

CS 489: Introduction to Natural Language Processing

Lecture 8: Neural Language Models

Instructor: Freda Shi

October 6, 2026



UNIVERSITY OF
WATERLOO

Outline

1. Language modeling as classification
2. Neural architectures: From N -gram to RNN and Transformer LMs
3. Training neural language models
4. Building a character-level LM

Recap: Where We Are

So far:

- **Lectures 1–2:** What is NLP? Tokenization (BPE, WordPiece, Unigram).
- **Lecture 3:** Word embeddings: distributional semantics, Word2Vec.
- **Lecture 4:** Supervised learning: logistic regression, neural classifiers.
- **Lecture 5:** Statistical language models: N -grams, smoothing, perplexity.
- **Lecture 6:** Neural network foundations: MLPs, backprop, RNNs, LSTMs, GRUs.
- **Lecture 7:** Transformers from scratch (attention, multi-head attention, encoder-decoder).

Today: **neural language models**: using neural networks to model

$P(x_t \mid x_1, \dots, x_{t-1})$.

- From N -gram LMs to RNN LMs to Transformer LMs.
- Training with teacher forcing and cross-entropy loss.
- Building a character-level LM from scratch.

Motivation: Why Go Neural?

Recall from Lecture 5 the limitations of statistical N -gram LMs:

- **Sparsity:** Most N -gram contexts are never observed.
 - Even with smoothing, many probabilities remain poorly estimated.
- **Fixed context:** A trigram model can only look at the previous 2 words.
 - “The students who passed the final _____” needs long-range context!
- **No learned word similarity:** “cat” and “kitten” have separate counts.
 - Smoothing/backoff share lower-order statistics, not semantic embeddings.
- **Exponential parameters:** An N -gram model over vocabulary V has $O(|V|^N)$ parameters.

Motivation: Why Go Neural?

Recall from Lecture 5 the limitations of statistical N -gram LMs:

- **Sparsity:** Most N -gram contexts are never observed.
 - Even with smoothing, many probabilities remain poorly estimated.
- **Fixed context:** A trigram model can only look at the previous 2 words.
 - “The students who passed the final _____” needs long-range context!
- **No learned word similarity:** “cat” and “kitten” have separate counts.
 - Smoothing/backoff share lower-order statistics, not semantic embeddings.
- **Exponential parameters:** An N -gram model over vocabulary V has $O(|V|^N)$ parameters.

Neural LMs address all of these via **continuous representations** and **parameter sharing**.

Language Modeling as Classification

Recall the chain rule decomposition of sequence probability:

$$P(\mathbf{x}_{1:n}) = P(x_1, x_2, \dots, x_n) = \prod_{i=1}^n \boxed{P(x_i \mid x_1, x_2, \dots, x_{i-1})}$$

Language Modeling as Classification

Recall the chain rule decomposition of sequence probability:

$$P(\mathbf{x}_{1:n}) = P(x_1, x_2, \dots, x_n) = \prod_{i=1}^n \boxed{P(x_i \mid x_1, x_2, \dots, x_{i-1})}$$

Each factor $P(x_i \mid x_1, \dots, x_{i-1})$ is a **classification problem**:

- **Input:** Context $x_1, \dots, x_{i-1}$.
- **Output:** A probability distribution over the vocabulary V .
- **Goal:** Assign high probability to the actual next token x_i .

Language Modeling as Classification

Recall the chain rule decomposition of sequence probability:

$$P(\mathbf{x}_{1:n}) = P(x_1, x_2, \dots, x_n) = \prod_{i=1}^n \boxed{P(x_i \mid x_1, x_2, \dots, x_{i-1})}$$

Each factor $P(x_i \mid x_1, \dots, x_{i-1})$ is a **classification problem**:

- **Input:** Context $x_1, \dots, x_{i-1}$.
- **Output:** A probability distribution over the vocabulary V .
- **Goal:** Assign high probability to the actual next token x_i .

We can use **any** classifier from the previous lectures:

- Linear model with features $\rightarrow$ neural N -gram model.
- RNN $\rightarrow$ RNN language model.
- Transformer decoder $\rightarrow$ GPT-style language model.

Neural N -Gram Language Models (Bengio et al., 2003)

Key idea: Replace the N -gram lookup table with a neural network.

For a trigram model, predict $P(x_t \mid x_{t-2}, x_{t-1})$:

Neural N -Gram Language Models (Bengio et al., 2003)

Key idea: Replace the N -gram lookup table with a neural network.

For a trigram model, predict $P(x_t \mid x_{t-2}, x_{t-1})$:

Step 1: Look up word embeddings:

$$\mathbf{c} = [\text{emb}(x_{t-2}); \text{emb}(x_{t-1})]$$

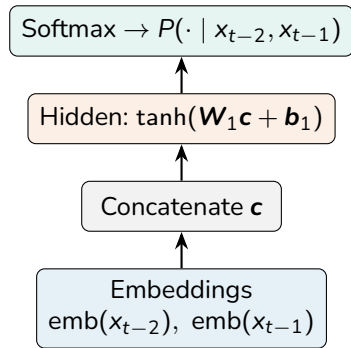
where $\text{emb}(\cdot) \in \mathbb{R}^d$ and $\mathbf{c} \in \mathbb{R}^{2d}$.

Step 2: Hidden layer (with tanh):

$$\mathbf{h} = \tanh(\mathbf{W}_1 \mathbf{c} + \mathbf{b}_1)$$

Step 3: Output distribution:

$$\mathbf{s} = \text{softmax}(\mathbf{W}_2 \mathbf{h} + \mathbf{b}_2) \in \mathbb{R}^{|V|}$$



Neural vs. Statistical N -Gram LMs

	Statistical N -gram LM	Neural N -gram LM
Parameters	$O(V ^N)$ (trigram: $\approx V ^3$)	$O(V (d + H) + H(N - 1)d)$
Scaling in N	Exponential	Linear for fixed d, H
Unseen contexts	Sparse counts; explicit smoothing.	Parameter sharing via embeddings.

d : embedding dimension; H : hidden size (both tunable).

Neural vs. Statistical N -Gram LMs

	Statistical N -gram LM	Neural N -gram LM
Parameters	$O(V ^N)$ (trigram: $\approx V ^3$)	$O(V (d + H) + H(N - 1)d)$
Scaling in N	Exponential	Linear for fixed d, H
Unseen contexts	Sparse counts; explicit smoothing.	Parameter sharing via embeddings.

d : embedding dimension; H : hidden size (both tunable).

Key advantage: generalization

With similar embeddings for *cat* and *kitten*, evidence for “the cat meowed” can help predict “the kitten meowed”.

Limitation of Neural N -Gram Models

The neural N -gram model still uses a **fixed context window**:

- A neural trigram only sees the previous 2 tokens.
- A neural 5-gram only sees the previous 4 tokens.
- Cannot capture dependencies like: “The **students** who passed the final exam **celebrated**.”

How to handle variable-length context?

Limitation of Neural N -Gram Models

The neural N -gram model still uses a **fixed context window**:

- A neural trigram only sees the previous 2 tokens.
- A neural 5-gram only sees the previous 4 tokens.
- Cannot capture dependencies like: “The **students** who passed the final exam **celebrated**.”

How to handle variable-length context?

1. **Average context embeddings** (bag-of-words):

$$P(x_i | x_{<i}) = \text{softmax}_v \left(\mathbf{W} \cdot \frac{1}{i-1} \sum_{j=1}^{i-1} \mathbf{w}_j \right), \text{ but this loses } \mathbf{word\ order}.$$

2. **Use a recurrent neural network**: process tokens sequentially, maintaining a hidden state.
3. **Use a Transformer decoder**: attend to all previous tokens in parallel.

RNN Language Models

Use an RNN to process the context **sequentially**, maintaining a hidden state:

Input embedding: $\mathbf{x}_t = \mathbf{U} \text{onehot}(x_t)$, where $\mathbf{U} \in \mathbb{R}^{d \times |V|}$ and $x_0 = \langle s \rangle$.

Hidden state update:

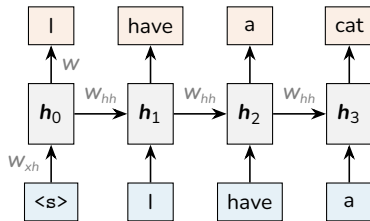
$$\mathbf{h}_t = f(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t)$$

Output distribution:

$$P_{\Theta}(x_t \mid x_1, \dots, x_{t-1}) = \text{softmax}_{x_t}(\mathbf{W}\mathbf{h}_{t-1})$$

Parameters: $\Theta = \{\mathbf{U}, \mathbf{W}_{xh}, \mathbf{W}_{hh}, \mathbf{W}\}$

- Same weights at each time step.
- $\mathbf{h}_t$ encodes the **entire** history $x_1, \dots, x_t$.
- No fixed context window!



Input: x_0, x_1, x_2, x_3 .

Target: predict x_1, x_2, x_3, x_4 .

RNN LM: Training

Loss at each time step:

$$L_t = -\log P_{\Theta}(x_t \mid x_1, \dots, x_{t-1})$$

RNN LM: Training

Loss at each time step:

$$L_t = -\log P_{\Theta}(x_t \mid x_1, \dots, x_{t-1})$$

Total loss for a sequence:

$$L = \sum_{t=1}^T L_t = \sum_{t=1}^T -\log P_{\Theta}(x_t \mid x_1, \dots, x_{t-1}) = -\log P_{\Theta}(\mathbf{x})$$

- This is the **negative log-likelihood**, which is exactly the cross-entropy loss.
- Compute $\nabla_{\Theta} L$ via **backpropagation through time** (BPTT). Minimize with Optimizer.

RNN LM: Training

Loss at each time step:

$$L_t = -\log P_{\Theta}(x_t \mid x_1, \dots, x_{t-1})$$

Total loss for a sequence:

$$L = \sum_{t=1}^T L_t = \sum_{t=1}^T -\log P_{\Theta}(x_t \mid x_1, \dots, x_{t-1}) = -\log P_{\Theta}(\mathbf{x})$$

- This is the **negative log-likelihood**, which is exactly the cross-entropy loss.
- Compute $\nabla_{\Theta} L$ via **backpropagation through time** (BPTT). Minimize with Optimizer.

Connection to perplexity: $\text{PPL}(\mathbf{x}) = \exp\left(\frac{L}{T}\right)$.

Lower loss $\Leftrightarrow$ lower perplexity $\Leftrightarrow$ better language model.

Teacher Forcing

Question: During training, what token do we feed into the update for h_t ?

Teacher Forcing

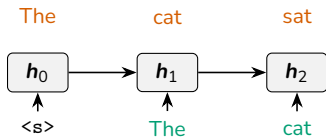
Question: During training, what token do we feed into the update for h_t ?

Option A: Free-running

- Feed the model's **own prediction** $\hat{x}_t$ to compute h_t .

Option B: Teacher forcing

- Feed the **ground-truth** x_t to compute h_t .
- The “teacher” corrects the input.
- Standard practice for training.



← ground-truth inputs (teacher forcing)

RNN LM: Strengths and Weaknesses

Strengths:

- Can process **variable-length** context.
- No fixed N -gram window.
- Parameter count is **independent** of sequence length.
- Useful neural language modeling baselines.

Weaknesses:

- **Sequential** hidden-state updates: $O(T)$ steps.
- **Effective context varies**: a [PTB/WikiText-2 LSTM study](#) found about 200 tokens on average.
- **Information bottleneck**: entire history compressed into h_t .
- Slow to train on modern hardware.

RNN LM: Strengths and Weaknesses

Strengths:

- Can process **variable-length** context.
- No fixed N -gram window.
- Parameter count is **independent** of sequence length.
- Useful neural language modeling baselines.

Weaknesses:

- **Sequential** hidden-state updates: $O(T)$ steps.
- **Effective context varies**: a [PTB/WikiText-2 LSTM study](#) found about 200 tokens on average.
- **Information bottleneck**: entire history compressed into h_t .
- Slow to train on modern hardware.

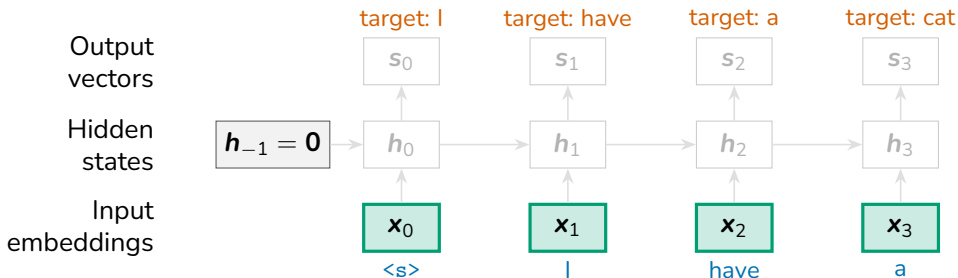
Historical note: RNN/LSTM LMs were important milestones in the 2010s (Mikolov et al., 2010; Zaremba et al., 2014; Merity et al., 2018).

⇒ Modern large LMs mostly use **Transformer decoders**.

RNN Training: What Can Run in Parallel?

Teacher-forced forward pass: all gold input tokens are known ($T = 4$).

$$h_t = f(W_{hh}h_{t-1} + W_{xh}x_t), \quad s_t = \text{softmax}(Wh_t).$$



Compute now Gray fill: available Faded: waiting

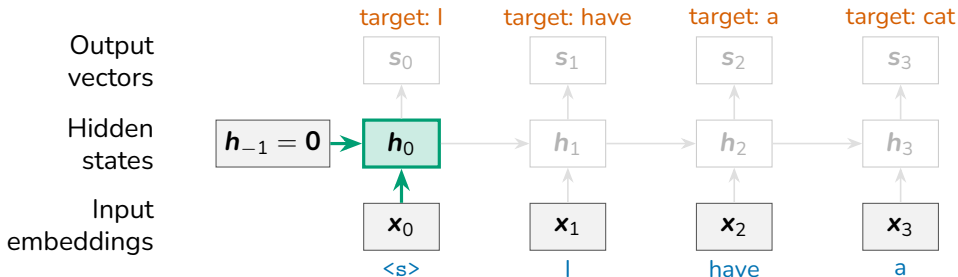
Stage 1: Compute x_0, x_1, x_2, x_3 in parallel.

Each box is one fixed-width vector computation.

RNN Training: What Can Run in Parallel?

Teacher-forced forward pass: all gold input tokens are known ($T = 4$).

$$h_t = f(W_{hh}h_{t-1} + W_{xh}x_t), \quad s_t = \text{softmax}(Wh_t).$$



Compute now Gray fill: available Faded: waiting

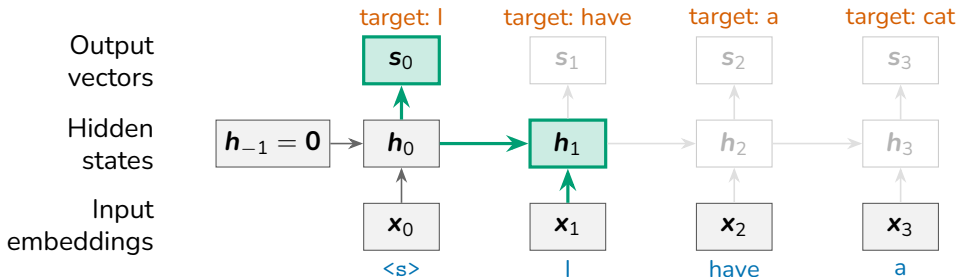
Stage 2: Compute h_0 ; later hidden states must wait.

$h_{-1} = 0$ is given, and x_0 is ready.

RNN Training: What Can Run in Parallel?

Teacher-forced forward pass: all gold input tokens are known ($T = 4$).

$$h_t = f(W_{hh}h_{t-1} + W_{xh}x_t), \quad s_t = \text{softmax}(Wh_t).$$



Compute now Gray fill: available Faded: waiting

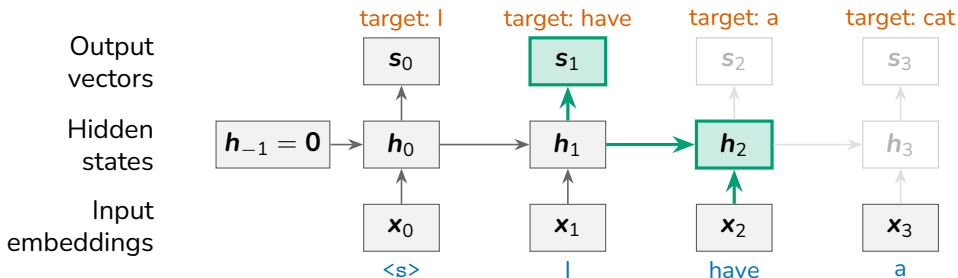
Stage 3: Compute h_1 and s_0 in parallel.

Both need h_0 , but neither depends on the other.

RNN Training: What Can Run in Parallel?

Teacher-forced forward pass: all gold input tokens are known ($T = 4$).

$$h_t = f(W_{hh}h_{t-1} + W_{xh}x_t), \quad s_t = \text{softmax}(Wh_t).$$



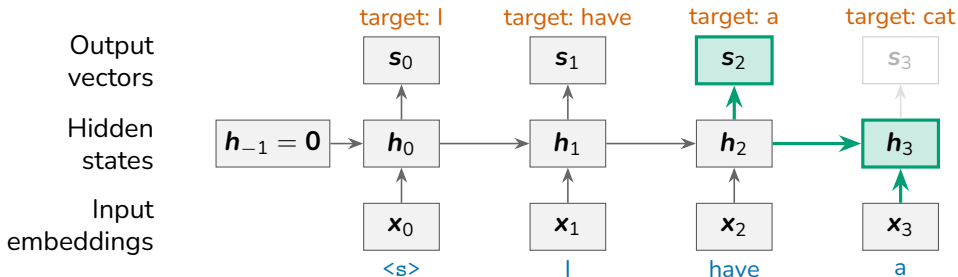
Compute now Gray fill: available Faded: waiting

Stage 4: Compute h_2 and s_1 in parallel.
Their shared dependency h_1 is now ready.

RNN Training: What Can Run in Parallel?

Teacher-forced forward pass: all gold input tokens are known ($T = 4$).

$$h_t = f(W_{hh}h_{t-1} + W_{xh}x_t), \quad s_t = \text{softmax}(Wh_t).$$



Compute now Gray fill: available Faded: waiting

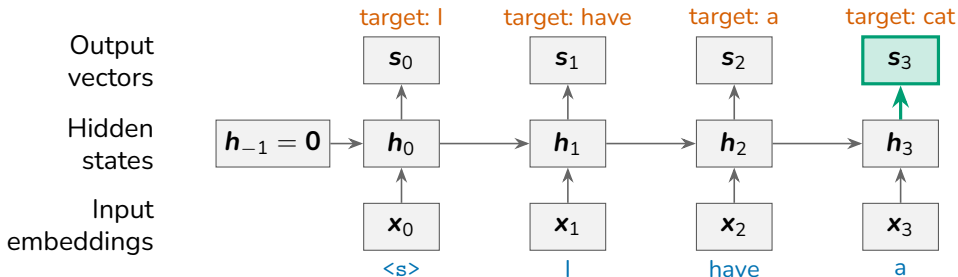
Stage 5: Compute h_3 and s_2 in parallel.

Their shared dependency h_2 is now ready.

RNN Training: What Can Run in Parallel?

Teacher-forced forward pass: all gold input tokens are known ($T = 4$).

$$h_t = f(W_{hh}h_{t-1} + W_{xh}x_t), \quad s_t = \text{softmax}(Wh_t).$$



Compute now Gray fill: available Faded: waiting

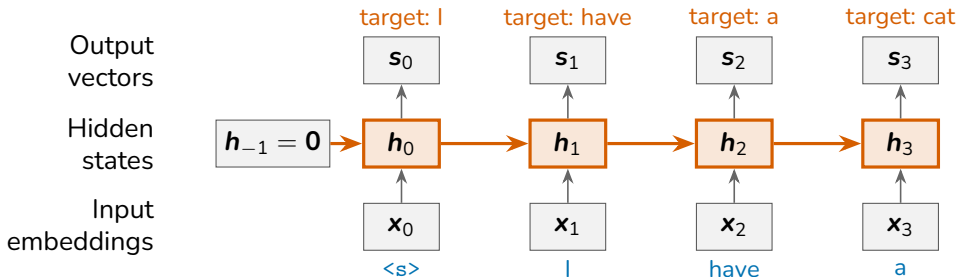
Stage 6: Compute s_3 ; all output distributions are ready.

Output projections overlap with the next hidden-state update.

RNN Training: What Can Run in Parallel?

Teacher-forced forward pass: all gold input tokens are known ($T = 4$).

$$h_t = f(W_{hh}h_{t-1} + W_{xh}x_t), \quad s_t = \text{softmax}(Wh_t).$$



Serial recurrence Gray fill: available Faded: waiting

Serial bottleneck: $h_0 \rightarrow \dots \rightarrow h_{T-1}$ requires T updates.

Forward: $T + 2 = \Theta(T)$; standard BPTT follows the reverse chain: $O(T)$.

Fixed model size, ideal parallelism; count sequential stages, not FLOPs.

From RNNs to Transformers for Language Modeling

A Transformer decoder uses **causal (masked) self-attention** (Lecture 7):

- Attend to current and earlier tokens **within the context window**.
- **No** recurrence → all positions computed in **parallel**.
- Combined with positional encodings to capture token order.

From RNNs to Transformers for Language Modeling

A Transformer decoder uses **causal (masked) self-attention** (Lecture 7):

- Attend to current and earlier tokens **within the context window**.
- **No** recurrence → all positions computed in **parallel**.
- Combined with positional encodings to capture token order.

This is exactly what we need for language modeling!

	RNN LM	Transformer LM
Context access	Via hidden state	Direct within window
Sequential training steps	$O(T)$	$O(1)$
Available history	No fixed window	Finite context window
Examples	RNN, LSTM	GPT-style decoder

Access does not guarantee effective use. At fixed depth, $O(1)$ counts training stages; self-attention still costs $O(T^2)$ per layer.

Transformer Decoder for Language Modeling

Tokens are columns: $E \in \mathbb{R}^{d \times k}$ for input $w_1, \dots, w_k$.

1. Embed + positional encoding:

$$E = (\text{emb}(w_1), \dots, \text{emb}(w_k)) + P$$

2. Causal attention (one head):

$$Q = W_q E, \quad K = W_k E, \quad V = W_v E$$

$$\tilde{E} = V \text{softmax}_{\text{col}} \left(\frac{K^\top Q}{\sqrt{d_k}} + M \right)$$

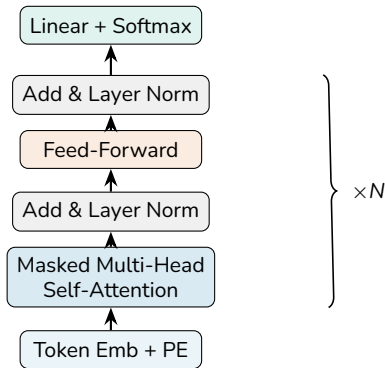
Softmax normalizes each column over keys.

$M_{ij} = 0$ if key $i \leq j$ (query); otherwise

$M_{ij} = -\infty$.

3. Predict next token:

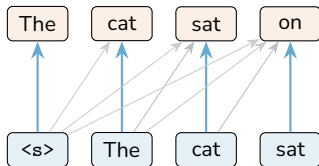
Use position k of the final decoder block's output to predict w_{k+1} .



Causal Masking in Detail

The key to using a Transformer for language modeling: the **causal mask** ensures position i can only attend to positions $\leq i$.

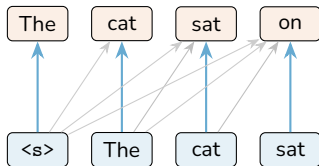
During training, all positions are predicted **simultaneously**:



Causal Masking in Detail

The key to using a Transformer for language modeling: the **causal mask** ensures position i can only attend to positions $\leq i$.

During training, all positions are predicted **simultaneously**:



Massive training speedup over RNNs:

- All T next-token predictions in a sequence are computed in a **single forward pass**.
- RNN: must process T steps **sequentially**.
- Transformer: all T steps in **parallel** (GPU friendly!).

GPT: Generative Pre-trained Transformer

GPT (Radford et al., 2018) = Transformer **decoder-only** for language modeling.

Architecture:

- 12 layers, $d = 768$, $H = 12$ heads.
- Learned positional embeddings.
- Context length: 512 tokens.
- ~ 117 M params, BPE (~ 40 K vocab).

Training:

- Pre-trained on BookCorpus (7K books).
- Objective: next-token prediction.

GPT: Generative Pre-trained Transformer

GPT (Radford et al., 2018) = Transformer **decoder-only** for language modeling.

Architecture:

- 12 layers, $d = 768$, $H = 12$ heads.
- Learned positional embeddings.
- Context length: 512 tokens.
- ~ 117 M params, BPE (~ 40 K vocab).

Training:

- Pre-trained on BookCorpus (7K books).
- Objective: next-token prediction.

Key insight: The same model can be **fine-tuned** for many tasks:

- Text classification
- Natural language inference
- Question answering
- Semantic similarity

This is the **pre-training** $\rightarrow$ **fine-tuning paradigm** that revolutionized NLP. (More in Lectures 9–10.)

The GPT Family

Model	Parameters	Layers	Context	Training Data
GPT (2018)	117M	12	512	BookCorpus
GPT-2 (2019)	1.5B	48	1024	WebText (40GB)
GPT-3 (2020)	175B	96	2048	Web + books + Wikipedia
GPT-4 (2023)	undisclosed	–	8K / 32K	undisclosed

GPT-4 Turbo (November 2023) later introduced a 128K context window.

The GPT Family

Model	Parameters	Layers	Context	Training Data
GPT (2018)	117M	12	512	BookCorpus
GPT-2 (2019)	1.5B	48	1024	WebText (40GB)
GPT-3 (2020)	175B	96	2048	Web + books + Wikipedia
GPT-4 (2023)	undisclosed	–	8K / 32K	undisclosed

GPT-4 Turbo (November 2023) later introduced a 128K context window.

Trend: scale up everything: parameters, data, compute.

- **GPT-2:** Can generate coherent paragraphs of text.
- **GPT-3:** Demonstrated few-shot in-context learning without fine-tuning.
- All are **decoder-only Transformers** with **causal masking**.

Cross-Entropy Loss for Language Modeling

The standard training objective for neural LMs is **cross-entropy loss**:

$$\mathcal{L}(\Theta) = - \sum_i \log P_{\Theta}(\mathbf{x}^{(i)}) = - \sum_i \sum_{t=1}^{T_i} \log P_{\Theta}(x_t^{(i)} \mid x_1^{(i)}, \dots, x_{t-1}^{(i)})$$

where i indexes training sequences and t indexes positions within each sequence.

Cross-Entropy Loss for Language Modeling

The standard training objective for neural LMs is **cross-entropy loss**:

$$\mathcal{L}(\Theta) = - \sum_i \log P_{\Theta}(\mathbf{x}^{(i)}) = - \sum_i \sum_{t=1}^{T_i} \log P_{\Theta}(x_t^{(i)} \mid x_1^{(i)}, \dots, x_{t-1}^{(i)})$$

where i indexes training sequences and t indexes positions within each sequence.

Per-token loss for a single sequence:

$$\mathcal{L}_t = -\log P_{\Theta}(x_t \mid x_{<t}) = -\log \text{softmax}(\mathbf{z}_{t-1})_{x_t} = -\mathbf{z}_{t-1}[x_t] + \log \sum_{v \in V} \exp(\mathbf{z}_{t-1}[v])$$

Here $\mathbf{z}_{t-1} \in \mathbb{R}^{|V|}$ contains logits for x_t , computed from the prefix $x_{<t}$.

Cross-Entropy Loss for Language Modeling

The standard training objective for neural LMs is **cross-entropy loss**:

$$\mathcal{L}(\Theta) = - \sum_i \log P_{\Theta}(\mathbf{x}^{(i)}) = - \sum_i \sum_{t=1}^{T_i} \log P_{\Theta}(x_t^{(i)} \mid x_1^{(i)}, \dots, x_{t-1}^{(i)})$$

where i indexes training sequences and t indexes positions within each sequence.

Per-token loss for a single sequence:

$$\mathcal{L}_t = -\log P_{\Theta}(x_t \mid x_{<t}) = -\log \text{softmax}(\mathbf{z}_{t-1})_{x_t} = -\mathbf{z}_{t-1}[x_t] + \log \sum_{v \in V} \exp(\mathbf{z}_{t-1}[v])$$

Here $\mathbf{z}_{t-1} \in \mathbb{R}^{|V|}$ contains logits for x_t , computed from the prefix $x_{<t}$.

- **Negative log-likelihood:** Maximize the probability of the training data.
- **Cross-entropy:** Minimize $H(\hat{P}, P_{\Theta}) = -\mathbb{E}_{\hat{P}}[\log P_{\Theta}]$.
- **KL divergence:** Minimize $D_{\text{KL}}(\hat{P} \parallel P_{\Theta})$ (up to a constant).

Training: Practical Considerations

1. **Optimizer:** AdamW (Kingma & Ba, 2015; Loshchilov & Hutter, 2019). Default LR: $\sim 10^{-4}$ to 3×10^{-4} , with warmup + cosine decay.
2. **Batching:** Pack multiple sequences into a batch. Pad or concatenate into fixed-length chunks. Typical: 256–2048 sequences.
3. **Held-out data:** Monitor **validation loss** to detect overfitting. Report perplexity on a separate test set.
4. **Gradient clipping:** Clip gradients to max norm (e.g., 1.0) to prevent exploding gradients.
5. **Weight tying:** Share embedding matrix and output projection (Press & Wolf, 2017; Inan et al., 2017): $\mathbf{z}_t = \mathbf{W}_{\text{emb}}^T \mathbf{h}_t$ gives logits for x_{t+1} . Can reduce parameters substantially; the savings depend on the model architecture.

Word-Level vs. Character-Level LMs

What tokens should the model predict?

	Word-level	Character-level
Vocabulary size	~30K–100K	~100–300
Sequence length	Short (words)	Long (characters)
Softmax cost	High ($ V $ large)	Low ($ V $ small)
Unknown words	OOV problem	Known alphabet only
Context needed	Fewer steps	Many more steps
Typical use	Traditional NLP	Research, multilingual

Word-Level vs. Character-Level LMs

What tokens should the model predict?

	Word-level	Character-level
Vocabulary size	~30K–100K	~100–300
Sequence length	Short (words)	Long (characters)
Softmax cost	High ($ V $ large)	Low ($ V $ small)
Unknown words	OOV problem	Known alphabet only
Context needed	Fewer steps	Many more steps
Typical use	Traditional NLP	Research, multilingual

In practice: Subwords balance vocabulary size and sequence length:

- Vocabulary of ~32K–100K subword tokens.
- Plain BPE/WordPiece may emit [UNK] for unseen characters.
- Full byte coverage (byte-level BPE or byte fallback) avoids character OOV.

But character-level LMs are great for **learning and experimentation!**

Building a Character-Level LM from Scratch

Let's walk through the key components of a character-level language model.

Setup:

- **Data:** A text corpus (e.g., Shakespeare, Wikipedia, code).
- **Vocabulary:** All unique characters in the corpus.
- **Model:** A small Transformer decoder (or LSTM).
- **Objective:** Predict the next character given the previous characters.

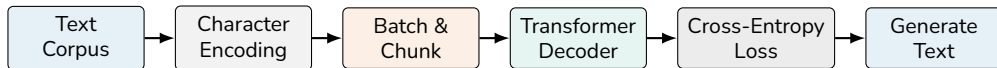
Building a Character-Level LM from Scratch

Let's walk through the key components of a character-level language model.

Setup:

- **Data:** A text corpus (e.g., Shakespeare, Wikipedia, code).
- **Vocabulary:** All unique characters in the corpus.
- **Model:** A small Transformer decoder (or LSTM).
- **Objective:** Predict the next character given the previous characters.

Pipeline:



Step 1: Data Preparation

Character encoding: Map each unique character to an integer index.

```
# Build vocabulary from text
text = open("shakespeare.txt").read()
chars = sorted(set(text)) # unique characters
vocab_size = len(chars)   # e.g., 65

# Character <-> integer mappings
char_to_idx = {ch: i for i, ch in enumerate(chars)}
idx_to_char = {i: ch for i, ch in enumerate(chars)}

# Encode the entire text
data = [char_to_idx[ch] for ch in text]
# e.g., "hello" -> [46, 43, 50, 50, 53]
```

Step 1: Data Preparation

Character encoding: Map each unique character to an integer index.

```
# Build vocabulary from text
text = open("shakespeare.txt").read()
chars = sorted(set(text)) # unique characters
vocab_size = len(chars)   # e.g., 65

# Character <-> integer mappings
char_to_idx = {ch: i for i, ch in enumerate(chars)}
idx_to_char = {i: ch for i, ch in enumerate(chars)}

# Encode the entire text
data = [char_to_idx[ch] for ch in text]
# e.g., "hello" -> [46, 43, 50, 50, 53]
```

Create training examples by chunking into fixed-length sequences:

- Input: `data[i : i+block_size]`
- Target: `data[i+1 : i+block_size+1]`
- The target is the input **shifted by one position**.

Step 2: Model Architecture (Transformer Decoder)

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

class CharLM(nn.Module):
    def __init__(self, vocab_size, d_model, n_heads, n_layers, block_size):
        super().__init__()
        self.tok_emb = nn.Embedding(vocab_size, d_model)
        self.pos_emb = nn.Embedding(block_size, d_model)
        self.blocks = nn.ModuleList([
            TransformerBlock(d_model, n_heads) for _ in range(n_layers)
        ])
        self.ln = nn.LayerNorm(d_model)
        self.head = nn.Linear(d_model, vocab_size)
```

Assume a provided TransformerBlock with causal self-attention, residuals, normalization, and an FFN.

Step 2: Model Forward Pass

Continuing the CharLM class:

```
def forward(self, x):                # x: (B, T) integers
    B, T = x.shape
    tok = self.tok_emb(x)             # (B, T, d_model)
    positions = torch.arange(T, device=x.device)
    pos = self.pos_emb(positions)     # (T, d_model)
    h = tok + pos                     # (B, T, d_model)
    for block in self.blocks:
        h = block(h)                 # causal self-attn + FFN
    h = self.ln(h)
    logits = self.head(h)             # (B, T, vocab_size)
    return logits
```

Step 3: Training Setup

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = CharLM(vocab_size=vocab_size, d_model=128, n_heads=4,
               n_layers=4, block_size=256).to(device)
optimizer = torch.optim.AdamW(model.parameters(), lr=3e-4)
model.train()
```

Assumptions:

- `get_batch` returns shifted input/target tensors of shape (B, T) with dtype `torch.long`.
- Set `max_steps` before training; use separate held-out text for validation.

Step 3: Training Loop

```
for step in range(max_steps):
    xb, yb = get_batch(data, block_size=256, batch_size=64)
    xb, yb = xb.to(device), yb.to(device)
    logits = model(xb)                                # (64, 256, vocab_size)
    loss = F.cross_entropy(
        logits.view(-1, vocab_size),                    # (64*256, vocab_size)
        yb.view(-1)                                     # (64*256,)
    )

    optimizer.zero_grad()
    loss.backward()
    torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
    optimizer.step()

    if step % 500 == 0:
        print(f"Step {step}, Loss: {loss.item():.4f}")
```

Step 4: Text Generation

```
@torch.inference_mode()
def generate(model, start_text, block_size, max_new_tokens=200):
    was_training = model.training
    model.eval()
    try:
        device = next(model.parameters()).device
        idx = torch.tensor([[char_to_idx[c] for c in start_text]],
                           dtype=torch.long, device=device)
        for _ in range(max_new_tokens):
            idx_crop = idx[:, -block_size:]
            logits = model(idx_crop)[:, -1, :]  # (1, vocab_size)
            probs = F.softmax(logits, dim=-1)
            next_idx = torch.multinomial(probs, num_samples=1)
            idx = torch.cat([idx, next_idx], dim=1)
        return ''.join([idx_to_char[i] for i in idx[0].tolist()])
    finally:
        model.train(was_training)
```

Use a nonempty, in-vocabulary prompt; block_size must fit the model.

Autoregressive Generation: Key Observations

- Each step uses only the latest `block_size` tokens (or fewer).
- We sample from the predicted distribution (more on decoding in Lecture 15).
- This is **autoregressive generation**: each new token becomes input for the next.

Autoregressive Generation: Key Observations

- Each step uses only the latest `block_size` tokens (or fewer).
- We sample from the predicted distribution (more on decoding in Lecture 15).
- This is **autoregressive generation**: each new token becomes input for the next.

Inference is not training:

- `model.eval()` disables training-time behavior such as dropout.
- `torch.inference_mode()` disables autograd bookkeeping.
- Restore the previous model mode when returning to training.

Character-Level LM: Published Training Samples

Source experiment: Karpathy (2015), a character-level **LSTM** on *War and Peace*. These are published excerpts, not outputs from the Transformer code above.

Iteration 100: Early character patterns.

tyntd-iafhatawiaoihrdemot lytdws e ,tfti, astai f ogoh eoase rrranbyne 'nhthnee e

Character-Level LM: Published Training Samples

Source experiment: Karpathy (2015), a character-level **LSTM** on *War and Peace*. These are published excerpts, not outputs from the Transformer code above.

Iteration 100: Early character patterns.

tyntd-iafhatawiaoihrdemot lytdws e ,tfti, astai f ogoh eoase rrranbyne 'nhthnee e

Iteration 500: Some common words.

we counter. He stutn co des. His stanted out one ofler that concossions and was
to gearang reay Jotrets and with fire colt otf paitt thin wall. Which das stimn

Character-Level LM: Published Training Samples

Source experiment: Karpathy (2015), a character-level **LSTM** on *War and Peace*. These are published excerpts, not outputs from the Transformer code above.

Iteration 100: Early character patterns.

tyntd-iafhatawiaoihrdemot lytdws e ,tfti, astai f ogoh eoase rrranbyne 'nhthnee e

Iteration 500: Some common words.

we counter. He stutn co des. His stanted out one ofler that concossions and was to gearang reay Jotrets and with fire colt of paitt thin wall. Which das stimn

Iteration 2,000: Names and longer words, but still incoherent.

"Why do what that day," replied Natasha, and wishing to himself the fact the princess, Princess Mary was easier, fed in had oftended him.
Pierre aking his soul came to the packs and drove up his father-in-law women.

Source: [Karpathy \(2015\)](#), *The Unreasonable Effectiveness of Recurrent Neural Networks*.

What Does the Character-Level LM Learn?

Progression of what the model captures during training:

1. **Character frequencies:** Common letters (e, t, a) appear more often.
2. **Character bigrams:** “th”, “he”, “in” are frequent pairs.
3. **Word boundaries:** Spaces appear at the right places.
4. **Common words:** “the”, “and”, “to” are generated correctly.
5. **Word morphology:** Suffixes like “-ing”, “-tion”, “-ly”.
6. **Local syntax:** Adjective-noun, verb-object patterns.
7. **Long-range structure:** Dialogue formatting, paragraph breaks.

What Does the Character-Level LM Learn?

Progression of what the model captures during training:

1. **Character frequencies:** Common letters (e, t, a) appear more often.
2. **Character bigrams:** “th”, “he”, “in” are frequent pairs.
3. **Word boundaries:** Spaces appear at the right places.
4. **Common words:** “the”, “and”, “to” are generated correctly.
5. **Word morphology:** Suffixes like “-ing”, “-tion”, “-ly”.
6. **Local syntax:** Adjective-noun, verb-object patterns.
7. **Long-range structure:** Dialogue formatting, paragraph breaks.

Takeaway: A neural LM learns a substantial amount of linguistic structure from **next-character prediction alone**, with no explicit grammar rules and no linguistic supervision.

This is the foundation of the **pre-training paradigm**.

Comparison: N -Gram vs. RNN vs. Transformer LMs

	N-Gram	RNN/LSTM	Transformer
Context	Fixed ($N-1$)	No fixed window	Bounded window
Parameters	$O(V ^N)$	$O(d^2 + d V)$	$O(d^2 + d V)$
Training	Count + smooth	Sequential (slow)	Parallel (fast)
Generalization	Smoothing, backoff	Via embeddings	Via embeddings + attention
Long-range	Very limited	Moderate (LSTM)	Strong
Scalability	Poor	Moderate	Excellent

Comparison: N -Gram vs. RNN vs. Transformer LMs

	N -Gram	RNN/LSTM	Transformer
Context	Fixed ($N-1$)	No fixed window	Bounded window
Parameters	$O(V ^N)$	$O(d^2 + d V)$	$O(d^2 + d V)$
Training	Count + smooth	Sequential (slow)	Parallel (fast)
Generalization	Smoothing, backoff	Via embeddings	Via embeddings + attention
Long-range	Very limited	Moderate (LSTM)	Strong
Scalability	Poor	Moderate	Excellent

Today's dominant approach: Transformer decoder-only LMs (GPT family).
But understanding N -grams and RNNs provides the **conceptual foundation!**

The Softmax Bottleneck

Setting: A single linear-softmax head, **without output bias**:

$$P(x_t = v \mid x_{<t}) = \frac{\exp(\mathbf{w}_v^\top \mathbf{h}_{t-1})}{\sum_{v' \in V} \exp(\mathbf{w}_{v'}^\top \mathbf{h}_{t-1})}$$

Here $\mathbf{h}_{t-1} \in \mathbb{R}^d$ encodes $x_{<t}$, and $\mathbf{w}_v$ is the output embedding for word v .

The Softmax Bottleneck

Setting: A single linear-softmax head, **without output bias**:

$$P(x_t = v \mid x_{<t}) = \frac{\exp(\mathbf{w}_v^\top \mathbf{h}_{t-1})}{\sum_{v' \in V} \exp(\mathbf{w}_{v'}^\top \mathbf{h}_{t-1})}$$

Here $\mathbf{h}_{t-1} \in \mathbb{R}^d$ encodes $x_{<t}$, and $\mathbf{w}_v$ is the output embedding for word v .

Rank limit: Across contexts c and tokens v , $\text{rank}(\log P) \leq d + 1$.

- Softmax normalization adds rank 1; an output bias raises the bound to $d + 2$.
- A higher-rank target log-probability matrix creates a **softmax bottleneck** (Yang et al., 2018).

The Softmax Bottleneck

Setting: A single linear-softmax head, **without output bias**:

$$P(x_t = v \mid x_{<t}) = \frac{\exp(\mathbf{w}_v^\top \mathbf{h}_{t-1})}{\sum_{v' \in V} \exp(\mathbf{w}_{v'}^\top \mathbf{h}_{t-1})}$$

Here $\mathbf{h}_{t-1} \in \mathbb{R}^d$ encodes $x_{<t}$, and $\mathbf{w}_v$ is the output embedding for word v .

Rank limit: Across contexts c and tokens v , $\text{rank}(\log P) \leq d + 1$.

- Softmax normalization adds rank 1; an output bias raises the bound to $d + 2$.
- A higher-rank target log-probability matrix creates a **softmax bottleneck** (Yang et al., 2018).

Options and scope:

- Mixture of softmaxes can represent higher-rank log-probabilities.
- Increasing d raises capacity, but also increases parameter count.
- If $d \geq |V|$, this rank bound is not restrictive (e.g., a small character vocabulary).

Summary

- Language modeling is a **classification** problem: predict $P(x_t \mid x_{<t})$ over vocabulary V .
- **Neural N -gram LMs** (Bengio et al., 2003): replace count tables with neural nets; generalize via embeddings, but use fixed context.
- **RNN LMs**: variable-length context sequentially; limited by vanishing gradients and sequential computation.
- **Transformer decoder LMs** (GPT-style): causal self-attention within a context window, with positions computed in **parallel during training**.
- **Training**: cross-entropy loss = negative log-likelihood. Teacher forcing feeds ground-truth tokens during training.
- **Character-level LMs**: learn linguistic structure from next-character prediction alone.
- Neural LMs encode **knowledge** about language, enabling the pre-training paradigm.

Readings

Required:

- Bengio et al. (2003). [A Neural Probabilistic Language Model](#). *JMLR*.
- Mikolov et al. (2010). [Recurrent neural network based language model](#). *Interspeech*.
- Radford et al. (2018). [Improving Language Understanding by Generative Pre-Training](#). OpenAI technical report.

Optional / further reading:

- Karpathy et al. (2016). [Visualizing and Understanding Recurrent Networks](#). *ICLR Workshop*.

Next Lecture

Masked LMs and Pretraining Objectives