LLM Exam Notes - ETH Zurich

Francesco Bondi

July 2025

Part I

1 Key Definitions

Language Model (LM)

- **Informally**, a language model is a collection of conditional probability distributions:

$$p(y \mid \mathbf{y})$$

  where y is a symbol and $\mathbf{y}$ is a prefix string over an augmented alphabet $\bar{\Sigma} = \Sigma \cup \{\text{eos}\}$. However, such informal definitions may not define a proper distribution over Σ^* because probability mass can leak to infinite sequences.

- **Formally**, a language model is a (discrete) probability distribution over all finite strings:

$$p_{\text{LM}} : \Sigma^* \to [0, 1], \quad \text{such that} \quad \sum_{y \in \Sigma^*} p_{\text{LM}}(y) = 1$$

- Language models are *definitionally tight*—i.e., they assign full probability mass to Σ^*. Measure theory is used to rigorously define this on countable domains.

Sequence Model (SM)

- A sequence model is more general than a language model. It defines conditional probabilities:

$$p_{\text{SM}}(y \mid \mathbf{y}) \quad \text{for } y \in \bar{\Sigma},\ \mathbf{y} \in \Sigma^*$$

- Unlike LMs, sequence models may place probability mass on **infinite sequences**, and hence define a probability space over $\Sigma^* \cup \Sigma^\infty$.

Tightness

- A locally normalized model p_{LN} derived from p_{SM} is **tight** if:

$$\sum_{y \in \Sigma^*} p_{\text{LN}}(y) = 1$$

- For a finite string $\mathbf{y} = y_1 \dots y_T$, this corresponds to:

$$p_{\text{LN}}(\mathbf{y}) = p_{\text{SM}}(\text{eos} \mid \mathbf{y}) \cdot \prod_{t=1}^{T} p_{\text{SM}}(y_t \mid \mathbf{y}_{<t})$$

- If this total sum is less than 1, the model is **non-tight**, leaking mass to infinite continuations.
- A common sufficient condition for tightness is *guaranteed termination*, such as ensuring the `eos` symbol has non-zero probability and will eventually occur.

Locally Normalized Language Model (LNM)

- A locally normalized language model defines:

$$p_{\text{LN}}(\mathbf{y}) = p_{\text{SM}}(\text{eos} \mid \mathbf{y}) \cdot \prod_{t=1}^{T} p_{\text{SM}}(y_t \mid \mathbf{y}_{<t})$$

- LNMs avoid global normalization by locally normalizing at each time step.
- They require an explicit `eos` symbol to mark string termination.
- **Any language model can be locally normalized**, but not every LNM is tight. Thus, the term "non-tight language model," though contradictory, is often used to describe improper LNMs.

Globally Normalized Language Models (GNLM)

Definition

- A GNLM defines probabilities from an **energy function**:

$$\hat{p}_{\text{GN}}(\mathbf{y}) : \Sigma^* \to \mathbb{R}$$

- The probability of string $\mathbf{y}$ is defined as:

$$p_{\text{LM}}(\mathbf{y}) = \frac{1}{Z_G} \exp\left(-\hat{p}_{\text{GN}}(\mathbf{y})\right)$$

- The **normalization constant** is:

$$Z_G = \sum_{\mathbf{y}' \in \Sigma^*} \exp\left(-\hat{p}_{\text{GN}}(\mathbf{y}')\right)$$

Normalizability and Tightness

- If $Z_G < \infty$, the GNLM is valid and tight.
- This is a major advantage: if the energy function is normalizable, the resulting model is guaranteed to be tight.
- However, computing Z_G is often intractable, since it sums over an infinite set.

Example: A Non-Normalizable GNLM

Let the energy function be:

$$\hat{p}(\mathbf{y}) = \log\left(\frac{1}{|\mathbf{y}|}\right) \quad \Rightarrow \quad \exp(-\hat{p}(\mathbf{y})) = |\mathbf{y}|$$

Then:

$$Z_G = \sum_{\mathbf{y}\in\Sigma^*} |\mathbf{y}| = \sum_{k=0}^{\infty} k \cdot |\Sigma|^k$$

For example, if $|\Sigma| = 3$, then:

$$Z_G = \sum_{k=0}^{\infty} k \cdot 3^k$$

This diverges (as it's a weighted geometric series with ratio $r > 1$). Therefore, the energy function is **not normalizable**, and the model is **not a valid GNLM**.

2 Tightness in Language Models

The concept of **tightness** is fundamental in formally defining and characterizing language models, particularly to ensure that they constitute valid probability distributions over finite strings.

At its core, a **language model (LM)** is rigorously defined as a **(discrete) probability distribution** p_{LM} over Σ^* (the set of all finite strings over an alphabet Σ). This definition inherently means that the probabilities of all finite strings must **sum to 1**:

$$\sum_{y \in \Sigma^*} p_{LM}(y) = 1$$

Any model satisfying this criterion is said to be **tight**. Conversely, if

$$\sum_{y \in \Sigma^*} p_{LM}(y) < 1$$

then the model is considered **non-tight**, meaning that some probability mass leaks to infinite sequences. Such a model does not, by definition, qualify as a true language model, despite colloquial usage.

2.1 General Language Models and Normalization

Globally Normalized Language Models (GNLMs)

A **globally-normalized language model (GNLM)** defines the probability of a string $\mathbf{y}$ using an **energy function** $\hat{p}_{GN}(\mathbf{y})$ and a **normalization constant** Z_G:

$$p_{LM}(\mathbf{y}) = \frac{1}{Z_G} \exp(-\hat{p}_{GN}(\mathbf{y})), \quad \text{where} \quad Z_G = \sum_{\mathbf{y}' \in \Sigma^*} \exp(-\hat{p}_{GN}(\mathbf{y}'))$$

The critical condition for a GNLM to be well-defined is that its energy function must be **normalizable**, i.e., $Z_G < \infty$. A significant advantage of normalizable GNLMs is that they **always induce a valid language model**, which implies they are **tight**. However, computing Z_G over an infinite set of strings is often computationally intractable.

Locally Normalized Language Models (LNMs)

A **locally-normalized language model (LNM)** defines:

$$p_{LN}(\mathbf{y}) = p_{SM}(\text{eos} \mid \mathbf{y}) \prod_{t=1}^{T} p_{SM}(y_t \mid \mathbf{y}_{<t})$$

LNMs avoid global normalization by normalizing only over the finite alphabet at each step. However, this local normalization does not guarantee tightness. An LNM might place non-zero probability on infinitely long sequences and thus fail to sum to 1 over Σ^*. It is **tight** if:

$$\sum_{y \in \Sigma^*} p_{LN}(y) = 1$$

While *any language model can be locally normalized*, the converse is not always true.

2.2 Conditions for Tightness in Specific Language Models

1. Probabilistic Finite-State Automata (PFSAs)

PFSAs are special weighted finite-state automata where:

- Initial state weights sum to 1.
- For each state, the sum of transition weights and its final weight equals 1.

Tightness Condition: A PFSA is tight *if and only if all accessible states are co-accessible*, where:

- An *accessible state* is reachable with non-zero probability from an initial state.
- A *co-accessible state* has a non-zero-weighted path to a final state.

Remarks:

- Globally normalized WFSAs (e.g., $p(y) = A(y)/Z_A$) are always tight if $Z_A < \infty$.
- Tight PFSAs and normalizable WFSAs are **equally expressive**.

2. Recurrent Neural Networks (RNNs)

RNNs update a hidden state h_t over time and use softmax to predict the next token.

Tightness Conditions:

- **General case:** A softmax RNN is tight if

$$s\|h_t\|_2 \leq \log t$$

  for all t, where s is related to the spread of embeddings.

- **Bounded dynamics:** A softmax RNN with a *bounded dynamics map* (e.g., using tanh or sigmoid) is tight.
- **Unbounded activations:** RNNs using ReLU **may not be tight**. For instance, if $h_t = \binom{t}{t}$, then $\|h_t\|_2 \sim t$ and tightness fails.
- However, ReLU-based RNNs can still be tight with proper parameter settings ensuring high 'eos' probability.

Undecidability:

- For rational-weighted Elman RNNs with infinite precision, **tightness is undecidable**. This follows from a reduction from the Halting Problem.

3. Transformers

Transformers compute token representations using self-attention and feedforward layers.

Tightness Condition: A **fixed-depth transformer with soft attention is always tight**.

Why?

- Symbol + positional embeddings form a compact set.
- Transformer layers are composed of continuous functions, mapping compact sets to compact sets.
- The softmax over these bounded representations assigns non-zero eos probability uniformly bounded below by $\delta > 0$.
- Thus:
$$\sum_t p(\text{eos} \mid y_{<t}) \geq \sum_t \delta = \infty$$
satisfying tightness (cf. Proposition 2.5.6).

Hard attention transformers: These may not be tight, particularly when relying on unique hard attention, which may cause confidence degradation with length and practical modeling issues.

2.3 Is a Given Language Model Tight?

- **GNLMs:** Tight *iff* normalization constant $Z_G < \infty$.
- **Example:** Let $\hat{p}(y) = \log \frac{1}{|y|} \Rightarrow \exp(-\hat{p}(y)) = |y|$. Then:
$$Z_G = \sum_{k=0}^{\infty} k \cdot 3^k = \infty$$
so the model is **not tight**.
- **PFSAs:** Tight *iff* all accessible states are co-accessible (decidable).
- **RNNs:**
  - Tight if softmax + bounded activations.
  - Tightness for ReLU must be explicitly verified.
  - For general Elman RNNs, tightness is undecidable.
- **Transformers:**
  - Soft-attention models are always tight.
  - Hard-attention models may not be, depending on architecture and task.

Conclusion

Tightness is a core requirement for language models to define valid probability distributions over finite strings. Its guarantees and computational implications vary significantly across model types:

- GNLMs are tight if their energy function is normalizable.
- LNMs must be checked for tightness unless derived from bounded models.
- RNNs and Transformers offer contrasting challenges and guarantees based on their recurrence or attention mechanisms.

3 Prefix Probabilities

What is the prefix probability of a string?

The **prefix probability** of a string, denoted as $\varpi(y)$, is formally defined as the sum of the probabilities of all finite strings that begin with y [previous response, 45, 46]. In essence, it quantifies the cumulative probability that y will serve as a prefix for any complete finite string yy' within the language, where y' can be any string, including the empty string, from the Kleene closure of the alphabet (Σ^*) [previous response].

Formally, for a language model p_{LM}, the prefix probability is expressed as:

$$\varpi(\mathbf{y}) \stackrel{\text{def}}{=} \sum_{\mathbf{y}' \in \mathbf{\Sigma}^*} \mathbf{p_{LM}}(\mathbf{y}\mathbf{y}') \quad \text{[previous response, 46]}$$

A fundamental property of prefix probability is that the prefix probability of the empty string ($\emptyset$) is always 1, i.e., $\varpi(\emptyset) = 1$ [previous response, 46].

How can you compute the prefix probability of a string in a locally-normalized language model?

To compute the prefix probability of a string within a locally-normalized language model (LNM), it's crucial to understand the concept of "tightness" in such models.

1. **Understanding Tightness in Locally-Normalized Language Models:** A locally-normalized language model (LNM) defines the probability of an entire string y using an autoregressive factorization based on conditional probabilities [1, 2]. This is typically expressed as:

$$p_{LN}(y) \stackrel{\text{def}}{=} p_{SM}(\text{eos}|y) \cdot \prod_{t=1}^{T} p_{SM}(y_t|y_{<t}) \quad \text{[1, 2]}$$

   Here, $p_{SM}(y_t|y_{<t})$ represents the conditional probability of the symbol y_t given the preceding prefix $y_{<t}$, and $p_{SM}(\text{eos}|y)$ is the probability that the string y is followed by an "end-of-sequence" symbol, effectively marking y as a complete string [1-3]. While individual conditional distributions $p_{SM}(\cdot|\text{context})$ for the next symbol (including the 'eos' symbol) always sum to one over the augmented alphabet $\Sigma \cup \{\text{eos}\}$ [3, 4], the sum of probabilities for all **finite strings** y over the Kleene closure Σ^* may **not** necessarily sum to 1 [1, 5].

   - If $\sum_{y \in \Sigma^*} p_{LN}(y) = 1$, the LNM is considered **tight** [2, 4, 6].
   - If $\sum_{y \in \Sigma^*} p_{LN}(y) < 1$, the LNM is **non-tight** [4, 6, 7]. This "leaks" probability mass to infinite sequences, meaning it does not represent a valid probability distribution over finite strings alone [1, 3, 5, 7-9]. The term "language model" without qualification typically refers only to tight language models; a non-tight language model is, by definition, not a language model in the strict sense [6].

2. **Computing Prefix Probability for a Tight Locally-Normalized Language Model:** For a tight LNM, the prefix probability $\varpi(y)$ can be computed using a recursive relationship [previous response, 49]. This formula expresses that the probability of y being a prefix is the sum of two components:

- The probability that y itself is a complete string (i.e., it is followed by the end-of-sequence symbol 'eos').
- The sum of probabilities that y continues with any symbol $a \in \Sigma$, multiplied by the prefix probability of the resulting longer string ya.

This recursive formula is given by:

$$\varpi(\mathbf{y}) = \sum_{\mathbf{a} \in \mathbf{\Sigma}} \mathbf{p_{SM}}(\mathbf{a}|\mathbf{y}) \cdot \varpi(\mathbf{ya}) + \mathbf{p_{SM}}(\text{eos}|\mathbf{y}) \quad \text{[previous response, 49]}$$

This method of computation is directly analogous to calculating **state-specific allsums** (also known as backward values) in Probabilistic Finite-State Automata (PFSAs) [previous response, 83]. For a PFSA A and a state q, the state-specific allsum $Z(A, q)$ is equivalent to $\varpi(y)$ if state q corresponds to prefix y [previous response, 83]. Lemma 4.1.2 provides the recursive formula for $Z(A, q)$ [10]:

$$\mathbf{Z}(\mathbf{A}, \mathbf{q}) = \sum_{\mathbf{q} \xrightarrow{\mathbf{a}/\mathbf{w}} \mathbf{q}'} \mathbf{w} \cdot \mathbf{Z}(\mathbf{A}, \mathbf{q}') + \rho(\mathbf{q}) \quad \text{[10]}$$

Here, w are transition weights (analogous to $p_{SM}(a|y)$), q' is the target state (analogous to ya), and $\rho(q)$ is the final weight of state q (analogous to $p_{SM}(\text{eos}|y)$) [10]. This approach is applicable to tight PFSAs, which are equally expressive as normalizable Weighted Finite-State Automata and induce globally normalized (and thus tight) language models [10].

3. **Computing Prefix Probability for a Non-Tight Locally-Normalized Language Model (Sequence Model):** If an LNM is non-tight, a positive probability mass is assigned to infinite sequences [7, 9]. In such cases, the definition of $\varpi(y)$ as the sum of probabilities *strictly for finite strings* starting with y becomes problematic. This is because such a sum would not account for the probability mass "lost" to infinite continuations. In a broader measure-theoretic framework, such models are referred to as **sequence models** (SMs) [11-14], which define a probability space over the set of both finite (Σ^*) and infinite (Σ^∞) sequences [13-15]. A true language model is then redefined as a sequence model where the probability of infinite sequences is zero, i.e., $P(\Sigma^\infty) = 0$ [14]. The conditional probabilities $p_{SM}(y|y)$ are still well-defined in non-tight sequence models [4], but the sum of probabilities over Σ^* does not equal 1 [4].

4 Representation-based Language Models

The Locally-Normalized Representation-Based Language Modeling Framework

Most modern language models are defined as **locally normalized models (LNMs)** [1]. In this framework, a sequence model $p_{SM}(y|y)$ is first defined, which models the conditional probability of the next possible symbol y given the context (history) y [1, 2]. The full probability of a string y in an LNM is then computed by multiplying these conditional probabilities, along with an end-of-sequence (eos) probability [3, 4]:

$$p_{LN}(y) \stackrel{\text{def}}{=} p_{SM}(\text{eos}|y) \cdot \prod_{t=1}^{T} p_{SM}(y_t|y_{<t}) \quad [3, 4]$$

where $y_{<t}$ is the prefix up to y_{t-1}, and y_0 is often denoted as a beginning-of-sequence (`bos`) symbol [4, 5].

The core idea of representation-based language modeling is to define $p_{SM}(y|y)$ in terms of the **similarity between learned numerical representations** of the symbols y and the context y [6]. The more compatible a symbol is with its context, the more probable it should be [6].

- **Vector Space Representations**: To quantify similarity, individual symbols and contexts are embedded as vectors in a **Hilbert space** [7, 8]. A Hilbert space is a complete inner product space [9]. This space allows for geometric notions of similarity, such as the angle between vectors [10].
  - A **representation function** $f : S \to V$ maps elements from a set S to vectors in a Hilbert space V [11].
  - For individual symbols $y \in \Sigma$, an **embedding function** $e(\cdot) : \Sigma \to \mathbb{R}^D$ (or symbol embedding function [12]) maps symbols to D-dimensional vectors, often implemented as a lookup into an embedding matrix E [12, 13]. These are sometimes called static symbol embeddings [14].
  - For contexts (strings) $y \in \Sigma^*$, a **context encoding function** $enc(\cdot) : \Sigma^* \to \mathbb{R}^D$ (or encoding function [15]) maps strings to D-dimensional vectors [15].
- **Compatibility and Logits**: The similarity between a symbol y and its context y is often measured using the **inner product** $e(y)^T enc(y)$ [16]. These similarity values, computed for all possible next symbols, form a vector whose entries are called **scores** or **logits** [16].
- **Projection onto the Simplex**: Since logits can be negative and do not necessarily sum to one, a **projection function** $f_{\Delta^{|\Sigma|-1}} : \mathbb{R}^D \to \Delta^{|\Sigma|-1}$ is used to transform them into a valid discrete probability distribution [16, 17]. The **probability simplex** Δ^{D-1} is the set of non-negative vectors in $\mathbb{R}^D$ whose components sum to 1 [18]. The most common choice for this projection function is the **softmax function** [19].

Combining these components, a representation-based locally normalized model defines the conditional probabilities as [19]:

$$p_{SM}(y_t|y_{<t}) \stackrel{\text{def}}{=} f_{\Delta^{|\Sigma|-1}}(E \cdot enc(y_{<t}))_{y_t} \quad [19]$$

where E is the symbol representation matrix [20] and the projection function is typically softmax [19]. The design choices for $e(y)$ and $enc(y)$ are crucial as they carry all the necessary information for determining next-symbol probabilities [4].

Tightness of Locally-Normalized Representation-based Models

A locally-normalized language model is considered **tight** if the sum of probabilities of all finite strings equals 1 [3, 21]. If this sum is less than 1, the model is non-tight and "leaks" probability mass to infinite sequences [3, 22]. A non-tight language model is not a language model in the strict sense [23].

For softmax-based representation models, a general result states that the model is tight if the maximum attainable norm of the context representation, $\max_{y \in \Sigma^t} \|enc(y)\|_2$, grows slower than $\log t$ for sufficiently large t [24, 25]. A simpler, sufficient condition for tightness is that the context encoding function $enc(y)$ is **uniformly bounded** [134, Corollary 5.1.1]. This is often the case in neural networks due to the choice of activation functions [26].

Training of Language Models and Their Optimization

The goal of the language modeling task is to **estimate the parameters of a model** p_M (a parameterized model p_ϵ) to approximate a ground-truth probability distribution p_{LM} based on a dataset D of text [27-29]. This is typically framed as an optimization problem [27].

- **Data**: A corpus $D = \{y^{(n)}\}_{n=1}^{N} \subset \Sigma^*$ is a collection of N strings [30]. A common assumption is that these strings are generated **independently and identically distributed (i.i.d.)** by p_{LM} [31].
- **Language Modeling Objectives**:
  - **Maximum Likelihood Estimation (MLE)**: This principle dictates that the optimal parameters are those that **maximize the likelihood** of observing the given data under the model [20]:

    $$\epsilon_{MLE} \stackrel{\text{def}}{=} \operatorname*{argmax}_{\epsilon \in \Omega} \mathcal{L}(\epsilon) \quad [20]$$

    In practice, the **log-likelihood** $\log \mathcal{L}(\epsilon)$ is maximized for numerical stability and convexity [20].
  - **Equivalence to Cross-Entropy**: Maximizing log-likelihood is equivalent to **minimizing the cross-entropy** $H(\tilde{p}_{LM}, p_\epsilon)$ between the empirical data distribution $\tilde{p}_{LM}$ and the model distribution p_ϵ [32, 33]. A consequence is that the model must assign positive probability mass to all samples observed in the training data, often leading to "mean-seeking behavior" where even "gibberish" sequences are assigned non-zero probability [34].
  - **Teacher Forcing**: During training, especially with cross-entropy loss, the model's predictions for the next symbol are conditioned on the **ground-truth prior context** from the data, even if the model made an incorrect prediction at an earlier step [35, 36]. This can lead to **exposure bias** during generation, where errors compound because the model is not exposed to its own generated outputs during training [36].

  - **Alternative Objectives**:
    * **Masked Language Modeling (MLM)**: Unlike standard LMs that predict the next symbol given prior context, MLM optimizes for per-symbol log-likelihood using **both preceding and succeeding context** [37, 38]. Models like BERT use MLM [39]. Although widely used, MLM does **not define a valid language model** in the strict sense (i.e., a probability distribution over Σ^*) [39].
    * **Other Divergence Measures**: Different divergence measures (e.g., DKL, power divergences) can be used, exhibiting different properties regarding how probability mass is spread (e.g., mean-seeking vs. mode-seeking behavior) [14].
    * **Auxiliary Prediction Tasks**: Joint optimization with additional tasks (e.g., next sentence prediction) can be used, though their formal relationship to language modeling and impact on model validity is unclear [40].

- **Optimization**:
  - **Data Splitting**: To ensure models generalize well to unseen data, data is split into **training set** (D_{train}), **test set** (D_{test}), and often a **validation set** (D_{val}) [41].
  - **Numerical Optimization**: Parameters are found iteratively using algorithms like **gradient descent** [42, 43]. These are typically **gradient-based**, using the gradient of the objective function with respect to current parameters to determine the update direction [43]. **Backpropagation** efficiently computes these gradients [44]. **Mini-batch gradient descent** uses small, randomly selected subsets of data for updates [45].
  - **Parameter Initialization**: The starting point ϵ_0 in the parameter space is critical, as it can significantly impact training dynamics and final model performance [46, 47].
  - **Regularization Techniques**: Modifications to learning algorithms intended to **increase generalization performance** and prevent **overfitting** [48, 49].
    * **Weight Decay** (ℓ_2 regularization): Adds a penalty to the loss for the ℓ_2 norm of parameters, discouraging high parameter values and promoting simpler models [50].
    * **Entropy Regularization**: Penalizes models for outputting low-entropy (peaky) distributions, encouraging more spread-out probabilities (e.g., label smoothing, confidence penalty) [51-53].
    * **Dropout**: Randomly "drops" (zeros out) variables during computation to prevent over-reliance on any single variable and improve robustness [53, 54].
    * **Batch and Layer Normalization**: Rescale variables within the network for training stability and better generalization [55]. Batch normalization normalizes across data points, while layer normalization normalizes across features [55].

Definition and Properties of the Softmax Function

The **softmax function** is a crucial projection function used in representation-based language models to transform real-valued "scores" (logits) into a valid discrete probability distribution over symbols [16, 17, 56, 57]. Its origin dates back to the Boltzmann distribution in statistical mechanics [58].

Definition of Softmax

Given a vector of scores $x = [x_1, \ldots, x_D]^T \in \mathbb{R}^D$, the softmax function with a temperature parameter $\phi > 0$ is defined for each component d as [58]:

$$\text{softmax}(x)_d = \frac{\exp(x_d/\phi)}{\sum_{k=1}^{D} \exp(x_k/\phi)} \quad [58]$$

The output of the softmax is a vector in the probability simplex Δ^{D-1}, meaning its components are non-negative and sum to 1 [18]. The temperature parameter ϕ (often set to 1) influences the "peakiness" of the distribution [58, 59].

Properties of Softmax

The softmax function possesses several desirable properties for its use in machine learning:

- **Limiting Behavior** (Theorem 3.1.2) [60, 61]:
  - As $\phi \to \infty$, the output approaches a **uniform distribution** over all D elements:

$$\lim_{\phi \to \infty} \text{softmax}(x) = \frac{1}{D}\mathbf{1} \quad [60]$$

  - As $\phi \to 0^+$, the output approaches a **one-hot vector** where all probability mass is placed on the element(s) with the maximum score (argmax operation, with ties broken deterministically, e.g., by choosing the lowest index) [59, 60]:

$$\lim_{\phi \to 0^+} \text{softmax}(x) = e_{\text{argmax}(x)} \quad [60]$$

    This is why it is sometimes informally called "softargmax" [59].
- **Variational Characterization** (Theorem 3.1.3) [62]: The softmax function can be viewed as the solution to an optimization problem: it is the probability distribution $p \in \Delta^{D-1}$ that **maximizes its similarity with the input scores** x (via $x^T p$) while being **regularized to produce a solution with high entropy** ($-\phi \sum p_d \log p_d$) [62, 63]:

$$\text{softmax}(x) = \underset{p \in \Delta^{D-1}}{\text{argmax}} \left(x^T p - \phi \sum_{d=1}^{D} p_d \log p_d \right) \quad [62]$$

  This implies that softmax generally leads to **non-sparse solutions**, meaning an output probability $\text{softmax}(x)_d$ can only be exactly 0 if the corresponding input score x_d is $-\infty$ [63].
- **Invariance to Constant Addition** (Theorem 3.1.4) [61]: Adding the same constant c to all input scores x does not change the softmax output:

$$\text{softmax}(x + c\mathbf{1}) = \text{softmax}(x) \quad [61, 64]$$

- **Differentiability** (Theorem 3.1.4) [61]: The softmax function is **continuous and differentiable everywhere**, and its derivative can be explicitly computed [61, 64, 65]. This property is crucial for gradient-based optimization in neural networks [61].

– **Rank Preservation** (Theorem 3.1.4) [64]: If an input score x_i is greater than or equal to x_j, then the corresponding softmax output $\text{softmax}(x)_i$ will also be greater than or equal to $\text{softmax}(x)_j$:

$$\text{If } x_i \geq x_j, \text{ then } \text{softmax}(x)_i \geq \text{softmax}(x)_j \quad [64]$$

While other projection functions exist (e.g., **sparsemax**, which can produce sparse distributions) [66], softmax remains the predominant choice due to its closed-form solution and these desirable properties, some of which are not met by alternatives (e.g., sparsemax is not everywhere differentiable) [58, 67].

5 Finite-State Language Models

Definitions and Computational Expressivity

A **finite-state language model (FSLM)** is formally defined as a language model that can be **represented by a Weighted Finite-State Automaton (WFSA)** [220]. This means that for a language model to be finite-state, there must exist a WFSA whose weighted language is identical to the language of the FSLM [220]. Informally, an FSLM is characterized by defining **only finitely many unique conditional distributions** $p_{LM}(y|y)$ [199]. This implies that there are only a finite number of contexts y that define the distribution over the next symbol [199]. WFSAs are characterized by a **finite set of states** Q [201, 208].

– **Computational Expressivity of FSLMs:**
  * **Regular Languages:** The set of languages that finite-state automata (FSA) can recognize is known as the class of **regular languages** [205, 206]. A language L is regular if and only if it can be recognized by an unweighted finite-state automaton [205]. This concept extends to **weighted regular languages**, which are defined by WFSAs [213].
  * **n-gram Models:** Finite-state language models are a **natural generalization of the well-known n-gram models** [197]. Every n-gram language model is, in fact, a WFSA (specifically, a probabilistic or substochastic one) [246]. This is because the **n-gram assumption** (where the probability of a symbol depends only on the previous $n-1$ symbols) implies a finite number of histories that need to be modeled, corresponding to the states of the automaton [247]. These models fall into the class of **strictly local languages** [558].
  * **Limitations for Human Language:** Despite their utility, FSLMs are **not sufficient for modeling human language** [199]. Human language contains structures, such as **arbitrarily deep recursive structures** (e.g., center embeddings like "The cat the dog barked at likes to cuddle") [108, 268, 269], that cannot be captured by a finite set of possible histories or states [269]. This unbounded increase in information that the model needs to "remember" to process matching terms correctly exceeds the finite memory of FSAs [269]. Phenomena like **cross-serial dependencies** found in languages like Swiss German also demonstrate that human language is more expressive than context-free grammars, and thus, by extension, finite-state models cannot describe them [138, 345, 348].
  * **Theoretical Utility:** Nevertheless, FSLMs are valuable as a **theoretical tool for understanding modern neural language models** [199]. Recurrent Neural Networks (RNNs) in a practical setting (fixed-point arithmetic, real-time operation) are, in fact, **equivalent to weighted finite-state automata with Heaviside activation functions** [384, 387, 459]. Even other RNN variants like GRUs are considered "rationally recurrent" and at most regular in some aspects [482].

Probabilities of Strings Under a Finite-State Language Model

There are two primary ways to define string probabilities under finite-state language models:

– **In a Probabilistic Finite-State Automaton (PFSA):**

    - A **PFSA** (Definition 4.1.17) is a specific type of WFSA where the initial weights of all states form a probability distribution (sum to 1), and for any state, the weights of its outgoing transitions (with any label) together with its final weight also form a valid discrete probability distribution (sum to 1) [217, 218]. Initial weights $\varrho(q)$, transition weights w, and final weights $\rho(q)$ must all be non-negative [217].
    - The **weight of a path** ϖ in a PFSA is considered the **probability of that path** [221].
    - The **probability of a string** y **(stringsum** $G(y)$**)** is defined as the **sum of the probabilities of all individual paths that recognize** y [222, 299].
    - These definitions **do not require any explicit normalization** over all possible paths or strings [222, 300]. This closely resembles how **locally normalized models (LNM)** are defined based on conditional probabilities [47, 299, 300]. The final weights $\rho(q)$ in a PFSA play an analogous role to the eos symbol, representing the probability of ending a string in that state [219].
- **In a General Weighted Finite-State Automaton (WFSA):**
    - For a general WFSA, string probabilities are defined using the concepts of **stringsum** $A(y)$ (also called string weight or acceptance weight) and **allsum** $Z(A)$ [212, 214].
    - The **stringsum** $A(y)$ of a string y under a WFSA A is the sum of the weights of all paths in A that yield y [212].
    - The **allsum** $Z(A)$ of a WFSA A is the total weight assigned to all possible strings, which is equivalent to the sum over the weights of all possible paths in the automaton [214].
    - A WFSA A is **normalizable** if its allsum $Z(A)$ is finite [215].
    - The **probability of a string** y under a normalizable WFSA A with non-negative weights is defined as $p_A(y) = A(y)/Z(A)$ [222]. These models are **globally normalized** and thus inherently **tight** by definition [209, 235, 301, 302].
    - **Computing the Allsum:** The allsum $Z(A)$ for a WFSA can be computed using matrix inversion, specifically as $(I - T)^{-1}$ where T is the full transition matrix of the automaton [224]. The runtime for this computation is cubic in the number of states ($O(|Q|^3)$) [225]. For acyclic WFSAs, the allsum can be computed in time linear in the number of transitions [225]. These algorithms are differentiable, enabling gradient-based training [225].
    - **Equivalence of PFSAs and WFSAs: Normalizable WFSAs with non-negative weights and tight PFSAs are equally expressive** [226, Theorem 4.1.1]. This means that any normalizable globally normalized FSLM can be locally normalized, and the locally normalized version will also be a finite-state model [231]. The transformation involves reweighting initial, final, and transition weights based on state-specific allsums [228].

Is a Given PFSA Tight?

Tightness in locally normalized language models (LNM) refers to whether the sum of probabilities of all finite strings in the language equals 1 [58, Definition 2.5.1]. If this sum is less than 1, the model is **non-tight**, meaning it "leaks" probability mass to infinite sequences [8, 35, 57].

For a **Probabilistic Finite-State Automaton (PFSA)**, tightness can be easily characterized:

- **A PFSA is tight if and only if all accessible states are also co-accessible** [236, Theorem 4.1.2].
  - An **accessible state** q is one for which there is a non-zero-weighted path to q from some initial state [84, 236].
  - A **co-accessible state** is one from which there is a non-zero-weighted path from q to some final state [84, 236].
  - The intuition for this condition is that if a WFSA is tight and an accessible state cannot reach a final state, the model will not be able to terminate after reaching that state, leading to non-tightness [236]. Conversely, if all accessible states are co-accessible, the Markov process represented by the PFSA is absorbed by the end-of-sequence (eos) or final state with probability 1, ensuring tightness [236]. The final weights $\rho(q)$ in a PFSA play an analogous role to the eos symbol, representing the probability of ending a string in that state [219].
- **Substochastic WFSAs:** A WFSA is **substochastic** if its initial and transition weights are non-negative, and for any state, the sum of weights of its outgoing transitions and its final weight is less than or equal to 1 [231, Definition 4.1.23]. For a trimmed substochastic WFSA, the sum of probabilities of finite strings is less than or equal to 1 [232, Theorem 4.1.3]. If a WFSA is trimmed (meaning useless states are removed [216]), and it's a probabilistic matrix (row sums to 1), then all its states must be co-accessible to lead to a final state, ensuring tightness [232, Theorem 4.1.3].
- **General Condition for LNMs:** A locally normalized sequence model (LNM) is tight if and only if the sum of probabilities of never-terminating sequences is zero. This is formally characterized by Theorem 2.5.3: an LNM is tight if and only if $\tilde{p}_{\text{eos}}(t) = 1$ for some t or $\sum_{t=1}^{\infty} \tilde{p}_{\text{eos}}(t) = \infty$ [105, Theorem 2.5.3], where $\tilde{p}_{\text{eos}}(t)$ is the probability of terminating at step t given no earlier termination [104]. Proposition 2.5.6 provides a useful sufficient condition: if $p_{LN}(\text{eos}|y) \geq f(t)$ for all $y \in \Sigma^t$ and for all t, and $\sum_{t=1}^{\infty} f(t) = \infty$, then the LNM is tight [95, Proposition 2.5.6].

Is a Given Language Finite-State?

A language is considered **finite-state** if it can be **recognized by an unweighted finite-state automaton** [184, 205, 206, Definition 4.1.5]. This means that if you can construct an FSA that accepts precisely the strings of that language, then the language is finite-state (or regular) [78].

- **Weighted Case:** A weighted language L is a **weighted regular language** if there exists a WFSA A such that $L = L(A)$ [197, 213, Definition 4.1.12].
- **Examples of Finite-State Languages:**
  - As discussed, **n-gram language models are indeed finite-state** [246, 247]. They model languages where the context for predicting the next symbol is finite (the previous $n-1$ symbols), which naturally maps to the finite states of an automaton [247]. These are also known as **strictly local languages** [558].
  - The language "First" defined as $L = \{y \in \Sigma^* | \Sigma = \{0, 1\}, y_1 = 1\}$ is a simple regular language, recognizable by an FSA [549, 557].

    - The language "Parity" defined as $L = \{y \in \Sigma^* | \Sigma = \{0, 1\}, y \text{ has odd number of } 1s\}$ is also a regular language [557].
- **Examples of Non-Finite-State Languages:** To determine if a language is finite-state, one often considers whether it requires **unbounded memory or recursive structures** to be recognized.
    - The language $L = \{a^n b^n | n \in \mathbb{N}_{\geq 0}\}$ is **not regular (finite-state)** [284]. This is because it requires remembering an unbounded number of 'a's to ensure an equal number of 'b's, which a finite-state automaton cannot do [284]. This language is, however, context-free [284].
    - Human languages generally are **not strictly finite-state** [199, 267]. Phenomena like **center embeddings** (e.g., "The cat the dog the mouse startled barked at likes to cuddle") involve arbitrarily deep recursion, requiring unbounded memory that finite-state automata lack [108, 268, 269].
    - **Cross-serial dependencies** (e.g., in Swiss German) are another example of linguistic phenomena that cannot be described by finite-state models [138, 345, 348]. These require formalisms beyond context-free grammars and pushdown automata [348, 349].
    - The language $L = \{a^n b^n c^n | n \in \mathbb{N}_{\geq 0}\}$ is not context-free, but context-sensitive [483, 484].
    - A language where $p_{LM}(a^n)$ follows a Poisson distribution (e.g., $p_{LM}(a^n) = e^{-\varrho} \frac{\varrho^n}{n!}$) is not context-free [341].
- **Distinguishing Finite-State Languages:** To formally prove a language is not regular (finite-state), one can use the **pumping lemma for regular languages** [284, 322].
- **Neural Models and Finite-State Languages:** While seemingly powerful, some transformer models, especially those using "unique hard attention," initially struggled to recognize simple regular languages like "Parity" or "First" [551, 556, 557]. This suggests that specific architectural choices within a model can impact its ability to recognize even simple finite-state languages. However, transformers can simulate n-gram models by using multiple attention heads to attend to specific previous positions in the sequence [535, 558, 560].

6 N-gram Language Models

N-gram language models are a classical framework in language modeling that impose a specific assumption on how the probability of a symbol depends on its preceding context.

Definitions

- **Informal Definition**: An n-gram language model (LM) determines the probability of a symbol (or word/token) based solely on the **preceding $n-1$ symbols**. This means that the influence of earlier parts of the sequence is limited to a fixed window.
- **Formal Definition (n-gram assumption)**: Given an alphabet Σ and a sequence of symbols $y = y_1 \dots y_T$, the conditional probability of the symbol y_t given its history $y_{<t}$ (all preceding symbols $y_1 \dots y_{t-1}$) is simplified to depend only on the $n-1$ previous symbols. This is formally expressed as:

$$p_{SM}(y_t|y_{<t}) = p_{SM}(y_t|y_{t-1}, \dots, y_{t-n+1})$$

  .
- **History or Context**: The sequence $y_{t-1}, \dots, y_{t-n+1}$ is referred to as the **history** or **context** for y_t .
- **Markov Assumption**: The n-gram assumption is synonymous with the $(n-1)^{th}$-order Markov assumption in the context of language modeling.
- **Edge Cases and Padding**: For sequences where the length of the history is less than $n-1$ (i.e., at the beginning of a string where $t < n$), the sequences are typically **padded with "beginning-of-sequence" (bos) symbols** to ensure a consistent context length of $n-1$.
- **Examples of N-grams**:
  - **Bigram Model (n=2)**: The probability of the next word depends only on the immediately preceding word. For example, $p_{SM}(y_t|y_{!t}) = p_{SM}(y_t|y_{t-1})$.
  - **Unigram Model (n=1)**: The probability of a word is independent of any preceding words.
- **Connection to Finite-State Automata (FSAs)**: Every n-gram language model can be represented as a **probabilistic finite-state automaton (PFSA)** or a substochastic one. In this representation, the states of the automaton correspond to all possible sequences of $n-1$ symbols (histories), and the transitions between states correspond to the conditional probabilities of observing the next symbol given that history. The final weights of the states in the WFSA play a role analogous to the 'eos' (end-of-sequence) symbol.
- **Subregular Languages**: N-gram models belong to a class of subregular languages known as **strictly local languages (SLn)**. This is because their patterns depend solely on blocks of consecutively occurring symbols, where each block is considered independently.

Parameter Estimation

The goal of language modeling is to estimate the parameters of a model p_M that approximates the ground-truth distribution p_{LM} based on observed data D.

- **Maximum Likelihood Estimation (MLE)**:
  - For a simple n-gram model, the parameters ϵ are the conditional probability distributions $\zeta_{y|y} = p_{SM}(y|y)$ for any context y of length $n-1$.
  - The optimal MLE solution for the conditional probabilities is given by the **relative frequencies** (count-based statistics) from the training data:

$$p_{SM}(y_n|y_{<n}) = \frac{C(y_1, \ldots, y_n)}{C(y_1, \ldots, y_{n-1})}$$

    where $C(y_1, \ldots, y_n)$ denotes the number of occurrences of the string $y_1, \ldots, y_n$ in the training corpus.
  - **Equivalence with Cross-Entropy**: Maximizing the log-likelihood of the data is equivalent to minimizing the cross-entropy loss between the empirical data distribution and the model's distribution.
- **Drawbacks of Simple MLE**:
  - **Zero Probabilities**: A significant problem with simple count-based MLE is that any n-gram not observed in the training data will be assigned a probability of zero. This leads to assigning zero probability to entire sentences if they contain an unobserved n-gram, which is often undesirable.
  - **Lack of Generalization**: Count-based n-gram models treat words as distinct, independent symbols in a lookup table, meaning they cannot inherently generalize semantic similarities between words. For example, if "dog," "puppy," and "kitten" are seen in similar contexts but "cat" is not, the model would assign a zero (or default, if smoothed) probability to "cat" in that context, even though it's semantically similar to the observed words.
- **Smoothing and Backoff Techniques**: To address the zero-probability problem, practical n-gram models often employ techniques like **smoothing** and **backoff** (which are beyond the scope of the provided notes).

Parameter-Sharing

To overcome the limitations of simple count-based n-gram models, particularly their inability to generalize based on semantic similarity and their exponential parameter growth, the concept of parameter-sharing through distributed representations was introduced.

- **Motivation for Parameter-Sharing**: The main motivation is to enable the model to account for relationships and similarities between words and generalize across different surface forms.
- **Distributed Word Representations (Embeddings)**: Instead of treating each word as a unique entry in a lookup table, words are associated with **vector representations** (embeddings), ‘e(y)‘. These embeddings are themselves parameters of the model and can be learned from the training data alongside the language modeling objective.

- **Neural N-gram Models**:
  - One of the first successful applications of this approach was by Bengio et al. (2003b). In this model, the context-encoding function 'enc' is implemented by a **neural network** that processes the previous $n-1$ word embeddings:

$$\text{enc}(y_{<t}) = \text{enc}(y_{t-1}, y_{t-2}, \ldots, y_{t-n+1})$$

    .
  - The conditional probability of the next symbol is then computed using a softmax function over the product of the output embedding matrix 'E' and the context encoding:

$$p_{SM}(y_t|y_{<t}) = \text{softmax}(E\,\text{enc}(y_{t-1}, \ldots, y_{t-n+1})^J + b)_{y_t}$$

    .
  - The specific form of the 'enc' function in Bengio et al. (2003b) is a feed-forward neural network:

$$\text{enc}(y_t, y_{t-1}, \ldots, y_{t-n+1}) = b + Wx + U\tanh(d + Hx)$$

    where 'x' is the concatenation of the context symbol embeddings.
- **Parameter Reduction**: This representation-based approach significantly reduces the number of parameters. While a lookup-table-based n-gram model requires $O(|\Sigma|^n)$ parameters, a representation-based n-gram model scales **linearly with** n, as it only requires adding additional rows to the parameter matrices (e.g., 'W', 'U', 'H').
- **Limitations**: Despite incorporating neural networks and parameter sharing, these models are still fundamentally n-gram models because they rely on a fixed, finite context length of n tokens. This means they cannot model dependencies that span beyond n words.
- **WFSA Representation**: These neural n-gram models can still be represented by a Weighted Finite-State Automaton (WFSA), where the weights on the transitions are parametrized by the neural network. This combines the theoretical understanding of formal language theory with the flexibility of neural networks.

7 Recurrent Neural Networks (RNNs)

Recurrent neural networks (RNNs) are a class of neural network models designed to process sequential data. They naturally model sequential input by evolving a hidden state that captures the context of previously seen symbols.

7.1 Elman RNNs

Elman RNNs, also called vanilla RNNs, are the simplest parameterization of recurrent dynamics.

- **Formal Definition:** An Elman sequence model

$$R = (\Sigma, D, U, V, E, b_h, h_0)$$

  is a D-dimensional RNN over an alphabet Σ, with dynamics:

$$h_t = \omega(Uh_{t-1} + Ve_1(y_t) + b_h)$$

  where:
  - $e_1(\cdot) : \Sigma \to \mathbb{R}^R$ is the embedding function.
  - ω is an element-wise activation function.
  - $b_h \in \mathbb{R}^D$ is the bias vector.
  - $U \in \mathbb{R}^{D \times D}$ is the recurrence matrix.
  - $V \in \mathbb{R}^{D \times R}$ is the input matrix.
  - $h_0 \in \mathbb{R}^D$ is the initial state.
- **Functionality:** Computes h_t as a non-linear function of h_{t-1} and the current input y_t.
- **Hidden State:** $h_t = \text{enc}_R(y_{1:t})$ acts as a compact summary of the sequence seen so far.
- **Embedding Tying:** The input embedding e_1 can share parameters with the output embedding matrix E.
- **Tightness:** Elman RNNs with bounded activation functions (e.g., tanh, sigmoid) and softmax projection are tight. If $s\|h_t\|_2 \leq \log t$ holds for all t, the RNN is tight.
- **Limitations:** Prone to vanishing/exploding gradients, hindering long-range dependency modeling. Motivated the creation of LSTMs and GRUs.

7.2 Heaviside Elman RNNs (HRNNs)

HRNNs use the Heaviside function as their activation.

- **Heaviside Function:**

$$H(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

- **Parameters:** Binary hidden activations; real or rational weights, possibly ∞ or $-\infty$.

- **Equivalence to dPFSAs:**
  - HRNNs are equivalent to deterministic probabilistic FSAs (dPFSAs), representing regular languages.
  - **HRNN ⇒ dPFSA:** Map HRNN hidden states (binary vectors) to PFSA states. Each transition in the PFSA corresponds to an HRNN update:

$$h_{t+1} = H(Uh_t + Ve(y_{t+1}) + b_h)$$

    where U, V, and b_h encode conjunctions for symbol/state transitions.
  - **dPFSA ⇒ HRNN (Minsky's Construction):**
    1. HRNN hidden state encodes (q_t, y_t) using one-hot encoding.
    2. Update rule simulates PFSA transition via Heaviside activation.
    3. Softmax (or sparsemax) output projection from Eh_t replicates PFSA's transition probabilities.
  - **Implications:**
    - HRNNs cannot represent non-deterministic PFSA behavior.
    - All RNNs implemented with finite precision are finite-state machines in practice.

7.3 Time Complexity in RNNs

- **Sequential Processing:** Computation of h_t requires $h_1, \ldots, h_{t-1}$. Inherently sequential, limiting parallelism and increasing training time.
- **Transformer Comparison:** Transformers compute all token encodings in parallel, giving them a training-time advantage over RNNs.
- **Space Complexity (Simulating FSAs):**
  - Minsky's Construction: Hidden state size $O(|Q||\Sigma|)$.
  - Dewdney: $O(|\Sigma||Q|^{3/4})$ via two-hot encodings.
  - Indyk: $O(|\Sigma|\sqrt{|Q|})$ for binary alphabets.
  - **Probabilistic Setting:** These optimizations do not generalize. HRNNs must still use $O(|Q|)$ dimensions to distinguish state-specific output distributions.

7.4 Constructing RNNs for Formal Languages

- **Regular Languages:** HRNNs are equivalent to dPFSAs ⇒ can recognize all deterministic regular languages.
- **Deterministic Context-Free Languages:**
  - Elman RNNs with infinite precision and saturated sigmoid can simulate single-stack PDAs.
  - The stack is encoded in one dimension of the hidden state using arithmetic operations (e.g., multiply/divide).
  - The hidden state includes:
    - Data (stack encoding)

- · Flags (top symbol / empty)
- · Configuration (state/input symbol)
- · Computation (update logic)
- · Acceptance (termination)

* Parameter matrices (U, V, b_h) are engineered to implement logic operations (Facts 5.2.2, 5.2.3).

– **Turing Completeness:**

* Two-stack PDAs are Turing complete.
* Elman RNNs (with infinite precision + saturated sigmoid) can simulate such PDAs ⇒ Turing complete.
* **Undecidable Properties:**
  - · Whether an RNN is tight [Thm 5.2.9]
  - · Finding highest-probability string [Thm 5.2.10]
  - · Equivalence of two RNNs [Thm 5.2.11]
  - · Minimal hidden size for a target language model [Thm 5.2.12]

8 Computational Expressivity of Elman RNNs

The computational expressivity of Elman Recurrent Neural Networks (RNNs) is highly dependent on the assumptions regarding precision and computation time.

8.1 Practical vs. Theoretical Assumptions

- **Finite Precision and Real-time Constraints:** When Elman RNNs are implemented using finite floating-point precision and operate in real-time (performing a constant number of operations per symbol), they are equivalent to *weighted finite-state automata (WFSAs)*. This places them at the bottom of the *weighted Chomsky hierarchy* and limits them to recognizing regular languages.
- **Infinite Precision and Unbounded Computation:** If we allow arbitrary precision and permit unbounded computation between symbol steps, Elman RNNs become *Turing complete*, attaining the highest computational expressivity.

8.2 What Language Models Can Heaviside RNNs Represent?

Heaviside Elman Networks (HRNNs) use the Heaviside function:

$$H(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

as the activation function.

- HRNNs are equivalent to *deterministic probabilistic finite-state automata (dPFSAs)*.
- Thus, HRNNs represent *regular distributions* and are strictly less expressive than nondeterministic PFSAs, which are equivalent to probabilistic regular grammars and Hidden Markov Models.

8.3 Minsky Construction and Space Complexity

Minsky's Construction (Lemma 5.2.2): A method to simulate any dPFSA using an HRNN.

- **Hidden State Encoding:** The hidden state h_t encodes the current PFSA state q_t and the triggering input y_t, using a one-hot representation of the pair (q_t, y_t).
- **Transition Function Simulation:**
  - The recurrence matrix U activates possible next states from q_t.
  - The input matrix V activates states reachable by the next input y_{t+1}.
  - The bias vector b_h (typically set to -1) ensures the Heaviside activation performs an *element-wise AND*.
  - The update rule is:
$$h_{t+1} = H(Uh_t + Ve(y_{t+1}) + b_h)$$
- **Probability Encoding:** The output matrix E is constructed so that softmax over Eh_t matches the PFSA's conditional probabilities from state q_t.

Space Complexity:

- Minsky's HRNN construction requires hidden state dimensionality:

$$D = O(|Q||\Sigma|)$$

- For unweighted FSAs, more efficient constructions exist:
  - **Dewdney:** $O(|\Sigma||Q|^{3/4})$
  - **Indyk:** $O(|\Sigma|\sqrt{|Q|})$, optimal for binary alphabets
- For probabilistic FSAs, the output matrix E must span $\mathbb{R}^{|Q|}$ to encode distinct distributions for each state.
- Thus, even in HRNNs, the hidden state dimensionality must scale linearly with $|Q|$ and $|\Sigma|$.

Implications of Determinism:

- HRNNs are less expressive than non-deterministic PFSAs.
- Simulating non-determinism requires determinization, potentially causing exponential state blowup.
- Due to finite-precision arithmetic, all RNNs implemented in practice behave as finite-state models, though they offer compact and parameter-efficient representations of large WFSAs.

8.4 What Is Required for Turing Completeness?

To achieve Turing completeness, Elman RNNs must abandon two practical constraints:

1. **Arbitrary Precision:** Infinite precision in weights and computations is needed to simulate unbounded memory structures (e.g., stacks).
2. **Unbounded Computation Time:** The model must be allowed to perform an unbounded number of operations between symbol steps (i.e., not real-time).

Under these conditions, Elman RNNs with the **saturated sigmoid** activation function (clipping outputs below 0 to 0 and above 1 to 1) become Turing complete.

Mechanism:

- Stack contents are encoded as a single real number (e.g., a binary fraction).
- Push and pop operations are simulated via multiplication and division on this value.
- A single stack suffices to simulate a PDA; with two stacks, the RNN can simulate a Turing machine.

8.5 Are Real-time RNNs Turing Complete?

No, real-time RNNs are *not* Turing complete.

- Real-time RNNs with finite-precision can only represent regular languages.
- They are functionally equivalent to WFSAs.
- Turing completeness requires relaxing both real-time and precision constraints.

9 Transformers: Definitions and Expressivity

9.1 Definitions and Attention Mechanism

A **transformer network** T is formally defined as a tuple $(\Sigma, D, \text{enc}_T)$, where:

- Σ is the alphabet of input symbols,
- D is the hidden dimension,
- enc_T is the transformer encoding function.

The transformer's hidden state h_t is defined as $\text{enc}_T(y_{1:t})$, and is used to compute the conditional probability:

$$p_{\text{SM}}(y_t \mid y_{<t}) = \text{softmax}(E \cdot \text{enc}_T(y_{<t}))$$

The core innovation in transformers is the **attention mechanism**.

Attention Function: Given a query q, key matrix K, and value matrix V, the attention output is:

$$\text{Att}(q, K, V) = \sum_i s_i v_i \quad \text{where } s = f^{D-1}(\{f(q, k_i)\}_{i=1}^T)$$

Here:

- $f(q, k_i)$ is the **scoring function**, often the scaled dot product: $f(q, k_i) = \frac{q \cdot k_i}{\sqrt{D}}$
- f^{D-1} is a projection (e.g., softmax) ensuring the weights sum to 1

Transformer Layer: Each layer processes an input sequence X to output a sequence Z:

$$Z = O(\text{Att}(Q(X), K(X), V(X))) + X$$

where O is a learned output transformation. Residual connections and layer normalization are also applied.

Full Transformer: A transformer consists of L stacked layers, with the first layer receiving input embeddings (including positional encodings), and each subsequent layer operating on the output of the previous.

Masked Self-Attention: For autoregressive modeling, a mask M is applied to the attention matrix $U = QK^\top$ to prevent positions i from attending to any $j > i$.

Multi-Head Attention: Multiple attention heads compute independent attention outputs, which are concatenated and projected to allow learning diverse contextual patterns.

Efficiency:

- Time Complexity: $O(T^2D)$
- Space Complexity: $O(T^2)$ (due to the attention matrix), plus $O(TD)$ for keys, queries, values, and outputs

9.2 Motivations Behind the Transformer Architecture

1. **Parallelization:** Unlike RNNs, transformers can compute representations of all positions in parallel.
2. **Context Compression Avoidance:** Transformers retain the full sequence of contextual embeddings instead of compressing into a fixed-size vector.
3. **Long-Range Dependencies:** Attention allows focusing directly on distant but relevant symbols.

9.3 Soft vs. Hard Attention

- **Soft Attention:** Uses softmax to distribute attention across all keys. Probabilities are always positive but never exactly 0 or 1.
- **Hard Attention:**
  - **Averaging Hard Attention (hardmaxavg):** Probability mass is uniformly divided among all keys with maximal scores.
  - **Unique Hard Attention (hardmaxuni):** One of the maximal keys is selected (possibly randomly).
- **Implications:** Hardmaxuni is less expressive than hardmaxavg, which can summarize over multiple max-scoring elements. This impacts language recognition capability.

9.4 Constructing Transformers for Specific Languages

Simulating n-gram Models: Transformers can simulate any n-gram model (Theorem 5.4.1) using:

- $n-1$ attention heads, each attending to a fixed offset.
- Positional encodings to identify locations.
- A scoring function $f(q,k) = -\|q-k\|_1$ to uniquely match keys at desired positions.
- A multi-layer perceptron (MLP) f_H to encode the resulting n-gram and index the correct row of the output matrix E.

Limitations on Recognizing Languages:

- Standard transformers (especially with unique hard attention) fail to recognize simple languages like Parity, First, and Dyck.
- These tasks require tracking unbounded memory or exact state, which is hindered by attention averaging and lack of explicit state.

9.5 Turing Completeness of Transformers

Theoretical Setting: With infinite precision and unbounded computation, transformers can be made Turing complete.

Mechanism:

- The internal state is embedded into the generated string, allowing the transformer to read it later (i.e., homomorphic simulation).
- Each symbol encodes part of the state; attention reads the whole prefix to recover state and choose the next output.
- This allows simulation of WFSAs, pushdown automata, and Turing machines.

10 Tokenization and Representational Challenges in Language Modeling

10.1 Definition of Tokens

In the context of language modeling, **tokens** serve as the fundamental building blocks or symbols. These can correspond to:

- **Words**, composed of one or more characters.
- **Subword units**, derived from data-driven tokenization schemes.
- **Symbols** from an abstract alphabet Σ.

A collection of such tokens forms a **vocabulary** (analogous to an alphabet in formal language theory). Sentences or utterances are constructed by concatenating tokens from the vocabulary, forming strings over Σ^*.

Formally, language models are defined as probability distributions over sequences of tokens (utterances), typically treated as strings over a finite vocabulary.

10.2 Byte-Pair Encoding (BPE) Algorithm

While not covered in the provided sources, we briefly describe the BPE algorithm, which is widely used in subword tokenization:

- **Motivation:** BPE balances vocabulary size and coverage by allowing frequent subword patterns to form new tokens, reducing out-of-vocabulary (OOV) issues.
- **Algorithm:**
  1. Start with a symbol vocabulary consisting of individual characters.
  2. Count all adjacent symbol pairs in the training corpus.
  3. Merge the most frequent pair into a new token.
  4. Repeat the process for a fixed number of iterations or until a desired vocabulary size is reached.
- **Result:** The learned vocabulary consists of frequent characters, subwords, and whole words. It enables tokenization into familiar patterns while keeping the vocabulary size manageable.

10.3 Challenges Related to Tokenization and Representation

Limitations of One-Hot Encodings

Tokens are often represented as one-hot vectors—basis vectors in $\mathbb{R}^V$ where V is the vocabulary size. These encodings suffer from:

- **High sparsity and dimensionality:** Inefficient for large vocabularies.
- **No semantic similarity:** All non-identical vectors have zero cosine similarity.

Limitations of n-gram Models

n-gram models assign conditional probabilities based solely on the last $n-1$ tokens. They:

- Treat words as atomic, ignoring semantic similarity.
- Assign zero probability to unseen combinations, even if semantically similar ones exist.

This motivates **distributed representations (embeddings)** that encode similarity and allow generalization beyond surface forms.

Transformers and Their Token-Related Challenges

Despite their success, transformer models face several challenges:

- **Failure to recognize simple formal languages:** Transformers with unique hard attention struggle with regular languages like PARITY, FIRST, and context-free languages like DYCK.
- **Confidence degradation with input length:** Soft attention averages over positions, which reduces focus on any specific symbol in long sequences.
- **Lack of inherent sequential order:** Transformers are position-agnostic. **Positional encodings** are needed to model word order (e.g., "The dog bit the man" $\neq$ "The man bit the dog").
- **Lack of internal state:** Unlike RNNs or automata, transformers do not update a persistent hidden state. Theoretical analyses simulate such state using **augmented alphabets**, shifting from model equivalence to homomorphic language equivalence.

11 Generation and Sampling Adapters in Language Modeling

11.1 Generation: Definition

In language modeling, **generation** refers to the process by which a model produces sequences of tokens (words or symbols). Formally, a language model is a probability distribution over sequences (utterances) formed by concatenating tokens from a vocabulary, analogous to strings over an alphabet Σ.

In **locally normalized language models (LNMs)**, generation is performed one symbol at a time by sampling from conditional distributions:

$$p_{LN}(y) = \left(\prod_{t=1}^{T} p_{\mathrm{SM}}(y_t \mid y_{1:t-1}) \right) \cdot p_{\mathrm{SM}}(\mathrm{eos} \mid y_{1:T})$$

Generation begins from a `bos` (beginning-of-sequence) symbol and continues by sampling until the special `eos` (end-of-sequence) symbol is emitted.

11.2 Sampling Adapters: Concepts and Motivation

While the term *sampling adapter* is not explicitly defined in the sources, it refers broadly to any mechanism that:

- **Modifies the sampling distribution** at generation time,
- **Adjusts the training objective** to improve generative performance,
- **Guides generation toward specific properties** such as diversity, confidence, or fidelity.

Motivations for Sampling Adapters:

1. **Mean-Seeking Behavior:** Cross-entropy loss penalizes zero-probability predictions infinitely, pushing models to spread mass across all plausible outcomes, sometimes including gibberish. Sampling adapters can steer the model away from low-quality outputs.
2. **Exposure Bias:** Models trained with teacher forcing only see ground-truth tokens during training. At inference, conditioning on their own outputs can lead to error accumulation. Sampling adapters address this mismatch between training and generation.
3. **Control over Output Properties:** Depending on the application, one might want more diverse outputs (exploratory behavior) or highly reliable ones (conservative mode-seeking). Adapters provide this control.

11.3 Examples of Sampling Adapters

1. Softmax Temperature

The softmax projection function can be adapted using a **temperature** parameter φ:

$$\text{softmax}_\varphi(z_i) = \frac{\exp(z_i/\varphi)}{\sum_j \exp(z_j/\varphi)}$$

- As $\varphi \to 0$, softmax becomes argmax: sharp, deterministic sampling.
- As $\varphi \to \infty$, softmax flattens: uniform distribution.

Temperature scaling enables control over the trade-off between coherence (low φ) and diversity (high φ).

2. Scheduled Sampling

- Initially uses **teacher forcing** (ground-truth context).
- Gradually transitions to using model outputs as inputs during training.
- Helps address **exposure bias**, allowing the model to learn from its own mistakes.

However, scheduled sampling is not a consistent estimator of the data distribution.

3. Alternative Divergence Measures

Standard Objective: Maximum likelihood estimation minimizes the forward KL divergence:

$$D_{\text{KL}}(p^* \parallel p) = \sum_y p^*(y) \log \frac{p^*(y)}{p(y)}$$

This encourages **mean-seeking** behavior.

Alternative Objective: Minimizing the reverse KL divergence:

$$D_{\text{KL}}(p \parallel p^*) = \sum_y p(y) \log \frac{p(y)}{p^*(y)}$$

yields **mode-seeking** behavior, focusing only on high-probability regions.

Motivation:

- Helps reduce low-quality generations by concentrating probability on well-formed text.
- Enables adaptation to downstream goals (e.g., user satisfaction, style consistency).
- Often implemented via reinforcement learning (e.g., REINFORCE).

Part II

12 Transfer Learning and Pretrained Language Models

12.1 Transfer Learning in Neural Language Models

Transfer learning refers to leveraging knowledge learned from a source task to improve learning in a target task. In neural networks, this concept dates back to the 1970s. In language modeling, transfer learning involves using a pretrained language model $p_{\text{LM}}(y; \theta)$ on a corpus D, and transferring parameters $\hat{\theta} \subseteq \theta$ to a target task T (e.g., $f : L \to Y$) for more efficient learning than randomly initialized θ'.

Key terms:

- **Pretrained model**: Model trained on the source task (language modeling).
- **Pretraining**: The initial training phase on large-scale unlabeled data.
- **Fine-tuning**: Updating pretrained parameters on a new task.
- **Prompting / In-context learning**: Adapting the model to new tasks using demonstrations in the context window without modifying parameters.
- **Multi-task learning**: Learning multiple tasks jointly rather than sequentially.

Language modeling is ideal for transfer learning because it is self-supervised and scales easily.

12.2 ELMo (Embeddings from Language Models)

ELMo (Peters et al., 2018) introduced context-dependent word embeddings by combining forward and backward LSTM language models.

Architecture and Pretraining:

- Two LMs: forward $p_{\text{LM}}(y_t \mid y_{<t})$ and backward $p^{\leftarrow}_{\text{LM}}(y_t \mid y_{>t})$, both with L stacked LSTM layers.
- Joint training to maximize both log-likelihoods.

Downstream Fine-tuning:

- Task-specific embeddings formed as convex combinations over layers: $h_t^{\text{task}} = [\overleftarrow{h}_t^{\text{LM}}; \overrightarrow{h}_t^{\text{LM}}]$.
- Concatenated with input tokens to existing task networks.

ELMo significantly improved benchmarks in QA, NLI, NER, coreference resolution, and sentiment analysis.

12.3 BERT (Bidirectional Encoder Representations from Transformers)

BERT (Devlin et al., 2019) is a multi-layer bidirectional Transformer encoder pretrained with two self-supervised objectives.

Pretraining Tasks:

1. **Masked Language Modeling (MLM)**: Predict masked tokens using full bidirectional context.
2. **Next Sentence Prediction (NSP)**: Classify whether sentence B follows sentence A.

Architecture:

- Transformer encoder with L layers, hidden size H, A attention heads.
- Two variants: $\text{BERT}_{\text{BASE}}$ (L=12, H=768, A=12, 110M params), $\text{BERT}_{\text{LARGE}}$ (L=24, H=1024, A=16, 340M params).

Input Representation:

- WordPiece embeddings (30k vocab), positional + segment embeddings.
- Special tokens: `[CLS]` for classification, `[SEP]` for sentence separation.

Fine-tuning:

- `[CLS]` token used for sequence classification tasks.
- All parameters updated during fine-tuning.

12.4 BERT Variants

RoBERTa

RoBERTa (Liu et al., 2019) retains BERT's architecture but improves pretraining:

- Larger datasets: CC-News, OpenWebText (¿160GB).
- Dynamic masking, larger batches, longer training (500k steps).

XLNet

XLNet (Yang et al., 2019) addresses BERT's pretraining/fine-tuning mismatch using:

- **Permutation language modeling** over all possible token orderings.
- **Autoregressive Transformer** with modified attention masks to preserve original sequence order.

ALBERT

ALBERT (Lan et al., 2019) improves parameter efficiency:

- **Factorized embeddings** and **cross-layer parameter sharing**.
- New pretraining task: **Sentence Order Prediction (SOP)**.

ELECTRA

ELECTRA (Clark et al., 2020) uses a **discriminative objective**:

- **Replaced Token Detection (RTD)**: Detect whether each token was replaced.
- Joint training of generator (MLM) and discriminator; only the latter is fine-tuned.

12.5 Decoder-Only Models: GPTs

GPT models are unidirectional decoder-only Transformers.

- **GPT (Radford et al., 2018)**: Pretrained on standard LM objective, fine-tuned for downstream tasks.
- **GPT-2 (Radford et al., 2019)**: Larger model, zero-shot generalization.
- **GPT-3 (Brown et al., 2020)**: 175B parameters, strong few-shot and in-context learning capabilities.

12.6 Encoder-Decoder (Seq2Seq) TLMs

T5 (Text-to-Text Transfer Transformer)

T5 (Raffel et al., 2020) frames all tasks as text-to-text:

- Input: task description + input string.
- Output: text response generated by decoder.
- Pretraining: **text infilling** (masking token spans).
- Trained on C4 corpus with supervised + unsupervised tasks.

BART (Bidirectional and Auto-Regressive Transformer)

BART (Lewis et al., 2020a):

- Combines encoder-decoder architecture with MLM-like objectives.
- Pretraining noise functions: text infilling + sentence permutation.
- Strong results on generation tasks (e.g., summarization).

13 Parameter-Efficient Fine-Tuning (PEFT)

Parameter-Efficient Fine-Tuning (PEFT) methods aim to adapt large language models (LLMs) using a small number of parameters. They address challenges of overfitting and high computational cost associated with full model fine-tuning, particularly when annotated data is scarce.

13.1 Motivation

The motivations for PEFT include:

- **Overfitting**: Tuning all parameters on small datasets can lead to poor generalization.
- **Cost**: Full fine-tuning is computationally expensive and memory-intensive.
- **Efficiency**: PEFT methods reduce the number of trainable parameters while maintaining competitive performance.

PEFT approaches are categorized as:

1. **Partial Fine-tuning**: Fine-tunes a subset of model parameters.
2. **Adapters**: Freezes the base model and adds new trainable modules.

13.2 BitFit

BitFit is a partial fine-tuning method that updates only the bias terms:

- **Scope**: Applies to attention layers and MLPs in transformers, such as biases in $Q(x) = W_q x + b_q$.
- **Parameter Reduction**: In BERT, updates only 0.04% of parameters.
- **Performance**: Retains over 95% of full fine-tuning performance on several tasks.
- **Limitations**: Mostly evaluated on small models; effectiveness for large models is uncertain.

13.3 Adapter Tuning

Adapter tuning introduces small trainable networks (adapters) into a frozen pre-trained model:

- **Structure**: A two-layer bottleneck MLP with down-projection, non-linearity, and up-projection:
$$h \leftarrow h + f(hW_{\text{down}})W_{\text{up}}$$
- **Placement**: After attention and feed-forward sublayers.
- **Training**: Only adapters, layer norms, and sometimes classifiers are updated.
- **Computational Overhead**: Adds latency due to sequential processing.

MAD-X Framework **MAD-X (Multiple ADapters for Cross-lingual transfer)** consists of:

- **Language Adapters**: Trained for each language via masked language modeling.
- **Task Adapters**: Stack on language adapters for downstream tasks.
- **Invertible Adapters**: Mitigate vocabulary mismatch; apply invertible transformations to embeddings.

13.4 Prefix Tuning

Prefix tuning appends trainable continuous vectors (prefixes) to the model input:

- **Mechanism**: Freezes pre-trained model; only prefix vectors are optimized.
- **Formulation**:
$$\log P(y \mid x; \theta, \phi) \quad \text{where } \phi \text{ are prefix parameters}$$
- **Hybrid Methods**: Combine with discrete prompts (e.g., AutoPrompt) or soft embedding tuning (e.g., P-tuning, PTR).

13.5 LoRA (Low-Rank Adaptation)

LoRA fine-tunes LLMs by learning a low-rank decomposition of weight updates:

- **Approach**: Keep weight matrix W fixed; optimize:
$$\Delta W = BA, \quad h = Wx + \frac{\alpha}{r} BAx$$
- **Initialization**: A initialized from Gaussian; B from zeros.
- **Advantages**:
  - Dramatic parameter reduction and training efficiency.
  - Maintains or exceeds full fine-tuning performance.
  - Flexibly applied across model layers.
  - Avoids inference-time latency overhead of adapters.

14 Prompting and Zero-Shot Inference with Language Models

Prompting and zero-shot inference are powerful techniques for utilizing large language models (LLMs) without requiring task-specific fine-tuning.

14.1 Definition of Prompting

Prompting is a method for eliciting behavior from a language model (LM) by crafting specific input sequences (prompts) that guide the model to produce desired outputs. This approach leverages the LM's probability distribution $P(x;\theta)$ to infer output y by modifying the input x into a prompt x' using a prompting function $f_{\text{prompt}}(\cdot)$. A *template* defines an input slot $[X]$ and an answer slot $[Z]$, and the model searches for:

$$\hat{z} = \arg\max_z P(f_{\text{fill}}(x', z); \theta)$$

where $f_{\text{fill}}(x', z)$ fills in the answer slot.

Prompt Engineering refers to the design of prompting functions:

- **Manual Prompts**: Hand-crafted templates for small-scale tasks.
- **Automated Prompts**:
  - **Discrete Prompts** (hard prompts): Natural language strings found through mining, paraphrasing, generation (e.g., with T5), or gradient-based methods like *AutoPrompt*.
  - **Continuous Prompts** (soft prompts): Vectors in embedding space prepended to the input (e.g., *Prefix Tuning*).
  - **Hybrid Methods**: Combine discrete and continuous prompts (e.g., *P-tuning*, *PTR*).

14.2 In-Context Learning

In-context learning is an emergent capability of large LMs to perform tasks without any gradient-based updates, by conditioning on a few demonstration examples in the prompt. Introduced with GPT-3, this phenomenon allows LMs to generalize to unseen tasks using only context.

Meta-optimization Viewpoint: Recent theories suggest that in-context learning acts as an implicit form of meta-learning, where demonstration examples induce a form of gradient-like adaptation via forward computation and attention mechanisms.

14.3 Prompting Strategies

Several prompting strategies have been proposed to enhance model reasoning:

14.3.1 Chain of Thought (CoT) Prompting

- Encourages models to generate intermediate reasoning steps.
- Improves performance on complex reasoning tasks.
- **Zero-shot CoT** is enabled by appending "Let's think step-by-step" to the prompt.
- Emergent property: effectiveness improves with model size.

14.3.2 Least to Most Prompting

- Decomposes a complex task into smaller subproblems.
- Each subproblem is solved sequentially, using previous solutions as input.
- Useful for tasks where CoT underperforms.

14.3.3 Program of Thought (PoT) Prompting

- Targets numerical reasoning and symbolic computation.
- LMs generate code-like reasoning (e.g., Python), executed externally.
- Improves accuracy on mathematical tasks.

14.3.4 Self-Consistency

- Replaces greedy decoding with sampling multiple reasoning paths.
- Aggregates final answers by majority vote.
- Based on the observation that valid solutions may stem from different correct reasoning paths.

14.4 Comparison and Conclusion

Prompting strategies that involve intermediate steps (e.g., CoT, Least to Most, PoT) generally outperform direct answer prediction. These techniques leverage the LM's internal reasoning capacity more effectively and align better with the model's training paradigm.

15 Multimodal Language Models and Vision-Language Models

In a world rich with diverse information beyond just text, **Multimodal Language Models** (MLMs), particularly **Vision-Language Models** (VLMs), aim to process and understand data from multiple modalities, most notably visual and textual information. These models extend the capabilities of traditional text-only language models by integrating visual perception, mimicking the multi-sensory nature of human cognition.

15.1 Motivation and Intuition

The primary goal of VLMs is to *develop algorithms that enable computers to learn from multimodal data*, particularly vision and language. While pre-trained text-only models have achieved significant success in natural language processing, human understanding typically relies on multi-sensory inputs. Incorporating visual signals allows models to better capture real-world semantics, making them applicable to tasks such as image captioning, visual question answering, and more.

15.2 Architectures of Vision-Language Models

A typical VLM consists of the following components:

- **Text Encoder**: Extracts textual features from the input. Popular encoders include BERT and RoBERTa. Tokenized text is embedded and optionally passed through additional Transformer layers.
- **Vision Encoder**: Extracts visual features from images. Three common types are:
  - **Object Detectors (OD)**: e.g., Faster R-CNN, which captures region-level features and location information.
  - **Convolutional Neural Networks (CNNs)**: e.g., ResNet-50, ResNet-101, ResNeXt-152, typically pre-trained on ImageNet or CLIP.
  - **Vision Transformers (ViT)**: e.g., ViLT, ALBEF. Images are split into patches and encoded using Transformer layers with position embeddings and a special `[CLS]` token.
- **Multimodal Fusion Module**: Integrates visual and textual representations.
  - **Merged Attention**: Concatenates text and visual features and feeds them into a shared Transformer block (e.g., VisualBERT, UNITER).
  - **Co-attention**: Uses separate Transformer blocks for each modality with cross-attention (e.g., LXMERT, ViLBERT). Preferred for long image sequences.
- **Decoder (optional)**: Some models (e.g., T5, BART) use encoder-decoder architectures for sequence generation tasks.

15.3 Pre-training Objectives and Contrastive Learning

VLMs are often pre-trained using self-supervised and multi-task objectives:

- **Masked Language Modeling (MLM)**: Randomly masks tokens in a text-image pair and predicts them. Variants include:
  - **Seq-MLM**: Adds causal masking for alignment with image captioning.
  - **Prefix-LM**: Allows bidirectional attention over a prefix and causal masking over the rest.
  - **Autoregressive LM**: Predicts tokens sequentially given the image and previous tokens.
- **Image-Text Matching (ITM)**: Predicts whether a given image and caption match using binary classification on the `[CLS]` embedding. Negative pairs are mined for contrastive effectiveness.
- **Image-Text Contrastive Learning (ITC)**: Trains dual encoders by maximizing the similarity of matched image-text pairs and minimizing mismatched ones. Formally, this maximizes:
$$\frac{\exp(\text{sim}(I, T)/\tau)}{\sum_{(I',T')\in\mathcal{B}} \exp(\text{sim}(I', T')/\tau)}$$
where τ is a temperature parameter, and sim is usually the dot product. Models like **CLIP** and ALIGN employ this objective.
- **Masked Image Modeling (MIM)**: Trains the model to reconstruct masked image patches given the rest of the image and all text. MIM may involve predicting original visual features or regressing to token embeddings.

15.4 CLIP: A Case Study in Contrastive Learning

CLIP (Contrastive Language-Image Pre-training) is a landmark model that advanced ITC significantly. It uses a dual-encoder structure (separate image and text encoders) and learns embeddings by aligning matching image-text pairs from a large-scale web dataset. By maximizing the dot product between correct pairs and minimizing for incorrect ones, CLIP enables:

- Training of vision encoders *from scratch* using language supervision.
- Reformulation of standard vision tasks (e.g., classification) into vision-language tasks.
- *Open-vocabulary recognition*, allowing classification of previously unseen categories.

CLIP's success illustrates the strength of language supervision and contrastive learning in scaling multimodal understanding.

16 Instruction Tuning and Reinforcement Learning from Human Feedback

Multimodal Language Models (MLMs), and particularly Vision-Language Models (VLMs), integrate data from multiple sensory modalities such as vision and language. Beyond their architectural and pretraining objectives, effective alignment of Language Models (LMs) with human preferences and desired behaviors is achieved through techniques like **Instruction Tuning** and **Reinforcement Learning from Human Feedback (RLHF)**.

16.1 Instruction Tuning

Instruction tuning is a fine-tuning paradigm where language models are trained on datasets annotated with *natural language instructions* describing the task. It serves as a bridge between traditional fine-tuning and prompting.

- **Mechanism:** For each task, an instruction is prepended to the model input. For instance, in sentiment analysis, the input might be: "Is the sentiment of this movie review positive or negative?" followed by the actual review.
- **Goal:** By exposing the model to diverse instruction-based tasks, it generalizes better to unseen tasks. This exploits the latent structure already acquired during large-scale pretraining.
- **FLAN Models:** The FLAN (Finetuned Language Net) family exemplifies instruction tuning, showing that combining diverse datasets, large model sizes, and natural instructions enhances zero-shot and few-shot performance. FLAN has been scaled to 540B parameters and 1836 tasks, improving reasoning and generalization.

16.2 Reinforcement Learning from Human Feedback (RLHF)

RLHF is a framework that uses human preferences to align LMs' behaviors with human values, addressing the misalignment between next-token prediction objectives and human-aligned outputs.

RLHF Pipeline

The standard RLHF procedure involves the following steps:

1. **Prompt Distribution:** Define a distribution of prompts for which human-aligned outputs are desirable.
2. **Supervised Fine-Tuning (SFT):** Train a supervised model π_{SFT} on human-written outputs, forming an initial aligned policy.
3. **Reward Model (RM) Training:**
   - Collect human preference data: pairs of outputs (y_w, y_l) for prompt x, where y_w is preferred.

– Train reward model $r_{\theta_{\text{RM}}}(x, y)$ using the Bradley-Terry loss:

$$\mathcal{L}(\theta_{\text{RM}}) = -\frac{1}{K}\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}}\left[\log\sigma\left(r_{\theta_{\text{RM}}}(x, y_w) - r_{\theta_{\text{RM}}}(x, y_l)\right)\right]$$

4. **Policy Optimization via PPO:**
   – Fine-tune the model $\pi_{\text{RL},\phi}$ using Proximal Policy Optimization (PPO) with the reward model as a signal.
   – The PPO objective includes a KL penalty to keep π_{RL} close to π_{SFT} and a language modeling loss to maintain fluency.

InstructGPT and **ChatGPT** are key examples trained using RLHF. InstructGPT used a 175B GPT-3 model for π_{SFT} and a 6B model for the RM. Human evaluations confirmed high compliance and output quality.

Alternative RLHF Approaches

Direct Preference Optimization (DPO): DPO bypasses the reward model and RL loop by directly fine-tuning the model using preference data:

$$\mathcal{L}_{\text{DPO}}(\theta) = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}}\left[\log\sigma\left(\beta\log\frac{\pi_\theta(y_w \mid x)}{\pi_{\text{SFT}}(y_w \mid x)} - \beta\log\frac{\pi_\theta(y_l \mid x)}{\pi_{\text{SFT}}(y_l \mid x)}\right)\right]$$

Here, β is a temperature-like hyperparameter. DPO increases the likelihood of preferred responses over dispreferred ones, without requiring reward estimation or sampling.

Best-of-n and Self-Consistency:

– During data collection, labelers may select the best completion from n generated samples (best-of-n).
– **Self-consistency** is a decoding strategy where n outputs are sampled (with temperature $T > 0$), and majority voting is used to select the final answer. This assumes different reasoning paths can converge on the same correct output.

These techniques are used both to collect high-quality training data and to improve inference robustness, especially in reasoning tasks.

17 Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) enhances language models by allowing them to access external knowledge sources during inference. This mitigates the limitation of traditional language models that store knowledge only in their parameters.

17.1 Parametric vs. Non-Parametric Models

- **Parametric Models:** These store knowledge implicitly within their parameters. Updating such models to reflect new information requires expensive re-training or fine-tuning.
- **Non-Parametric Models:** These query external knowledge bases at inference time (e.g., Wikipedia, knowledge graphs). The LM generates a query, retrieves relevant content, and conditions its response on that content.

17.2 Benefits of RAG

- **Expanded Knowledge Capacity:** Enables access to a broader, dynamically updateable knowledge base.
- **Efficient Knowledge Updates:** External sources can be updated without modifying the LM.
- **Interpretability and Verifiability:** Retrieved documents can be inspected, making outputs more explainable.
- **Improved Efficiency:** Smaller non-parametric models can outperform larger parametric ones.
- **Reduced Hallucination:** Grounding responses in retrieved facts helps mitigate factual errors.

17.3 Retrieval Methods

Sparse Retrieval:

- **TF-IDF / BM25:** Rank documents based on term frequency and rarity. Limitation: lexical matching only, high-dimensional sparse vectors.

Dense Retrieval (e.g., DPR):

- **Dual Encoder:** Maps queries and documents into vector space via encoders.
- **Similarity Function:** Dot product between vector encodings:

$$\text{sim}(q, d) = \text{encoder}(q)^T \cdot \text{encoder}(d)$$

- **Contrastive Learning:** Optimize embeddings so that positive pairs are close and negatives are distant.
- **Metrics:** Recall@k, Mean Reciprocal Rank (MRR).

17.4 Integrating Retrieved Information

Early Fusion (REALM):

- Retrieved documents z are concatenated with input x to form a single sequence.
- The model is trained to predict y via $p(y \mid z, x)$.
- Joint training of retriever and predictor improves effectiveness.

Intermediate Fusion (RETRO):

- Incorporates retrieved text into intermediate Transformer layers.
- Uses **Chunked Cross-Attention (CCA)** between input chunks and retrieved neighbors.
- **RETRO-blocks** interleave standard Transformer layers to attend to retrieval content.

Late Fusion (kNN-LM):

- Builds a **datastore** of training prefix representations and their target words.
- At inference, retrieves k nearest neighbors based on current prefix.
- Combines LM distribution with retrieval distribution:

$$p(y \mid x) = (1 - \lambda) p_{\text{LM}}(y \mid x) + \lambda p_{\xi}(y \mid x)$$

- **Dynamic Gating:** A learned vector g adjusts λ depending on prefix frequency.

Prompting-Based Fusion:

- Retrieved documents are simply prepended to the model input.
- This aligns with **in-context learning**, using relevant content as additional prompt material.

Part III

18 Security and Attacks on Large Language Models

Large Language Models (LLMs) are vulnerable to a range of attacks, including **adversarial examples**, **data poisoning**, and **prompt injection**. Understanding these threats and their mitigation is critical to ensuring the safety and reliability of LLMs.

18.1 Adversarial Examples

Definition. An *adversarial example* is an input that has been subtly modified to cause a model to err. For classification tasks, given an input x and true label y, the goal is to find a perturbation $\delta \in S$ such that $f(x + \delta) \neq y$. In LLMs, the aim is to modify the input x such that $f(x) \in B$ for some undesirable output set B.

Optimization Techniques.

- **Projected Gradient Descent (PGD)** for classifiers maximizes a loss function (e.g., cross-entropy) by iteratively updating δ within constraint set S.
- **Greedy Coordinate Gradient (GCG)** for LLMs operates on discrete token spaces by appending an adversarial suffix δ to the prompt and iteratively substituting tokens to maximize a prefix loss:

$$\mathcal{L}_{\text{prefix}} = \sum_{t=1}^{k} \text{CE}(f(x)_t, y_t)$$

Threat Models.

- **White-box:** Full gradient and model access.
- **Black-box:** Attacks via transferability or query-based approximations.
- **Logit Access:** Complete or top-k logit access can be exploited using logit bias to reconstruct model internals.

Defenses and Limitations.

- **Perplexity Filters:** Reject high-perplexity prompts; bypassed via regularized loss.
- **Jailbreak Detectors:** Classify prompts as harmful or benign; vulnerable to adversarial transfer.
- **Adversarial Training:** Retrains model on adversarial samples.
- **Representation Engineering:** Monitors internal activations; bypassed with benign-looking jailbreaks.

18.2 Data Poisoning and Backdoor Attacks

Backdooring a Classifier. Poison a subset of training data using a trigger pattern and desired output. Even *clean-label attacks* are possible using adversarial perturbations.

Poisoning RLHF. Poison the reward model training data to misalign outputs:

$$\mathcal{L}_{\text{RM}} = -\frac{1}{K}\mathbb{E}_{(x,y_w,y_l)}\left[\log \sigma(r_\theta(x, y_w) - r_\theta(x, y_l))\right]$$

Poisoning Pretraining. Exploit publicly maintained datasets:

- **Image Datasets:** Use domain-squatting to alter images at expired URLs.
- **Wikipedia Dumps:** Inject malicious edits just before scheduled dumps.
- **Formatted Dialogs:** Plant harmful examples as assistant-style chats.

18.3 Prompt Injection

Definition. Prompt injection tricks LLMs into executing malicious instructions embedded within otherwise benign-looking inputs.

Examples. Injecting commands into:

- Natural language instructions (e.g., translation tasks).
- Calendar events or user data processed by LLM agents.

Defenses.

- **Delimiters:** Syntax-based separation like `[DATA]`—limited effectiveness.
- **Detectors:** Classifiers to flag injected prompts—vulnerable to adversarial bypasses.
- **Instruction Hierarchy:** System prompt > user prompt > model outputs > external data.
- **System-Level Isolation:**
  - **Dual LLM Pattern:** A privileged LLM handles control flow; a quarantined LLM handles untrusted data.
  - **CaMeL:** Encodes and enforces safe dataflow via formal code representations.

19 Logits, Watermarking, and Model Stealing in LLMs

19.1 Logits

In neural language models, **logits** are the unnormalized prediction scores output by the model before applying a projection function such as the softmax.

- Given an input sequence, the model computes logits $l_t(k)$ for each vocabulary token k at position t. The corresponding probability is:

$$p_t(k) = \frac{\exp(l_t(k))}{\sum_{i \in V} \exp(l_t(i))}$$

- The softmax function may be scaled by a temperature parameter φ:

$$p_t(k) = \frac{\exp(l_t(k)/\varphi)}{\sum_i \exp(l_t(i)/\varphi)}$$

- Logits arise from inner products like $E \cdot \text{enc}(y)$, and are interpreted as *compatibility scores* between the context and the output token.

19.2 Watermarking in Language Models

Watermarking techniques embed hidden statistical signals in LLM-generated text to identify or trace its origin.

Hard Red List Watermarking.

- Vocabulary V is partitioned into green and red lists, e.g., using a PRNG seeded on the prior token.
- The model is restricted to sample only from the green list.
- Detection is based on a z-test for the number of green tokens. For lists of equal size:

$$\mathbb{P}(\text{T green tokens}) = \left(\frac{1}{2}\right)^T$$

- Removal typically requires editing 25% of tokens.

Soft Red List Watermarking.

- Instead of banning red tokens, a constant δ is added to green token logits:

$$p_t(k) \propto \begin{cases} \exp(l_t(k) + \delta) & \text{if } k \in G_t \\ \exp(l_t(k)) & \text{if } k \in R_t \end{cases}$$

- Maintains quality on low-entropy sequences while preserving detectability.

Public vs. Private Watermarking.

- **Public:** Anyone can verify using known list generation algorithms.
- **Private:** Lists are generated using a keyed pseudorandom function F_K; detection requires access to the secret key K.

19.3 Model Stealing

Model stealing aims to replicate the functionality or parameters of a target LLM using only API access.

Logistic Regression Stealing.

- If confidence scores are returned, each query yields a linear equation. $d+1$ queries suffice to solve for the d-dimensional weights and bias.
- With label-only access, a binary search can locate decision boundaries for parameter recovery.

Distillation and Parameter Recovery in Deep Networks.

- Stealing deep models becomes model distillation: train a new model $\hat{f}$ on pairs $(x, f(x))$.
- For ReLU networks, linear regions of input space can sometimes be exploited to extract parameters (in theory).

Stealing Transformers via Logits.

- Transformers compute output as $\text{softmax}(W \cdot g(p))$, where $g(p) \in \mathbb{R}^h$ is the hidden representation.
- Using full logit access, collect logit vectors for n prompts and form a matrix $Q = WH$. Then:
  - $\text{rank}(Q) = h$ reveals hidden size.
  - Apply SVD: $Q = U\Sigma V^T$ to recover $\tilde{W} \approx W$.
- With top-k access and a logit bias API, individual logits can be recovered:

$$l_t(i) = \text{logit}(i) - \text{bias}(i)$$

20 Privacy in Large Language Models

Large Language Models (LLMs) can satisfy various notions of privacy that address concerns over training on sensitive data, commercial confidentiality, and deployment security. This section surveys multiple privacy paradigms, attack strategies, and defense mechanisms.

20.1 Forms of Privacy for LLMs

1. Cryptographic Privacy.

- **Secure Training:** Enables multiple parties to jointly compute a model without revealing their private data.
- **Secure Inference:** A server computes $f(x)$ for a client without learning x, and the client learns only $f(x)$.
- **Challenges:** Cryptographic protocols require model quantization and polynomial activation approximations, leading to high computational overhead.

2. Learning from Encoded Data.

- Clients send encoded inputs $E(x_i)$ to a server for training.
- **Instance-Hiding:** Techniques like InstaHide and TextHide obscure raw inputs but have been broken by recent attacks.
- **Federated Learning:** Aggregates client gradients without raw data access. However, **gradient reconstruction attacks** can recover individual samples from gradients.

3. Privacy Backdoors.

- **In Federated Learning:** A malicious server crafts updates that cause gradient aggregation to leak private data.
- **In Pretraining:** A malicious actor distributes a poisoned pre-trained model that leaks data when fine-tuned on sensitive information.

20.2 Privacy Attacks

1. Data Extraction.

- **Goal:** Recover memorized training examples by prompting the model.
- **Observation:** Base LLMs frequently emit memorized sequences, especially as model size increases.
- **Defense Bypass:** Aligned models like ChatGPT may not emit training data with simple prompts but can still be coerced into doing so.

2. Membership Inference Attacks (MIAs).

- **Goal:** Decide if a data point x was in the training set.
- **Statistical Framing:** Formulated as a hypothesis test using a test statistic (e.g., model loss on x).
- **LiRA:** Uses shadow models to approximate the likelihood distributions under each hypothesis.
- **Challenges:**
  - Difficult loss selection for LLMs.
  - Shadow models are costly to train.
  - Evaluations often confound distribution shifts with true membership.
- **Proof of Membership:**
  - **Canaries:** Inject unique strings and check for low loss.
  - **Extraction:** High-entropy verbatim output strongly indicates memorization.

3. Inference-Time Attacks.

- LLMs may extract sensitive information (e.g., locations, PII) from images or structured text inputs.

20.3 Heuristic Defenses

- **Memorization Filters:** Use string matching or decoding constraints to block known training substrings.
- **Deduplication:** Remove duplicate training examples to reduce memorization likelihood.
- **Limitations:** Filters can be bypassed by paraphrasing, and may reveal membership via output suppression.

20.4 Differential Privacy (DP)

Definition. A randomized algorithm M is ε-differentially private if for any neighboring datasets D_1, D_2 differing by one row and any event S:

$$\Pr[M(D_1) \in S] \leq e^{\varepsilon} \cdot \Pr[M(D_2) \in S]$$

Key Properties.

1. **Post-Processing Invariance:** $f(M(D))$ is still ε-DP.
2. **Composition:** Multiple DP mechanisms compose linearly in privacy cost.

The Laplace Mechanism.

- Adds noise scaled to ℓ_1-sensitivity of the query function and ε.

20.5 Differentially Private SGD (DP-SGD)

- **Process:** Clip each sample's gradient, add Gaussian noise, and update parameters.
- **Privacy Guarantee:** Model remains approximately unchanged by adding/removing a single training point.
- **Challenges for LLMs:**
  - High compute/memory cost due to per-sample gradients.
  - Ambiguity in defining the "individual" (word, document, author).

20.6 Public Pretraining and Private Fine-Tuning

- **Public Pretraining:** General knowledge is acquired from large-scale, publicly available corpora.
- **Private Fine-Tuning:** When adapting to sensitive domains, techniques like DP-SGD ensure that the fine-tuned model avoids memorization.
- This stands in contrast to **privacy backdoors**, where pretraining is used to induce memorization during downstream fine-tuning.