VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING


## Natural Language Processing - Exercise (CO3086)

---

## Lab 11

# Math Exercises

**Semester 2, Academic Year 2024 - 2025**

---

Teacher: Bui Khanh Vinh
Students: Nguyen Quang Phu - 2252621

HO CHI MINH CITY, APRIL 2025

# Contents

# Chapter 1

# Problem Description and Solution

## 1.1 Problem 1

### 1.1.1 Description

Why are vanishing or exploding gradients an issue for RNNs?

### 1.1.2 Solution

Vanishing or exploding gradients are major issues in training RNNs due to the nature of backpropagation through time (BPTT). These problems occur when the gradients calculated during backpropagation either decrease rapidly (vanishing gradients) or increase rapidly (exploding gradients) as they are propagated through many time steps.

**Vanishing Gradients**:

- **Difficulty in Learning Long-Term Dependencies**: When gradients become very small, the network struggles to propagate information through multiple time steps, making it difficult to learn long-term dependencies.
- **Weight Updates Become Negligible**: Extremely small gradients lead to negligible updates during backpropagation, resulting in very slow or stalled training.
- **Poor Gradient Flow**: Information fails to flow effectively through the network, leading to degraded performance and limited learning capacity.

**Exploding Gradients**:

- **Unstable Training**: Extremely large gradients cause weights to grow exponentially during training, leading to numerical instability and potentially causing the model to diverge.

- **Loss Function Saturation**: High gradients result in rapidly increasing loss values, making it difficult for the model to converge to a minimum.
- **Ineffective Learning**: Learning becomes highly unstable, with oscillating or diverging loss curves, making it hard to achieve optimal performance.

**Common Cause**: Both issues arise due to the repeated multiplication of gradients during backpropagation through time (BPTT), especially when using activation functions with steep or flat gradients (e.g., sigmoid, tanh).

**Additional Information**: Mitigation Techniques:

- **Gradient Clipping**: Limits gradient values within a threshold to prevent explosion.
- **Using Gated Architectures**: Models like LSTMs and GRUs help regulate the gradient flow by introducing gating mechanisms.
- **Proper Weight Initialization**: Techniques like Xavier or He initialization help stabilize training.
- **Batch Normalization**: Normalizing activations helps prevent extreme gradients.

## 1.2 Problem 2

### 1.2.1 Description

**GRUs.** In class, we learned about RNNs and an extension — Gated Recurrent Units. GRUs can adaptively reset or update its "memory" of previous states. The feedforward computation for a GRU is given by

$$z_t = \sigma(W_z x_t + U_z h_{t-1})$$

$$r_t = \sigma(W_r x_t + U_r h_{t-1})$$

$$\hat{h}_t = \tanh(W x_t + r_t \circ U h_{t-1})$$

$$h_t = (1 - z_t) \circ h_{t-1} + z_t \circ \hat{h}_t$$

(a) Show that for the sigmoid function $\sigma(x) = \frac{1}{1+\exp(-x)}$, $\sigma(-x) = 1 - \sigma(x)$.

(b) True/False. If the update gate $z_t$ is close to 0, the net does not update its state significantly. (Explain)

(c) True/False. If the update gate $z_t$ is close to 1 and the reset gate $r_t$ is close to 0, the net remembers the past state very well. (Explain)

### 1.2.2 Solution

**(a) Show that for the sigmoid function $\sigma(x) = \frac{1}{1+\exp(-x)}$, it holds that $\sigma(-x) = 1 - \sigma(x)$.**

To prove this identity, we start from the definition of the sigmoid function:

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

Now, substituting $-x$ into the function:

$$\sigma(-x) = \frac{1}{1 + \exp(x)}$$

Next, we express $\sigma(-x)$ in terms of $\sigma(x)$. Consider the reciprocal form of $\sigma(x)$:

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

Rearranging this equation:

$$1 - \sigma(x) = 1 - \frac{1}{1 + \exp(-x)}$$

$$= \frac{(1 + \exp(-x)) - 1}{1 + \exp(-x)}$$

$$= \frac{\exp(-x)}{1 + \exp(-x)}$$

Multiplying numerator and denominator by $\exp(x)$:

$$= \frac{1}{1 + \exp(x)}$$

which is exactly $\sigma(-x)$, proving that:

$$\sigma(-x) = 1 - \sigma(x)$$

This property is useful in various neural network optimizations, including GRUs and back-propagation, as it simplifies computations involving the sigmoid function.

**(b) True/False. If the update gate $z_t$ is close to 0, the net does not update its state significantly.**

**True.** The update equation for the hidden state in a GRU is:

$$h_t = (1 - z_t) \circ h_{t-1} + z_t \circ \hat{h}_t$$

where $z_t$ is the update gate. If $z_t \approx 0$, then:

$$h_t \approx (1 - 0) \circ h_{t-1} + 0 \circ \hat{h}_t = h_{t-1}$$

This means that the hidden state $h_t$ remains almost unchanged from $h_{t-1}$. In this case:

- The update gate $z_t$ prevents the network from incorporating new information from $x_t$.
- The previous hidden state $h_{t-1}$ is retained, leading to minimal updates.
- This allows GRUs to preserve long-term dependencies when needed, avoiding unnecessary overwriting of useful information.

Thus, when $z_t \approx 0$, the GRU effectively ignores the current input and maintains the previous memory, making minimal updates.

**(c) True/False. If the update gate $z_t$ is close to 1 and the reset gate $r_t$ is close to 0, the net remembers the past state very well.**

**False.** If $z_t \approx 1$, the update equation becomes:

$$h_t = (1 - 1) \circ h_{t-1} + 1 \circ \hat{h}_t = \hat{h}_t$$

This means that the new hidden state $h_t$ is completely replaced by the candidate hidden state $\hat{h}_t$, discarding $h_{t-1}$.

Additionally, the reset gate $r_t$ affects the computation of $\hat{h}_t$:

$$\hat{h}_t = \tanh(Wx_t + r_t \circ Uh_{t-1})$$

If $r_t \approx 0$, then:

$$r_t \circ Uh_{t-1} \approx 0$$

which simplifies to:

$$\hat{h}_t = \tanh(Wx_t)$$

This means that $\hat{h}_t$ is determined solely by the input $x_t$, with no contribution from $h_{t-1}$. Therefore:

- The past state $h_{t-1}$ is not remembered at all.
- The network strongly depends on the current input $x_t$.
- The memory reset eliminates previous information, making the GRU behave more like a feedforward network.

Thus, the statement is false because when $z_t \approx 1$ and $r_t \approx 0$, the past state is completely overwritten by new information, rather than being remembered.

## 1.3 Problem 3

### 1.3.1 Description

Here are the defining equations for a LSTM cell.

$$i_t = \sigma(W^{(i)}x_t + U^{(i)}h_{t-1})$$

$$f_t = \sigma(W^{(f)}x_t + U^{(f)}h_{t-1})$$

$$o_t = \sigma(W^{(o)}x_t + U^{(o)}h_{t-1})$$

$$\tilde{c}_t = \tanh\left(W^{(c)}x_t + U^{(c)}h_{t-1}\right)$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$$

$$h_t = o_t \circ \tanh(c_t)$$

Recall that $\circ$ denotes element-wise multiplication and that $\sigma$ denotes the sigmoid function.

(a) (True/False) If $x_t$ is the 0 vector, then $h_t = h_{t-1}$. (Explain)

(b) (True/False) If $f_t$ is very small or zero, then error will not be back-propagated to earlier time steps. (Explain)

(c) (True/False) The entries of $f_t, i_t, o_t$ are non-negative. (Explain)

(d) (2 points) (True/False) $f_t, i_t, o_t$ can be viewed as probability distributions. (i.e., their entries are non-negative and their entries sum to 1.) (Explain)

### 1.3.2 Solution

(a) (2 points) (True/False) If $x_t$ is the 0 vector, then $h_t = h_{t-1}$.

**False.** While it might seem that if the input $x_t$ is zero, the hidden state remains unchanged, this is generally not true. The LSTM update equations involve **learnable weights** $(W, U)$ and **non-linear transformations** (sigmoid and tanh functions), which still operate even when $x_t = 0$.

- The forget gate $f_t$, input gate $i_t$, and output gate $o_t$ depend on the previous hidden state $h_{t-1}$.

- Even if $x_t = 0$, the memory cell update equation:

$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$$

still modifies $c_t$, which in turn affects $h_t = o_t \circ \tanh(c_t)$.

Thus, $h_t \neq h_{t-1}$ in general, since the transformations applied to $h_{t-1}$ still occur.

(b) (2 points) (True/False) If $f_t$ is very small or zero, then error will not be back-propagated to earlier time steps.

**False.** Even if $f_t$ is very small or zero, error can still be back-propagated to earlier time steps because the input gate $i_t$ and the candidate cell state $\tilde{c}_t$ both depend on the hidden state $h_{t-1}$. Therefore, gradients can still flow backward through these connections even when the forget gate $f_t$ is near zero.

(c) (2 points) (True/False) The entries of $f_t, i_t, o_t$ are non-negative.

**True.** Each of these gates is computed using the sigmoid activation function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Since the sigmoid function outputs values in the range $(0, 1)$, all entries of $f_t, i_t, o_t$ are strictly between 0 and 1. This ensures:

- The forget gate $f_t$ never makes the memory cell completely negative.
- The input gate $i_t$ properly scales new candidate memory values.
- The output gate $o_t$ correctly modulates the hidden state $h_t$.

Thus, all entries of $f_t, i_t, o_t$ are non-negative by design.

(d) (2 points) (True/False) $f_t, i_t, o_t$ can be viewed as probability distributions. (i.e., their entries are non-negative and their entries sum to 1.)

**False.** Although entries of $f_t, i_t, o_t$ are **non-negative**, they **do not necessarily sum to 1**.

- A probability distribution requires that all elements sum to 1, but in an LSTM, each element of $f_t, i_t, o_t$ is independently computed using the sigmoid function.
- Since the sigmoid function is applied **element-wise**, the sum of entries across a vector is not constrained to 1.

Thus, these gates are not probability distributions; rather, they act as independent gating mechanisms to control the flow of information in the LSTM.

**Summary of Answers:**

- (a) **False** – $h_t \neq h_{t-1}$ even if $x_t = 0$.
- (b) **False** – Small $f_t$ leads to vanishing gradients.
- (c) **True** – Sigmoid ensures $f_t, i_t, o_t \in (0, 1)$.
- (d) **False** – These are not probability distributions.

## 1.4 Problem 4

### 1.4.1 Description

To address the problem of vanishing and exploding gradients, we can use a different kind of recurrent cell – the LSTM cell (standing for "long short term memory"). The layout of the cell is shown in Figure 4. The LSTM has two states which are passed between timesteps: a "cell memory" $C$ and the hidden state $h$. The LSTM update is given as follows:

$$f_t = \sigma(x_t W_f + h_{t-1} W'_f)$$

$$i_t = \sigma(x_t W_i + h_{t-1} W'_i)$$

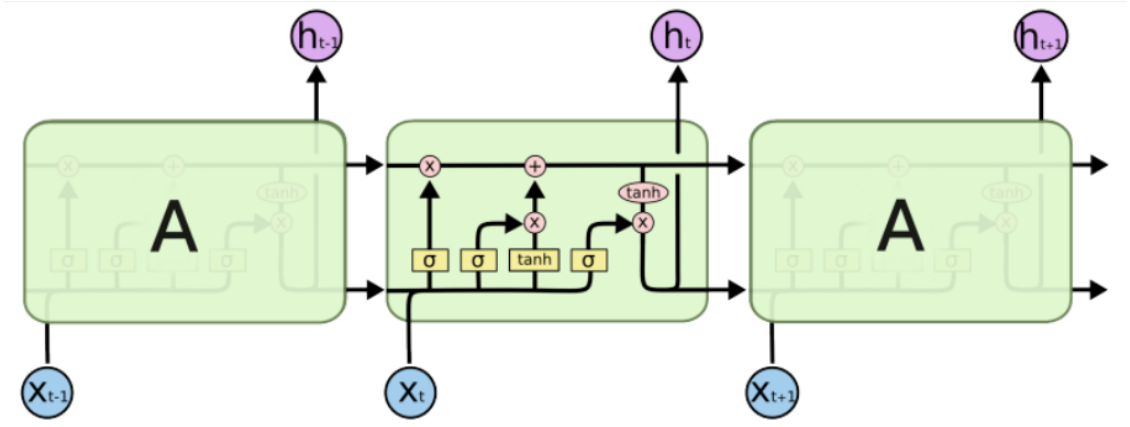
$$o_t = \sigma(x_t W_o + h_{t-1} W'_o)$$

$$\tilde{C}_t = \tanh(x_t W_g + h_{t-1} W'_g)$$

$$C_t = f_t \circ C_{t-1} + i_t \circ \tilde{C}_t$$

$$h_t = \tanh(C_t) \circ o_t$$

where $\circ$ represents the Hadamard Product (elementwise multiplication).




(a) Denote the final cost function as $J$. Compute the gradient $\frac{\partial J}{\partial W_g}$ using a combination of the following gradients,

$$\frac{\partial h_t}{\partial h_{t-1}}, \quad \frac{\partial h_{t-1}}{\partial W_g}, \quad \frac{\partial J}{\partial h_t}, \quad \frac{\partial C_t}{\partial W_g}, \quad \frac{\partial C_{t-1}}{\partial W_g}, \quad \frac{\partial C_t}{\partial C_{t-1}}, \quad \frac{\partial C_t}{\partial \tilde{C}_t}, \quad \frac{\partial h_t}{\partial o_t}$$

(b) Using the previously derived gradient, which part of $\frac{\partial J}{\partial W_g}$ allows LSTMs to mitigate the vanishing gradient problem?

### 1.4.2 Solution

**(a) Compute the gradient $\frac{\partial J}{\partial W_g}$ using a combination of given gradients.**

To compute $\frac{\partial J}{\partial W_g}$, we use the chain rule of differentiation, propagating the gradient of the cost function $J$ back through the LSTM cell. Given the available gradients:

$$\frac{\partial h_t}{\partial h_{t-1}}, \quad \frac{\partial h_{t-1}}{\partial W_g}, \quad \frac{\partial J}{\partial h_t}, \quad \frac{\partial C_t}{\partial W_g}, \quad \frac{\partial C_{t-1}}{\partial W_g}, \quad \frac{\partial C_t}{\partial C_{t-1}}, \quad \frac{\partial C_t}{\partial \tilde{C}_t}, \quad \frac{\partial h_t}{\partial o_t}$$

we apply the chain rule to express $\frac{\partial J}{\partial W_g}$:

$$\frac{\partial J}{\partial W_g} = \sum_{t=1}^{T} \frac{\partial J}{\partial h_t} \frac{\partial h_t}{\partial W_g}$$

Expanding $\frac{\partial h_t}{\partial W_g}$:

$$\frac{\partial h_t}{\partial W_g} = \frac{\partial h_t}{\partial C_t} \left( \frac{\partial C_t}{\partial W_g} + \frac{\partial C_t}{\partial C_{t-1}} \frac{\partial C_{t-1}}{\partial W_g} \right) + \frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial W_g}$$

where:

- $\frac{\partial h_t}{\partial C_t} = o_t \cdot (1 - \tanh^2(C_t))$ (derivative of tanh).
- $\frac{\partial C_t}{\partial C_{t-1}} = f_t$ (contribution from the forget gate).
- $\frac{\partial C_t}{\partial W_g} = i_t \cdot \frac{\partial \tilde{C}_t}{\partial W_g}$, where $\frac{\partial \tilde{C}_t}{\partial W_g}$ follows from the tanh activation function.
- $\frac{\partial h_t}{\partial h_{t-1}}$ accounts for the dependency of $h_t$ on previous hidden states.

After expanding, we merge together and get the equation of:

$$\frac{\partial J}{\partial W_g} = \sum_{t=1}^{T} \frac{\partial J}{\partial h_t} \left[ o_t(1 - \tanh^2(C_t)) \left( i_t \frac{\partial \tilde{C}_t}{\partial W_g} + f_t \frac{\partial C_{t-1}}{\partial W_g} \right) + \frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial W_g} \right]$$

This equation shows how gradients propagate back through both the hidden state $h_t$ and memory cell $C_t$, allowing error signals to affect earlier time steps.

**(b) Which part of $\frac{\partial J}{\partial W_g}$ helps LSTMs mitigate the vanishing gradient problem?**

The key term in the gradient that helps mitigate the vanishing gradient problem is:

$$\frac{\partial C_t}{\partial C_{t-1}} = f_t$$

Since the forget gate $f_t$ determines how much information from the previous memory cell $C_{t-1}$ is retained, it plays a crucial role in **gradient preservation**. If $f_t \approx 1$, then:

$$\frac{\partial C_t}{\partial C_{t-1}} \approx 1$$

which means gradients can flow **unhindered through time**, reducing the vanishing gradient effect. This allows LSTMs to maintain long-term dependencies more effectively compared to vanilla RNNs, where repeated multiplication by small values causes exponential decay of gradients.

**Key insights:**

- When $f_t$ is close to 1, the memory cell state $C_t$ acts like an **identity function**, preserving gradients over many time steps.
- This behavior is similar to **skip connections** in ResNets, where identity mappings help avoid gradient degradation.
- Unlike standard RNNs, where gradients diminish due to repeated squashing by activation functions, LSTMs maintain gradient magnitude through explicit gating mechanisms.

Thus, **the forget gate mechanism is fundamental in mitigating vanishing gradients**, enabling LSTMs to retain long-range dependencies effectively.

## 1.5 Problem 5

### 1.5.1 Description

(a) Explain how we incorporate self-attention into an RNN model at a high-level. m

(b) Consider a form of attention that matches query $q$ to keys $k_1, \ldots, k_t$ in order to attend over associated values $v_1, \ldots, v_t$. If we have multiple queries $q_1, \ldots, q_n$, how can we write this version of attention in matrix notation?

### 1.5.2 Solution

#### (a) Incorporating Self-Attention into an RNN Model

Recurrent Neural Networks (RNNs) process sequences by maintaining a hidden state that is updated at each timestep using the previous hidden state and the current input. However, RNNs struggle to capture long-range dependencies due to vanishing gradients.

Self-attention improves RNNs by:

- Allowing each token to attend to all others, rather than just relying on past hidden states.
- Computing attention scores for all timesteps and updating hidden representations based on weighted sums.
- Enabling parallel computation, unlike sequential RNN updates.

To integrate self-attention, we follow these steps:

1. **Compute Attention Scores**: For each timestep $t$, compute the attention scores $e_{t,l}$ for all previous hidden states $h_l$ using a scoring function:

   $$e_{t,l} = \text{score}(h_t, h_l)$$

   Common scoring functions include:

   $$\text{Dot Product:} \quad e_{t,l} = h_t^\top h_l$$

   $$\text{Additive:} \quad e_{t,l} = V^\top \tanh(W_1 h_t + W_2 h_l)$$

   $$\text{Scaled Dot Product:} \quad e_{t,l} = \frac{h_t^\top h_l}{\sqrt{d}}$$

2. **Compute Attention Weights**: Apply the softmax function to obtain a probability distribution over all previous states:

   $$\alpha_{t,l} = \frac{\exp(e_{t,l})}{\sum_j \exp(e_{t,j})}$$

3. **Compute Attention Output**: Compute a weighted sum of all previous hidden states using the attention weights:
$$a_t = \sum_{j=1}^{t-1} \alpha_{t,j} h_j$$

4. **Update Hidden State**: Combine the attention output $a_t$ with the RNN's original hidden state update to enhance information retention and learning:
$$h_t = \text{RNN}(x_t, h_{t-1}) + a_t$$

5. **Hybrid Model**: By combining RNNs for sequential processing with self-attention for enhanced context learning, the model can better capture long-range dependencies.

### (b) Matrix Notation for Multiple Queries

For a single query $q$, attention scores are:

$$\alpha_i = \frac{\exp(q^\top k_i)}{\sum_{j=1}^{t} \exp(q^\top k_j)}$$

The output is:

$$\text{Attention}(q, K, V) = \sum_{i=1}^{t} \alpha_i v_i$$

For multiple queries $Q = [q_1, q_2, \ldots, q_n]^\top$, we represent keys and values as:

$$K = [k_1, k_2, \ldots, k_t]^\top, \quad V = [v_1, v_2, \ldots, v_t]^\top$$

The attention scores matrix is (where Softmax is computed row-wise):

$$A = \text{softmax}(QK^\top)$$

The final output is:

$$\text{Attention}(Q, K, V) = AV$$

where $A$ contains attention weights, enabling efficient computation for multiple queries.

## 1.6 Problem 6

### 1.6.1 Description

In practice, Transformers use a Scaled Self-Attention. Suppose $q, k \in \mathbb{R}^d$ are two random vectors with $q, k \sim \mathcal{N}(\mu, \sigma^2 I)$, where $\mu \in \mathbb{R}^d$ and $\sigma \in \mathbb{R}^+$.

(a) Define $\mathbb{E}[q^\top k]$ in terms of $\mu, \sigma, d$.

(b) Define $\text{Var}(q^\top k)$ in terms of $\mu, \sigma, d$.

(c) Let $s$ be the scaling factor on the dot product. We would like $\mathbb{E}[q^\top k/s]$ to scale linearly with $d$. What should $s$ be in terms of $\mu, \sigma, d$?

(d) Briefly explain what would happen to the variance of dot product if $s = 1$.

### 1.6.2 Solution

**1. Computing $\mathbb{E}[q^\top k]$**

We want to compute the expected value of the dot product $q^\top k$, where $q, k \in \mathbb{R}^d$ and $q_i, k_i \sim \mathcal{N}(\mu_i, \sigma^2)$.

Expressing the Dot Product:

$$q^\top k = \sum_{i=1}^{d} q_i k_i$$

Applying Linearity of Expectation:

$$\mathbb{E}[q^\top k] = \mathbb{E}\left[\sum_{i=1}^{d} q_i k_i\right] = \sum_{i=1}^{d} \mathbb{E}[q_i k_i]$$

Since $q_i$ and $k_i$ are drawn independently from $\mathcal{N}(\mu_i, \sigma^2)$, we can compute the expectation of their product as:

$$\mathbb{E}[q_i k_i] = \mathbb{E}[q_i]\mathbb{E}[k_i] = \mu_i \mu_i = \mu_i^2$$

Summing Over All Dimensions to get the final answer as:

$$\mathbb{E}[q^\top k] = \sum_{i=1}^{d} \mu_i^2 = \mu^\top \mu$$

**2. Computing $\mathbf{Var}(q^\top k)$**

We want to compute the variance of dot product $q^\top k$. According to definition of variance:

$$\mathrm{Var}(q^\top k) = \mathbb{E}[(q^\top k)^2] - \mathbb{E}[q^\top k]^2$$

We have already computed $\mathbb{E}[q^\top k] = \mu^\top \mu$, so we proceed by focusing on first term $\mathbb{E}[(q^\top k)^2]$. Using the matrix product identity:

$$(q^\top k)^2 = q^\top k k^\top q = \mathrm{Tr}(q q^\top k k^\top)$$

By applying the linearity of expectation and using the independence of $q$ and $k$:

$$\mathbb{E}[\mathrm{Tr}(q q^\top k k^\top)] = \mathrm{Tr}(\mathbb{E}[q q^\top]\mathbb{E}[k k^\top])$$

Since $q$ and $k$ are independent and both are drawn from a multivariate normal distribution:

$$\mathbb{E}[q q^\top] = \mathbb{E}[q]\mathbb{E}[q]^\top + \mathrm{Cov}(q) = \mu\mu^\top + \sigma^2 I$$

$$\mathbb{E}[k k^\top] = \mathbb{E}[k]\mathbb{E}[k]^\top + \mathrm{Cov}(k) = \mu\mu^\top + \sigma^2 I$$

Thus, the expectation becomes:

$$\mathrm{Tr}\left((\mu\mu^\top + \sigma^2 I)(\mu\mu^\top + \sigma^2 I)\right)$$

Expanding using the distributive property of matrix multiplication:

$$= \mathrm{Tr}\left(\mu\mu^\top\mu\mu^\top + \sigma^2\mu\mu^\top + \sigma^2\mu\mu^\top + \sigma^4 I\right)$$

Using the cyclic property of the trace $\mathrm{Tr}(AB) = \mathrm{Tr}(BA)$:

$$= (\mu^\top\mu)^2 + 2\sigma^2\mu^\top\mu + d\sigma^4$$

Finally, subtracting $\mathbb{E}[q^\top k]^2 = (\mu^\top\mu)^2$:

$$\mathrm{Var}(q^\top k) = 2d\sigma^2\mu^\top\mu + d\sigma^4$$

**3. Choosing the Scaling Factor $s$**

We want the expectation of the scaled dot product $\mathbb{E}[q^\top k/s]$ to scale linearly with the dimension $d$. Recall that from part (a), we derived:

$$\mathbb{E}[q^\top k] = \mu^\top \mu$$

Now, consider the case when $\mu = 0$ (zero-mean vectors). From part (b), we found:

$$\text{Var}(q^\top k) = 2d\sigma^2 \mu^\top \mu + d\sigma^4$$

When $\mu = 0$, the variance simplifies to:

$$\text{Var}(q^\top k) = d\sigma^4$$

Since we want $\mathbb{E}[q^\top k/s]$ to scale linearly with $d$, we need to normalize by a factor that compensates for the dependence on $d$. The standard deviation of the dot product grows as $\sqrt{d}$. Therefore, to maintain a consistent scale across varying $d$:

$$s = \sqrt{d}$$

This is why Transformers use a scaling factor of $s = \sqrt{d}$ in scaled self-attention.

**4. Effect of Setting $s = 1$**

If we set the scaling factor $s = 1$, meaning no scaling is applied, then the variance of the dot product $q^\top k$ will grow linearly with the dimensionality $d$.

From our earlier result, when $\mu = 0$:

$$\text{Var}(q^\top k) = d\sigma^4$$

If we do not scale the dot product by $\sqrt{d}$ (i.e., $s = 1$), the magnitude of the variance increases **linearly** with $d$. As $d$ becomes large, this results in:

- **Large variance** in the dot product, causing extremely high or low values to be passed to the softmax function.
- **Highly skewed softmax outputs**, where one or few entries dominate, leading to poor learning and unstable gradients during training.
- **Gradient explosion** during backpropagation, making optimization difficult.

By scaling the dot product by $\sqrt{d}$, we effectively **normalize the variance**, making the training process stable even for large $d$.

## 1.7 Problem 7

### 1.7.1 Description

1. What is the reason for positional encoding? How is it typically implemented?
2. What is the advantage of multi-head attention? Give some examples of structures that can be found using multi-head attention.
3. For input sequences of length $M$ and output sequences of length $N$, what are the complexities of:
   (a) Encoder Self-Attention
   (b) Decoder-Encoder Attention
   (c) Decoder Self-Attention

   Further let $k$ be the hidden dimension of the network.
4. Do activation of the encoder depend on decoder activation? How much additional computation is needed to translate a source sequence into a different target language, in terms of $M$ and $N$?

### 1.7.2 Solution

#### 1. Reason for Positional Encoding and Implementation

Transformers do not have a built-in sense of word order because they lack recurrence (as in RNNs) or convolutional operations (as in CNNs). Instead, **positional encoding** is used to introduce order information into the input sequence.

**Reason for Positional Encoding:**

- Since self-attention is permutation-invariant, positional encoding helps the model distinguish different positions in the sequence.
- It allows the model to understand sequential relationships between words.
- It enables learning dependencies based on positions, crucial for capturing long-range dependencies.

**Implementation of Positional Encoding:**

A common implementation is to use **sinusoidal positional encoding**:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$

where:

- $pos$ is the position index of the token in the sequence,
- $i$ is the dimension index,
- $d$ is the total embedding dimension.

These values are added directly to the token embeddings before being fed into the transformer.

## 2. Advantages of Multi-Head Attention and Example Structures

### Advantages of Multi-Head Attention:

Multi-head attention enables the model to focus on different parts of the input sequence simultaneously. The main advantages are:

- **Improved Representation Power:** Each head learns different types of dependencies in the sequence.
- **Better Feature Extraction:** Different attention heads can capture different syntactic and semantic relationships.
- **Stability in Learning:** It prevents over-reliance on a single attention pattern.
- **Parallel Computation:** Unlike recurrent networks, self-attention allows efficient parallel computation.

### Example Structures Detected Using Multi-Head Attention:

- **Syntactic Structures:** Multi-head attention can capture relationships like subject-verb agreement or coreference resolution.
- **Semantic Relationships:** It can differentiate between different meanings of words based on context.
- **Translation Alignments:** In machine translation, different heads can focus on different word alignments.

### 3. Complexity of Self-Attention and Cross-Attention

Let:

- $M$ be the length of the input sequence,
- $N$ be the length of the output sequence,
- $k$ be the hidden dimension size.

#### (a) Encoder Self-Attention Complexity:

The encoder applies self-attention over the entire input sequence of length $M$. The key computations are:

- **Query, Key, and Value Computation:**

  In self-attention, for each token in the input sequence, we need to compute the **Query**, **Key**, and **Value** vectors.

  Since there are $M$ tokens, each of dimension $k$, the projection matrices for $Q$, $K$, and $V$ have sizes $k \times k$. Therefore, the complexity for computing these projections is:

$$O(Mk^2)$$

- **Dot-Product Attention Calculation:**

  The attention mechanism requires computing similarity scores between every pair of tokens. For a sequence of length $M$, this requires:

  - Forming a matrix of size $M \times M$ containing attention scores for all pairs of tokens and For each pair, we compute a dot product of dimension $k$.

  Therefore, the complexity is:

$$O(M^2k)$$

**Total Complexity:**

$$\boldsymbol{O(M^2k + Mk^2) = O(M^2k)}$$

The $M^2k$ term dominates because typically $M > k$.

#### (b) Decoder-Encoder Attention Complexity:

Decoder-encoder attention refers to mechanism where each token in decoder attends to all tokens in the encoder. This occurs during training or when generating tokens in sequence.

- **Query, Key, and Value Computation:**

  - Queries are generated from the $N$ tokens in the decoder, requiring projection matrices of size $k \times k$. Therefore, the complexity is:

$$O(Nk^2)$$

- Keys and Values are generated from the $M$ tokens in the encoder, requiring similar projections. Therefore, the complexity is:

$$O(Mk^2)$$

- **Dot-Product Attention Calculation:**

  - The decoder attends to all tokens of the encoder. Thus, for each of the $N$ decoder tokens, attention is computed over $M$ encoder tokens.

  - The resulting attention matrix has size $N \times M$.

  - For each pair, we perform a dot product of dimension $k$. Therefore, complexity is:

$$O(MNk)$$

**Total Complexity:**

$$\boldsymbol{O(Nk^2 + Mk^2 + MNk) = O(MNk)}$$

The $MNk$ term dominates since $M$ and $N$ are usually larger than $k$.

**(c) Decoder Self-Attention Complexity:**

The decoder applies self-attention over all previous tokens generated so far. It applies a **masked self-attention** mechanism to ensure that each token can only attend to previous tokens, not future tokens.

- **Query, Key, and Value Computation:**

  - For each of the $N$ tokens in the decoder, we compute the **Query**, **Key**, and **Value** vectors.

  - The projection matrices for $Q$, $K$, and $V$ are of size $k \times k$.

  - Therefore, the complexity for this step is:

$$O(Nk^2)$$

- **Dot-Product Attention Calculation:**

  - The attention mechanism requires computing similarity scores between every pair of tokens within the output sequence.

  - For a sequence of length $N$, the attention matrix is of size $N \times N$.

  - Each similarity score requires a dot-product operation of size $k$, resulting in:

$$O(N^2k)$$

**Total Complexity:**

$$\boldsymbol{O(N^2k + Nk^2) = O(N^2k)}$$

The $N^2k$ term dominates for large $N$.

**Summary of Complexities:**

- **Encoder Self-Attention:** $O(M^2k)$
- **Decoder-Encoder Attention:** $O(MNk)$
- **Decoder Self-Attention:** $O(N^2k)$

### 4. Does Encoder Activation Depend on Decoder Activation? Computational Cost of Translation

**Dependency Between Encoder and Decoder Activations**

The question asks whether **encoder activations** depend on **decoder activations**. Let's break down the architecture:

- **Encoder:**
  - The encoder processes the entire source sequence of length $M$ in **parallel**.
  - It applies **self-attention** layers and feed-forward networks to generate a sequence of **encoder activations** (or representations).
  - These representations are independent of the decoder, i.e., the encoder does not need any information from the decoder to complete its computations.
- **Decoder:**
  - The decoder takes the **encoder outputs** and performs **cross-attention** between its own activations and the encoder activations.
  - This means the **decoder depends on the encoder**, but the **encoder does not depend on the decoder**.

**Conclusion:** The encoder activations do not depend on the decoder activations.

**Additional Computation Needed for Translation**

We want to determine the amount of additional computation needed to **translate a source sequence of length $M$ into a target sequence of length $N$.**

**Components Involved**

- **Encoder Self-Attention Complexity:** $O(M^2k)$
- **Decoder Self-Attention Complexity:** $O(N^2k)$
- **Decoder-Encoder Attention Complexity:** $O(MNk)$

Since we are focusing on the additional computation needed to generate the target sequence, we only consider the **Decoder Self-Attention** and **Decoder-Encoder Attention**.

$$\text{Total Computation} = \text{Decoder Self-Attention} + \text{Decoder-Encoder Attention}$$

$$= O(N^2k) + O(MNk)$$

Simplifying to ignore constant terms $k$:

$$\boldsymbol{O(N^2) + O(MN)}$$

**Explanation of Terms**

- $O(N^2)$: This comes from the **Decoder Self-Attention** mechanism, where each output token attends to all previous tokens within the decoder. This grows quadratically with the output length $N$.
- $O(MN)$: This comes from the **Decoder-Encoder Attention**, where each decoder token attends to all $M$ encoder tokens. This grows linearly with both $M$ and $N$.

**Conclusion**: Additional computation required to translate source sequence is $O(MN + N^2)$.