Due Date: March 28th, 23:00 2025

Instructions

- For all questions that are not graded only on the answer, show your work! Any problem without work shown will get no marks regardless of the correctness of the final answer.
- Please try to use a document preparation system such as LaTeX. If you write your answers by hand, note that you risk losing marks if your writing is illegible without any possibility of regrade, at the discretion of the grader.
- Submit your answers electronically via the course GradeScope. Incorrectly assigned answers can be given 0 automatically at the discretion of the grader. To assign answers properly, please:
  - Make sure that the top of the first assigned page is the question being graded.
  - Do not include any part of answer to any other questions within the assigned pages.
  - Assigned pages need to be placed in order.
  - For questions with multiple parts, the answers should be written in order of the parts within the question.
- Questions requiring written responses should be short and concise when necessary.Unnecessary wordiness will be penalized at the grader's discretion.
- Please sign the agreement below.
- It is your responsibility to follow updates to the assignment after release. All changes will be visible on Overleaf and Piazza.
- Any questions should be directed towards the TAs for this assignment (theoretical part): *Alireza Dizaji, Pascal Tikeng.*

I acknowledge I have read the above instructions and will abide by them throughout this assignment. I further acknowledge that any assignment submitted without the following form completed will result in no marks being given for this portion of the assignment.

Signature: _____

Name: _____

UdeM Student ID: _____

Question 1 (6pts)**.** (**Normalization**)

Batch normalization, layer normalization and instance normalization all involve calculating the mean $\boldsymbol{\mu}$ and variance $\boldsymbol{\sigma^2}$ with respect to different subsets of the tensor dimensions. Given the following 3D tensor, calculate the corresponding mean and variance tensors for each normalization technique: $\boldsymbol{\mu}_{batch}$, $\boldsymbol{\mu}_{layer}$, $\boldsymbol{\mu}_{instance}$, $\boldsymbol{\sigma}^2_{batch}$, $\boldsymbol{\sigma}^2_{layer}$, and $\boldsymbol{\sigma}^2_{instance}$.

$$\left[\begin{bmatrix}2,5,1\\4,3,4\end{bmatrix},\begin{bmatrix}1,2,4\\1,2,2\end{bmatrix},\begin{bmatrix}3,1,2\\3,2,4\end{bmatrix},\begin{bmatrix}3,4,3\\5,2,3\end{bmatrix},\right]$$

The size of this tensor is 4 x 2 x 3, which corresponds to the batch size, number of input channels, and number of features, respectively.

CODE:

Solution to Normalization Exercise

Python Code for Calculation

```
import numpy as np

Given 3D tensor (Batch Size x Channels x Features)

tensor = np.array([
[[2, 5, 1], [4, 3, 4]],
[[1, 2, 4], [1, 2, 2]],
[[3, 1, 2], [3, 2, 4]],
[[3, 4, 3], [5, 2, 3]]
])

Batch Normalization: Mean and variance computed across batch dimension (axis=0)

mu_batch = np.mean(tensor, axis=0, keepdims=True)
var_batch = np.var(tensor, axis=0, keepdims=True)

Layer Normalization: Mean and variance computed across feature dimension (axis=-1)

mu_layer = np.mean(tensor, axis=-1, keepdims=True)
var_layer = np.var(tensor, axis=-1, keepdims=True)

Instance Normalization: Mean and variance computed per instance across channels and feature

mu_instance = np.mean(tensor, axis=(1, 2), keepdims=True)
```

```
var_instance = np.var(tensor, axis=(1, 2), keepdims=True)

print("Batch Mean:\n", mu_batch)
print("Batch Variance:\n", var_batch)
print("\nLayer Mean:\n", mu_layer)
print("Layer Variance:\n", var_layer)
print("\nInstance Mean:\n", mu_instance)
print("Instance Variance:\n", var_instance)

tensor.shape # (4, 2, 3)
```

Computed Results

Batch Normalization:

$$\boldsymbol{\mu}_{batch} = \begin{bmatrix}2.25 & 3.0 & 2.5\ 3.25 & 2.25 & 3.25\end{bmatrix}$$

$$\boldsymbol{\sigma}^{\mathbf{2}}_{batch} = \begin{bmatrix}0.6875 & 2.5 & 1.25\ 2.1875 & 0.1875 & 0.6875\end{bmatrix}$$

Layer Normalization:

$$\boldsymbol{\mu}_{layer} = \begin{bmatrix}2.67 & 3.67\ 2.33 & 1.67\ 2.00 & 3.00\ 3.33 & 3.00\end{bmatrix}$$

$$\boldsymbol{\sigma}^{\mathbf{2}}_{layer} = \begin{bmatrix}2.89 & 0.22\ 1.56 & 0.22\ 0.67 & 0.67\ 0.22 & 2.00\end{bmatrix}$$

Instance Normalization:

$$\boldsymbol{\mu}_{instance} = \begin{bmatrix}3.17\ 2.00\ 2.50\ 3.33\end{bmatrix}$$

$$\boldsymbol{\sigma}^{\mathbf{2}}_{instance} = \begin{bmatrix}2.14\ 1.11\ 0.92\ 2.22\end{bmatrix}$$

Question 2 (20pts)**.** (**Decoding**)

Suppose that we have a vocabulary containing N possible words, including a special token `<BOS>` to indicate the beginning of a sentence. Recall that in general, a language model with a full context can be written as

$$p(w_1, w_2, \ldots, w_T \mid w_0) = \prod_{t=1}^{T} p(w_t \mid w_0, \ldots, w_{t-1}).$$

We will use the notation $\boldsymbol{w}_{0:t-1}$ to denote the (partial) sequence $(w_0, \ldots, w_{t-1})$. Once we have a fully trained language model, we would like to generate realistic sequences of words from our language model, starting with our special token `<BOS>`. In particular, we might be interested in generating the most likely sequence $\boldsymbol{w}^\star_{1:T}$ under this model, defined as

$$\boldsymbol{w}^\star_{1:T} = \arg\max_{\boldsymbol{w}_{1:T}} p(\boldsymbol{w}_{1:T} \mid w_0 = \texttt{<BOS>}). \tag{1}$$

For clarity we will drop the explicit conditioning on w_0, assuming from now on that the sequences always start with the `<BOS>` token.

1. (2 pts) How many possible sequences of length $T + 1$ starting with the token `<BOS>` can be generated in total? Give an exact expression, without the O notation. Note that the length $T+1$ here includes the `<BOS>` token.
   Answer:
   The total number of possible sequences of length $T + 1$ starting with the token is given by:

   $$N^T.$$

   Since the first word is fixed as , the remaining T positions can each be filled with any of the N words in the vocabulary.

2. (2 pts) To determine the most likely sequence $\boldsymbol{w}^\star_{1:T}$, one could perform exhaustive enumeration of all possible sequences and select the one with the highest joint probability (as defined in equation 1). Comment on the feasibility of this approach. Is it scalable with vocabulary size?
   Answer:
   Exhaustive enumeration involves evaluating all N^T possible sequences and selecting the one with the highest joint probability. However, this approach is not scalable due to the following reasons:
   Time Complexity:
   The computational time grows exponentially with T, as:

   $$O(N^T)$$

   Space Complexity:
   Storing all N^T sequences and their probabilities is infeasible for large N and T.

   Example Consider a case where:

   - Vocabulary size: $N = 10,000$
   - Sequence length: $T = 10$

The total number of sequences to evaluate is:

$$10,000^{10} = 10^{40}$$

This makes exhaustive enumeration computationally unfeasable.

3. (3 pts) In light of the difficulties associated with exhaustive enumeration, it becomes essential to employ practical search strategies to obtain a reasonable approximation of the most likely sequence.

   In order to generate B sequences having high likelihood, one can use a heuristic algorithm called Beam search decoding, whose pseudo-code is given in Algorithm 1 below

Algorithm 1: Beam search decoding

Input: A language model $p(\boldsymbol{w}_{1:T} \mid w_0)$, the beam width B
Output: B sequences $\boldsymbol{w}_{1:T}^{(b)}$ for $b \in \{1, \dots, B\}$
Initialization: $w_0^{(b)} \leftarrow$ `<BOS>` for all $b \in \{1, \dots, B\}$
Initial log-likelihoods: $l_0^{(b)} \leftarrow 0$ for all $b \in \{1, \dots, B\}$
for $t = 1$ **to** T **do**
  for $b = 1$ **to** B **do**
    for $j = 1$ **to** N **do**
      $s_b(j) \leftarrow l_{t-1}^{(b)} + \log p(w_t = j \mid \boldsymbol{w}_{0:t-1}^{(b)})$
  for $b = 1$ **to** B **do**
    Find (b', j) such that $s_{b'}(j)$ is the b-th largest score
    Save the partial sequence b': $\widetilde{\boldsymbol{w}}_{0:t-1}^{(b)} \leftarrow \boldsymbol{w}_{0:t-1}^{(b')}$
    Add the word j to the sequence b: $w_t^{(b)} \leftarrow j$
    Update the log-likelihood: $l_t^{(b)} \leftarrow s_{b'}(j)$
  Assign the partial sequences: $\boldsymbol{w}_{0:t-1}^{(b)} \leftarrow \widetilde{\boldsymbol{w}}_{0:t-1}^{(b)}$ for all $b \in \{1, \dots, B\}$

What is the time complexity of Algorithm 1? Its space complexity? Write the algorithmic complexities using the O notation, as a function of T, B, and N. Is this a practical decoding algorithm when the size of the vocabulary is large?
Answer:

Time Complexity At each time step t, the algorithm computes scores for $B \times N$ candidates, where:

- B is the beam width.
- N is the vocabulary size.

This process is repeated for T time steps. Thus, the overall time complexity is:

$$O(T \cdot B \cdot N)$$

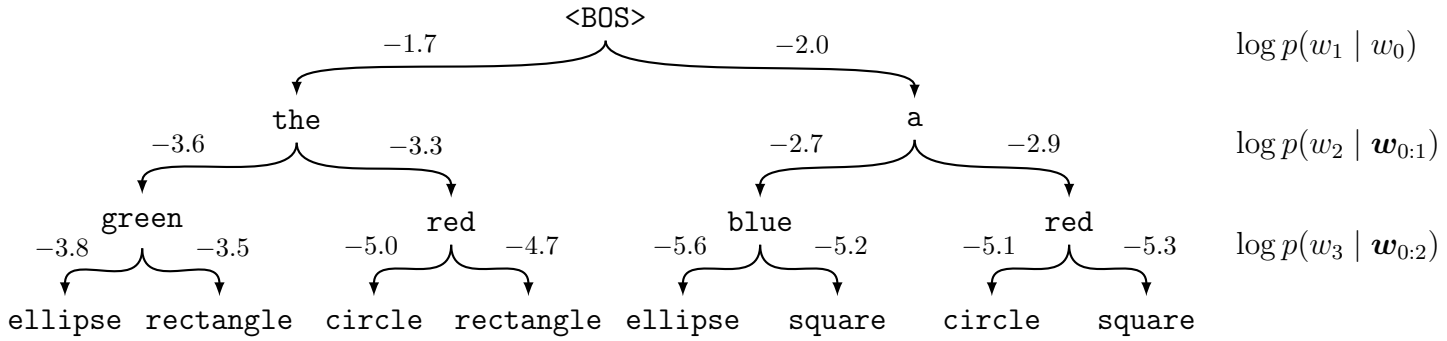
Space Complexity The algorithm needs to store B sequences of length T and their corresponding log-likelihoods. Therefore, the space complexity is:

$$O(B \cdot T)$$

The decoding is practical because :

- **Scalability:** Time complexity $O(T \cdot B \cdot N)$ is linear in N, making it feasible for large vocabularies.
- **Efficiency:** Unlike exhaustive enumeration, beam search reduces the number of evaluated sequences while maintaining high-quality results.
- **Memory Usage:** Space complexity $O(B \cdot T)$ is manageable for reasonable values of B and T.

4. (10 pts) The different sequences that can be generated with a language model can be represented as a tree, where the nodes correspond to words and edges are labeled with the log-probability $\log p(w_t \mid \boldsymbol{w}_{0:t-1})$, depending on the path $\boldsymbol{w}_{0:t-1}$. In this question, consider the following language model (where the low probability paths have been removed for clarity)



(a) (2 pts) Greedy decoding is a simple algorithm where the next word $\overline{w}_t$ is selected by maximizing the conditional probability $p(w_t \mid \overline{\boldsymbol{w}}_{0:t-1})$ (with $\overline{w}_0 =$ <BOS>)

$$\overline{w}_t = \arg\max_{w_t} \log p(w_t \mid \overline{\boldsymbol{w}}_{0:t-1}).$$

Find $\overline{\boldsymbol{w}}_{1:3}$ using greedy decoding on this language model, and its log-likelihood $\log p(\overline{\boldsymbol{w}}_{1:3})$. Greedy decoding selects the word with the highest log-probability at each step. Starting from <BOS>, we choose the next word as:

- $\arg\max$ after <BOS>:

$$p(w_1 \mid \texttt{<BOS>}) = \max(-1.7, -2.0) = -1.7 \quad \Rightarrow \quad w_1 = \text{“the”}$$

- $\arg\max$ after “the”:

$$p(w_2 \mid w_1) = \max(-3.6, -3.3) = -3.3 \quad \Rightarrow \quad w_2 = \text{“red”}$$

- $\arg\max$ after “red”:

$$p(w_3 \mid w_{0:2}) = \max(-5.0, -4.7) = -4.7 \quad \Rightarrow \quad w_3 = \text{“rectangle”}$$

Thus, the greedy decoding sequence is:

$$\overline{\boldsymbol{w}}_{1:3} = (\text{the}, \text{red}, \text{rectangle})$$

with a total log-likelihood:

$$\log p(\overline{\boldsymbol{w}}_{1:3}) = (-1.7) + (-3.3) + (-4.7) = -9.7$$

(b) (8 pts) Apply beam search decoding with a beam width $B = 2$ to this language model, and find $\boldsymbol{w}^{(1)}_{1:3}$ and $\boldsymbol{w}^{(2)}_{1:3}$, together with their respective log-likelihoods.
Beam search maintains the top B sequences at each step.

Step 1: Expanding from `<BOS>`

- "the": -1.7
- "a": -2.0

Top 2 paths: **"the"**, **"a"**

Step 2: Expanding candidates

- "the → green": -5.3
- "the → red": -5.0 (selected)
- "a → blue": -4.7 (selected)
- "a → red": -4.9

Top 2 paths: **"a → blue"** (-4.7), **"the → red"** (-5.0)

Step 3: Expanding candidates

- "a → blue → ellipse": -10.3
- "a → blue → square": -9.9 (selected)
- "the → red → circle": -10.0
- "the → red → rectangle": -9.7 (selected)

Final Sequences:

$$\boldsymbol{w}^{(1)}_{1:3} = (\text{the}, \text{red}, \text{rectangle}), \qquad \log p(\boldsymbol{w}^{(1)}_{1:3}) = -9.7$$
$$\boldsymbol{w}^{(2)}_{1:3} = (\text{a}, \text{blue}, \text{square}), \qquad \log p(\boldsymbol{w}^{(2)}_{1:3}) = -9.9$$

Thus, beam search identifies the same best sequence as greedy decoding but also provides an alternative high-probability sequence.

5. (3 pts) Please highlight the primary limitation that stands out to you for each of the discussed methods (greedy decoding and beam search).
Greedy Decoding can get stuck in suboptimal sequences because it makes locally optimal choices at each step without considering the global structure of the sequence. This myopic decision-making can lead to poor overall performance, especially in cases where a slightly lower-probability choice early on could lead to a much better sequence later.

Beam Search is computationally more expensive than greedy decoding, especially for larger beam widths. Additionally, it can still miss globally optimal sequences if the beam width is too small, as it only explores a limited number of paths. This trade-off between computational cost and search quality is a key challenge.

Question 3 (17pts)**.** (**RNNs**)

Consider the following Bidirectional RNN:

$$\begin{aligned}\boldsymbol{h}_t^{(f)} &= \tanh(\boldsymbol{W}^{(f)}\boldsymbol{x}_t + \boldsymbol{U}^{(f)}\boldsymbol{h}_{t-1}^{(f)})\\ \boldsymbol{h}_t^{(b)} &= \tanh(\boldsymbol{W}^{(b)}\boldsymbol{x}_t + \boldsymbol{U}^{(b)}\boldsymbol{h}_{t+1}^{(b)})\\ \boldsymbol{y}_t &= \boldsymbol{V}^{(f)}\boldsymbol{h}_t^{(f)} + \boldsymbol{V}^{(b)}\boldsymbol{h}_t^{(b)}\end{aligned}$$

where $\boldsymbol{W}^{(f)}, \boldsymbol{W}^{(b)}, \boldsymbol{U}^{(f)}, \boldsymbol{U}^{(b)}, \boldsymbol{V}^{(f)}, \boldsymbol{V}^{(b)} \in \mathbb{R}^{d\times d}$, and $\boldsymbol{x}_i, \boldsymbol{h}_i^{(f)}, \boldsymbol{h}_i^{(b)} \in \mathbb{R}^d, \forall i \in [T]$ where the superscripts f and b correspond to the forward and backward RNNs respectively and tanh denotes the hyperbolic tangent function. Let $\boldsymbol{z}_t$ be the true target of the prediction $\boldsymbol{y}_t$ and consider the sum of squared loss $L = \sum_t L_t$ where $L_t = ||\boldsymbol{z}_t - \boldsymbol{y}_t||_2^2$.

In this question, our goal is to obtain an expression for the gradients $\nabla_{\boldsymbol{W}^{(b)}} L$ and $\nabla_{\boldsymbol{U}^{(f)}} L$.

1. (2 pts) First, complete the following computational graph for this RNN, unrolled for 3 time steps (from $t = 1$ to $t = 3$). Label each node with the corresponding hidden unit and each edge with the corresponding weight. Note that it includes the initial hidden states for both the forward and backward RNNs.

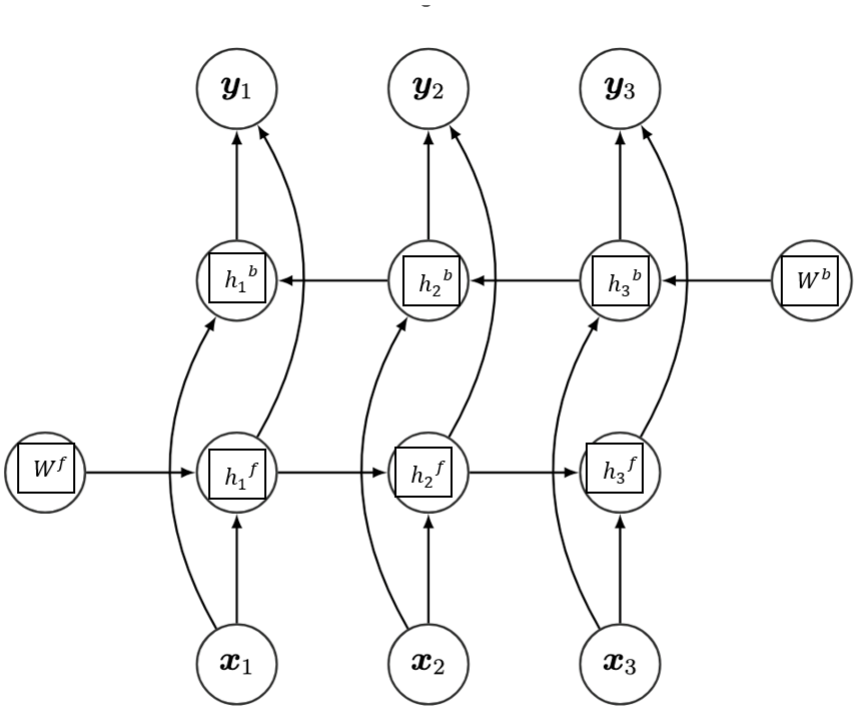


FIGURE 1 – Computational graph of the bidirectional RNN unrolled for three timesteps.

2. (10 pts) Using total derivatives we can express the gradients $\nabla_{\boldsymbol{h}_t^{(f)}} L$ and $\nabla_{\boldsymbol{h}_t^{(b)}} L$ recursively in terms of $\nabla_{\boldsymbol{h}_{t+1}^{(f)}} L$ and $\nabla_{\boldsymbol{h}_{t-1}^{(b)}} L$ as follows:

$$\nabla_{\boldsymbol{h}_t^{(f)}} L = \nabla_{\boldsymbol{h}_t^{(f)}} L_t + \left(\frac{\partial \boldsymbol{h}_{t+1}^{(f)}}{\partial \boldsymbol{h}_t^{(f)}}\right)^\top \nabla_{\boldsymbol{h}_{t+1}^{(f)}} L$$

$$\nabla_{\boldsymbol{h}_t^{(b)}} L = \nabla_{\boldsymbol{h}_t^{(b)}} L_t + \left(\frac{\partial \boldsymbol{h}_{t-1}^{(b)}}{\partial \boldsymbol{h}_t^{(b)}} \right)^\top \nabla_{\boldsymbol{h}_{t-1}^{(b)}} L$$

(a) (8 pts) Derive the expression for $\nabla_{\boldsymbol{h}_t^{(f)}} L_t$, $\nabla_{\boldsymbol{h}_t^{(b)}} L_t$, $\left(\frac{\partial \boldsymbol{h}_{t+1}^{(f)}}{\partial \boldsymbol{h}_t^{(f)}} \right)_{ij}$ and $\left(\frac{\partial \boldsymbol{h}_{t-1}^{(b)}}{\partial \boldsymbol{h}_t^{(b)}} \right)_{ij}$. **Answer**

Solving $\nabla_{\boldsymbol{h}_t^{(f)}} L_t$ and $\nabla_{\boldsymbol{h}_t^{(b)}} L_t$

The loss function is given by:

$$L = \sum_t \|\boldsymbol{z}_t - \boldsymbol{y}_t\|_2^2,$$

where the loss at time step t is:

$$L_t = \|\boldsymbol{z}_t - \boldsymbol{y}_t\|_2^2.$$

Since the output is computed as:

$$\boldsymbol{y}_t = \boldsymbol{V}^{(f)} \boldsymbol{h}_t^{(f)} + \boldsymbol{V}^{(b)} \boldsymbol{h}_t^{(b)},$$

we derive the gradients of L_t with respect to the forward and backward hidden states, $\boldsymbol{h}_t^{(f)}$ and $\boldsymbol{h}_t^{(b)}$, respectively.

Gradient with respect to $\boldsymbol{h}_t^{(f)}$

Applying the chain rule:

$$\nabla_{\boldsymbol{h}_t^{(f)}} L_t = \frac{\partial L_t}{\partial \boldsymbol{y}_t} \frac{\partial \boldsymbol{y}_t}{\partial \boldsymbol{h}_t^{(f)}}.$$

1. Compute $\frac{\partial L_t}{\partial \boldsymbol{y}_t}$:

$$\frac{\partial L_t}{\partial \boldsymbol{y}_t} = \frac{\partial}{\partial \boldsymbol{y}_t} \|\boldsymbol{z}_t - \boldsymbol{y}_t\|_2^2 = 2(\boldsymbol{y}_t - \boldsymbol{z}_t).$$

2. Compute $\frac{\partial \boldsymbol{y}_t}{\partial \boldsymbol{h}_t^{(f)}}$:

$$\frac{\partial \boldsymbol{y}_t}{\partial \boldsymbol{h}_t^{(f)}} = \boldsymbol{V}^{(f)}.$$

3. Combine the results:

$$\nabla_{\boldsymbol{h}_t^{(f)}} L_t = 2\boldsymbol{V}^{(f)\top}(\boldsymbol{y}_t - \boldsymbol{z}_t).$$

Gradient with respect to $\boldsymbol{h}_t^{(b)}$

Similarly, applying the chain rule:

$$\nabla_{\boldsymbol{h}_t^{(b)}} L_t = \frac{\partial L_t}{\partial \boldsymbol{y}_t} \frac{\partial \boldsymbol{y}_t}{\partial \boldsymbol{h}_t^{(b)}}.$$

1. Compute $\frac{\partial L_t}{\partial \boldsymbol{y}_t}$ (same as above):

$$\frac{\partial L_t}{\partial \boldsymbol{y}_t} = 2(\boldsymbol{y}_t - \boldsymbol{z}_t).$$

2. Compute $\frac{\partial \boldsymbol{y}_t}{\partial \boldsymbol{h}_t^{(b)}}$:

$$\frac{\partial \boldsymbol{y}_t}{\partial \boldsymbol{h}_t^{(b)}} = \boldsymbol{V}^{(b)}.$$

3. Combine the results:

$$\nabla_{\boldsymbol{h}_t^{(b)}} L_t = 2\boldsymbol{V}^{(b)^\top}(\boldsymbol{y}_t - \boldsymbol{z}_t).$$

Final Expressions:

$$\nabla_{\boldsymbol{h}_t^{(f)}} L_t = 2\boldsymbol{V}^{(f)^\top}(\boldsymbol{y}_t - \boldsymbol{z}_t),$$
$$\nabla_{\boldsymbol{h}_t^{(b)}} L_t = 2\boldsymbol{V}^{(b)^\top}(\boldsymbol{y}_t - \boldsymbol{z}_t).$$

Solving $\left(\frac{\partial \boldsymbol{h}_{t+1}^{(f)}}{\partial \boldsymbol{h}_t^{(f)}}\right)_{ij}$ and $\left(\frac{\partial \boldsymbol{h}_{t-1}^{(b)}}{\partial \boldsymbol{h}_t^{(b)}}\right)_{ij}$
The forward and backward hidden state update equations are given by:

$$\boldsymbol{h}_t^{(f)} = \tanh(\boldsymbol{W}^{(f)}\boldsymbol{x}_t + \boldsymbol{U}^{(f)}\boldsymbol{h}_{t-1}^{(f)})$$
$$\boldsymbol{h}_t^{(b)} = \tanh(\boldsymbol{W}^{(b)}\boldsymbol{x}_t + \boldsymbol{U}^{(b)}\boldsymbol{h}_{t+1}^{(b)}).$$

Derivative of Forward Hidden State: Taking the derivative of the forward hidden state equation with respect to $\boldsymbol{h}_t^{(f)}$:

$$\frac{\partial \boldsymbol{h}_{t+1}^{(f)}}{\partial \boldsymbol{h}_t^{(f)}} = \frac{\partial \tanh(\boldsymbol{W}^{(f)}\boldsymbol{x}_{t+1} + \boldsymbol{U}^{(f)}\boldsymbol{h}_t^{(f)})}{\partial \boldsymbol{h}_t^{(f)}}.$$

Using the derivative of the hyperbolic tangent function:

$$\frac{d}{dz}\tanh(z) = 1 - \tanh^2(z),$$

we apply the chain rule:

$$\frac{\partial \boldsymbol{h}_{t+1}^{(f)}}{\partial \boldsymbol{h}_t^{(f)}} = \operatorname{diag}(1 - \boldsymbol{h}_{t+1}^{(f)} \odot \boldsymbol{h}_{t+1}^{(f)}) \cdot \boldsymbol{U}^{(f)}.$$

Thus, the element-wise expression is:

$$\left(\frac{\partial \boldsymbol{h}_{t+1}^{(f)}}{\partial \boldsymbol{h}_t^{(f)}}\right)_{ij} = (1 - (\boldsymbol{h}_{t+1}^{(f)})_i^2)U_{ij}^{(f)}.$$

Derivative of Backward Hidden State: Similarly, taking the derivative of the backward hidden state equation with respect to $\boldsymbol{h}_t^{(b)}$:

$$\frac{\partial \boldsymbol{h}_{t-1}^{(b)}}{\partial \boldsymbol{h}_t^{(b)}} = \frac{\partial \tanh(\boldsymbol{W}^{(b)}\boldsymbol{x}_{t-1} + \boldsymbol{U}^{(b)}\boldsymbol{h}_t^{(b)})}{\partial \boldsymbol{h}_t^{(b)}}.$$

Applying the chain rule:

$$\frac{\partial \boldsymbol{h}_{t-1}^{(b)}}{\partial \boldsymbol{h}_t^{(b)}} = \operatorname{diag}(1 - \boldsymbol{h}_{t-1}^{(b)} \odot \boldsymbol{h}_{t-1}^{(b)}) \cdot \boldsymbol{U}^{(b)}.$$

Thus, the element-wise expression is:

$$\left(\frac{\partial \boldsymbol{h}_{t-1}^{(b)}}{\partial \boldsymbol{h}_t^{(b)}}\right)_{ij} = (1 - (\boldsymbol{h}_{t-1}^{(b)})_i^2) U_{ij}^{(b)}.$$

Final Expressions:

$$\left(\frac{\partial \boldsymbol{h}_{t+1}^{(f)}}{\partial \boldsymbol{h}_t^{(f)}}\right)_{ij} = (1 - (\boldsymbol{h}_{t+1}^{(f)})_i^2) U_{ij}^{(f)},$$
$$\left(\frac{\partial \boldsymbol{h}_{t-1}^{(b)}}{\partial \boldsymbol{h}_t^{(b)}}\right)_{ij} = (1 - (\boldsymbol{h}_{t-1}^{(b)})_i^2) U_{ij}^{(b)}.$$

(b) (2 pts) Now using prev section, drive the expression for Jacobian matrices $\frac{\partial \boldsymbol{h}_{t+1}^{(f)}}{\partial \boldsymbol{h}_t^{(f)}}$ and $\frac{\partial \boldsymbol{h}_{t-1}^{(b)}}{\partial \boldsymbol{h}_t^{(b)}}$.
Answered in previous section?
Solving $\left(\frac{\partial \boldsymbol{h}_{t+1}^{(f)}}{\partial \boldsymbol{h}_t^{(f)}}\right)_{ij}$ and $\left(\frac{\partial \boldsymbol{h}_{t-1}^{(b)}}{\partial \boldsymbol{h}_t^{(b)}}\right)_{ij}$
The forward and backward hidden state update equations are given by:

$$\boldsymbol{h}_t^{(f)} = \tanh(\boldsymbol{W}^{(f)} \boldsymbol{x}_t + \boldsymbol{U}^{(f)} \boldsymbol{h}_{t-1}^{(f)})$$
$$\boldsymbol{h}_t^{(b)} = \tanh(\boldsymbol{W}^{(b)} \boldsymbol{x}_t + \boldsymbol{U}^{(b)} \boldsymbol{h}_{t+1}^{(b)}).$$

Derivative of Forward Hidden State: Taking the derivative of the forward hidden state equation with respect to $\boldsymbol{h}_t^{(f)}$:

$$\frac{\partial \boldsymbol{h}_{t+1}^{(f)}}{\partial \boldsymbol{h}_t^{(f)}} = \frac{\partial \tanh(\boldsymbol{W}^{(f)} \boldsymbol{x}_{t+1} + \boldsymbol{U}^{(f)} \boldsymbol{h}_t^{(f)})}{\partial \boldsymbol{h}_t^{(f)}}.$$

Using the derivative of the hyperbolic tangent function:

$$\frac{d}{dz} \tanh(z) = 1 - \tanh^2(z),$$

we apply the chain rule:

$$\frac{\partial \boldsymbol{h}_{t+1}^{(f)}}{\partial \boldsymbol{h}_t^{(f)}} = \operatorname{diag}(1 - \boldsymbol{h}_{t+1}^{(f)} \odot \boldsymbol{h}_{t+1}^{(f)}) \cdot \boldsymbol{U}^{(f)}.$$

Thus, the element-wise expression is:

$$\left(\frac{\partial \boldsymbol{h}_{t+1}^{(f)}}{\partial \boldsymbol{h}_t^{(f)}}\right)_{ij} = (1 - (\boldsymbol{h}_{t+1}^{(f)})_i^2) U_{ij}^{(f)}.$$

Derivative of Backward Hidden State: Similarly, taking the derivative of the backward hidden state equation with respect to $\boldsymbol{h}_t^{(b)}$:

$$\frac{\partial \boldsymbol{h}_{t-1}^{(b)}}{\partial \boldsymbol{h}_t^{(b)}} = \frac{\partial \tanh(\boldsymbol{W}^{(b)} \boldsymbol{x}_{t-1} + \boldsymbol{U}^{(b)} \boldsymbol{h}_t^{(b)})}{\partial \boldsymbol{h}_t^{(b)}}.$$

Applying the chain rule:

$$\frac{\partial \boldsymbol{h}_{t-1}^{(b)}}{\partial \boldsymbol{h}_t^{(b)}} = \text{diag}(1 - \boldsymbol{h}_{t-1}^{(b)} \odot \boldsymbol{h}_{t-1}^{(b)}) \cdot \boldsymbol{U}^{(b)}.$$

Thus, the element-wise expression is:

$$\left(\frac{\partial \boldsymbol{h}_{t-1}^{(b)}}{\partial \boldsymbol{h}_t^{(b)}}\right)_{ij} = (1 - (\boldsymbol{h}_{t-1}^{(b)})_i^2) U_{ij}^{(b)}.$$

Final Expressions:

$$\left(\frac{\partial \boldsymbol{h}_{t+1}^{(f)}}{\partial \boldsymbol{h}_t^{(f)}}\right)_{ij} = (1 - (\boldsymbol{h}_{t+1}^{(f)})_i^2) U_{ij}^{(f)},$$
$$\left(\frac{\partial \boldsymbol{h}_{t-1}^{(b)}}{\partial \boldsymbol{h}_t^{(b)}}\right)_{ij} = (1 - (\boldsymbol{h}_{t-1}^{(b)})_i^2) U_{ij}^{(b)}.$$

3. (5 pts) Now derive $\nabla_{\boldsymbol{W}^{(b)}} L$ and $\nabla_{\boldsymbol{U}^{(f)}} L$ as functions of $\nabla_{\boldsymbol{h}_t^{(b)}} L$ and $\nabla_{\boldsymbol{h}_t^{(f)}} L$, respectively.
Hint: It might be useful to consider the contribution of the weight matrices when computing the recurrent hidden unit at a particular time t and how those contributions might be aggregated.

Deriving $\nabla_{\boldsymbol{W}^{(b)}} L$ and $\nabla_{\boldsymbol{U}^{(f)}} L$

We aim to compute the gradients of the loss function with respect to the weight matrices $\boldsymbol{W}^{(b)}$ and $\boldsymbol{U}^{(f)}$, which govern the backward and forward hidden state updates, respectively.

Gradient of L with respect to $\boldsymbol{W}^{(b)}$

The backward hidden state update equation is:

$$\boldsymbol{h}_t^{(b)} = \tanh(\boldsymbol{W}^{(b)} \boldsymbol{x}_t + \boldsymbol{U}^{(b)} \boldsymbol{h}_{t+1}^{(b)}).$$

Applying the chain rule, the gradient of the loss with respect to $\boldsymbol{W}^{(b)}$ is:

$$\nabla_{\boldsymbol{W}^{(b)}} L = \sum_t \frac{\partial L}{\partial \boldsymbol{h}_t^{(b)}} \frac{\partial \boldsymbol{h}_t^{(b)}}{\partial \boldsymbol{W}^{(b)}}.$$

Since:

$$\frac{\partial \boldsymbol{h}_t^{(b)}}{\partial \boldsymbol{W}^{(b)}} = \text{diag}(1 - \boldsymbol{h}_t^{(b)} \odot \boldsymbol{h}_t^{(b)}) \boldsymbol{x}_t^\top,$$

we obtain:

$$\nabla_{\boldsymbol{W}^{(b)}} L = \sum_t \text{diag}(1 - \boldsymbol{h}_t^{(b)} \odot \boldsymbol{h}_t^{(b)}) \nabla_{\boldsymbol{h}_t^{(b)}} L \boldsymbol{x}_t^\top.$$

Gradient of L with respect to $\boldsymbol{U}^{(f)}$

The forward hidden state update equation is:

$$\boldsymbol{h}_t^{(f)} = \tanh(\boldsymbol{W}^{(f)} \boldsymbol{x}_t + \boldsymbol{U}^{(f)} \boldsymbol{h}_{t-1}^{(f)}).$$

Applying the chain rule, the gradient of the loss with respect to $\boldsymbol{U}^{(f)}$ is:

$$\nabla_{\boldsymbol{U}^{(f)}} L = \sum_t \frac{\partial L}{\partial \boldsymbol{h}_t^{(f)}} \frac{\partial \boldsymbol{h}_t^{(f)}}{\partial \boldsymbol{U}^{(f)}}.$$

Since:

$$\frac{\partial \boldsymbol{h}_t^{(f)}}{\partial \boldsymbol{U}^{(f)}} = \text{diag}(1 - \boldsymbol{h}_t^{(f)} \odot \boldsymbol{h}_t^{(f)}) \boldsymbol{h}_{t-1}^{(f)\top},$$

we obtain:

$$\nabla_{\boldsymbol{U}^{(f)}} L = \sum_t \text{diag}(1 - \boldsymbol{h}_t^{(f)} \odot \boldsymbol{h}_t^{(f)}) \nabla_{\boldsymbol{h}_t^{(f)}} L \boldsymbol{h}_{t-1}^{(f)\top}.$$

Final Expressions:

$$\nabla_{\boldsymbol{W}^{(b)}} L = \sum_t \text{diag}(1 - \boldsymbol{h}_t^{(b)} \odot \boldsymbol{h}_t^{(b)}) \nabla_{\boldsymbol{h}_t^{(b)}} L \boldsymbol{x}_t^\top,$$

$$\nabla_{\boldsymbol{U}^{(f)}} L = \sum_t \text{diag}(1 - \boldsymbol{h}_t^{(f)} \odot \boldsymbol{h}_t^{(f)}) \nabla_{\boldsymbol{h}_t^{(f)}} L \boldsymbol{h}_{t-1}^{(f)\top}.$$

Question 4 (18pts)**.** (**Low-rank adaptation**)

In this question, the goal is to study the low-rank adaptation techniques. Low-rank adaptation techniques are parameter-efficient fine-tuning methods that adapt large pre-trained models to specific tasks by adding small, trainable low-rank matrices to the frozen base weights, which is a common practice to reduce the memory usage with slight to zero changes on the performance. Here, you

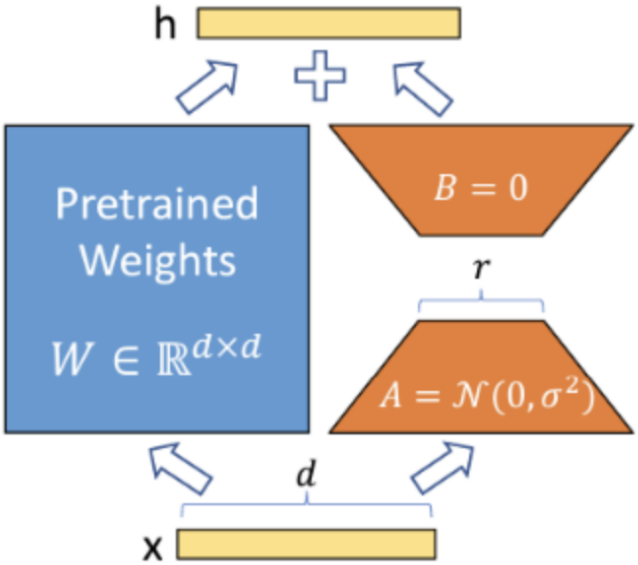


Figure 1: LoRA architecture

will study two well-known low-rank adaptation techinques, LoRA and QLoRA; You can access their papers by clicking here and here. Provide your answers for the following questions:

1. (2 pts) How do QLoRA and LoRA differ in their implementation of low-rank updates?
ANSWER:
While LoRA stands for *Low-Rank Adaptation*, QLoRA stands for *Quantized Low-Rank Adaptation*. Like LoRA, it is a parameter-efficient fine-tuning method that adapts large pre-trained models to specific tasks by introducing small trainable low-rank matrices. However, they differ in implementation, particularly in terms of quantization and memory efficiency.

Key Differences Between LoRA and QLoRA:

Feature	**LoRA**	**QLoRA**
Quantization	No quantization of base model	Base model quantized to 4-bit (NF4)
Memory Efficiency	Reduces trainable parameters, but base model remains in full precision	Further reduces memory via 4-bit quantization and Double Quantization
Adapter Placement	Typically added to specific layers (e.g., attention layers)	Added to all linear layers of the Transformer
Performance	Matches full fine-tuning performance	Matches 16-bit fine-tuning performance despite 4-bit quantization
Use Cases	Suitable for large models with moderate memory constraints	Suitable for very large models on limited hardware (e.g., single GPU)

2. (2 pts) How does the choice of rank affect the performance and memory usage in both QLoRA and LoRA?
Answer:

   (a) **Higher rank:** Better performance, but increased memory usage.

   (b) **Lower rank:** Reduced memory usage, but potential underperformance on complex tasks.

   (c) **QLoRA:** Leverages quantization to support higher ranks, enabling fine-tuning of large models with limited memory.

3. (2 pts) What are the trade-offs in speed and latency between QLoRA and LoRA during inference?

Answer:
The choice between LoRA and QLoRA during inference involves trade-offs in speed, latency, and memory usage. LoRA offers faster inference with no additional latency, making it ideal for real-time applications, but at the cost of higher memory usage. QLoRA reduces memory usage significantly but incurs higher latency and slower inference due to dequantization and additional computations for the low-rank adapters.

Key Trade-offs in Speed and Latency:

Aspect	**LoRA**	**QLoRA**
Speed	Minimal overhead; similar to original model	Slower due to dequantization and adapter computations
Latency	No additional latency (adapters can be merged)	Higher latency due to dequantization and separate adapter computations
Memory Usage	Higher (base model in full precision)	Lower (base model in 4-bit precision)
Inference Cost	Low computational cost	Higher computational cost

4. (12 pts) In this section, you will examine the differences in time and space complexities when applying LoRA and QLoRA for fine-tuning a Transformer architecture that includes Linear Feed-Forward network (LFFN) and multi-head attention (MHA) layers:

$$\begin{aligned}
&\boldsymbol{H}^l = \text{LFFN}^{(l)}(\text{MHA}^{(l)}(\boldsymbol{H}^{l-1})) \\
&\text{LFFN}^{(l)}(\boldsymbol{X}) = \boldsymbol{X}\boldsymbol{W}^{F,l} + \boldsymbol{X}\boldsymbol{A}^{F,l}(\boldsymbol{B}^{F,l})^T \\
&\text{MHA}^{(l)}(\boldsymbol{H}) = [\text{head}^{l,1}(\boldsymbol{H}), \text{head}^{l,2}(\boldsymbol{H}), ..., \text{head}^{l,h}(\boldsymbol{H})]\boldsymbol{W}^{O,l} \\
&\text{head}^{l,i}(\boldsymbol{H}) = softmax(\frac{\boldsymbol{Q}^{l,i}(\boldsymbol{K}^{l,i})^T}{\sqrt{d}})\boldsymbol{V}^{l,i} \\
&\boldsymbol{Q}^{l,i} = \boldsymbol{H}\boldsymbol{W}^{Q,l,i} + \boldsymbol{H}\boldsymbol{A}^{Q,l,i}(\boldsymbol{B}^{Q,l,i})^T \\
&\boldsymbol{K}^{l,i} = \boldsymbol{H}\boldsymbol{W}^{K,l,i} + \boldsymbol{H}\boldsymbol{A}^{K,l,i}(\boldsymbol{B}^{K,l,i})^T \\
&\boldsymbol{V}^{l,i} = \boldsymbol{H}\boldsymbol{W}^{V,l,i} + \boldsymbol{H}\boldsymbol{A}^{V,l,i}(\boldsymbol{B}^{V,l,i})^T
\end{aligned}$$

All pretrained parameters ($\boldsymbol{W}^*$) are frozen, where $\boldsymbol{W}^{Q,*}, \boldsymbol{W}^{K,*}, \boldsymbol{W}^{V,*}, \boldsymbol{W}^{F,*} \in \mathbb{R}^{d\times d}$, while $W^{O,*} \in \mathbb{R}^{dh\times d}$, and $\boldsymbol{A}^*, \boldsymbol{B}^* \in \mathbb{R}^{d\times r}$. By default, the precision used is 32-bit floating point (FP), while QLoRA first quantizes the pretrained parameters to m-bit integer precision for fine-tuning, where the quantization of a single matrix $\boldsymbol{W}$ is formulated as following:

$$\tilde{\boldsymbol{W}} = \text{quant}(\boldsymbol{W}) = \left\lfloor \frac{\boldsymbol{W}}{\Delta} \right\rfloor, \Delta = \frac{max(|\boldsymbol{W}|)}{2^{(m-1)} - 1}$$

and the dequantization of the matrix $\tilde{\boldsymbol{W}}$ to the original precision is computed by:

$$\boldsymbol{W} = \text{dequant}(\Delta, \tilde{\boldsymbol{W}}) = \Delta\tilde{\boldsymbol{W}}$$

Given the embedding dimension d, low-rank dimension $r \ll d$, h heads, and the context length n, in a single forward pass:

(a) (9 pts) Compute the time and space complexities of applying QLoRA and LoRA to FFN and MHA separately. In this problem, space complexity includes only the trainable parameters. *Hint: Ensure that operands at each stage are in the same precision.*
Let n denote the sequence length (number of tokens), d the hidden dimension, r the LoRA rank (with $r \ll d$), and h the number of attention heads. The space complexity refers solely to the additional trainable parameters introduced by LoRA/QLoRA. The time complexity is computed for a single forward pass, ensuring all operands are in the same precision during computation.

- **FFN – LoRA**
  LoRA introduces two trainable low-rank matrices $A \in \mathbb{R}^{d\times r}$ and $B \in \mathbb{R}^{r\times d}$ per linear projection, resulting in a space complexity of $2dr$. The time complexity for the original FFN is $\mathcal{O}(nd^2)$. The additional LoRA term adds $\mathcal{O}(ndr)$, yielding a total time complexity of $\mathcal{O}(nd^2 + ndr)$.
- **FFN – QLoRA**
  QLoRA shares the same trainable parameter structure as LoRA ($2dr$), so the space complexity is identical. However, the pretrained weight matrix is quantized (e.g., to 4-bit) and dequantized during computation. Dequantization incurs a one-time cost of $\mathcal{O}(d^2)$, but since the actual computation is still done in float32, the total forward pass time remains $\mathcal{O}(nd^2 + ndr)$, same as LoRA.

- **MHA – LoRA**
  For each attention head, LoRA is applied to the query, key, and value projections. Each projection introduces $2dr$ trainable parameters, resulting in $6hdr$ total for h heads. The time complexity includes: projection computations $\mathcal{O}(3nd^2 + 3ndr)$, attention score and value multiplication $\mathcal{O}(n^2d)$, and the final output projection $\mathcal{O}(nd^2h)$. This gives a total time complexity of $\mathcal{O}(h(nd^2 + n^2d) + nd^2h)$.
- **MHA – QLoRA**
  QLoRA again retains the same trainable space complexity as LoRA: $6hdr$. Quantized pretrained weights for Q/K/V projections are dequantized to FP32 for use in matrix multiplications. Dequantization adds a minor cost of $\mathcal{O}(3d^2)$, but the primary operations are the same. Thus, the overall time complexity remains $\mathcal{O}(h(nd^2 + n^2d) + nd^2h)$.

Summary:

Layer	**Method**	**Time Complexity / Space Complexity**
FFN	LoRA / QLoRA	$\mathcal{O}(nd^2 + ndr)$ / $2dr$
MHA	LoRA / QLoRA	$\mathcal{O}(h(nd^2 + n^2d) + nd^2h)$ / $6hdr$

(b) (3 pts) If you were assigned to fine-tune using low-rank adapters and had the choice of applying either LoRA or QLoRA to each of these two modules, under the following conditions, which would you select and why? Fill in the table below based on your answer.
Answer: Under memory-constrained scenarios, QLoRA is preferred because it quantizes the pretrained weights (e.g., to 4-bit), significantly reducing memory usage. For inference speed, LoRA is slightly better due to the absence of dequantization overhead. However, when both constraints matter, QLoRA strikes the best balance by enabling efficient fine-tuning with negligible speed loss and substantial memory savings.

	MHA	LFFN
Memory limit	QLoRA	QLoRA
Faster inference	LoRA	LoRA
Both	QLoRA	QLoRA