot h^e_j[k]$$

High dot product indicates high similarity; low (or negative) indicates low similarity.

Normalization with softmax. The raw scores are converted to a probability distribution:

$$\alpha_{ij} = \text{softmax}_j \left(\text{score}(h^d_{i-1}, h^e_j)\right) = \frac{\exp(\text{score}(h^d_{i-1}, h^e_j))}{\sum_{k=1}^{n} \exp(\text{score}(h^d_{i-1}, h^e_k))}$$

The attention weights $\alpha_{i1}, \alpha_{i2}, \ldots, \alpha_{in}$ sum to 1 and represent a distribution over source positions.

7.4 Computing the Context Vector

Complete attention computation at decoder step i:

1. **Compute relevance scores** for each encoder state:

$$s_j = h^d_{i-1} \cdot h^e_j \quad \text{for } j = 1, \ldots, n$$

2. **Normalize to attention weights:**

$$\alpha_{ij} = \frac{\exp(s_j)}{\sum_{k=1}^{n} \exp(s_k)}$$

3. **Compute weighted context:**

$$c_i = \sum_{j=1}^{n} \alpha_{ij} h^e_j$$

4. **Update decoder hidden state:**

$$h^d_i = g(\hat{y}_{i-1}, h^d_{i-1}, c_i)$$

Bilinear attention. A more expressive variant learns a weight matrix W_s:

$$\text{score}(h^d_{i-1}, h^e_j) = h^d_{i-1} W_s h^e_j$$

This allows the encoder and decoder to have different dimensionalities and learns which aspects of similarity matter for the task.

Attention as soft alignment. The attention weights can be interpreted as a soft alignment between source and target positions. Visualizing α_{ij} often reveals linguistically meaningful correspondences—verbs attending to verbs, adjectives to adjectives—learned without explicit alignment supervision

8 Transformers

This section introduces the Transformer architecture, which replaces recurrence with self-attention.

8.1 From Recurrence to Self-Attention

RNNs process sequences sequentially: h_t depends on h_{t-1}, which depends on h_{t-2}, and so on. This sequential dependency has two drawbacks:

1. **No parallelization.** Each time step must wait for the previous step to complete
2. **Long paths for distant dependencies.** Information from position 1 must flow through all intermediate positions to reach position n

Transformers replace recurrence with **self-attention**: each position can directly attend to all other positions, enabling parallel computation and constant-length dependency paths.

Self-attention computes a weighted combination of *all* positions in the input:

$$y_i = \sum_{j=1}^{n} \alpha_{ij} x_j$$

where the attention weights α_{ij} are computed from the inputs themselves (hence "self"-attention).

8.2 Query, Key, and Value Projections

Transformer attention introduces three distinct roles for each position:

Role	Symbol	Purpose	Projection Matrix
Query	q_i	What position i is looking for	W^Q
Key	k_i	What position i offers to be matched against	W^K

Role	Symbol	Purpose	Projection Matrix
Value	v_i	What position i contributes if matched	W^V

Each input x_i is projected into all three roles:

$$q_i = W^Q x_i$$

$$k_i = W^K x_i$$

$$v_i = W^V x_i$$

where $W^Q, W^K, W^V \in \mathbb{R}^{d \times d}$ are learned projection matrices.

Intuition. The query represents what information position i needs; keys represent what information each position provides; values represent the actual content to be aggregated.

8.3 Scaled Dot-Product Self-Attention

Attention scores are computed as dot products between queries and keys:

$$\text{score}(x_i, x_j) = q_i \cdot k_j = (W^Q x_i) \cdot (W^K x_j)$$

Scaling. For high-dimensional vectors, dot products can grow large, pushing softmax into regions with vanishing gradients. The scores are scaled by $\sqrt{d_k}$:

$$\text{score}(x_i, x_j) = \frac{q_i \cdot k_j}{\sqrt{d_k}}$$

Output computation:

$$\alpha_{ij} = \text{softmax}_j \left(\frac{q_i \cdot k_j}{\sqrt{d_k}} \right)$$

$$y_i = \sum_{j=1}^{n} \alpha_{ij} v_j$$

8.4 Matrix Form

For efficiency, self-attention is computed in matrix form. Given input matrix $X \in \mathbb{R}^{n \times d}$ (one row per position):

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

where $Q, K, V \in \mathbb{R}^{n \times d}$.

Self-attention formula:

$$\text{SelfAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Breakdown:

- $QK^T \in \mathbb{R}^{n \times n}$: all pairwise query-key dot products
- $\text{softmax}(\cdot)$: row-wise normalization to attention weights
- $(\cdot)V$: weighted combination of values

Causal masking. For autoregressive generation, position i should not attend to future positions $j > i$. This is enforced by setting future attention scores to $-\infty$ before softmax:

$$\text{score}(x_i, x_j) = \begin{cases} \frac{q_i \cdot k_j}{\sqrt{d_k}} & \text{if } j \leq i \\ -\infty & \text{if } j > i \end{cases}$$

8.5 Multi-Head Attention

A single attention head may not capture all relevant relationships. **Multi-head attention** runs H attention heads in parallel, each with its own projection matrices:

$$\text{head}_h = \text{SelfAttention}(XW_h^Q, XW_h^K, XW_h^V)$$

The heads are concatenated and projected:

$$\text{MultiHead}(X) = [\text{head}_1 \oplus \text{head}_2 \oplus \ldots \oplus \text{head}_H]W^O$$

where $W^O \in \mathbb{R}^{Hd_v \times d}$ projects the concatenated heads back to dimension d.

Intuition. Different heads can attend to different aspects: one head might capture syntactic dependencies, another semantic similarities, another positional patterns.

8.6 Transformer Blocks

A complete **Transformer block** combines:

1. **Multi-head self-attention**
2. **Feedforward network (FFN)**: two linear layers with nonlinearity
3. **Residual connections**: adding input to output
4. **Layer normalization**: normalizing activations

Block equations:

$$z = \text{LayerNorm}(x + \text{MultiHeadAttn}(x))$$

$$y = \text{LayerNorm}(z + \text{FFN}(z))$$

Layer normalization:

$$\mu = \frac{1}{d}\sum_{i=1}^{d} x_i$$

$$\sigma = \sqrt{\frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2}$$

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

where γ and β are learned scale and shift parameters.

8.7 Positional Encodings

Self-attention is permutation-invariant: it treats inputs as a *set*, not a *sequence*. To inject positional information, **positional encodings** are added to input embeddings:

$$x_i' = x_i + p_i$$

where p_i encodes position i.

Sinusoidal encodings (original Transformer):

$$p_{i,2k} = \sin\left(\frac{i}{10000^{2k/d}}\right)$$

$$p_{i,2k+1} = \cos\left(\frac{i}{10000^{2k/d}}\right)$$

Different dimensions encode position at different frequencies, allowing the model to learn relative positions.

Learned encodings. Alternatively, positional encodings can be learned parameters, one vector per position up to a maximum sequence length.

8.8 Transformer Language Models

Applying Transformers to language modeling:

1. Token embeddings + positional encodings → input
2. Stack of Transformer blocks → contextualized representations
3. Linear layer + softmax → next-word probability distribution

Training: Same self-supervised objective as RNNLMs:

$$L = \frac{1}{T} \sum_{t=1}^{T} -\log \hat{y}_t[w_{t+1}]$$

Advantages over RNNs:

- Fully parallelizable (no sequential dependencies)
- Constant-length paths for all dependencies
- Scales to very deep architectures (hundreds of layers)

9 Conclusion

This section summarizes the key concepts and their relationships.

This chapter has developed the theoretical foundations of deep learning for sequential language processing, progressing from recurrent architectures to attention-based Transformers.

Key contributions:

1. **Recurrent Neural Networks** introduce temporal structure through the hidden state recurrence $h_t = g(Uh_{t-1} + Wx_t)$. The hidden state carries forward information from all

preceding inputs, enabling language modeling without fixed context windows.

2. **RNN Language Models** are trained via cross-entropy loss $L = -\log \hat{y}_t[w_{t+1}]$, where the model learns to assign high probability to the correct next word. Weight tying ($V = E^T$) reduces parameters and improves performance.

3. **Vanishing gradients** limit RNNs' ability to capture long-range dependencies. Gradients shrink exponentially during backpropagation through time, preventing learning from distant context.

4. **LSTMs** solve this via gating mechanisms: the forget gate f_t controls memory deletion, the input gate i_t controls memory updates, and the output gate o_t controls exposure. The cell state c_t provides an additive gradient path.

5. **Encoder-decoder models** separate encoding (processing input) from decoding (generating output), enabling sequence-to-sequence tasks like translation. The context vector $c = h_n^e$ bridges encoder and decoder.

6. **Attention** addresses the bottleneck by computing dynamic context vectors $c_i = \sum_j \alpha_{ij} h_j^e$. Attention weights measure relevance via dot products, allowing the decoder to focus on different source positions at each step.

7. **Transformers** replace recurrence with self-attention: $\text{SelfAttention}(Q, K, V) = \text{softmax}(QK^T/\sqrt{d_k})V$. Query-key-value projections, multi-head attention, layer normalization, and positional encodings complete the architecture that underlies modern large language models.

The progression from RNNs to Transformers represents a fundamental shift: from sequential processing with implicit context (hidden states) to parallel processing with explicit context (attention weights). Transformers' scalability has enabled the development of large language models with hundreds of billions of parameters, achieving unprecedented performance across NLP tasks.

10 References

Bahdanau, D., Cho, K. H., & Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. In *Proceedings of ICLR*.

Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of EMNLP*.

Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14(2), 179–211.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8), 1735–1780.

Jurafsky, D., & Martin, J. H. (2024). *Speech and Language Processing* (3rd ed.). Chapter 8.

Mikolov, T., Karafiát, M., Burget, L., Černocký, J., & Khudanpur, S. (2010). Recurrent neural network based language model. In *Proceedings of INTERSPEECH*.

Schuster, M., & Paliwal, K. K. (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45, 2673–2681.

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in NeurIPS*.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. In *Advances in NeurIPS*.

Appendix A: RNN Forward Pass Pseudocode

```
function FORWARD_RNN(x, network) returns output sequence y:
    Input:
        x: input sequence [x_1, x_2, ..., x_T]
        network: contains E, W, U, V matrices

    h_0 ← zero vector of dimension d_h

    for t ← 1 to T do:
        # Embedding lookup
        e_t ← E · x_t

        # Hidden state update
        h_t ← g(U · h_{t-1} + W · e_t)

        # Output computation
        y_t ← softmax(V · h_t)
    return y = [y_1, y_2, ..., y_T]
```

Complexity:

- Time: $O(T \cdot d^2)$ — T steps, each with matrix-vector products of dimension d
- Space: $O(T \cdot d)$ — storing all hidden states for backpropagation

Appendix B: LSTM Gate Equations Summary

Input:

- Current input: x_t
- Previous hidden state: h_{t-1}
- Previous cell state: c_{t-1}

Gates:

Gate	Equation	Purpose
Forget	$f_t = \sigma(U_f h_{t-1} + W_f x_t + b_f)$	What to remove from cell
Input	$i_t = \sigma(U_i h_{t-1} + W_i x_t + b_i)$	What to add to cell
Output	$o_t = \sigma(U_o h_{t-1} + W_o x_t + b_o)$	What to expose

Cell and hidden state updates:

Computation	Equation
Candidate content	$g_t = \tanh(U_g h_{t-1} + W_g x_t + b_g)$
Cell state	$c_t = f_t \odot c_{t-1} + i_t \odot g_t$
Hidden state	$h_t = o_t \odot \tanh(c_t)$

Output:

- Current hidden state: h_t
- Current cell state: c_t

Parameters per LSTM layer:

- 4 weight matrices of shape $(d_h \times d_{in})$: W_f, W_i, W_o, W_g
- 4 weight matrices of shape $(d_h \times d_h)$: U_f, U_i, U_o, U_g
- 4 bias vectors of shape (d_h): b_f, b_i, b_o, b_g
- Total: $4(d_h \times d_{in} + d_h \times d_h + d_h) = 4d_h(d_{in} + d_h + 1)$

Appendix C: Transformer Self-Attention Computation

Input: Sequence $X \in \mathbb{R}^{n \times d}$

Step 1: Project to Q, K, V

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

Step 2: Compute attention scores

$$S = \frac{QK^T}{\sqrt{d_k}} \in \mathbb{R}^{n \times n}$$

Step 3: Apply causal mask (for autoregressive)

$$S_{ij} = \begin{cases} S_{ij} & \text{if } j \leq i \\ -\infty & \text{if } j > i \end{cases}$$

Step 4: Softmax to attention weights

$$A = \text{softmax}(S) \in \mathbb{R}^{n \times n}$$

(Row-wise softmax; each row sums to 1)

Step 5: Weighted combination of values

$$Y = AV \in \mathbb{R}^{n \times d}$$

Complexity:

- Time: $O(n^2 d)$ — dominated by QK^T computation
- Space: $O(n^2)$ — storing attention matrix

Glossary

Term	Definition
Attention	Mechanism computing weighted context from all encoder states
Autoregressive Generation	Generating tokens sequentially, each conditioned on previous outputs
Backpropagation Through Time (BPTT)	Backpropagation applied to unrolled RNN
Bidirectional RNN	RNN processing sequence in both directions; $h_t = [\overrightarrow{h}_t; \overleftarrow{h}_t]$
Causal Masking	Preventing attention to future positions in autoregressive models
Cell State	LSTM's long-term memory c_t, updated via gating
Context Vector	Encoder summary passed to decoder; $c = h_n^e$ or weighted sum
Cross-Entropy Loss	$L = -\log \hat{y}[y^*]$; loss for classification/language modeling
Encoder-Decoder	Architecture separating input encoding from output generation
Forget Gate	LSTM gate controlling what to remove from cell state
Gating	Using sigmoid-weighted masks to control information flow
Hidden State	RNN's internal representation h_t at time t
Input Gate	LSTM gate controlling what to add to cell state
Layer Normalization	Normalizing activations: $\mathrm{LN}(x) = \gamma \frac{x-\mu}{\sigma} + \beta$
LSTM	Long Short-Term Memory; gated RNN architecture
Multi-Head Attention	Parallel attention heads capturing different relationships
Output Gate	LSTM gate controlling what to expose as hidden state
Positional Encoding	Vector added to embeddings to encode sequence position
Query, Key, Value	Transformer projections for attention computation
Recurrent Neural Network (RNN)	Network with recurrent connections; $h_t = g(Uh_{t-1} + Wx_t)$
Residual Connection	Adding input to output: $y = x + f(x)$
Scaled Dot-Product Attention	$\mathrm{softmax}(QK^T/\sqrt{d_k})V$
Self-Attention	Attention where queries, keys, and values come from same sequence
Sequence-to-Sequence	Mapping input sequence to output sequence of different length
Stacked RNN	Multiple RNN layers; output of layer l is input to layer $l+1$
Teacher Forcing	Using gold tokens as decoder input during training
Transformer	Architecture replacing recurrence with self-attention
Unrolling	Visualizing RNN as feedforward network across time
Vanishing Gradients	Gradients shrinking exponentially during BPTT
Weight Tying	Using same matrix for input embeddings and output projection