Generative Models for NLP

Language Models

Nadi Tomeh

24/1/25

Outline

- Introduction to Language Models
- Vocabulary and Tokenization
- Applications
- Parametrization and Estimation
- n-Gram Language Models
- Addressing Data Sparsity in n-Gram Models
- Evaluation Metrics for Language Models
- Toy Example
- Generation Strategies for Language Models
- Feed-Forward Neural Language Models
- Training
- Recurrent Neural Networks (RNNs)
- LSTMs and GRUs
- What's Next?

Outline

- Introduction to Language Models
- Vocabulary and Tokenization
- Applications
- Parametrization and Estimation
- n-Gram Language Models
- Addressing Data Sparsity in n-Gram Models
- Evaluation Metrics for Language Models
- Toy Example
- Generation Strategies for Language Models
- Feed-Forward Neural Language Models
- Training
- Recurrent Neural Networks (RNNs)
- LSTMs and GRUs
- What's Next?

What is a Language Model?

Definition (Language Model)

A **language model** is a function that defines a joint probability distribution $p(w_1, w_2, \ldots, w_n)$, over an ordered sequence of tokens $\boldsymbol{w} = (w_1, w_2, \ldots, w_n)$. Each $w_k \in \mathcal{V}$, a finite set of tokens called the **vocabulary**. A valid language model must satisfy the constraint:

$$\sum_{\boldsymbol{w} \in \mathcal{W}} p(\boldsymbol{w}) = 1,$$

where $\mathcal{W} \subseteq \mathcal{V}^*$ is the (possibly infinite) set of all token sequences (recall the definition of a **formal language**).

Chain Rule of Probability

Using the chain rule, we can factorize this joint probability as:

$$p(w_1, w_2, \ldots, w_n) = \prod_{k=1}^{n} p\big(w_k \mid w_1, \ldots, w_{k-1}\big).$$

Each term $p(w_k \mid w_1, \ldots, w_{k-1})$ is a conditional probability of the current token given all previous tokens.

- **Interpretation:**
  - The model measures how "natural" or likely a sequence is.
  - Each factor $p(w_k \mid w_1, \ldots, w_{k-1})$ represents how likely the next token w_k is given the context $(w_1, \ldots, w_{k-1})$.

Generative vs. Discriminative Models: Basic Concepts

Generative Models

A **generative model** aims to learn the joint probability $p(\boldsymbol{x}, \boldsymbol{y})$, where $\boldsymbol{x}$ represents the observed data (e.g., a sequence of tokens) and $\boldsymbol{y}$ represents labels, latent variables, or outputs (can be *structured*).

- A **language model** is generative because it learns $p(w_1, \ldots, w_n)$, i.e., the probability of entire sequences.
- Once a generative model is learned, you can derive $p(\boldsymbol{x} \mid \boldsymbol{y}) = \frac{p(\boldsymbol{x}, \boldsymbol{y})}{p(\boldsymbol{y})}$, and $p(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{p(\boldsymbol{x}, \boldsymbol{y})}{p(\boldsymbol{x})}$.
- The marginal probability of $\boldsymbol{x}$, necessary for computing $p(\boldsymbol{y} \mid \boldsymbol{x})$, is obtained by summing (or integrating) over all possible values of $\boldsymbol{y}$:

$$p(\boldsymbol{x}) = \sum_{\boldsymbol{y}} p(\boldsymbol{x}, \boldsymbol{y}) \quad \text{(discrete case)}$$

Discriminative Models

A **discriminative model** directly learns the conditional probability $p(\boldsymbol{y} \mid \boldsymbol{x})$, without modeling the joint distribution $p(\boldsymbol{x}, \boldsymbol{y})$ or the data likelihood $p(\boldsymbol{x})$.

- A discriminative **text classifier** takes an input sequence $\boldsymbol{x} = (w_1, w_2, \ldots, w_n)$ and predict a class label y (e.g., *positive* or *negative* sentiment) and directly models $p(y \mid \boldsymbol{x})$.

Outline

- Introduction to Language Models
- Vocabulary and Tokenization
- Applications
- Parametrization and Estimation
- n-Gram Language Models
- Addressing Data Sparsity in n-Gram Models
- Evaluation Metrics for Language Models
- Toy Example
- Generation Strategies for Language Models
- Feed-Forward Neural Language Models
- Training
- Recurrent Neural Networks (RNNs)
- LSTMs and GRUs
- What's Next?

Tokenization: Definitions and Approaches

What is Tokenization?

- **Tokenization** is the process of splitting text into smaller units, called *tokens*, which serve as the atomic input to language models.
- A *token* can be:
  - A full word
  - A subword or morpheme
  - A single character (especially in low-resource or highly morphologically rich languages)

Key Considerations

- **Vocabulary Size**:
  - Large vocabulary $\Longrightarrow$ fewer unknowns or Out-Of-Vocaulary (OOV) tokens, but increases parameter count.
  - Small vocabulary $\Longrightarrow$ risk of high OOV rates, or reliance on subword tokens.
- **Handling Unknown Words**:
  - Use a special `<unk>` token, or fallback to character-level tokens.
- **Granularity**:
  - **Word-Level**: Simplest, but OOV issues can be severe.
  - **Subword-Level** (BPE, WordPiece, SentencePiece): Balances coverage and vocabulary size.
  - **Character-Level**: No OOVs, but leads to longer sequences and sometimes slower training.

Byte-Pair Encoding (BPE): Algorithm and Vocabulary Evolution I

Core Idea of BPE

- **Byte-Pair Encoding (BPE)** is a data compression technique adapted for tokenization.
- Iteratively merges the most frequent pair of symbols (characters or subwords) into a single token.
- Produces a subword-based vocabulary that reduces out-of-vocabulary issues while controlling vocabulary size.

Algorithm (High-Level Steps)

1. **Initialize Vocabulary** $\mathcal{V}_0$:
   - Each unique character is its own token (e.g., `l`, `o`, `v`, `e`, `c`, `a`, `t`, `s`, plus any spaces or special markers).
2. **Count Pair Frequencies**:
   - Scan the training text for adjacent token pairs (e.g., `l`+`o`, `o`+`v`, etc.).
3. **Merge Most Frequent Pair**:
   - Combine that pair into a single token (e.g., `o_v`).
   - Update your text (i.e., each occurrence of `o v` becomes the new merged token).
   - Add this newly merged token to your vocabulary $\mathcal{V}_1$.
4. **Repeat** for *n* merges or until desired vocabulary size is reached.

Byte-Pair Encoding (BPE): Algorithm and Vocabulary Evolution II

Vocabulary & Text Evolution (Simplified Example)

Training Text (repeated twice): `i love love cats`
Initial vocabulary $\mathcal{V}_0$ (characters only):

$$\mathcal{V}_0 = \{\texttt{i}, \texttt{l}, \texttt{o}, \texttt{v}, \texttt{e}, \texttt{c}, \texttt{a}, \texttt{t}, \texttt{s}\}$$

Step 1: Most frequent adjacent pair is $\texttt{l} + \texttt{o}$.

$$\text{Merge } (\texttt{l}, \texttt{o}) \to \texttt{l\_o}. \quad \mathcal{V}_1 = \mathcal{V}_0 \cup \{\texttt{l\_o}\}.$$

Text now becomes: `i l_o v e l_o v e c a t s`
Step 2: Next frequent pair might be $\texttt{l\_o} + \texttt{v}$.

$$\text{Merge } (\texttt{l\_o}, \texttt{v}) \to \texttt{l\_o\_v}. \quad \mathcal{V}_2 = \mathcal{V}_1 \cup \{\texttt{l\_o\_v}\}.$$

Text now becomes: `i l_o_v e l_o_v e c a t s`
Step 3: Merge $\texttt{l\_o\_v} + \texttt{e}$ to form `l_o_v_e`, etc.
Over several merges, common subwords like `cat` or `love` end up as single tokens.
Final Vocabulary $\mathcal{V}_n$ (after n merges):

$$\mathcal{V}_n = \{\texttt{i}, \texttt{l\_o\_v\_e}, \texttt{c\_a\_t\_s}, \dots\}$$

Outline

- Introduction to Language Models
- Vocabulary and Tokenization
- Applications
- Parametrization and Estimation
- n-Gram Language Models
- Addressing Data Sparsity in n-Gram Models
- Evaluation Metrics for Language Models
- Toy Example
- Generation Strategies for Language Models
- Feed-Forward Neural Language Models
- Training
- Recurrent Neural Networks (RNNs)
- LSTMs and GRUs
- What's Next?

Applications

Evaluating Text Likelihood

- Given a sequence $\boldsymbol{w} = (w_1, w_2, \ldots, w_n)$, compute its probability: $p(\boldsymbol{w}) = \prod_{k=1}^{n} p(w_k \mid w_1, \ldots, w_{k-1})$.
- **Use Cases:**
  - **Speech Recognition & Machine Translation:** Re-rank candidate outputs based on their probabilities.
  - **Error Correction:** Identify unlikely sequences as potential errors.
  - **Quality Assessment:** Evaluate fluency and coherence of text in various applications.

Text Generation

- **Next-Token Prediction:** Iteratively extend the sequence $(w_1, w_2, \ldots, w_{k-1}) \to (w_1, w_2, \ldots, w_{k-1}, w_k)$ until a stopping criterion is met.
- Used for dialogue systems, creative content creation, and auto-completion.

Generalization

Used for modeling any kind of sequences: code, time series, etc.

Outline

- Introduction to Language Models
- Vocabulary and Tokenization
- Applications
- Parametrization and Estimation
- n-Gram Language Models
- Addressing Data Sparsity in n-Gram Models
- Evaluation Metrics for Language Models
- Toy Example
- Generation Strategies for Language Models
- Feed-Forward Neural Language Models
- Training
- Recurrent Neural Networks (RNNs)
- LSTMs and GRUs
- What's Next?

Parameterization of Language Models

Parameterized Probability

Rather than directly specifying $p(w_k \mid w_1, \ldots, w_{k-1})$, language models introduce a set of parameters $\boldsymbol{\theta}$ to define:

$$p_{\boldsymbol{\theta}}(w_k \mid w_1, \ldots, w_{k-1}).$$

- The joint probability over a sequence is then parameterized as:

$$p_{\boldsymbol{\theta}}(w_1, \ldots, w_n) = \prod_{k=1}^{n} p_{\boldsymbol{\theta}}(w_k \mid w_1, \ldots, w_{k-1}).$$

- Parameterization allows using various model architectures:
  - **n-gram** models: Use fixed-context frequency counts with parameters derived from observed counts.
  - **Neural networks**: Use parameters $\boldsymbol{\theta}$ to encode complex dependencies (e.g., in RNNs, Transformers).
- The goal: Find $\boldsymbol{\theta}$ that best captures the underlying language patterns.

Estimation of Parameters from Data

Statistical Estimation

In **probability theory**, the distribution $p(\boldsymbol{x})$ is assumed known, and we derive properties (e.g., expectations, variances) from that distribution. In **statistics**, the distribution is *unknown*, and we **estimate** its parameters or form based on observed data $\mathcal{D}$.

Maximum Likelihood Estimation (MLE)

Given a training corpus $\mathcal{D} = \{\boldsymbol{w}^{(i)}\}_{i=1}^{N}$, estimate parameters by maximizing the likelihood:

$$\hat{\boldsymbol{\theta}}_{\mathrm{MLE}} = \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{N} p_{\boldsymbol{\theta}}(\boldsymbol{w}^{(i)}).$$

Equivalently, maximize the log-likelihood:

$$\hat{\boldsymbol{\theta}}_{\mathrm{MLE}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{N} \log p_{\boldsymbol{\theta}}(\boldsymbol{w}^{(i)}) = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{N} \sum_{k=1}^{n^{(i)}} \log p_{\boldsymbol{\theta}}(w_k^{(i)} \mid w_1^{(i)}, \ldots, w_{k-1}^{(i)}).$$

where $n^{(i)}$ is the length of sequence i. Optimization is typically performed using gradient-based methods (e.g., stochastic gradient descent) and backpropagation for neural models.

Outline

- Introduction to Language Models
- Vocabulary and Tokenization
- Applications
- Parametrization and Estimation
- n-Gram Language Models
- Addressing Data Sparsity in n-Gram Models
- Evaluation Metrics for Language Models
- Toy Example
- Generation Strategies for Language Models
- Feed-Forward Neural Language Models
- Training
- Recurrent Neural Networks (RNNs)
- LSTMs and GRUs
- What's Next?

Markov Assumption in Language Modeling

Full Conditional Probability

Recall the chain rule for a sequence $\boldsymbol{w} = (w_1, w_2, \ldots, w_n)$:

$$p(\boldsymbol{w}) = \prod_{k=1}^{n} p(w_k \mid w_1, \ldots, w_{k-1}).$$

Markov Assumption

The **Markov assumption** simplifies this by assuming that the probability of the next token depends only on a finite history of previous tokens:

$$p(w_k \mid w_1, \ldots, w_{k-1}) \approx p(w_k \mid w_{k-n+1}, \ldots, w_{k-1}),$$

where n is the **order** of the Markov model.

- This *finite memory* assumption reduces computational complexity and makes estimation from data feasible.
- It introduces conditional independence: w_k is independent of tokens beyond the last $n-1$ given the recent history.
- Leads directly to n-**gram models**, where probabilities are estimated based on limited context of length $n-1$.

Parametrization of n-Gram Models Using Categorical Distributions

Parametrization

- For each possible $(n-1)$-gram context $\boldsymbol{c} = (w_{k-n+1}, \ldots, w_{k-1})$, define a **categorical distribution**:

$$p_{\boldsymbol{\theta}}(w_k \mid \boldsymbol{c}) = \theta_{\boldsymbol{c},w_k}, \quad \text{where} \quad \sum_{w_k \in \mathcal{V}} \theta_{\boldsymbol{c},w_k} = 1.$$

- $\theta_{\boldsymbol{c},w_k}$ represents the probability of observing w_k given the history $\boldsymbol{c}$.
- For each context $\boldsymbol{c}$, the model stores a parameter vector:

$$\boldsymbol{\theta}_{\boldsymbol{c}} = \left(\theta_{\boldsymbol{c},w_1}, \theta_{\boldsymbol{c},w_2}, \ldots, \theta_{\boldsymbol{c},w_{|\mathcal{V}|}}\right),$$

  which lies in the $|\mathcal{V}|$-dimensional **probability simplex**.

Parameters of n-Gram Models

Number of Parameters

- Total parameters:

$$|\mathcal{V}|^{n-1} \cdot (|\mathcal{V}| - 1),$$

  where:
  - $|\mathcal{V}|^{n-1}$: Number of possible $(n-1)$-token contexts.
  - $|\mathcal{V}| - 1$: Free parameters per context (due to the simplex constraint).

Parameter Estimation

Parameters are estimated using **maximum likelihood** in *closed form*:

$$\hat{\theta}_{\boldsymbol{c},w_k} = \frac{\text{count}(\boldsymbol{c}, w_k)}{\text{count}(\boldsymbol{c})},$$

where:

- $\text{count}(\boldsymbol{c}, w_k)$: Number of times $(\boldsymbol{c}, w_k)$ appears in the training corpus $\mathcal{D}$.
- $\text{count}(\boldsymbol{c}) = \sum_{w_k \in \mathcal{V}} \text{count}(\boldsymbol{c}, w_k)$: Total occurrences of $\boldsymbol{c}$.

Deriving the MLE for n-Gram Language Models I

Log-Likelihood for n-Gram Models

Given a training corpus $\mathcal{D} = \{\boldsymbol{w}^{(i)}\}_{i=1}^{M}$, where each sequence $\boldsymbol{w}^{(i)} = (w_1^{(i)}, \ldots, w_{n^{(i)}}^{(i)})$, the log-likelihood of the parameters $\boldsymbol{\theta}$ is:

$$\mathcal{L}(\boldsymbol{\theta}; \mathcal{D}) = \sum_{i=1}^{M} \log p_{\boldsymbol{\theta}}(\boldsymbol{w}^{(i)}) = \sum_{i=1}^{M} \sum_{k=1}^{n^{(i)}} \log p_{\boldsymbol{\theta}}(w_k^{(i)} \mid \boldsymbol{c}_k^{(i)}),$$

where $\boldsymbol{c}_k^{(i)} = (w_{k-n+1}^{(i)}, \ldots, w_{k-1}^{(i)})$ is the $(n-1)$-token context.

Maximizing the Log-Likelihood

Substitute $p_{\boldsymbol{\theta}}(w_k \mid \boldsymbol{c}) = \theta_{\boldsymbol{c}, w_k}$:

$$\mathcal{L}(\boldsymbol{\theta}; \mathcal{D}) = \sum_{\boldsymbol{c} \in \mathcal{V}^{N-1}} \sum_{w \in \mathcal{V}} \text{count}(\boldsymbol{c}, w) \log \theta_{\boldsymbol{c}, w}.$$

Subject to the constraint that for each context $\boldsymbol{c}$,

$$\sum_{w \in \mathcal{V}} \theta_{\boldsymbol{c}, w} = 1.$$

Deriving the MLE for *n*-Gram Language Models II

Solving with Lagrange Multipliers

Define the Lagrangian:

$$\mathcal{L}'(\boldsymbol{\theta}, \boldsymbol{\lambda}) = \sum_{\boldsymbol{c} \in \mathcal{V}^{N-1}} \sum_{w \in \mathcal{V}} \mathsf{count}(\boldsymbol{c}, w) \log \theta_{\boldsymbol{c},w} + \sum_{\boldsymbol{c} \in \mathcal{V}^{N-1}} \lambda_{\boldsymbol{c}} \left(1 - \sum_{w \in \mathcal{V}} \theta_{\boldsymbol{c},w}\right).$$

Taking the derivative w.r.t. $\theta_{\boldsymbol{c},w}$ and setting to zero:

$$\frac{\partial \mathcal{L}'}{\partial \theta_{\boldsymbol{c},w}} = \frac{\mathsf{count}(\boldsymbol{c}, w)}{\theta_{\boldsymbol{c},w}} - \lambda_{\boldsymbol{c}} = 0 \quad \Longrightarrow \quad \theta_{\boldsymbol{c},w} = \frac{\mathsf{count}(\boldsymbol{c}, w)}{\lambda_{\boldsymbol{c}}}.$$

Enforce the normalization constraint:

$$\sum_{w \in \mathcal{V}} \theta_{\boldsymbol{c},w} = 1 \quad \Longrightarrow \quad \lambda_{\boldsymbol{c}} = \mathsf{count}(\boldsymbol{c}).$$

Subtitute $\lambda_{\boldsymbol{c}}$ back to get the MLE estimate for $\theta_{\boldsymbol{c},w}$:

$$\hat{\theta}_{\boldsymbol{c},w} = \frac{\mathsf{count}(\boldsymbol{c}, w)}{\mathsf{count}(\boldsymbol{c})}.$$

Order of *n*-Gram Models

Definition of **Order**

- An ***n*-gram model** uses the last $n-1$ tokens to predict the next token:

$$p(w_k \mid w_1, \ldots, w_{k-1}) \;\approx\; p(w_k \mid w_{k-N+1}, \ldots, w_{k-1}).$$

- The integer n is called the **order** of the model. For example:
  - $n = 1$: **Unigram** model (context-free).
  - $n = 2$: **Bigram** model (1-token context).
  - $n = 3$: **Trigram** model (2-token context).

Impact of Model Order

- **Higher order** (n large):
  - Captures longer-range dependencies in text.
  - Increases the number of parameters dramatically, leading to potential *data sparsity*.
- **Lower order** (n small):
  - Fewer parameters, simpler to estimate from limited data.
  - May miss important context (lacks expressive power).

Outline

- Introduction to Language Models
- Vocabulary and Tokenization
- Applications
- Parametrization and Estimation
- n-Gram Language Models
- Addressing Data Sparsity in n-Gram Models
- Evaluation Metrics for Language Models
- Toy Example
- Generation Strategies for Language Models
- Feed-Forward Neural Language Models
- Training
- Recurrent Neural Networks (RNNs)
- LSTMs and GRUs
- What's Next?

Data Sparsity in *n*-Gram Models

Where Does Sparsity Come From?

- The vocabulary $\mathcal{V}$ can be large (tens or hundreds of thousands of tokens).
- As N grows, so does the number of possible $(n-1)$-token contexts: $|\mathcal{V}|^{n-1}$.
- Many valid $(n-1)$-gram contexts may appear *zero or very few times* in the training data $\mathcal{D}$.

Consequences of Data Sparsity

- **Zero Counts**: Some $(n-1)$-gram contexts are never observed, leading to

$$\hat{\theta}_{\boldsymbol{c},w} = \frac{\text{count}(\boldsymbol{c}, w)}{\text{count}(\boldsymbol{c})} = 0 \quad \text{(no observed tokens)}.$$

- **Poor Generalization**: A context not seen in training has probability 0, causing the entire probability of any sentence containing such a context to also become 0.
- **Need for Smoothing**: Techniques like Laplace, Kneser–Ney, or Good–Turing adjust counts to avoid assigning zero probability.
- **Memory and Computation**: Large $|\mathcal{V}|^{n-1}$ means storing and computing vast tables for $\theta_{\boldsymbol{c},w}$.

Laplace Smoothing (Add-One Smoothing)

Motivation

- Pure MLE often assigns *zero* probability to unseen $(n-1)$-gram contexts.
- **Smoothing** redistributes probability mass to ensure every event has a nonzero probability.

Formula for Add-One Smoothing

- Original MLE estimate:

$$\hat{\theta}_{\boldsymbol{c},w} = \frac{\text{count}(\boldsymbol{c}, w)}{\text{count}(\boldsymbol{c})}.$$

- Add-One smoothing (Laplace):

$$\hat{\theta}^{\text{Laplace}}_{\boldsymbol{c},w} = \frac{\text{count}(\boldsymbol{c}, w) + 1}{\text{count}(\boldsymbol{c}) + |\mathcal{V}|}.$$

- Each $(\boldsymbol{c}, w)$ is treated as if it appeared at least once.
- Denominator adds $|\mathcal{V}|$ to account for adding 1 for each possible token w.
- Eliminates zero probabilities.

Problems with Laplace Smoothing

Uniform Distribution for Unobserved Contexts and Over-Smoothing Rare Contexts

- For unobserved $(n-1)$-gram contexts $\boldsymbol{c}$ ($\text{count}(\boldsymbol{c}) = 0$), Laplace smoothing assigns:

$$\hat{\theta}_{\boldsymbol{c},w}^{\text{Laplace}} = \frac{1}{|\mathcal{V}|},$$

  resulting in a uniform distribution across the vocabulary.
- This fails to capture any linguistic structure or dependencies in the data.
- For rare contexts (e.g., $\text{count}(\boldsymbol{c}) = 2$), smoothing redistributes too much probability to unseen tokens.

High Sensitivity to Vocabulary Size and Model's Order

- The denominator $(\text{count}(\boldsymbol{c}) + |\mathcal{V}|)$ grows with $|\mathcal{V}|$, making the smoothed probabilities heavily dependent on the vocabulary size.
- As n increases, the number of possible $(n-1)$-gram contexts grows exponentially: $|\mathcal{V}|^{n-1}$.
- Even large corpora cannot cover this space, leading to unrealistic distributions for unseen or rare contexts.

Handling Unknown Words

Out-of-Vocabulary (OOV) Words

- Even large training corpora cannot cover every word form or proper noun.
- Any word ω *not observed* in training is **out-of-vocabulary** (OOV).
- **Issue:** If OOV word appears in testing (or real-world usage), the *n*-gram model has zero probability for any sequence containing ω.

<UNK> Token

- A common approach is to **preemptively** replace low-frequency words in the training data with a special symbol `<UNK>`.
- This maps all rare or unobserved words to a single `<UNK>` token, effectively reducing vocabulary size.
- `<UNK>` is then treated like any other token in the *n*-gram model, allowing the model to handle previously unseen words during inference.

- **Threshold Method**:
  - If count$(\omega) < \tau$, replace ω with `<UNK>` in training.
  - Choose τ (e.g., 1, 2, 5) based on data scale and performance.
- **Vocabulary Pruning**:
  - Keep only the top $\alpha\%$ most frequent words and map the rest to `<UNK>`.

Linear Interpolation of n-gram Models

Motivation

- Pure **MLE** or simple smoothing (e.g., Laplace) can still suffer from zero probabilities for higher-order n-grams with low counts.
- **Interpolation** combines multiple context lengths (orders) rather than "backing off" only when higher-order counts are insufficient.
- Offers a continuous blend of *all* available contexts, reducing the abruptness of pure backoff.

General Interpolation Formula (Trigram Example)

Suppose you want to interpolate among unigram ($N = 1$), bigram ($N = 2$), and trigram ($N = 3$) models:

$$p_{\mathrm{interp}}(w_k \mid w_{k-2}, w_{k-1}) = \lambda_3\, p_{\mathrm{MLE}}(w_k \mid w_{k-2}, w_{k-1}) + \lambda_2\, p_{\mathrm{MLE}}(w_k \mid w_{k-1}) + \lambda_1\, p_{\mathrm{MLE}}(w_k),$$

where:

- $\sum_{i=1}^{3} \lambda_i = 1$.
- $p_{\mathrm{MLE}}(\cdot)$ are the standard MLE estimates for each context size, can use smoothing.
- $\{\lambda_i\}$ can be tuned on a held-out *validation set* (e.g., maximize likelihood or minimize perplexity).
- Often, λ_i depend on context counts so that higher-order models get more weight when data is sufficient.

Special Tokens: <s> and </s>

Purpose of Special Tokens

- <s>: Marks the **start of a sentence**.
- </s>: Marks the **end of a sentence**.

Motivation

- **Defining Sentence Boundaries:** <s> and </s> provide explicit delimiters for sequences.
- **Context Padding for *n*-Gram Models:**
  - For *n*-gram models, prepend $(n-1)$ '<s>' tokens to the beginning of a sentence.
  - Example (Bigram Model): $p(w_1, w_2, w_3) \approx p(w_1 \mid \texttt{<s>})p(w_2 \mid w_1)p(w_3 \mid w_2)p(\texttt{</s>} \mid w_3)$.
- **Termination in Generation:** Models recognize </s> as the endpoint for generated sequences, preventing infinite loops.

Outline

- Introduction to Language Models
- Vocabulary and Tokenization
- Applications
- Parametrization and Estimation
- n-Gram Language Models
- Addressing Data Sparsity in n-Gram Models
- **Evaluation Metrics for Language Models**
- Toy Example
- Generation Strategies for Language Models
- Feed-Forward Neural Language Models
- Training
- Recurrent Neural Networks (RNNs)
- LSTMs and GRUs
- What's Next?

Entropy of a Discrete Distribution I

Intuition

- **Entropy** measures the average uncertainty or surprise of a random variable.
- In language modeling, it reflects how predictable or unpredictable the tokens are under a distribution.

Formal Definition

Let X be a discrete random variable with a probability mass function $p(x)$ over some set $\mathcal{V}$. The **entropy** $H(X)$ is defined as:

$$H(X) = -\sum_{x \in \mathcal{V}} p(x) \log p(x).$$

- The base of the logarithm determines the units:
  - Base 2: Entropy is measured in **bits**.
  - Base e: Entropy is measured in **nats**.
- **Bits:** The number of binary (yes/no) questions needed, on average, to identify an outcome of X.
- High entropy $\Rightarrow$ high unpredictability; low entropy $\Rightarrow$ more predictability.

Entropy of a Discrete Distribution II

Example

- Suppose $\mathcal{V} = \{\texttt{cat}, \texttt{dog}, \texttt{mouse}\}$ with $p(\texttt{cat}) = 0.5$, $p(\texttt{dog}) = 0.3$, $p(\texttt{mouse}) = 0.2$. Then

$$H(X) = -\left[0.5 \log 0.5 \ + \ 0.3 \log 0.3 \ + \ 0.2 \log 0.2\right].$$

- If base 2, $H(X) \approx 1.485\,\text{bits}$.

Interpreting Binary Questions

- Suppose X represents a random word from $\{\texttt{cat}, \texttt{dog}, \texttt{mouse}\}$:
- To identify the outcome of X using yes/no questions:
  - Q1: Is it cat? ($p(\texttt{cat}) = 0.5$) - If yes, stop (probability 0.5). - If no, proceed (probability 0.5).
  - Q2: Is it dog? ($p(\texttt{dog}) = 0.3$) - If yes, stop (probability 0.3). - If no, stop at mouse (probability 0.2).
- **Expected Number of Questions:**

$$H(X) \approx 1.485 \text{ bits} \quad \text{(on average, slightly fewer than 2 binary questions).}$$

Cross-Entropy and the Derivation of KL-Divergence I

Cross-Entropy Definition

Let $p(x)$ be the *true* distribution and $q(x)$ be a *model* distribution over the same set $\mathcal{V}$. The **cross-entropy** $H(p, q)$ is:

$$H(p, q) = -\sum_{x \in \mathcal{V}} p(x) \log q(x).$$

- Measures how well the model q "fits" the true data p.
- If $q = p$, then $H(p, q) = H(p)$, the entropy of p.

Derivation of KL-Divergence

Starting from the cross-entropy:

$$H(p,q) = -\sum_{x\in\mathcal{V}} p(x)\,\log\,q(x),$$

we can rewrite:

$$-\sum_{x\in\mathcal{V}} p(x)\,\log\,q(x) = -\sum_{x\in\mathcal{V}} p(x)\,\log\,p(x)\;-\;\sum_{x\in\mathcal{V}} p(x)\,\log\!\left(\frac{q(x)}{p(x)}\right).$$

Observe that

$$\log\,q(x) = \log\,p(x) + \log\!\left(\tfrac{q(x)}{p(x)}\right).$$

Therefore,

$$H(p,q) = \underbrace{-\sum_{x\in\mathcal{V}} p(x)\,\log\,p(x)}_{=H(p)} \;+\; \underbrace{\sum_{x\in\mathcal{V}} p(x)\,\log\!\left(\frac{p(x)}{q(x)}\right)}_{=D_{\mathrm{KL}}(p\,\|\,q)}.$$

Cross-Entropy and the Derivation of KL-Divergence III

KL-Divergence

We define the **Kullback–Leibler (KL) divergence** as

$$D_{\mathrm{KL}}(p \,\|\, q) \;=\; \sum_{x \in \mathcal{V}} p(x) \,\log \frac{p(x)}{q(x)} \;\geq\; 0.$$

Hence, we obtain the well-known relationship:

$$\boxed{H(p, q) \;=\; H(p) \;+\; D_{\mathrm{KL}}(p \,\|\, q).}$$

- $D_{\mathrm{KL}}(p \,\|\, q) = 0$ if and only if $p = q$.
- Minimizing cross-entropy $\Leftrightarrow$ Minimizing KL-divergence.

Using Cross-Entropy for LM Evaluation I

Empirical vs. Model Distribution

- We have a **test set** of sequences:

$$\mathcal{D}_{\text{test}} = \left\{ \left(w_1^{(i)}, w_2^{(i)}, \ldots, w_{n^{(i)}}^{(i)} \right) \right\}_{i=1}^{M}.$$

- Let $N = \sum_{i=1}^{M} n^{(i)}$ be the total number of tokens across all sequences.
- The *empirical distribution* $\hat{p}$ places probability $\frac{1}{N}$ on each token $w_k^{(i)}$ in $\mathcal{D}_{\text{test}}$.
- Our **language model** is a distribution $p_{\theta}(w_k \mid w_{1:k-1})$ over the next token given its context.

Using Cross-Entropy for LM Evaluation II

Per-Token Cross-Entropy and Negative Log-Likelihood

- **Cross-Entropy:**

$$H(\hat{p},\, p_{\boldsymbol{\theta}}) = -\sum_{i=1}^{N} \frac{1}{N} \log p_{\boldsymbol{\theta}}(w^{(i)}),$$

  where each $w^{(i)}$ is treated as an i.i.d. sample from $\hat{p}$.
- Equivalently,

$$H(\hat{p}, p_{\boldsymbol{\theta}}) = -\frac{1}{N} \sum_{i=1}^{M} \sum_{k=1}^{n^{(i)}} \log\Big(p_{\boldsymbol{\theta}}(w_k^{(i)} \mid w_{1:k-1}^{(i)})\Big).$$

- This **per-token cross-entropy** is exactly the **average negative log-likelihood** of the test set under $p_{\boldsymbol{\theta}}$.
- $\downarrow$ **Lower cross-entropy** $\Rightarrow$ the model assigns *higher probability* to the observed tokens.

Perplexity as a Measure of LM Quality

Definition of Perplexity

- Perplexity is an exponentiation of the cross-entropy, providing a more intuitive scale.
- If using natural logs,
$$\mathrm{PP}(p_\theta) = \exp\Big(H(\hat{p}, p_\theta)\Big).$$
- If using base-2 logs,
$$\mathrm{PP}(p_\theta) = 2^{\,H(\hat{p}, p_\theta)}.$$

Why Perplexity is Intuitive

- **Average Branching Factor:**
  - Imagine each token prediction as choosing among equally likely options.
  - Perplexity says "on average, how many distinct choices does the model effectively consider?"
  - A perplexity of 1 means the model is *never* uncertain; larger values indicate greater uncertainty.

Outline

- Introduction to Language Models
- Vocabulary and Tokenization
- Applications
- Parametrization and Estimation
- n-Gram Language Models
- Addressing Data Sparsity in n-Gram Models
- Evaluation Metrics for Language Models
- Toy Example
- Generation Strategies for Language Models
- Feed-Forward Neural Language Models
- Training
- Recurrent Neural Networks (RNNs)
- LSTMs and GRUs
- What's Next?

Example: Toy Corpus and Tri-Gram Model Setup I

Corpus & Vocabulary

Toy Corpus $\mathcal{D}$ consists of three sentences, each prepended with two `<s>`:

<s> <s> i love cats </s>

<s> <s> i love dogs </s>

<s> <s> cats chase mice </s>

Vocabulary $\mathcal{V}$: $\{\texttt{<s>}, \texttt{i}, \texttt{love}, \texttt{cats}, \texttt{dogs}, \texttt{chase}, \texttt{mice}, \texttt{</s>}\}$.

Trigram Model Assumption

- For each position k, we model $p_\theta(w_k \mid w_{k-2}, w_{k-1})$.
- Example: In `<s> <s> i love cats </s>`, the third token `i` is predicted by $p_\theta(\texttt{i} \mid \texttt{<s>}, \texttt{<s>})$.
- We will collect all (2-token context, next token) counts from $\mathcal{D}$ and apply MLE:

$$\hat{\theta}_{\boldsymbol{c},w} = \frac{\text{count}(\boldsymbol{c}, w)}{\text{count}(\boldsymbol{c})}.$$

Example: Toy Corpus and Tri-Gram Model Setup II

Context-Next Token Counts

Below is a **partial** table of contexts ($\boldsymbol{c} = (w_{k-2}, w_{k-1})$) and how often each next token appears:

Context (w_{k-2}, w_{k-1})	**Next Token**	**Count**	**Sum Over Next Toks**	**MLE Probability**
(<s>, <s>)	i	2	3	$\frac{2}{3} \approx 0.67$
(<s>, <s>)	cats	1		$\frac{1}{3} \approx 0.33$
(<s>, i)	love	2	2	$\frac{2}{2} = 1.0$
(i, love)	cats	1	2	$\frac{1}{2} = 0.5$
(i, love)	dogs	1		$\frac{1}{2} = 0.5$
(love, cats)	</s>	1	1	1.0
...				

Note: Fill out this table for *all* observed 2-token contexts in the corpus (omitting zero-count contexts not observed, or using smoothing).

Example: Toy Corpus and Tri-Gram Model Setup III

Probability of a New Sentence

Test Sentence: `<s> <s> i love mice </s>`

- Using chain rule for trigrams:

$$p_\theta(\texttt{<s> <s> i love mice </s>}) = p_\theta(\texttt{i} \mid \texttt{<s> <s>}) \times p_\theta(\texttt{love} \mid \texttt{<s> i}) \times p_\theta(\texttt{mice} \mid \texttt{i love}) \times p_\theta(\texttt{</s>} \mid \texttt{love mice}).$$

- Since $p_\theta(\texttt{mice} \mid \texttt{i}, \texttt{love}) = 0$, then the entire product is zero *unless* we apply smoothing.

Cross-Entropy & Perplexity Computation

- Let N be total tokens in `<s> <s> i love mice </s>` (which is 6).
- **Per-token cross-entropy** $= -\frac{1}{6}\sum_{k=1}^{5} \log p_\theta(w_k \mid w_{k-2}, w_{k-1})$.
- **Perplexity** $= \exp($cross-entropy$)$.
- **Example:** If $p_\theta\big(\texttt{mice} \mid \texttt{i}, \texttt{love}\big) = 0$, perplexity is infinite.

Outline

- Introduction to Language Models
- Vocabulary and Tokenization
- Applications
- Parametrization and Estimation
- n-Gram Language Models
- Addressing Data Sparsity in n-Gram Models
- Evaluation Metrics for Language Models
- Toy Example
- Generation Strategies for Language Models
- Feed-Forward Neural Language Models
- Training
- Recurrent Neural Networks (RNNs)
- LSTMs and GRUs
- What's Next?

Decoding Algorithm: Greedy vs. Sampling (with Temperature)

Input:

- Trained language model $p_\theta(w_k \mid w_1, \ldots, w_{k-1})$. Initial context c_{init} (e.g., (<s>, <s>) for a trigram model).
- Decoding strategy: choose either `greedy` or `sampling`. (Optional) Temperature T for sampling.

Algorithm:

1. Initialize context $\boldsymbol{c} \leftarrow \boldsymbol{c}_{\text{init}}$ and set *sequence* $\leftarrow []$.
2. **Repeat**
   2.1 Compute $p_{\boldsymbol{\theta}}(w \mid \boldsymbol{c})$ for all $w \in \mathcal{V}$.
   2.2 **if** strategy is `greedy`:
   $$w^* \leftarrow \arg\max_{w \in \mathcal{V}} p_{\boldsymbol{\theta}}(w \mid \boldsymbol{c}).$$
   2.3 **else if** strategy is `sampling`:
   $$w^* \sim p_{\boldsymbol{\theta}}^{(T)}(w \mid \boldsymbol{c}) = \frac{p_{\boldsymbol{\theta}}(w \mid \boldsymbol{c})^{1/T}}{\sum_{w' \in \mathcal{V}} p_{\boldsymbol{\theta}}(w' \mid \boldsymbol{c})^{1/T}}.$$
   2.4 Append w^* to *sequence* and update context $\boldsymbol{c}$.
3. **Until** $w^* =$ </s>.
4. **Return** *sequence* (optionally excluding special tokens like <s> and </s>).

Generation Strategies for Language Models II

Greedy vs. Sampling

- **Sampling**:
  - At each step, sample the next token w_k from $p_{\boldsymbol{\theta}}(w_k \mid w_1, \ldots w_{k-1})$.
  - **Pros:** Can produce diverse, creative outputs.
  - **Cons:** May generate nonsensical or low-probability tokens if distribution is broad.
- **Greedy Decoding**:
  - Always pick the token w_k with the highest probability $\arg\max p_\theta(w_k \mid w_1, \ldots w_{k-1})$.
  - **Pros:** Fastest method, easy to implement.
  - **Cons:** Often gets stuck in repetitive or sub-optimal sequences (lack of diversity).
- **Temperature Scaling**
  - Effects of T: $T > 1$: Flattens the distribution, increasing randomness. $T < 1$: Sharpens the distribution.
  - **Pros:** Fine-grained control over output randomness.
  - **Cons:** Requires careful tuning of T for desired behavior.

Other Sampling Strategies

- **Top-k:** Restrict sampling to the k most probable tokens at each step.
- **Nucleus (Top-p):** Sample from the smallest set of tokens whose cumulative probability exceeds p.

Generation Strategies for Language Models III

Beam Search Algorithm

Input:

Trained language model $p_{\boldsymbol{\theta}}(w_k \mid w_1, \ldots, w_{k-1})$.
Initial context $\boldsymbol{c}_{\text{init}}$ (e.g., (<s>, <s>)).
Beam size B (number of parallel hypotheses to maintain) and maximum length L.

Algorithm:

Initialize *candidates* $\leftarrow \{(\boldsymbol{c}_{\text{init}}, 0)\}$, where each candidate is a tuple of context and log-probability.
Initialize *final_sequences* $\leftarrow []$.
Repeat:
For each candidate $(\boldsymbol{c}, \textit{score})$ in *candidates*:
Compute $p_{\boldsymbol{\theta}}(w \mid \boldsymbol{c})$ for all $w \in \mathcal{V}$.
Extend $\boldsymbol{c}$ with each w, forming new candidates:
$(\boldsymbol{c} + w, \text{score} + \log p_{\boldsymbol{\theta}}(w \mid \boldsymbol{c}))$.
If $w =$ </s>:
Move $(\boldsymbol{c} + w, \textit{score})$ to *final_sequences*.
Retain the top B candidates by *score* for the next step.
Until: All B candidates end with </s> or maximum length is reached.
Return: The highest-scoring sequence from *final_sequences*.

Outline

- Introduction to Language Models
- Vocabulary and Tokenization
- Applications
- Parametrization and Estimation
- n-Gram Language Models
- Addressing Data Sparsity in n-Gram Models
- Evaluation Metrics for Language Models
- Toy Example
- Generation Strategies for Language Models
- **Feed-Forward Neural Language Models**
- Training
- Recurrent Neural Networks (RNNs)
- LSTMs and GRUs
- What's Next?

Problems with Categorical n-Gram Parametrization

Exponential Growth of Parameters

- A categorical n-gram model requires a unique parameter $\theta_{\boldsymbol{c},w}$ for each context $\boldsymbol{c}$ (of length $n-1$) and next word w. Total number of parameters is exponential in the context length $n-1$:

$$|\mathcal{V}|^{n-1} \cdot (|\mathcal{V}| - 1).$$

Sparsity and Zero Probabilities

- For most possible n-grams, the count $\text{count}(\boldsymbol{c}, w) \approx 0$, causing $\theta_{\boldsymbol{c},w} \approx 0$ for many $(\boldsymbol{c}, w)$ if no smoothing is used to modify the counts.

Lookup Table Representation $p(w \mid \boldsymbol{c}) = \theta_{\boldsymbol{c},w}$

- **Input:** Each word in the key n-gram can be seen as a **one-hot vector** $\mathbf{1}_w \in \{0,1\}^{|\mathcal{V}|}$. The n-gram $(\boldsymbol{c}_1, \ldots, \boldsymbol{c}_{n-1}, w)$ can be seen as a concatenation of n one-hot word vectors:

$$\boldsymbol{x} = \left[\mathbf{1}_{\boldsymbol{c}_1}; \ldots; \mathbf{1}_{\boldsymbol{c}_{n-1}}; \mathbf{1}_w\right] \in \mathbb{R}^{|\mathcal{V}| \cdot n}.$$

- There is no intrinsic notion of similarity between different contexts in this representation.

Feed-Forward Neural Network Parametrization

Input Mapping

- Each word w in the context $\boldsymbol{c}$ is mapped to an embedding $\boldsymbol{e}_w \in \mathbb{R}^d$, with $d \ll |\mathcal{V}|$.
- Computed by projecting one-hot vectors through an embedding matrix $\boldsymbol{E} \in \mathbb{R}^{|\mathcal{V}| \times d}$: $\boldsymbol{e}_w = \boldsymbol{E}^\top \mathbf{1}_w$.
- Embeddings of the $n-1$ context words are concatenated

$$\boldsymbol{x} = \left[\boldsymbol{e}_{c_1}; \ldots; \boldsymbol{e}_{c_{n-1}} \right] \in \mathbb{R}^{d \cdot (n-1)}.$$

Hidden Layer

$$\boldsymbol{h} = \tanh\left(\boldsymbol{W}^{(h)} \boldsymbol{x} + \boldsymbol{b}^{(h)}\right), \quad \boldsymbol{W}^{(h)} \in \mathbb{R}^{m \times (d \cdot (n-1))}, \ \boldsymbol{b}^{(h)} \in \mathbb{R}^m.$$

- Computes a continuous context embedding $\boldsymbol{h} \in \mathbb{R}^m$. Learns to mix features from the word embeddings.

Output Layer

$$p(w \mid \boldsymbol{c}) = \boldsymbol{p}_w \quad \textit{where} \quad \boldsymbol{p} \in \mathbb{R}^{|\mathcal{V}|}, \ \boldsymbol{p} = \mathrm{softmax}\left(\boldsymbol{W}^{(o)} \boldsymbol{h} + \boldsymbol{b}^{(o)}\right), \quad \boldsymbol{W}^{(o)} \in \mathbb{R}^{|\mathcal{V}| \times m}, \ \boldsymbol{b}^{(o)} \in \mathbb{R}^{|\mathcal{V}|}.$$

Comparison: Categorical vs. Neural Parametrization I

Parameter Sets

Categorical *n*-gram:

$$\boldsymbol{\theta}_{\text{cat}} = \left\{ \theta_{\boldsymbol{c},w} \mid \boldsymbol{c} \in \mathcal{V}^{n-1},\ w \in \mathcal{V} \right\}$$

Neural LM:

$$\boldsymbol{\theta}_{\text{nn}} = \left\{ \boldsymbol{E} \in \mathbb{R}^{|\mathcal{V}| \times d},\ \boldsymbol{W}^{(h)} \in \mathbb{R}^{m \times (d \cdot (n-1))},\ \boldsymbol{b}^{(h)} \in \mathbb{R}^{m},\ \boldsymbol{W}^{(o)} \in \mathbb{R}^{|\mathcal{V}| \times m},\ \boldsymbol{b}^{(o)} \in \mathbb{R}^{|\mathcal{V}|} \right\}.$$

- Here, $\boldsymbol{h} \in \mathbb{R}^m$ is the hidden context embedding. Neural LM uses far fewer parameters due to sharing across contexts.

Comparison: Categorical vs. Neural Parametrization II

Practical Example of Parameter Sizes

Assume a vocabulary size $|\mathcal{V}| = 10{,}000$, embedding dimension $d = 300$, hidden layer size $m = 500$, and $n = 3$ (trigram).

- **Categorical Trigram:**

$$\text{Parameters} \approx |\mathcal{V}|^2 \cdot (|\mathcal{V}| - 1) \approx 10{,}000^2 \cdot 9{,}999 \approx 10^{12}.$$

- **Neural Trigram:**

$$\begin{aligned}
&|\mathcal{V}| \times d + m \times ((n-1) \cdot d) + m + |\mathcal{V}| \times m + |\mathcal{V}| \\
&\approx 10{,}000 \times 300 + 500 \times (2 \times 300) + 500 + 10{,}000 \times 500 + 10{,}000 \\
&\approx 3 \times 10^6 + 300{,}000 + 500 + 5 \times 10^6 + 10{,}000 \\
&\approx 8.3 \times 10^6 \text{ parameters.}
\end{aligned}$$

Feature Exatraction

Feature Mixing in Hidden Layers

$$\underbrace{\begin{pmatrix} \cdots & \underbrace{W^{(h)}_{j,i_1}}_{\text{large weight}} & \cdots & \underbrace{W^{(h)}_{j,i_2}}_{\text{large weight}} & \cdots \end{pmatrix}}_{\text{row } j \text{ of } \boldsymbol{W}^{(h)}} \times \underbrace{\begin{pmatrix} \vdots \\ \underbrace{x_{i_1}}_{\text{dim } i_1} \\ \vdots \\ \underbrace{x_{i_2}}_{\text{dim } i_2} \\ \vdots \end{pmatrix}}_{\boldsymbol{x}} \longrightarrow h_j = \tanh\left(\sum_{i=1}^{d\cdot(n-1)} W^{(h)}_{j,i}\, x_i \,+\, b^{(h)}_j \right).$$

- Each row $\boldsymbol{W}^{(h)}_{j,\cdot}$ selectively combines specific dimensions of the input $\boldsymbol{x}$.
- Larger weights $\left|W^{(h)}_{j,i}\right|$ amplify embedding dimensions (e.g., those tied to nouns or adjectives).
- Thus, h_j can learn a particular pattern by focusing on relevant parts of $\boldsymbol{x}$.

Outline

- Introduction to Language Models
- Vocabulary and Tokenization
- Applications
- Parametrization and Estimation
- n-Gram Language Models
- Addressing Data Sparsity in n-Gram Models
- Evaluation Metrics for Language Models
- Toy Example
- Generation Strategies for Language Models
- Feed-Forward Neural Language Models
- **Training**
- Recurrent Neural Networks (RNNs)
- LSTMs and GRUs
- What's Next?

Training a Feed-Forward Neural LM I

Training Corpus and Empirical Distribution

- Let $\mathcal{D} = \{\boldsymbol{w}^{(i)}\}_{i=1}^{N}$ be a set of N sentences (or sequences), each $\boldsymbol{w}^{(i)} = (w_1^{(i)}, \ldots, w_{T_i}^{(i)})$.
- The **empirical distribution** $\hat{p}(\boldsymbol{w})$ places probability $\frac{1}{N}$ on each training sentence $\boldsymbol{w}^{(i)}$.

Cross-Entropy ⇔ Maximum Likelihood

- Our model $p_{\boldsymbol{\theta}}(\boldsymbol{w})$ assigns a probability to any sentence $\boldsymbol{w}$. **Cross-entropy** between $\hat{p}$ and $p_{\boldsymbol{\theta}}$:

$$H(\hat{p}, p_{\boldsymbol{\theta}}) = -\sum_{i=1}^{N} \frac{1}{N} \log p_{\boldsymbol{\theta}}\left(\boldsymbol{w}^{(i)}\right).$$

- Minimizing $H(\hat{p}, p_{\boldsymbol{\theta}}) \Leftrightarrow \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{N} p_{\boldsymbol{\theta}}\left(\boldsymbol{w}^{(i)}\right)$, i.e. **maximum likelihood estimation (MLE)**.
- This objective is also known as the **negative log-likelihood (NLL)**:

$$\ell(\boldsymbol{\theta}) = -\sum_{i=1}^{N} \log p_{\boldsymbol{\theta}}\left(\boldsymbol{w}^{(i)}\right).$$

Training a Feed-Forward Neural LM II

Chain Rule and n-grams

- In a feed-forward LM with context size $n-1$:

$$p_{\boldsymbol{\theta}}\left(\boldsymbol{w}^{(i)}\right) = \prod_{k=1}^{T_i} p_{\boldsymbol{\theta}}\left(w_k^{(i)} \mid w_{k-n+1}^{(i)}, \ldots, w_{k-1}^{(i)}\right).$$

- Each term $p_{\boldsymbol{\theta}}(w_k \mid \boldsymbol{c}_k)$ is computed via:

$$\boldsymbol{c}_k \mapsto \boldsymbol{x} \mapsto \boldsymbol{h} \mapsto \boldsymbol{p} = \mathrm{softmax}\left(\boldsymbol{W}^{(o)}\boldsymbol{h} + \boldsymbol{b}^{(o)}\right), \quad p_{\boldsymbol{\theta}}(w_k \mid \boldsymbol{c}_k) = \boldsymbol{p}_{w_k}.$$

Loss Over the Entire Corpus

$$\ell(\boldsymbol{\theta}) = -\sum_{i=1}^{N} \log p_{\boldsymbol{\theta}}\left(\boldsymbol{w}^{(i)}\right) = -\sum_{i=1}^{N}\sum_{k=1}^{T_i} \log p_{\boldsymbol{\theta}}\left(w_k^{(i)} \mid \boldsymbol{c}_k^{(i)}\right).$$

- Minimizing $\ell(\boldsymbol{\theta})$ sums the negative log-probabilities over all context-target pairs $(\boldsymbol{c}_k, w_k)$.
- **Single Pair Loss:** $\ell(\boldsymbol{\theta}; \boldsymbol{c}, w) = -\log p_{\boldsymbol{\theta}}(w \mid \boldsymbol{c})$.

Training a Feed-Forward Neural LM III

Parameter Update Rule

- Use gradient-based methods (SGD, Adam, etc.) to update parameters:

$$\theta \leftarrow \theta - \eta \nabla_{\theta} \ell(\boldsymbol{\theta}).$$

- η is the learning rate. In practice, Adam or RMSProp handle adaptive step sizes and momentum.

Forward, Loss, and Backprop

Forward Pass:

$$\boldsymbol{x} = \left[\boldsymbol{e}_{w_{k-n+1}}; \ldots; \boldsymbol{e}_{w_{k-1}}\right],\ \boldsymbol{h} = \tanh(\boldsymbol{W}^{(h)}\,\boldsymbol{x} + \boldsymbol{b}^{(h)}),\ \boldsymbol{z} = \boldsymbol{W}^{(o)}\,\boldsymbol{h} + \boldsymbol{b}^{(o)},\ \boldsymbol{p} = \mathrm{softmax}(\boldsymbol{z}).$$

$$\ell(\boldsymbol{\theta}; \boldsymbol{c}, w) = -\log \boldsymbol{p}_w.$$

Backward Pass:

- Derive $\nabla_{\boldsymbol{z}}\ell$ from the softmax derivative, propagate to $\boldsymbol{h}$ and $\boldsymbol{x}$ via chain rule (through $\tanh$, matrix multiplies).
- Accumulate gradients for $\boldsymbol{W}^{(h)}, \boldsymbol{b}^{(h)}, \boldsymbol{W}^{(o)}, \boldsymbol{b}^{(o)}$, and $\boldsymbol{E}$ (embedding matrix).

Training a Feed-Forward Neural LM IV

Mini-Batch Training and Vectorization

- Instead of processing one $(\boldsymbol{c}, w)$ at a time, we group examples into mini-batches (e.g., size 32).
- **Vectorization:**
  - Stack the $\boldsymbol{x}$ vectors of multiple examples into a matrix $\boldsymbol{X}$.
  - Compute $\boldsymbol{W}^{(h)}\boldsymbol{X}$ (and subsequent layers) in parallel for the whole batch.
- Average gradients over the mini-batch, then update parameters, resulting in more stable training and GPU efficiency.

Detailed Gradient Derivations I

Setup:

$$\ell(\boldsymbol{\theta}; \boldsymbol{c}, w) = -\log p_{\boldsymbol{\theta}}(w \mid \boldsymbol{c}), \quad p_{\boldsymbol{\theta}}(w \mid \boldsymbol{c}) = \mathrm{softmax}(\boldsymbol{z})_w, \quad \boldsymbol{z} = \boldsymbol{W}^{(o)}\,\boldsymbol{h} + \boldsymbol{b}^{(o)},$$

$$\boldsymbol{h} = \tanh(\boldsymbol{W}^{(h)}\,\boldsymbol{x} + \boldsymbol{b}^{(h)}), \quad \boldsymbol{x} = \left[\boldsymbol{e}_{w_{k-n+1}}, \ldots, \boldsymbol{e}_{w_{k-1}}\right].$$

where $\boldsymbol{p} = \mathrm{softmax}(\boldsymbol{z})$ and $\boldsymbol{y} \in \{0,1\}^{|\mathcal{V}|}$ is the one-hot vector for the correct word w. Then:

1. **Gradient w.r.t. output logits $\boldsymbol{z}$**:

$$\frac{\partial \ell}{\partial z_j} = \frac{\partial}{\partial z_j}\left[-\log(\boldsymbol{p}_w)\right] = p_j - y_j \quad (\text{for } j = 1, \ldots, |\mathcal{V}|).$$

2. **Output layer parameters**:

$$\nabla_{\boldsymbol{W}^{(o)}} \ell = (\boldsymbol{p} - \boldsymbol{y})\,\boldsymbol{h}^{\top}, \quad \nabla_{\boldsymbol{b}^{(o)}} \ell = \boldsymbol{p} - \boldsymbol{y}.$$

3. **Hidden layer gradient**:

$$\nabla_{\boldsymbol{h}} \ell = (\boldsymbol{W}^{(o)})^{\top}(\boldsymbol{p} - \boldsymbol{y}).$$

   Then apply chain rule for $\tanh$:

$$\nabla_{\boldsymbol{z}^{(h)}} \ell = \left(1 - \tanh^2(\boldsymbol{z}^{(h)})\right) \odot \nabla_{\boldsymbol{h}} \ell,$$

   where $\boldsymbol{z}^{(h)} = \boldsymbol{W}^{(h)}\,\boldsymbol{x} + \boldsymbol{b}^{(h)}$.

Detailed Gradient Derivations II

4. **Hidden layer parameters**:

$$\nabla_{\boldsymbol{W}^{(h)}} \ell = \nabla_{\boldsymbol{z}^{(h)}} \ell \, \boldsymbol{x}^\top, \quad \nabla_{\boldsymbol{b}^{(h)}} \ell = \nabla_{\boldsymbol{z}^{(h)}} \ell.$$

5. **Embedding matrix *E***: Backprop through $\boldsymbol{x}$ (the concatenation of each context word's embedding). Each relevant row in $\boldsymbol{E}$ is updated according to $\dfrac{\partial \ell}{\partial \boldsymbol{e}_{w_i}}$.

Outline

- Introduction to Language Models
- Vocabulary and Tokenization
- Applications
- Parametrization and Estimation
- n-Gram Language Models
- Addressing Data Sparsity in n-Gram Models
- Evaluation Metrics for Language Models
- Toy Example
- Generation Strategies for Language Models
- Feed-Forward Neural Language Models
- Training
- Recurrent Neural Networks (RNNs)
- LSTMs and GRUs
- What's Next?

Motivation: Moving Beyond Fixed Context Size

Limitations of Feed-Forward LM

- **Fixed Window**: A feed-forward LM uses a context of size $n-1$. Any dependency beyond $n-1$ tokens is *not* captured.
- **Long-Distance Dependencies in Language**:
  - Example:

    *The **car** that I drove yesterday **broke down** this morning.*

    The mention of "car" is quite far from the point where we describe what happened to it.

Recurrent Neural Networks (RNNs)

- Designed to capture *variable-length* contexts and long-distance dependencies by maintaining a **hidden state** that updates at each time step.
- The RNN hidden state plays the role of **memory**, combining information from all previous tokens.

Elman RNN: Detailed Equations I

Notation and Setup

- Let $\boldsymbol{w} = (w_1, w_2, \ldots, w_T)$ be a tokenized sequence.
- At each time step t, the RNN processes the *embedding* $\boldsymbol{x}_t \in \mathbb{R}^d$ of the current token w_t.
- Maintains a hidden state $\boldsymbol{h}_t \in \mathbb{R}^m$ capturing *all previously seen* tokens, thus overcoming the fixed-window limitation.

Forward Pass of an Elman RNN

$$\boldsymbol{h}_t = \tanh\Big(\boldsymbol{W}_{xh}\,\boldsymbol{x}_t \;+\; \boldsymbol{W}_{hh}\,\boldsymbol{h}_{t-1} \;+\; \boldsymbol{b}_h\Big), \qquad \boldsymbol{h}_0 = \mathbf{0} \text{ (or learned)}.$$

- $\boldsymbol{W}_{xh} \in \mathbb{R}^{m \times d}$: transforms current input $\boldsymbol{x}_t$ (as in feed-forward LMs).
- $\boldsymbol{W}_{hh} \in \mathbb{R}^{m \times m}$: **new** recurrent connection, combining the previous state $\boldsymbol{h}_{t-1}$.
- $\boldsymbol{b}_h \in \mathbb{R}^m$: bias term.
- $\tanh$: typical nonlinear activation; other choices (ReLU, etc.) are possible.

Elman RNN: Detailed Equations II

Key Difference vs. Feed-Forward LM

- Unlike a feed-forward LM (which sees only a fixed window of size $n-1$), the RNN recurrently incorporates $\boldsymbol{h}_{t-1}$ through $\boldsymbol{W}_{hh}$.
- This enables the network to (in principle) use an unbounded context.

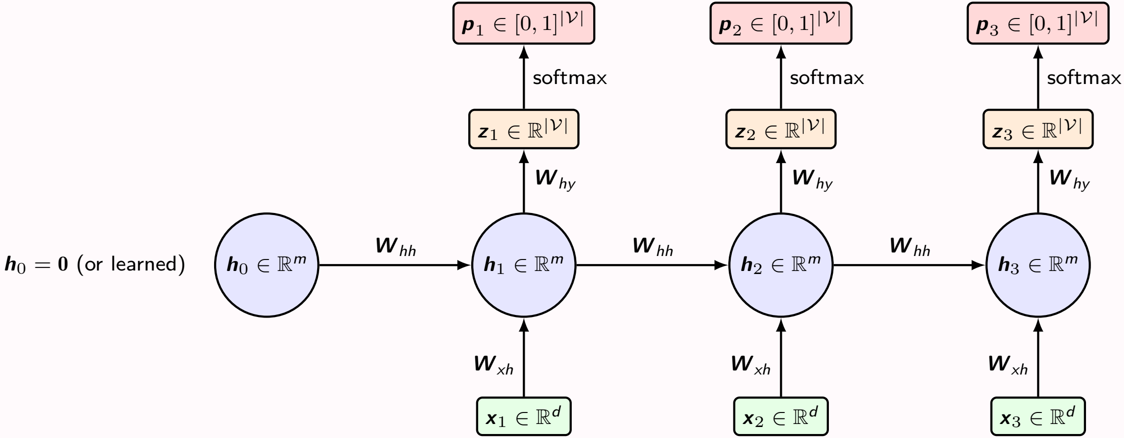
Output and Next-Word Distribution

$$\boldsymbol{z}_t = \boldsymbol{W}_{hy}\,\boldsymbol{h}_t + \boldsymbol{b}_y, \quad \boldsymbol{p}_t = \mathrm{softmax}(\boldsymbol{z}_t), \quad p_{\boldsymbol{\theta}}(w_{t+1} \mid w_{1\ldots t}) = \boldsymbol{p}_{t,\,w_{t+1}}.$$

- $\boldsymbol{z}_t \in \mathbb{R}^{|\mathcal{V}|}$: output logits for next token at time t.
- $\boldsymbol{p}_t \in \mathbb{R}^{|\mathcal{V}|}$: next-token probability distribution via softmax.
- $\boldsymbol{W}_{hy} \in \mathbb{R}^{|\mathcal{V}|\times m}$, $\boldsymbol{b}_y \in \mathbb{R}^{|\mathcal{V}|}$.

Unrolling RNN in time I

A Diagram



Unrolling RNN in time II

In Equations

- For inputs $\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \boldsymbol{x}_4$, each $\boldsymbol{x}_t \in \mathbb{R}^d$.
- The hidden state at the final time step ($\boldsymbol{h}_4 \in \mathbb{R}^m$) unfolds as:

$$
\begin{aligned}
\boldsymbol{h}_4 = \tanh\Big(& \boldsymbol{W}_{hh} \tanh\Big(\\
& \boldsymbol{W}_{hh} \tanh\Big(\\
& \boldsymbol{W}_{hh} \tanh\Big(\\
& \boldsymbol{W}_{hh}\, \boldsymbol{h}_0 + \boldsymbol{W}_{xh}\, \boldsymbol{x}_1 + \boldsymbol{b}_h \Big) \\
& + \boldsymbol{W}_{xh}\, \boldsymbol{x}_2 + \boldsymbol{b}_h \Big) \\
& + \boldsymbol{W}_{xh}\, \boldsymbol{x}_3 + \boldsymbol{b}_h \Big) \\
& + \boldsymbol{W}_{xh}\, \boldsymbol{x}_4 + \boldsymbol{b}_h \Big)
\end{aligned}
$$

- Logits at time step 4 ($\boldsymbol{z}_4 \in \mathbb{R}^{|\mathcal{V}|}$): $\boldsymbol{z}_4 = \boldsymbol{W}_{hy}\, \boldsymbol{h}_4 + \boldsymbol{b}_y$.
- Notice that $\boldsymbol{h}_4$ depends on $\boldsymbol{h}_0$ and all prior inputs $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_4$, each influencing the hidden state through multiple nested $\tanh$ transformations.

Comparison with Feed-Forward LM Architecture

Drawbacks of Feed-Forward LM

- **Fixed window**: $(w_{k-n+1}, \dots, w_{k-1}) \to$ concat embeddings $\to$ hidden layer $\to$ softmax$(\dots)$.
- **Limitations**:
  - Cannot look beyond $(n-1)$ tokens of context.
  - Parameter explosion if n is large.
  - No built-in mechanism to capture long-distance or variable-length dependencies.

RNN LM Advantages

- **Implicitly unbounded context**: $\boldsymbol{h}_t$ in principle encodes all previous tokens $(w_1, \dots, w_{t-1})$.
- **Shared parameters over time steps**: leads to statistical strength and fewer parameters for large contexts than a large-window feed-forward LM.
- **Recurrent updating**: $\boldsymbol{h}_t$ evolves recursively, capturing sequential correlations in language.

Training an RNN LM and Its Challenges I

Objective and Unrolling in Time

- Similar to feed-forward LMs, we define a training set $\mathcal{D} = \{\boldsymbol{w}^{(i)}\}_{i=1}^{N}$ of sequences $\boldsymbol{w}^{(i)} = (w_1^{(i)}, \ldots, w_{T_i}^{(i)})$.
- Our RNN LM factorizes $p_{\boldsymbol{\theta}}(\boldsymbol{w})$ via:

$$p_{\boldsymbol{\theta}}(w_1, \ldots, w_T) = \prod_{t=1}^{T} p_{\boldsymbol{\theta}}(w_t \mid w_1, \ldots, w_{t-1}).$$

- **Unrolled Computation:**
  - A hidden state $\boldsymbol{h}_t \in \mathbb{R}^m$ is computed at each time t: $\boldsymbol{h}_t = f(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t)$ (e.g. Elman update with $\tanh$).
  - Output logits $\boldsymbol{z}_t = \boldsymbol{W}_{hy}\, \boldsymbol{h}_t + \boldsymbol{b}_y$, probabilities $\boldsymbol{p}_t = \mathrm{softmax}(\boldsymbol{z}_t)$.
- **Loss over entire sequence**:

$$\ell(\boldsymbol{\theta}; \boldsymbol{w}) = -\sum_{t=1}^{T} \log p_{\boldsymbol{\theta}}(w_t \mid w_{1:t-1}).$$

Training an RNN LM and Its Challenges II

Backprop Through Time (BPTT)

- We sum (or average) over all time steps and all sequences:

$$\ell(\boldsymbol{\theta}) = \sum_{i=1}^{N} \sum_{t=1}^{T_i} -\log p_{\boldsymbol{\theta}}\left(w_t^{(i)} \mid w_{1:t-1}^{(i)}\right).$$

- **Gradient Computation**:
  - We *unroll* the RNN across time steps $1 \dots T$.
  - Apply backprop to each unrolled connection, known as **BPTT**.
  - Accumulate gradients $\nabla \boldsymbol{W}_{xh}, \nabla \boldsymbol{W}_{hh}, \nabla \boldsymbol{W}_{hy}, \dots$.
- **Parameter Updates**:

$$\theta \leftarrow \theta - \eta \nabla_{\theta} \ell(\boldsymbol{\theta}),$$

typically in mini-batches for efficiency.

Training an RNN LM and Its Challenges III

Challenges: Vanishing/Exploding Gradients

- **Vanishing Gradients:**
  - When $\|\boldsymbol{W}_{hh}\| < 1$, backprop terms can decay exponentially over many steps.
  - The model struggles to learn long-term dependencies.
- **Exploding Gradients:**
  - When $\|\boldsymbol{W}_{hh}\| > 1$, gradients can grow exponentially, causing instability.
  - Common solutions: gradient clipping, careful initialization.
- Both issues arise because gradients repeatedly multiply through $\boldsymbol{W}_{hh}$ across time.
- **Recurrent Architectures (LSTM/GRU)** partially address these challenges with gating.

Detailed Gradients for one sequence: $\nabla_{\boldsymbol{h}_t}\ell(\boldsymbol{\theta};\boldsymbol{w})$

Goal: Gradient w.r.t. Hidden State

- Let us call $L = \ell(\boldsymbol{\theta};\boldsymbol{w}) = -\sum_{t=1}^{T}\log p_{\boldsymbol{\theta}}(w_t \mid w_{1:t-1})$.
- We want $\frac{\partial L}{\partial \boldsymbol{h}_t}$, the gradient of the total sequence loss L wrt. the hidden state $\boldsymbol{h}_t$:

$$\frac{\partial L}{\partial \boldsymbol{h}_t} = \frac{\partial L_t}{\partial \boldsymbol{h}_t} + \frac{\partial L_{t+1}}{\partial \boldsymbol{h}_t} + \ldots + \frac{\partial L_T}{\partial \boldsymbol{h}_t}.$$

- Summing direct and indirect contributions:

$$\frac{\partial L}{\partial \boldsymbol{h}_t} = \underbrace{\frac{\partial L_t}{\partial \boldsymbol{h}_t}}_{\textbf{direct from step } t} + \sum_{k=t+1}^{T} \underbrace{\frac{\partial L_k}{\partial \boldsymbol{h}_t}}_{\textbf{indirect from future steps } k>t} = \frac{\partial L_t}{\partial \boldsymbol{h}_t} + \sum_{k=t+1}^{T} \frac{\partial L_k}{\partial \boldsymbol{h}_{t+1}} \frac{\partial \boldsymbol{h}_{t+1}}{\partial \boldsymbol{h}_t}.$$

- Often simplified as a **recursive formula**:

$$\frac{\partial L}{\partial \boldsymbol{h}_t} = \frac{\partial L_t}{\partial \boldsymbol{h}_t} + \frac{\partial L}{\partial \boldsymbol{h}_{t+1}} \frac{\partial \boldsymbol{h}_{t+1}}{\partial \boldsymbol{h}_t}.$$

Computing the Recursive Gradient I

Hidden State Update

- Recall the simple Elman RNN:

$$\boldsymbol{h}_{t+1} = \tanh(\boldsymbol{a}_{t+1}), \quad \boldsymbol{a}_{t+1} = \boldsymbol{W}_{hh}\, \boldsymbol{h}_t \;+\; \boldsymbol{W}_{xh}\, \boldsymbol{x}_{t+1} \;+\; \boldsymbol{b}_h.$$

- We compute:

$$\frac{\partial \boldsymbol{h}_{t+1}}{\partial \boldsymbol{h}_t} = \underbrace{\frac{\partial \boldsymbol{h}_{t+1}}{\partial \boldsymbol{a}_{t+1}}}_{\text{diag}(1-\tanh^2(\boldsymbol{a}_{t+1}))} \cdot \underbrace{\frac{\partial \boldsymbol{a}_{t+1}}{\partial \boldsymbol{h}_t}}_{\boldsymbol{W}_{hh}}.$$

Computing the Recursive Gradient II

Chain Rule in Detail

- Since $\boldsymbol{h}_{t+1} = \tanh(\boldsymbol{a}_{t+1})$ and $\boldsymbol{W}_{xh}\,\boldsymbol{x}_{t+1}$ and $\boldsymbol{b}_h$ are constants wrt. $\boldsymbol{h}_t$:

$$\frac{\partial \boldsymbol{h}_{t+1}}{\partial \boldsymbol{a}_{t+1}} = \text{diag}\big(1 - \tanh^2(\boldsymbol{a}_{t+1})\big), \qquad \frac{\partial \boldsymbol{a}_{t+1}}{\partial \boldsymbol{h}_t} = \boldsymbol{W}_{hh}$$

- Combine:

$$\frac{\partial \boldsymbol{h}_{t+1}}{\partial \boldsymbol{h}_t} = \text{diag}\big(1 - \tanh^2(\boldsymbol{a}_{t+1})\big)\,\boldsymbol{W}_{hh}.$$

- Then the gradient update:

$$\frac{\partial L}{\partial \boldsymbol{h}_t} = \frac{\partial L_t}{\partial \boldsymbol{h}_t} + \frac{\partial L}{\partial \boldsymbol{h}_{t+1}}\,\frac{\partial \boldsymbol{h}_{t+1}}{\partial \boldsymbol{h}_t}.$$

$$= \frac{\partial L_t}{\partial \boldsymbol{h}_t} + \frac{\partial L}{\partial \boldsymbol{h}_{t+1}}\,\text{diag}(1 - \tanh^2(\boldsymbol{a}_{t+1}))\,\boldsymbol{W}_{hh}.$$

- Adjust for shape (often a transpose factor). Final form:

$$\boxed{\frac{\partial L}{\partial \boldsymbol{h}_t} = \frac{\partial L_t}{\partial \boldsymbol{h}_t} + \boldsymbol{W}_{hh}^\top\Big[\text{diag}\big(1 - \tanh^2(\boldsymbol{a}_{t+1})\big)\,\frac{\partial L}{\partial \boldsymbol{h}_{t+1}}\Big].}$$

Unrolling the Recursion

Repeated Application

- Applying the recurrence from t to $t+1$, $t+2$, etc. yields:

$$\begin{aligned}\frac{\partial L}{\partial \boldsymbol{h}_t} &= \frac{\partial L_t}{\partial \boldsymbol{h}_t} + \boldsymbol{W}_{hh}^\top \boldsymbol{\Phi}'_{t+1} \frac{\partial L}{\partial \boldsymbol{h}_{t+1}} \\ &= \frac{\partial L_t}{\partial \boldsymbol{h}_t} + \boldsymbol{W}_{hh}^\top \boldsymbol{\Phi}'_{t+1} \left(\frac{\partial L_{t+1}}{\partial \boldsymbol{h}_{t+1}} + \boldsymbol{W}_{hh}^\top \boldsymbol{\Phi}'_{t+2} \frac{\partial L}{\partial \boldsymbol{h}_{t+2}} \right) \\ &\vdots \\ &= \sum_{k=t}^{T} \left(\left(\prod_{j=t+1}^{k} \boldsymbol{W}_{hh}^\top \boldsymbol{\Phi}'_j \right) \frac{\partial L_k}{\partial \boldsymbol{h}_k} \right)\end{aligned}$$

- $\boldsymbol{\Phi}'_j$ denotes $\mathrm{diag}(1 - \tanh^2(\boldsymbol{a}_j))$.
- This product across many steps can **vanish** if $\|\boldsymbol{W}_{hh}\| < 1$ or **explode** if $\|\boldsymbol{W}_{hh}\| > 1$.

Vanishing & Exploding Gradients (Recap)

Vanishing Gradients

- If $\|\boldsymbol{W}_{hh}\|_2 < 1$, repeated multiplication shrinks gradients **exponentially** with distance:

$$\left\| \frac{\partial L}{\partial \boldsymbol{h}_t} \right\| \leq \left(\|\boldsymbol{W}_{hh}\|_2 \, \gamma \right)^{(k-t)} \left\| \frac{\partial L_k}{\partial \boldsymbol{h}_k} \right\|.$$

- Hard to learn long-term dependencies.

Exploding Gradients

- If $\|\boldsymbol{W}_{hh}\|_2 > 1$, norms can blow up:

$$\left\| \frac{\partial L}{\partial \boldsymbol{h}_t} \right\| \geq \left(\|\boldsymbol{W}_{hh}\|_2 \, \gamma \right)^{(k-t)} \left\| \frac{\partial L_k}{\partial \boldsymbol{h}_k} \right\|.$$

- Causes numerical instability; we often do **gradient clipping**.

Mitigating Gradient Problems

Common Strategies

- **Gradient Clipping**:
  - Restricts the norm $\|\nabla_\theta \ell\|$ to a predefined threshold.
  - Prevents numeric overflow when gradients become large (exploding gradients).
- **Initialization Techniques**:
  - Properly initializing $\boldsymbol{W}_{hh}, \boldsymbol{W}_{xh}$ etc. to maintain stable gradient propagation.
  - Use orthogonal or unitary matrices for $\boldsymbol{W}_{hh}$, e.g. $\boldsymbol{W}_{hh}\boldsymbol{W}_{hh}^\top = \mathbf{I}$.
    - Preserves the norm: $\|\boldsymbol{W}_{hh}\boldsymbol{x}\| = \|\boldsymbol{x}\|$.
    - Helps combat vanishing/exploding gradients.
- **Activation Functions**:
  - ReLU or similar (e.g. Leaky ReLU) can reduce gradient decay compared to $\tanh$.
  - For instance, $\mathrm{ReLU}(x) = \max(0, x)$, derivative is 1 for $x > 0$, allowing large gradient flow.
- **Advanced RNN Architectures**:
  - **LSTM** Introduces a *cell state* and gating mechanisms to preserve long-term information.
  - **GRU** A simpler variant of LSTM, also addresses gradient issues through gating.

Outline

- Introduction to Language Models
- Vocabulary and Tokenization
- Applications
- Parametrization and Estimation
- n-Gram Language Models
- Addressing Data Sparsity in n-Gram Models
- Evaluation Metrics for Language Models
- Toy Example
- Generation Strategies for Language Models
- Feed-Forward Neural Language Models
- Training
- Recurrent Neural Networks (RNNs)
- LSTMs and GRUs
- What's Next?

LSTM Architecture: Scalar and Vector Forms I

Scalar Equations (Conceptual)

- For each time t, an LSTM maintains c_t (the *cell state*) and h_t (the *hidden state*).
- Example (scalar version):

$$
\begin{aligned}
c_t &= f_t \cdot c_{t-1} \;+\; i_t \cdot z_t, && \text{cell state} \\
h_t &= o_t \; \psi(c_t), && \text{hidden output} \\
z_t &= \varphi(\tilde{z}_t), && \tilde{z}_t = w_z^\top x_t \;+\; r_z \, h_{t-1} \;+\; b_z, \\
i_t &= \sigma(\tilde{i}_t), && \tilde{i}_t = w_i^\top x_t \;+\; r_i \, h_{t-1} \;+\; b_i, \\
f_t &= \sigma(\tilde{f}_t), && \tilde{f}_t = w_f^\top x_t \;+\; r_f \, h_{t-1} \;+\; b_f, \\
o_t &= \sigma(\tilde{o}_t), && \tilde{o}_t = w_o^\top x_t \;+\; r_o \, h_{t-1} \;+\; b_o.
\end{aligned}
$$

- σ is the logistic sigmoid, φ could be $\tanh$. This form highlights the gating logic: i_t (input gate), f_t (forget gate), and o_t (output gate).

LSTM Architecture: Scalar and Vector Forms II

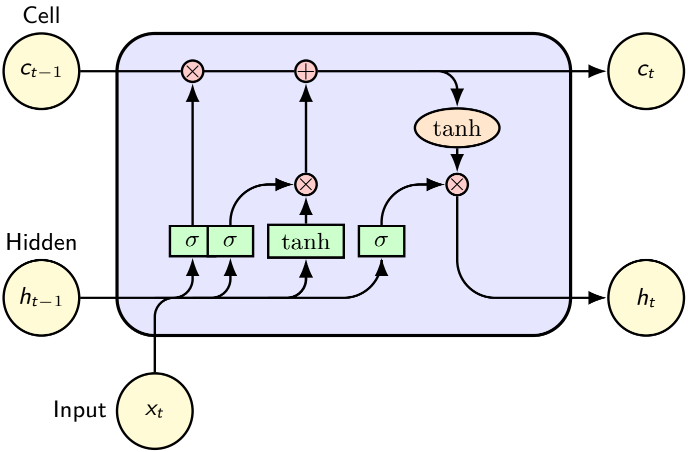
Vector Form (Practical Implementation)

- In practice, we combine scalar gates into vector/matrix operations. For each time t:

$$
\begin{aligned}
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} \;+\; \mathbf{i}_t \odot \mathbf{z}_t, \quad \mathbf{h}_t = \mathbf{o}_t \odot \psi(\mathbf{c}_t), \\
\mathbf{z}_t &= \varphi\big(\mathbf{W}_z\, \mathbf{x}_t + \mathbf{R}_z\, \mathbf{h}_{t-1} + \mathbf{b}_z\big), \\
\mathbf{i}_t &= \sigma\big(\mathbf{W}_i\, \mathbf{x}_t + \mathbf{R}_i\, \mathbf{h}_{t-1} + \mathbf{b}_i\big), \\
\mathbf{f}_t &= \sigma\big(\mathbf{W}_f\, \mathbf{x}_t + \mathbf{R}_f\, \mathbf{h}_{t-1} + \mathbf{b}_f\big), \\
\mathbf{o}_t &= \sigma\big(\mathbf{W}_o\, \mathbf{x}_t + \mathbf{R}_o\, \mathbf{h}_{t-1} + \mathbf{b}_o\big).
\end{aligned}
$$

- $\mathbf{x}_t \in \mathbb{R}^d, \quad \mathbf{h}_t, \mathbf{c}_t \in \mathbb{R}^m$.
- $\mathbf{W}_* \in \mathbb{R}^{m\times d}$, $\mathbf{R}_* \in \mathbb{R}^{m\times m}$, $\mathbf{b}_* \in \mathbb{R}^m$.
- Each gate $\mathbf{i}_t, \mathbf{f}_t, \mathbf{o}_t \in \mathbb{R}^m$ controls how info flows in/out of the cell state $\mathbf{c}_t$.

Illustration of an LSTM Cell Structure



How the LSTM's Constant Error Carousel (CEC) Addresses Vanishing Gradients I

Constant Error Carousel in LSTM

- The key update rule in LSTMs for the cell state $\mathbf{c}_t \in \mathbb{R}^m$:

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

  where $\odot$ is element-wise multiplication.
  - **Additive** updates (rather than purely multiplicative) avoid exponential shrinking of gradients.
  - Each component $\mathbf{f}_t, \mathbf{i}_t, \mathbf{z}_t$ is computed via gates (e.g. σ or $\tanh$).

How the LSTM's Constant Error Carousel (CEC) Addresses Vanishing Gradients II

Gradient Flow Through CEC

- **Recursive Gradient Equation:**

$$\frac{\partial L}{\partial \mathbf{c}_t} = \frac{\partial L_t}{\partial \mathbf{c}_t} + \left(\frac{\partial L}{\partial \mathbf{c}_{t+1}} \odot \mathbf{f}_{t+1} \right).$$

- **Unrolling the Recursion**:

$$\begin{aligned} \frac{\partial L}{\partial \mathbf{c}_t} &= \frac{\partial L_t}{\partial \mathbf{c}_t} + \left[\frac{\partial L_{t+1}}{\partial \mathbf{c}_{t+1}} + \left(\frac{\partial L}{\partial \mathbf{c}_{t+2}} \odot \mathbf{f}_{t+2} \right) \right] \odot \mathbf{f}_{t+1} \\ &= \frac{\partial L_t}{\partial \mathbf{c}_t} + \left(\frac{\partial L_{t+1}}{\partial \mathbf{c}_{t+1}} \odot \mathbf{f}_{t+1} \right) + \left(\frac{\partial L}{\partial \mathbf{c}_{t+2}} \odot \mathbf{f}_{t+2} \odot \mathbf{f}_{t+1} \right) + \dots \\ &= \sum_{k=t}^{T} \left(\frac{\partial L_k}{\partial \mathbf{c}_k} \odot \prod_{j=t+1}^{k} \mathbf{f}_j \right). \end{aligned}$$

- Each term is modulated by the product of forget gates $\mathbf{f}_j \in [0, 1]^m$, which can preserve gradient flow if $\mathbf{f}_j \approx 1$. This prevents the exponential decay of gradients, thus solving the vanishing gradient problem.

Gated Recurrent Units (GRUs)

Motivation

- **Simplify the LSTM architecture**: Reduce the number of gates and parameters while still addressing vanishing gradients.
- **Combine Forget and Input gates** into a single *update* gate to decide how much past information to keep or overwrite.
- Often yields comparable performance to LSTM with a simpler structure and sometimes trains faster.

Key Differences from LSTM

- **No separate cell state $\boldsymbol{c}_t$.** GRU keeps a single hidden state vector $\boldsymbol{h}_t$.
- **Two main gates**:
  - $\boldsymbol{z}_t$ (*update gate*): controls how much of the previous hidden state to retain.
  - $\boldsymbol{r}_t$ (*reset gate*): decides how strongly to forget the old hidden state.
- **Fewer parameters** than LSTM, potentially faster convergence.

GRU Architecture I

Gate Definitions

$$\boldsymbol{z}_t = \sigma\Big(\boldsymbol{W}_z\,\boldsymbol{x}_t + \boldsymbol{R}_z\,\boldsymbol{h}_{t-1} + \boldsymbol{b}_z\Big) \quad (\textit{update gate}),$$

$$\boldsymbol{r}_t = \sigma\Big(\boldsymbol{W}_r\,\boldsymbol{x}_t + \boldsymbol{R}_r\,\boldsymbol{h}_{t-1} + \boldsymbol{b}_r\Big) \quad (\textit{reset gate}).$$

$$\tilde{\boldsymbol{h}}_t = \tanh\Big(\boldsymbol{W}_h\,\boldsymbol{x}_t + \boldsymbol{R}_h\big(\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1}\big) + \boldsymbol{b}_h\Big),$$

$$\boldsymbol{h}_t = (1 - \boldsymbol{z}_t) \odot \tilde{\boldsymbol{h}}_t \;+\; \boldsymbol{z}_t \odot \boldsymbol{h}_{t-1}.$$

- $\boldsymbol{z}_t$ blends old vs. new information: when $\boldsymbol{z}_t \approx 1$, we preserve more of $\boldsymbol{h}_{t-1}$.
- $\boldsymbol{r}_t$ gates how much of $\boldsymbol{h}_{t-1}$ is used in creating $\tilde{\boldsymbol{h}}_t$.

Parameter Shapes

- $\boldsymbol{W}_* \in \mathbb{R}^{m\times d},\ \boldsymbol{R}_* \in \mathbb{R}^{m\times m},\ \boldsymbol{b}_* \in \mathbb{R}^m$.
- Each gate has its own $\boldsymbol{W}_*, \boldsymbol{R}_*, \boldsymbol{b}_*$, e.g. $\boldsymbol{W}_z, \boldsymbol{W}_r, \boldsymbol{W}_h, \boldsymbol{R}_z, \boldsymbol{R}_r, \boldsymbol{R}_h$.

Outline

- Introduction to Language Models
- Vocabulary and Tokenization
- Applications
- Parametrization and Estimation
- n-Gram Language Models
- Addressing Data Sparsity in n-Gram Models
- Evaluation Metrics for Language Models
- Toy Example
- Generation Strategies for Language Models
- Feed-Forward Neural Language Models
- Training
- Recurrent Neural Networks (RNNs)
- LSTMs and GRUs
- What's Next?

Next Lecture: Attention and Transformer models

Attention Mechanisms

- **Purpose**: Enable models to dynamically focus on relevant parts of the input.
- **Types of Attention**:
  - **Additive (Bahdanau) Attention**
  - **Multiplicative (Dot-Product) Attention**
  - **Scaled Dot-Product Attention**
- **Key Equation**:

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

- **Applications**: Machine translation, text summarization, question answering.

Transformer Architectures

- **Core Components**:
  - **Encoder-Decoder Structure**
  - **Multi-Head Self-Attention**
  - **Position-wise Feed-Forward Networks**
  - **Positional Encoding**
- **Multi-Head Attention**:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h) W^O$$

where $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$

- **Advantages**:
  - Parallelization over sequence length
  - Captures long-range dependencies effectively
  - Scalable to large datasets and models
- **Impact**: Foundation for state-of-the-art models like BERT, GPT, and more.