Homework 3: Recurrent Neural Networks (RNN) Language Modeling (LM)

Daryoush Safe

October 2025

1 Recurrent Neural Networks (RNNs) & LSTMs

1.1 Vanishing and Exploding Gradients

(a) We wish to analyze the effect of the first hidden layer, h_1, on the t-th hidden layer, h_t. For simplicity, we assume the activation functions are identity functions, which, while not realistic, allows us to isolate the effect of the weight matrices.

The recurrence relation for the hidden state at time t is given by:

$$h_t = W_h h_{t-1} + W_x x_t$$

Substituting the expression for h_{t-1}:

$$h_{t-1} = W_h h_{t-2} + W_x x_{t-1}$$

yields:

$$h_t = W_h(W_h h_{t-2} + W_x x_{t-1}) + W_x x_t = W_h^2 h_{t-2} + W_h W_x x_{t-1} + W_x x_t$$

By unrolling this recurrence back to the first time step, the component of h_t specifically influenced by the initial input x_1 is:

$$h_t^{(x_1)} = W_h^{t-1} W_x x_1$$

This shows that the influence of x_1 is modulated by the matrix W_h raised to the power $t-1$.

To analyze the long-term behavior, we consider the eigendecomposition of W_h, assuming it is diagonalizable:

$$W_h = U \Lambda U^{-1}$$

where Λ is a diagonal matrix whose entries are the eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$ of W_h.

For any eigenvector u_i, we have:

$$W_h u_i = \lambda_i u_i$$

For a general vector v expressed in the eigenbasis as $v = \alpha_1 u_1 + \alpha_2 u_2 + \cdots + \alpha_n u_n$, it follows that:

$$W_h v = \alpha_1 \lambda_1 u_1 + \alpha_2 \lambda_2 u_2 + \cdots + \alpha_n \lambda_n u_n$$

and therefore:

$$W_h^t v = \alpha_1 \lambda_1^t u_1 + \alpha_2 \lambda_2^t u_2 + \cdots + \alpha_n \lambda_n^t u_n$$

As $t \to \infty$, the term with the largest eigenvalue magnitude will dominate the sum. Let $m = \arg\max_j |\lambda_j|$. Then:

$$\lim_{t\to\infty} W_h^t v \approx \alpha_m \lambda_m^t u_m$$

This asymptotic behavior explains the vanishing and exploding gradient problem in RNNs:

- If $|\lambda_m| > 1$, the term λ_m^t grows exponentially, leading to **exploding gradients**.
- If $|\lambda_m| < 1$, the term λ_m^t decays exponentially, leading to **vanishing gradients**.

The core issue in RNNs is the **recursive multiplication** of the weight matrix W_h over many time steps. This recursive structure, which is essential for processing sequences, causes gradients to scale with powers of the eigenvalues. In contrast, standard Multi-Layer Perceptrons (MLPs) do not have this temporal recursion; the depth is fixed, and gradients flow through a static, non-recurrent composition of functions, making them less susceptible to these specific exponential dynamics.

(b) The standard RNN equations are:

$$h_t = W_{hh} h_{t-1} + W_{hx} x_t$$
$$z_t = \tanh(h_t)$$

The gradient with respect to W_{hh} accumulates across all time steps:

$$\frac{\partial L_T}{\partial W_{hh}} = \sum_{k=1}^{T} \frac{\partial L_T}{\partial z_T} \frac{\partial z_T}{\partial h_T} \left(\prod_{j=k+1}^{T} \frac{\partial h_j}{\partial h_{j-1}} \right) \frac{\partial h_k}{\partial W_{hh}}$$

The critical term is the product of Jacobians:

$$\frac{\partial h_T}{\partial h_k} = \prod_{j=k+1}^{T} \frac{\partial h_j}{\partial h_{j-1}} = \prod_{j=k+1}^{T} W_{hh} \cdot \mathrm{diag}(\tanh'(h_{j-1}))$$

Where:

- $\frac{\partial h_j}{\partial h_{j-1}} = W_{hh} \cdot \text{diag}(\tanh'(h_{j-1}))$
- $\tanh'(h_{j-1}) = 1 - \tanh^2(h_{j-1})$ is the element-wise derivative

The long-term dependency problem comes from:

$$\left| \prod_{j=k+1}^{T} \frac{\partial h_j}{\partial h_{j-1}} \right| \leq |W_{hh}|^T \cdot |\tanh'|_\infty^T$$

Since $\| \tanh' \|_\infty \leq 1$ and typically < 1, this product tends to vanish when T is large. If $\|W_{hh}\| > 1$, it can explode instead.

(c) A primary solution is the use of gated recurrent units, such as those found in Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) networks. The core idea is to replace the recursive matrix multiplications in the vanilla RNN with a combination of element-wise multiplications and summations, which allows for more stable gradient flow.

- A primary solution is the use of gated recurrent units, such as those found in Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) networks. The core idea is to replace the recursive matrix multiplications in the vanilla RNN with a combination of element-wise multiplications and summations, which allows for more stable gradient flow.

  The LSTM introduces a cell state C_t, which acts as a conveyor belt for long-term information, and three gating mechanisms to regulate the flow of information. This architecture uses approximately four times the parameters of a vanilla RNN to implement this gating capability.

  The key equations for an LSTM cell are:

$$\begin{aligned}
f_t &= \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) && \text{(Forget Gate)} \\
i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) && \text{(Input Gate)} \\
o_t &= \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) && \text{(Output Gate)} \\
g_t &= \tanh(W_g \cdot [h_{t-1}, x_t] + b_g) && \text{(Cell Gate)} \\
C_t &= f_t \odot C_{t-1} + i_t \odot g_t && \text{(Cell State Update)} \\
h_t &= o_t \odot \tanh(C_t) && \text{(Hidden State Update)}
\end{aligned}$$

  The forget gate (f_t) determines what information to discard from the previous cell state. The input gate (i_t) and cell gate (g_t) work together to decide what new information to store. The additive interaction in the cell state update, $C_t = f_t \odot C_{t-1} + i_t \odot g_t$, is crucial. It allows gradients to flow backwards through time largely unimpeded by the forget gate, mitigating the vanishing gradient problem and enabling long-term dependencies.

The Gated Recurrent Unit (GRU) offers a similar gating mechanism with a slightly simplified structure, combining the input and forget gates into a single update gate. It achieves comparable performance to LSTMs in many scenarios while being computationally more efficient.

- To directly address the problem of exploding gradients, a common and effective technique is *gradient clipping*. This technique caps the norm of the gradient during backpropagation, preventing parameter updates from becoming destructively large while preserving the gradient's direction.
- The choice of activation function is critical for stable RNN training. While vanilla RNNs can, in theory, use any activation function, some are more suitable than others.

  The **Sigmoid** function (σ) is prone to rapid saturation, as its derivatives approach zero for large positive or negative inputs. This characteristic severely limits the network's ability to build and maintain long-term memory.

  The **ReLU** function, while successful in deep feedforward networks, can lead to exploding activations in RNNs when the recurrent weights are positive, as its output is unbounded above.

  The **Hyperbolic Tangent (Tanh)** function is generally the best choice in practice. It saturates more slowly than the sigmoid function, providing a more stable and larger region where its derivative is non-zero. This property makes it more effective at propagating gradients over multiple time steps.

1.2 Long Short-Term Memory (LSTM) Architecture

(a) The key equations for an LSTM cell are:

$$\begin{aligned}
f_t &= \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) && \text{(Forget Gate)} \\
i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) && \text{(Input Gate)} \\
o_t &= \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) && \text{(Output Gate)} \\
g_t &= \tanh(W_g \cdot [h_{t-1}, x_t] + b_g) && \text{(Cell Gate)} \\
C_t &= f_t \odot C_{t-1} + i_t \odot g_t && \text{(Cell State Update)} \\
h_t &= o_t \odot \tanh(C_t) && \text{(Hidden State Update)}
\end{aligned}$$

(b) The LSTM introduces a cell state C_t, which acts as a conveyor belt for long-term information, and three gating mechanisms to regulate the flow of information. This architecture uses approximately four times the parameters of a vanilla RNN to implement this gating capability.

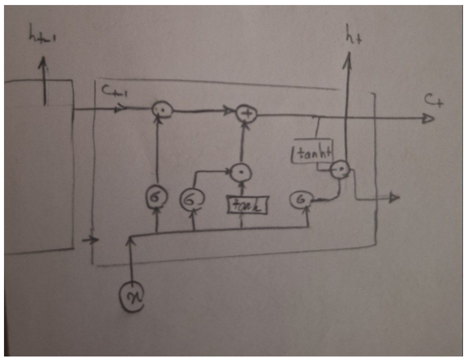

Figure 1: 1.2.1 answer

The key equations for an LSTM cell are:

$$
\begin{aligned}
f_t &= \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) && \text{(Forget Gate)} \\
i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) && \text{(Input Gate)} \\
o_t &= \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) && \text{(Output Gate)} \\
g_t &= \tanh(W_g \cdot [h_{t-1}, x_t] + b_g) && \text{(Cell Gate)} \\
C_t &= f_t \odot C_{t-1} + i_t \odot g_t && \text{(Cell State Update)} \\
h_t &= o_t \odot \tanh(C_t) && \text{(Hidden State Update)}
\end{aligned}
$$

The forget gate (f_t) determines what information to discard from the previous cell state. The input gate (i_t) and cell gate (g_t) work together to decide what new information to store. The additive interaction in the cell state update, $C_t = f_t \odot C_{t-1} + i_t \odot g_t$, is crucial. It allows gradients to flow backwards through time largely unimpeded by the forget gate, mitigating the vanishing gradient problem and enabling long-term dependencies.

The sigmoid activation function, defined as $\sigma(x) = \frac{1}{1+e^{-x}}$, produces outputs in the range $[0, 1]$. This bounded output range makes it ideally suited for gating mechanisms:

- In the **forget gate**, σ produces values between 0 and 1 that determine the fraction of each component in the previous cell state C_{t-1} to retain. A value near 0 indicates complete forgetting, while a value near 1 indicates complete retention.
- In the **input gate**, σ regulates how much of the new candidate information should be incorporated into the cell state.
- In the **output gate**, σ controls how much of the current cell state should be exposed to the hidden state h_t.

In all cases, the sigmoid function serves as a *soft attention mechanism*, providing differentiable, element-wise control over information flow.

The hyperbolic tangent function, defined as $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, produces outputs in the range $[-1, +1]$. This symmetric range around zero provides several advantages:

- **Zero-centered outputs**: Unlike sigmoid, tanh produces outputs with mean zero, which helps maintain stable gradients during backpropagation and prevents systematic bias in one direction.
- **Gentle saturation**: While both functions saturate, tanh saturates more gradually than sigmoid, providing a larger region where derivatives are significantly non-zero. This property allows gradients to flow more effectively during training.
- **Natural regularization**: The bounded range $[-1, 1]$ naturally constrains the magnitude of activations, acting as an implicit regularizer that prevents activations from growing uncontrollably.

In LSTM architectures, tanh is typically used for the cell gate g_t and for scaling the cell state to produce the hidden state, as these applications benefit from its symmetric, bounded nature and stable gradient properties.

(c) In Vanilla RNNs, a single shared weight matrix W_{hh} governs the influence of previous hidden states on the current state. As established in Question 1, the recurrence relation $h_t = W_{hh}h_{t-1} + W_{hx}x_t$ leads to recursive matrix multiplication during backpropagation. This results in the gradient term containing factors of W_{hh}^T, which causes the gradients to either vanish exponentially when the spectral radius $\rho(W_{hh}) < 1$ or explode exponentially when $\rho(W_{hh}) > 1$.

Long Short-Term Memory (LSTM) networks address this fundamental limitation through the introduction of a **cell state** C_t, which serves as a continuous internal memory that propagates information across time steps with minimal transformation.

The key innovation lies in the cell state update equation:

$$C_t = f_t \odot C_{t-1} + i_t \odot g_t$$

Rather than relying on a single weight matrix, LSTMs employ four specialized gating mechanisms:

- **Forget Gate** (f_t): Determines what information to remove from the cell state
- **Input Gate** (i_t): Controls what new information to store in the cell state
- **Cell Gate** (g_t): Creates candidate values for updating the cell state
- **Output Gate** (o_t): Regulates what information to expose via the hidden state

The combination of additive state updates and gated control mechanisms enables LSTMs to maintain stable gradient flow over long sequences, effectively mitigating the vanishing gradient problem while preserving the ability to learn long-term dependencies.

1.3 Gated Recurrent Unit (GRU) vs. LSTM

(a) The Gated Recurrent Unit (GRU) is a gated recurrent architecture that provides performance comparable to LSTM while offering a more streamlined design with fewer parameters. GRU achieves this through two key gating mechanisms that regulate information flow.

The GRU cell is defined by the following equations:

$$\begin{aligned} z_t &= \sigma(W_z \cdot [h_{t-1}, x_t] + b_z) && \text{(Update Gate)} \\ r_t &= \sigma(W_r \cdot [h_{t-1}, x_t] + b_r) && \text{(Reset Gate)} \\ \hat{h}_t &= \tanh(W_h \cdot [r_t \odot h_{t-1}, x_t] + b_h) && \text{Candidate Activation} \\ h_t &= (1 - z_t) \odot \hat{h}_t + z_t \odot h_{t-1} && \text{Hidden State Update} \end{aligned}$$

The update gate $z_t \in [0, 1]$ determines the balance between preserving previous information and incorporating new knowledge. It functions as an interpolation controller:

- When $z_t \approx 1$: The hidden state primarily retains information from h_{t-1}, effectively "remembering" past states
- When $z_t \approx 0$: The hidden state is largely replaced by the candidate activation $\hat{h}_t$, allowing significant updates with new information

The reset gate $r_t \in [0, 1]$ controls how much of the previous hidden state contributes to the candidate activation:

- When $r_t \approx 0$: The gate "resets" the influence of h_{t-1} on the candidate activation, allowing the unit to discard irrelevant historical information when processing new inputs
- When $r_t \approx 1$: The gate fully incorporates the previous hidden state, maintaining contextual information necessary for understanding the current input

This interpolation mechanism $h_t = (1 - z_t) \odot \hat{h}_t + z_t \odot h_{t-1}$ provides a direct pathway for gradients to flow through time, mitigating the vanishing gradient problem in a manner similar to the LSTM's additive cell state.

(b) GRU simplifies the LSTM architecture by combining the input and forget gates into a single update gate and eliminating the separate cell state. This reduction in gating complexity results in fewer parameters while maintaining competitive performance on most sequence modeling tasks.

The unified update mechanism provides an efficient compromise between expressive power and computational efficiency.

There is no definitive theoretical proof establishing the universal superiority of either LSTM or GRU architectures across all domains. Empirical evidence demonstrates that the relative accuracy and performance of these models are highly task-dependent and dataset-specific.

In summary, while LSTM offers more sophisticated control mechanisms for long-term memory, GRU provides a compelling alternative with better computational complexity and parameter efficiency, making the choice fundamentally application-dependent. A practical recommendation is to use empirical validation on a held-out validation set remains the most reliable method for architecture selection for a specific application.

1.4 Backpropagation Through Time (BPTT)

(a) Backpropagation Through Time (BPTT) consists of two sequential phases: a forward pass to compute losses and a backward pass to compute gradients and update parameters.

During the forward pass, the network processes the input sequence sequentially, computing hidden states and output predictions at each time step. For a Vanilla RNN, the hidden state at time t is computed as:

$$h_t = \tanh(W_{hh}h_{t-1} + W_{hx}x_t + b_h)$$

This recurrence relation demonstrates that the current hidden state h_t depends on both the current input x_t and the previous hidden state h_{t-1} through the shared weight matrix W_{hh}. This parameter sharing across time steps is a defining characteristic of RNNs that enables them to process sequences of arbitrary length.

The total loss over the entire sequence of length T is computed as the sum of individual losses at each time step:

$$\mathcal{L} = \sum_{t=1}^{T} \ell(\hat{y}_t, y_t)$$

The backward pass computes gradients with respect to all parameters by propagating error signals backwards through both the network layers and the temporal dimension. Due to parameter sharing across time, the gradient for W_{hh} accumulates contributions from all time steps. The gradient of the total loss with respect to the recurrent weight matrix W_{hh} is given by:

$$\frac{\partial \mathcal{L}}{\partial W_{hh}} = \sum_{k=1}^{T} \frac{\partial \mathcal{L}}{\partial h_T} \frac{\partial h_T}{\partial h_k} \frac{\partial h_k}{\partial W_{hh}}$$

This can be expanded using the chain rule through time:

$$\frac{\partial \mathcal{L}}{\partial W_{hh}} = \sum_{k=1}^{T} \frac{\partial \mathcal{L}}{\partial h_T} \left(\prod_{j=k+1}^{T} \frac{\partial h_j}{\partial h_{j-1}} \right) \frac{\partial^+ h_k}{\partial W_{hh}}$$

(b) In standard Backpropagation Through Time (BPTT), the gradient computation requires maintaining the complete computational graph across all time steps. For long sequences, Storing all intermediate hidden states and activations for all time steps requires huge memory and has much time complexity. Also in terms of vanishing gradients, the influence of distant time steps on current gradients becomes exponentially attenuated due to recursive matrix multiplication. Truncated Backpropagation Through Time (T-BPTT) addresses these limitations by approximating the full gradient computation using only a fixed window of recent history.

2 Word Embeddings & Language Modeling

2.1 Word Embeddings

(a) natural language consists of discrete symbolic units—words and sentences—that cannot be directly processed by these models. This representational gap necessitates a mapping from discrete linguistic symbols to continuous vector spaces where mathematical operations can be performed.

The most straightforward approach to word representation is one-hot encoding. In this scheme, each word in a vocabulary of size V is represented as a sparse binary vector $\mathbf{v} \in \{0, 1\}$. Despite its conceptual simplicity, one-hot encoding suffers from significant limitations:

- Dimensionality Curse
- Memory Inefficiency
- Semantic Sparsity: All words are orthogonal—the dot product between any two distinct word vectors is zero, preventing the representation of semantic relationships.

Dense word embeddings address these limitations by mapping words to continuous, low-dimensional vector spaces. Instead of sparse high-dimensional vectors, words are represented as dense vectors $\mathbf{e} \in \mathbb{R}^d$ where $d \ll V$. By using this approach similar words have similar vector representations (small Euclidean distance) and Semantic relationships can be expressed as vector arithmetic (e.g., king − man + woman ≈ queen).

Dense embeddings overcome the limitations of one-hot encoding by distributing semantic information across all dimensions of the vector, creating representations that are both computationally efficient and linguistically meaningful.

(b) Both Word2Vec and GloVe models learn distributed representations where each word in the vocabulary is mapped to a unique dense vector in a continuous space. The core principle underlying these models is that the embedding vector for a word is determined by both the word itself and its linguistic context. The context is typically defined as a sliding window of surrounding words within a fixed radius, which can be effectively captured using a delayed many-to-many architectural approach.

Both models employ a masked language modeling objective, where the training task involves predicting words with certain positions masked or omitted. Given the context words, the model predicts the target word, and the loss is computed by comparing the prediction against the ground truth value.

The training process can be formalized as:

- For a given context window $C = \{w_{t-k}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+k}\}$
- The model predicts the target word w_t (or vice versa in skip-gram)
- The loss function measures the discrepancy between predicted and actual word distributions:

$$\mathcal{L}(\theta) = -\log P(w_t|C;\theta)$$

- Model parameters θ (embedding matrices) are updated via gradient descent to minimize this loss

These training methodologies are grounded in the distributional hypothesis, posits that words that occur in similar contexts tend to have similar meanings. Formally, this can be expressed as:

The mathematical instantiation of this principle states that if two words w_i and w_j have similar probability distributions over their context words, then their embedding vectors $\mathbf{v}_i$ and $\mathbf{v}_j$ should be geometrically close in the vector space.

This theoretical foundation ensures that the learned embeddings capture meaningful semantic and syntactic relationships, enabling algebraic operations in the vector space (e.g., king $-$ man $+$ woman $\approx$ queen).

(c)
- **Continuous Bag-of-Words (CBOW)**: Predicts a target word given its surrounding context words. The model averages the context word embeddings and uses this representation to predict the missing center word.
- **Skip-gram**: Predicts context words given a target word. This approach uses the center word to predict the surrounding context words within the window.

In case of handling rare words the available contextual information is inherently limited, making it difficult for CBOW to accurately predict these words from their sparse contexts. But skip-gram works better on handling rare words. Even with limited occurrences, a rare word's embedding can be effectively updated each time it appears as the center word, as it must predict multiple context words in each training instance.

As a computational point of view CBOW is more efficient because it has to predict fewer masked words.

2.2 Language Modeling

(a) A Statistical Language Model is a probabilistic framework that assigns likelihoods to sequences of linguistic tokens. Its fundamental objective is to estimate the probability distribution over the next token in a sequence given the preceding context.

The training process can be formalized as:

- For a given context window $C = \{w_{t-k}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+k}\}$
- The model predicts the target word w_t (or vice versa in skip-gram)
- The loss function measures the discrepancy between predicted and actual word distributions:

$$\mathcal{L}(\theta) = -\log P(w_t | C; \theta)$$

- Model parameters θ (embedding matrices) are updated via gradient descent to minimize this loss

(b) RNN-based language models are trained using an **autoregressive** approach that processes sequences sequentially. The training procedure follows these steps:

(a) **Initialization**: The model input begins with a special `[start]` token, and the initial hidden state h_0 is typically initialized to zeros.

(b) **Sequential Prediction**: At each time step t, the model consumes the previous token and predicts the next token in the sequence:

- Input: Token from position $t - 1$ (or `[start]` token for $t = 1$)
- Target: Token at position t

(c) **Loss Computation and Weight Update**: The model's prediction is compared against the ground truth token using a cross-entropy loss function, and gradients are computed to update the model parameters via backpropagation through time (BPTT).

(d) **Teacher Forcing**: During training, the input for the next time step is the **ground truth token** from the previous position rather than the model's own prediction. This training strategy, known as teacher forcing, stabilizes training and accelerates convergence.

Since RNNs operate on continuous vector spaces, discrete input tokens must be converted into dense vector embeddings. Also for the output the hidden state h_t (which can be viewed as a contextual embedding) is projected to the vocabulary space:

$$z_t = W_{hy} h_t + b_y$$

A softmax function is applied to convert the logits z_t into a probability distributions. This distribution represents the model's belief about the next token given the preceding context.

(c) Perplexity is a metric that measures the uncertainty of a model's predictions. Specifically, in language models, it quantifies how well the model predicts the next word in a sequence. When a model makes a prediction, it assigns probabilities to possible next words.

Mathematically, perplexity is calculated as:

$$\text{Perplexity} = 2^{H(p)}$$

where $H(p)$ is the entropy of the model's predictions.

Entropy $H(p) = -\sum_{x \in C} P(x) \log_2^{p(x)}$ measures the level of uncertainty in the model's output. Lower entropy means the model is more certain about its predictions and therefore, the perplexity is lower.

Perplexity indicates the level of confidence the model has in its prediction—lower perplexity suggests higher confidence and better performance in predicting the next word, while higher perplexity signals more uncertainty and less reliability. In simple terms, perplexity represents the number of potential options the model is considering when making its prediction.

Click here for source

2.3 Contextual vs. Non-Contextual Embeddings

(a) The meaning of words is fundamentally context-dependent. Consider the word "bank", which can refer to:

- A financial institution in monetary contexts: "I deposited money at the **bank**"
- The land alongside a river: "We sat on the river **bank**"

So each token's representation adapts based on its relationship with all other tokens in the sequence. This called Context-Aware Representations.

This linguistic phenomenon demonstrates that semantic interpretation requires dynamic consideration of the surrounding context. Traditional recurrent models struggle to adequately capture these contextual relationships, particularly over longer sequences.

In conventional RNN models, the hidden state h_t attempts to compress the entire history of previous states $\{h_1, h_2, \ldots, h_{t-1}\}$ into a fixed-dimensional vector:

$$h_t = f(W_{hh}h_{t-1} + W_{xh}x_t)$$

This approach suffers from several critical limitations. The fixed-length hidden state vector becomes a bottleneck, forcing the model to discard potentially relevant historical information. All previous hidden states contribute equally through shared weight matrices, ignoring varying degrees of relevance to the current context. The compressed representation tends to prioritize recent information, making it difficult to capture long-range dependencies effectively and the model cannot dynamically emphasize specific parts of the input sequence that are most relevant to the current processing step.

Attention mechanisms address these limitations by enabling the model to dynamically focus on different parts of the input sequence when processing each element. The core idea is to compute the current representation as a weighted combination of all previous hidden states, where the weights are determined by relevance.

The attention-based context vector c_t for time step t is computed as:

$$c_t = \sum_{i=1}^{t-1} \alpha_{t,i} h_i$$

where the attention weights $\alpha_{t,i}$ represent the relevance of hidden state h_i to the current state h_t. These weights are calculated using a similarity functions.

(b) Models like ELMo or BERT generate dynamic, context-aware representations using attention mechanisms.

e attention mechanism enables models to dynamically focus on different parts of the input sequence that are most relevant to the current processing task. The core innovation is computing each token's representation as a weighted combination of all other tokens in the sequence:

$$\text{ContextualizedEmbedding}(w_i) = \sum_{j=1}^{n} \alpha_{ij} \cdot \text{Representation}(w_j)$$

where the attention weights α_{ij} quantify the relevance of word w_j to word w_i so each word can selectively attend to other words based on semantic and syntactic relationships, rather than applying uniform treatment.

2.4 Softmax and Output Layer

(a) Output the hidden state h_t (which can be viewed as a contextual embedding) is projected to the vocabulary space:

$$z_t = W_{hy}h_t + b_y$$

A softmax function is applied to convert the logits z_t into a probability distributions. This distribution represents the model's belief about the next token given the preceding context.

$$s(z_t) = \frac{e^{z(t)}}{\sum_{i=1}^{T} e^{z(i)}}$$

(b) The standard formula of the softmax:

$$s(z_t) = \frac{e^{z(t)}}{\sum_{i=1}^{T} e^{z(i)}}$$

The computational bottleneck lies in the $\sum_{i=1}^{n} e^{z(i)}$ term which is need to calculate for every n embedding vector in the vocabulary. For a large vocabularies, this operation becomes too expensive. Since this calculation must be performed for every training example, and each training example requires a forward pass (to compute the loss) and a backward pass (to compute gradients), the softmax dominates the training time.

Negative Sampling converts the problem into binary classification. Instead of computing probability across the entire vocabulary, train the model to push the true word closer to the context vector and push a small set of negative words farther away. This eliminates the need for normalizing across all vocabulary tokens.