CSCI-GA 2572: Deep Learning, Fall 2025

Transformer

Aditeya Baral
N19186654

Problem 1.1: Attention

Problem a

Solution

The standard dot-product attention involves the following operations:

1. Compute the attention scores: $S = K^T Q \in \mathbb{R}^{m \times n}$
   - This gives us the dot products between each key and each query
2. Apply the scaled softargmax function column-wise: $A = \text{softargmax}_\beta(S) \in \mathbb{R}^{m \times n}$
   - Each column of A contains the attention weights for the corresponding query
   - The softargmax is applied to each column independently
3. Compute the weighted sum of values: $H = VA \in \mathbb{R}^{t \times n}$
   - This produces the final output by combining value vectors according to the attention weights

Output dimension. The output $H \in \mathbb{R}^{t \times n}$: t rows (matching the dimension of the value vectors) and n columns (one output vector for each of the n queries).

Problem b

Solution

The scale β in the softargmax function controls the sharpness or smoothness of the attention distribution:

- **Large β :** The softargmax becomes sharper (more peaked). The attention weights concentrate on the key(s) with the highest similarity to the query, approaching a hard selection (one-hot distribution). This makes the output h closer to a single value vector. As $\beta \to \infty$, the attention becomes a hard argmax, selecting only the value corresponding to the key with maximum similarity.
- **Small β :** The softargmax becomes smoother (more uniform). The attention weights are more evenly distributed across all keys, regardless of their similarity scores. This makes the output h a more balanced mixture of multiple value vectors. As $\beta \to 0$, the attention weights become uniform (equal weight to all keys), resulting in a simple average of all value vectors.

Convenient choice of β. A convenient and commonly used choice is $\beta = \frac{1}{\sqrt{d}}$, where d is the

dimension of the queries and keys. When queries and keys have unit variance in each dimension, their dot products have variance approximately d. Scaling by $\frac{1}{\sqrt{d}}$ normalizes the dot products to have unit variance, preventing the softargmax from saturating (becoming too sharp) when d is large. This keeps gradients well-behaved during training.

Problem c

Solution

The attention operation preserves a value vector v_i in the output h_j when the attention weights are concentrated (sharp) such that the i-th attention weight is close to 1 and all other weights are close to 0. This occurs when:

- The query q_j has much higher similarity (dot product) with key k_i compared to all other keys
- Mathematically: $k_i^T q_j \gg k_l^T q_j$ for all $l \neq i$
- The resulting attention weight satisfies $a_{ij} \approx 1$ and $a_{lj} \approx 0$ for $l \neq i$
- This gives $h_j \approx v_i$ (the output preserves the value vector)

Choice of scale β. To preserve value vectors, we want β to be **large** (i.e., $\beta \to \infty$). A large β makes the softargmax very sharp, concentrating the attention weight on the single key with highest similarity to the query, effectively performing a hard selection.

Type of attention. This refers to **hard attention** (or **sharp attention**), where the attention mechanism selects one value vector rather than blending multiple values.

Implementation using fully connected architectures.

- Using a hard argmax operation instead of softmax to select the single highest-scoring value
- Alternatively, using a very large temperature parameter (equivalent to large β) in the softmax to approximate hard selection
- Or using Gumbel-softmax or straight-through estimators to enable gradient-based learning with hard selections

Problem d

Solution

The attention operation dilutes different value vectors to generate a new output h_j when the attention weights are distributed (smooth) such that multiple attention weights have significant non-zero values. This occurs when:

- The query q_j has similar similarity (dot product) with multiple keys
- Mathematically: $k_i^T q_j \approx k_l^T q_j$ for multiple indices i, l
- The resulting attention weights are distributed: multiple a_{ij} values are non-negligible
- This gives $h_j = \sum_{i=1}^{m} a_{ij} v_i$ as a weighted average (mixture) of multiple value vectors

Choice of scale β. To diffuse as much as possible, we want β to be **small** (i.e., $\beta \to 0$). A small β makes the softargmax very smooth and uniform, distributing attention weights nearly equally across

all keys regardless of their similarity scores. In the limit $\beta \to 0$, we get uniform attention weights $a_{ij} = \frac{1}{m}$ for all i.

Type of attention. This refers to **soft attention** (or **smooth attention**), where the attention mechanism blends multiple value vectors together rather than selecting a single one.

Implementation using fully connected architectures.

- Using a softmax with very small temperature parameter (equivalent to small β) to produce uniform weights
- Simply computing a uniform average of all value vectors (when $\beta \to 0$)
- Using dropout on attention weights to encourage diffusion across multiple values during training

Problem e

Solution

When we perturb one key such that $\hat{k}_i = k_i + \epsilon$ where ϵ is zero-mean Gaussian noise with small variance, the output H changes as follows:

- The perturbed key affects the attention scores for all queries: $\hat{s}_{ij} = \hat{k}_i^T q_j = k_i^T q_j + \epsilon^T q_j$ for all $j = 1, \ldots, n$
- This causes a small change in the attention weights in the i-th row of the attention matrix A. Specifically, the attention weight a_{ij} (weight of value v_i for query q_j) changes for all queries.
- Consequently, **all output columns** h_j (for $j = 1, \ldots, n$) are affected, since each output is computed as $h_j = \sum_{l=1}^{m} a_{lj} v_l$, and the weights a_{ij} have changed for all j.
- The magnitude of change in each h_j depends on:
  - The original attention weight a_{ij} (if a_{ij} was already small, the effect is minimal)
  - The scale β (larger β makes the attention more sensitive to score changes)
  - The magnitude of $\epsilon^T q_j$ (how much the perturbation aligns with each query)
- Since the perturbation is zero-mean, the expected change in H is zero, but there will be variance in the outputs.

Summary. Perturbing a single key k_i affects **all output vectors** h_j, but the effect is typically small and proportional to the attention weight a_{ij} that key k_i receives for each query.

Problem f

Solution

When we perturb one query such that $\hat{q}_i = q_i + \epsilon$ where ϵ is zero-mean Gaussian noise with small variance, the output H changes as follows:

- The perturbed query affects the attention scores for all keys: $\hat{s}_{ji} = k_j^T \hat{q}_i = k_j^T q_i + k_j^T \epsilon$ for all $j = 1, \ldots, m$
- This causes a change in the attention weights in the i-th column of the attention matrix A. Specifically, all attention weights a_{ji} (for $j = 1, \ldots, m$) in column i are affected.

- Consequently, **only the single output column** h_i is affected, since $h_i = \sum_{j=1}^{m} a_{ji} v_j$ depends on the i-th column of attention weights.
- All other output vectors h_j for $j \neq i$ remain unchanged because they depend on different columns of the attention matrix.
- The magnitude of change in h_i depends on:
  - The scale β (larger β makes the attention more sensitive to score changes)
  - The magnitude of perturbations $k_j^T \epsilon$ for all keys
  - How the perturbation redistributes attention weights among the value vectors
- Since the perturbation is zero-mean, the expected change in h_i is zero, but there will be variance in the outputs.

Difference from the previous case. **Perturbation to key k_i affects all output vectors h_j** for $j = 1, \ldots, n$ (affects one row of attention matrix), while **perturbation to query q_i affects only one output vector h_i** (affects one column of the attention matrix). This difference arises because each key participates in computing attention for *all* queries, while each query only computes attention for its *own* corresponding output.

Problem g

Solution

When we scale a single key by a factor $\alpha > 1$, the output H changes as follows:

- The scaled key affects the attention scores for all queries: $\hat{s}_{ij} = \hat{k}_i^T q_j = \alpha k_i^T q_j = \alpha s_{ij}$ for all $j = 1, \ldots, n$
- This multiplicatively increases all attention scores involving key k_i by factor α.
- After applying softargmax, the attention weight a_{ij} for the scaled key increases for all queries, while the attention weights for all other keys decrease correspondingly (since attention weights must sum to 1 in each column).
- Specifically, for each query q_j:
  - If key k_i already had high similarity with q_j, scaling amplifies this, making $a_{ij} \to 1$
  - The output h_j shifts closer to value v_i
  - Other value vectors contribute less to h_j
- The effect is more pronounced when:
  - α is large (stronger scaling)
  - β is large (sharper attention, more sensitive to score differences)
  - Key k_i already had relatively high similarity with the queries
- As $\alpha \to \infty$, the attention weights approach $a_{ij} \to 1$ and $a_{lj} \to 0$ for $l \neq i$ (for all j), meaning all outputs $h_j \to v_i$.

Summary. **Scaling key k_i by $\alpha > 1$ increases its attention weight for all queries**, causing all output vectors to shift closer to the corresponding value v_i. This is a systematic bias that affects the entire output matrix H.

Problem 1.2: Multi-headed Attention

Problem a

Solution

Given queries $Q \in \mathbb{R}^{d \times n}$, keys $K \in \mathbb{R}^{d \times m}$, and values $V \in \mathbb{R}^{t \times m}$, the multi-headed attention involves the following operations:

1. **Linear projections for each head:** For each head $i = 1, \ldots, h$, project the queries, keys, and values:

$$Q_i = W_i^Q Q \in \mathbb{R}^{d_k \times n}$$
$$K_i = W_i^K K \in \mathbb{R}^{d_k \times m}$$
$$V_i = W_i^V V \in \mathbb{R}^{d_v \times m}$$

   where $W_i^Q \in \mathbb{R}^{d_k \times d}$, $W_i^K \in \mathbb{R}^{d_k \times d}$, and $W_i^V \in \mathbb{R}^{d_v \times t}$ are learnable projection matrices. Typically, $d_k = d/h$ and $d_v = t/h$.

2. **Compute attention for each head:** For each head $i = 1, \ldots, h$, compute single-head attention:

$$S_i = K_i^T Q_i \in \mathbb{R}^{m \times n}$$
$$A_i = \text{softargmax}_\beta(S_i) \in \mathbb{R}^{m \times n}$$
$$H_i = V_i A_i \in \mathbb{R}^{d_v \times n}$$

   Note that softargmax_β incorporates the scaling, typically with $\beta = \frac{1}{\sqrt{d_k}}$ in standard scaled dot-product attention.

3. **Concatenate heads:** Concatenate the outputs from all heads:

$$H_{\text{concat}} = [H_1; H_2; \ldots; H_h] \in \mathbb{R}^{(h \cdot d_v) \times n}$$

4. **Final linear projection:** Apply a final linear transformation:

$$H = W^O H_{\text{concat}} \in \mathbb{R}^{t \times n}$$

   where $W^O \in \mathbb{R}^{t \times (h \cdot d_v)}$ is a learnable output projection matrix.

The output $H \in \mathbb{R}^{t \times n}$ has the same dimensions as single-head attention, with t rows and n columns.

Problem b

Solution

Multiple convolutional filters (or feature maps) in convolutional neural networks are similar to multi-headed attention.

Similarities:

- **Multiple parallel transformations**. Just as multi-headed attention uses multiple heads to process the input with different learned projections, CNNs use multiple convolutional filters to extract different features from the same input in parallel. Each filter learns to detect different patterns or features.

- **Different representation subspaces**. In multi-headed attention, each head projects queries, keys, and values into different learned subspaces (via W_i^Q, W_i^K, W_i^V), allowing the model to attend to different aspects of the input. Similarly, each convolutional filter operates in a different learned representation space, capturing different features (edges, textures, patterns, etc.).
- **Concatenation and combination**. Multi-headed attention concatenates outputs from all heads and applies a final linear transformation (W^O). Similarly, CNNs stack feature maps from different filters along the channel dimension, and subsequent layers combine these features through further convolutions or fully connected layers.
- **Increased model capacity**. Both mechanisms increase the model's representational capacity by allowing it to capture multiple types of information simultaneously, rather than being limited to a single transformation.
- **Parameter efficiency**. Both distribute parameters across multiple "views": multi-head attention splits the embedding dimension across heads (typically $d_k = d/h$), while CNNs use many small filters rather than one large transformation, making them more parameter-efficient while maintaining expressiveness.

Key Difference: While conceptually similar, CNN filters extract local spatial patterns, whereas attention heads can attend globally across all input positions, allowing for long-range dependencies.

Problem 1.3: Self Attention

Problem a

Solution

Given an input $C \in \mathbb{R}^{e \times n}$ where e is the embedding dimension and n is the sequence length, the queries, keys, values, and output for multi-headed scaled dot-product self-attention are defined as follows:

Self-attention setup: In self-attention, all queries, keys, and values are derived from the same input C via learnable linear projections W_i^Q, W_i^K, and W_i^V:

$$Q = C \in \mathbb{R}^{e \times n}$$
$$K = C \in \mathbb{R}^{e \times n}$$
$$V = C \in \mathbb{R}^{e \times n}$$

Multi-headed self-attention operations:

1. **Linear projections for each head:** For each head $i = 1, \ldots, h$, project the queries, keys, and values:

$$Q_i = W_i^Q C \in \mathbb{R}^{d_k \times n}$$
$$K_i = W_i^K C \in \mathbb{R}^{d_k \times n}$$
$$V_i = W_i^V C \in \mathbb{R}^{d_v \times n}$$

   where $W_i^Q \in \mathbb{R}^{d_k \times e}$, $W_i^K \in \mathbb{R}^{d_k \times e}$, and $W_i^V \in \mathbb{R}^{d_v \times e}$ are the learnable projection matrices. Typically, $d_k = e/h$ and $d_v = e/h$.

2. **Compute attention for each head:** For each head $i = 1, \ldots, h$, compute single-head attention:

$$S_i = K_i^T Q_i \in \mathbb{R}^{n \times n}$$
$$A_i = \text{softargmax}_\beta(S_i) \in \mathbb{R}^{n \times n}$$
$$H_i = V_i A_i \in \mathbb{R}^{d_v \times n}$$

   Note that softargmax_β incorporates the scaling, typically with $\beta = \frac{1}{\sqrt{d_k}}$.

3. **Concatenate heads:** Concatenate the outputs from all heads:

$$H_{\text{concat}} = [H_1; H_2; \ldots; H_h] \in \mathbb{R}^{(h \cdot d_v) \times n}$$

4. **Final linear projection:** Apply a final linear transformation:

$$H = W^O H_{\text{concat}} \in \mathbb{R}^{e \times n}$$

   where $W^O \in \mathbb{R}^{e \times (h \cdot d_v)}$ is a learnable output projection matrix.

The output $H \in \mathbb{R}^{e \times n}$ has the same dimensions as the input C, with e rows (embedding dimension) and n columns (sequence length).

Problem b

Solution

What is positional encoding.
Positional encoding is a mechanism to inject information about the position or order of elements in a sequence into the model. Since self-attention operations are permutation-invariant (the attention weights depend only on pairwise dot-products of embeddings and the model has no inherent sense of token order), they treat the input as an unordered set. Thus, positional encodings are necessary to provide the model with information about the sequential structure of the data. Positional encodings are typically added to the input embeddings before feeding them into the attention layers.

Difference between absolute and relative positional encoding.

- **Absolute positional encoding**. Encodes the absolute position of each token in the sequence. Each position i is assigned a unique encoding vector PE_i that is added to the input embedding. The encoding depends only on the position index itself, not on the relationships between positions.

  Examples: Sinusoidal encodings (e.g., $PE_{(pos,2i)} = \sin(pos/10000^{2i/d})$) or learned position embeddings.

- **Relative positional encoding:** Encodes the relative distances or relationships between pairs of positions rather than their absolute positions. The encoding captures information about how far apart two tokens are (e.g., position i is 3 steps before position j), which is incorporated directly into the attention mechanism.

  Examples: Relative position biases added to attention scores, or learnable relative position embeddings.

When to use absolute positional encoding.
Absolute positional encoding is appropriate when:

- The absolute position in the sequence carries important meaning (e.g., the first word of a sentence, specific positions in structured data)
- The input sequences have fixed or similar lengths during training and inference
- The task benefits from knowing the exact position of each token

When to use relative positional encoding.
Relative positional encoding is more appropriate when:

- The relationships between tokens matter more than their absolute positions (e.g., syntactic dependencies in language)
- The model needs to generalize to sequences longer than those seen during training (better length extrapolation)
- The task involves translation invariance, where patterns can occur at any position in the sequence
- Local context and relative distances are more important than global position information

Problem c

Solution

Self-attention can behave like an **identity layer** or **permutation layer** when the attention weights form a permutation matrix (i.e., each row and column of the attention matrix A has exactly one entry equal to 1 and all other entries equal to 0).

Identity layer case.
The self-attention layer behaves as an identity layer when:

- The attention matrix $A = I$ (identity matrix), where $a_{ij} = 1$ if $i = j$ and $a_{ij} = 0$ otherwise
- Each query attends only to itself with attention weight 1
- This occurs when each token has maximal similarity with itself and minimal similarity with all other tokens in the sequence
- The output becomes $H_i = V_i A_i = V_i I = V_i$, where each output is simply the corresponding value vector
- If the value projection W^V and output projection W^O are identity matrices (or their composition is identity), then $H = C$ (perfect identity)

Permutation layer case.
The self-attention layer behaves as a permutation layer when:

- The attention matrix A is a permutation matrix, where each row has exactly one entry equal to 1
- Each query attends to exactly one key (possibly different from itself) with attention weight 1
- This occurs when each query has maximal similarity with one specific key and minimal similarity with all others
- The output becomes $H_i = V_i A_i$, where each output is a reordered version of the value vectors
- For example, if $a_{21} = 1$ and all other entries in column 1 are 0, then the first output $h_1 = v_2$ (the output at position 1 comes from position 2)
- If value and output projections preserve this structure, the layer effectively permutes the input sequence

How this is achieved:

- Large scale parameter $\beta \to \infty$ (very sharp attention) to create one-hot attention weights
- Appropriate query and key projections such that each query has much higher dot product with one specific key compared to all others

Problem d

Solution

Self-attention can behave like a convolution layer when the attention weights form a banded or local pattern, where each position primarily attends to itself and a fixed-size neighborhood of nearby positions. This effectively acts like a 1D convolution over the sequence embeddings, where each position aggregates information from a fixed-size local neighborhood.

Assume we use positional encoding where the similarity between query and key depends on their relative distance. Each position attends to a local window of positions around itself.

Convolution-like behavior.
The self-attention layer behaves like a convolution with kernel size k when:

- For each position i, the attention weights a_{ji} are significant only when $|j - i| \leq \lfloor k/2 \rfloor$ (i.e., position j is within a local window around position i)
- The attention weights are approximately uniform or follow a fixed pattern within this window, and near zero outside the window
- The attention matrix A has a banded structure: non-zero entries are concentrated along the diagonal in a band of width k
- Each output $h_i = \sum_j a_{ji} v_j \approx \sum_{j=i-\lfloor k/2 \rfloor}^{i+\lfloor k/2 \rfloor} a_{ji} v_j$ is a weighted combination of values from the local neighborhood

How this is achieved:

- **Positional encoding:** Use relative positional encodings that bias the attention scores based on distance. Query-key similarities are higher for nearby positions and lower for distant positions.
- **Learned projections:** The projection matrices W^Q and W^K are learned such that queries have high similarity with keys in their local neighborhood
- **Moderate scale parameter β:** A moderate β value creates soft attention that is concentrated but not completely one-hot, distributing weights across the local window
- **Local attention masks (optional):** Explicitly mask out attention to positions outside a fixed window, forcing strict locality

Example. For a kernel size $k = 3$, position i attends primarily to positions $\{i - 1, i, i + 1\}$ with attention weights approximately $[0.25, 0.5, 0.25]$, and negligible weights for all other positions. This mimics a 1D convolution with a kernel of size 3.

Problem e

Solution

For real-time automatic speech recognition, we need to modify the attention mechanism to prevent the model from accessing future information. The standard self-attention allows each position to attend to all positions in the sequence (including future positions), which is not suitable for real-time scenarios where future input has not yet been observed.

Solution: Causal (Masked) Self-Attention
We need to implement **causal self-attention** (also called **masked self-attention** or **autoregressive attention**), which ensures that each position can only attend to itself and previous positions in the sequence. This is performed as follows:

1. **Apply an attention mask:** Before applying the softargmax function, mask out (set to $-\infty$) the attention scores for future positions.

   For each query at position i, set $s_{ji} = -\infty$ for all $j > i$, where s_{ji} is the attention score between key j and query i.

2. **Modified attention computation:** The attention computation becomes:

$$\begin{aligned} S_i &= K_i^T Q_i \in \mathbb{R}^{n \times n} \\ \tilde{S}_i &= S_i + M \quad \text{(apply mask)} \\ A_i &= \text{softargmax}_\beta(\tilde{S}_i) \in \mathbb{R}^{n \times n} \\ H_i &= V_i A_i \in \mathbb{R}^{d_v \times n} \end{aligned}$$

   where $M \in \mathbb{R}^{n \times n}$ is the causal mask matrix defined as:

$$M_{ji} = \begin{cases} 0 & \text{if } j \leq i \\ -\infty & \text{if } j > i \end{cases}$$

3. **Effect:** After applying softargmax, the attention weights $a_{ji} = 0$ for all $j > i$, ensuring that position i only attends to positions $1, 2, \ldots, i$ (current and past positions only).

4. **Result:** The attention matrix A becomes lower triangular (including diagonal), where each position can only attend to previous and current positions, making the model suitable for real-time streaming applications.

This causal masking ensures that the model processes the speech input in a streaming fashion, producing transcriptions incrementally as new audio arrives, without requiring access to future context.

Problem 1.4: Transformer

Problem a

Solution

1. **No recurrence - fully parallel architecture**.
   - **RNNs/LSTMs:** Process sequences sequentially, one time step at a time. Each hidden state h_t depends on the previous hidden state h_{t-1}, creating a sequential dependency that prevents parallel computation.
   - **Transformer:** Eliminates recurrence entirely. All positions in the sequence are processed simultaneously using self-attention, allowing full parallelization during both training and inference. This significantly reduces training time.
2. **Direct long-range dependencies**.
   - **RNNs/LSTMs:** Information must propagate sequentially through many time steps to connect distant positions. The path length between any two positions is $O(n)$ where n is the sequence length, making it difficult to capture long-range dependencies due to vanishing/exploding gradients.
   - **Transformer:** Self-attention allows direct connections between any two positions in the sequence with a path length of $O(1)$. Every token can directly attend to every other token, making it much easier to model long-range dependencies.
3. **Self-attention mechanism vs. recurrent connections**.
   - **RNNs/LSTMs:** Use recurrent connections and gating mechanisms (e.g., forget gates, input gates) to maintain and update hidden states over time. Context is accumulated through sequential processing.
   - **Transformer:** Uses self-attention mechanisms to compute representations by attending to all positions in the input simultaneously. Each position's representation is a weighted combination of all other positions based on learned attention weights.
4. **Positional information handling**.
   - **RNNs/LSTMs:** Sequential processing inherently provides positional information - the model knows the order of tokens through the temporal processing structure.
   - **Transformer:** Since there is no sequential processing, positional encodings must be explicitly added to input embeddings to provide the model with information about token positions in the sequence.
5. **Computational complexity**.
   - **RNNs/LSTMs:** Sequential computation requires $O(n)$ sequential operations for a sequence of length n, limiting parallelization. Complexity per layer is $O(n \cdot d^2)$ where d is the hidden dimension.
   - **Transformer:** Self-attention has complexity $O(n^2 \cdot d)$ per layer, but all operations can be parallelized. For typical sequences where $n < d$, this is efficient and the parallelization provides significant speed advantages.

Problem b

Solution

Self-attention is a mechanism that computes a representation of a sequence by relating different positions within the same sequence to each other. For each position in the sequence, self-attention determines how much attention (or weight) to assign to every other position, including itself, and creates an output representation as a weighted combination of all positions.

Mathematical formulation. Given an input sequence, self-attention computes:

- Queries (Q), Keys (K), and Values (V) from the same input through learned linear projections
- Attention scores by computing similarity between queries and keys
- Attention weights by applying softmax to the scaled scores
- Output representations as weighted sums of values based on attention weights

The operation can be expressed as: $\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$

Importance in the Transformer model.

1. **Capturing contextual relationships**. Self-attention allows each token to directly interact with and gather information from all other tokens in the sequence, enabling the model to understand the context and relationships between words regardless of their distance.
2. **Handling long-range dependencies**. Unlike RNNs where information must propagate through many sequential steps, self-attention provides direct connections ($O(1)$ path length) between any two positions, making it much more effective at capturing long-range dependencies without the gradient degradation issues.
3. **Parallel computation**. Self-attention operates on all positions simultaneously rather than sequentially, enabling efficient parallelization during training. This drastically reduces training time compared to sequential models.
4. **Dynamic and context-dependent representations**. The attention weights are computed dynamically based on the input content, allowing the model to adaptively focus on relevant parts of the sequence for each position. The same word can have different representations depending on its context.
5. **Interpretability**. Attention weights provide some interpretability by showing which parts of the input the model focuses on when producing each output, offering insights into the model's decision-making process.
6. **Foundation for the entire architecture**. Self-attention is the core building block that replaces recurrence in the Transformer, enabling the model to be both powerful and efficient. It is used in both the encoder (for processing input) and decoder (for processing output and attending to encoder representations).

Problem c

Solution

Multi-head attention extends the self-attention mechanism by running multiple attention operations (called "heads") in parallel, each with different learned linear projections. Instead of performing a single attention function, the model performs multiple attention operations simultaneously and combines their results.

Mechanism.
For h attention heads:

1. Each head i has its own learned projection matrices W_i^Q, W_i^K, and W_i^V that project the input into queries, keys, and values
2. Each head computes attention independently: $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$
3. The outputs from all heads are concatenated: $\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$
4. A final linear projection W^O combines the concatenated outputs

Typically, the dimension of each head is reduced (e.g., $d_k = d_{\text{model}}/h$) so that the total computational cost is similar to single-head attention with full dimensionality.

Benefits of multi-head attention:

1. **Multiple representation subspaces**. Different heads can learn to attend to different types of relationships and capture different aspects of the input. Each head operates in its own learned representation subspace, allowing the model to jointly attend to information from different representation subspaces at different positions.
2. **Diverse attention patterns**. Different heads can focus on different linguistic or semantic phenomena. For example, one head might focus on syntactic relationships (e.g., subject-verb agreement), another on semantic relationships (e.g., co-reference), and another on positional relationships (e.g., nearby words).
3. **Increased model capacity**. Multiple heads increase the model's expressive power without significantly increasing computational cost (since each head works with reduced dimensions). This allows the model to capture more complex patterns and relationships in the data.
4. **Robustness and redundancy**. Having multiple heads provides redundancy - if one head fails to capture important information, other heads may compensate. This makes the model more robust to various inputs and scenarios.
5. **Better gradient flow**. Multiple parallel pathways for information flow can improve gradient propagation during backpropagation, contributing to more stable training.
6. **Parallel specialization.** Different heads can specialize in capturing different ranges of dependencies (short-range vs. long-range) or different types of contextual information, leading to richer and more comprehensive representations.

Problem d

Solution

The Transformer model includes position-wise feed-forward neural networks (FFN) that are applied to each position separately and identically. These FFNs are inserted after the attention sub-layers in both the encoder and decoder.

The feed-forward network consists of two linear transformations with a ReLU activation in between:

$$\text{FFN}(x) = \text{ReLU}(xW_1 + b_1)W_2 + b_2$$

where:

- The first linear layer expands the dimensionality (typically from d_{model} to d_{ff}, where $d_{ff} = 4 \times d_{\text{model}}$ in the original paper, e.g., 2048)
- ReLU activation introduces non-linearity
- The second linear layer projects back down to the model dimension d_{model} (e.g., 512)
- The same FFN is applied independently to each position in the sequence

Purpose of feed-forward networks.

1. **Non-linear transformations**. While attention mechanisms are good at aggregating information across positions, they are primarily linear operations (weighted sums). The two-layer FFN with non-linearity adds essential non-linearity through the ReLU activation, significantly increasing the model's expressive power and ability to learn complex patterns. This provides the universal approximation capability, allowing the Transformer to model intricate functions and relationships in the data.
2. **Position-wise feature transformation**. The FFN processes each position independently, allowing the model to transform the features at each position based on the contextual information gathered by the attention layer. This refines and enriches the representations after attention has mixed information across positions.
3. **Increased model capacity**. The expansion to a higher-dimensional space (d_{ff}) in the intermediate layer provides additional capacity for the model to learn complex transformations. The bottleneck structure (expand-then-compress) allows for rich intermediate representations.
4. **Feature mixing within positions**. While self-attention mixes information across different positions in the sequence, the FFN mixes information across different feature dimensions within each position, providing a complementary form of computation.

Problem e

Solution

1. **Residual Connections (Skip Connections)**. The Transformer uses residual connections around each sub-layer (both self-attention and feed-forward networks). The output of each sub-layer is added to its input before being passed to the next layer:

   $$\text{Output} = \text{LayerNorm}(x + \text{Sublayer}(x))$$

   where x is the input to the sub-layer and $\text{Sublayer}(x)$ is the function computed by the sub-layer itself.

How it helps with vanishing/exploding gradients.

- Residual connections create direct paths for gradients to flow backward through the network during backpropagation
- Gradients can bypass the sub-layers through the identity connection, preventing them from vanishing when multiplied through many layers
- The gradient of a residual block includes an additive identity term: $\frac{\partial}{\partial x}(x + F(x)) = I + \frac{\partial F(x)}{\partial x}$, ensuring that gradients are at least 1, which prevents complete vanishing
- This allows training of very deep networks (the Transformer encoder and decoder each have multiple layers stacked) without severe gradient degradation

2. **Layer Normalization**. Layer normalization is applied after each residual connection. It normalizes the inputs across the feature dimension for each example independently:

$$\text{LayerNorm}(x) = \gamma \odot \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta$$

where μ and σ^2 are the mean and variance computed across the feature dimension, and γ and β are learned scale and shift parameters.

How it helps with vanishing/exploding gradients.

- Layer normalization re-centers and re-scales activations at each layer, preventing them from growing too large (exploding) or becoming too small (vanishing) as they propagate through the network
- By normalizing the inputs to each sub-layer, it ensures that the activations remain in a reasonable range, which stabilizes the gradients during backpropagation
- It reduces the sensitivity of the network to initialization and learning rates, making training more stable
- The normalization helps maintain consistent gradient magnitudes across layers, preventing the gradients from either exploding or vanishing as they backpropagate through the deep network

Combined effect: Together, residual connections and layer normalization facilitate stable training that allows the Transformer to be trained effectively despite its depth and complexity, addressing the challenges of vanishing and exploding gradients that affect deep architectures.

Problem 1.5: Vision Transformer

Problem a

Solution

Key difference in handling input images. The key difference between Vision Transformer (ViT) and traditional CNNs lies in how they process spatial information in images. The ViT converts the 2D image understanding problem into a sequence modeling problem, similar to how Transformers process text, whereas CNNs maintain and exploit the 2D spatial structure throughout:

- **CNNs**. Process images using local receptive fields through convolutional operations. Convolution filters slide across the image, extracting local features hierarchically. Spatial relationships are captured through the inductive bias of locality and translation equivariance built into the convolutional operations. CNNs process images in a spatially continuous manner, maintaining the 2D spatial structure throughout the network.
- **Vision Transformer (ViT)**. Treats the image as a sequence of patches rather than a continuous 2D grid. The image is divided into fixed-size non-overlapping patches (e.g., 16×16 pixels), each patch is flattened into a vector and linearly embedded, and these patch embeddings are then processed as a sequence by the Transformer encoder using self-attention. ViT does not have built-in spatial inductive biases, it learns spatial relationships purely through self-attention on the patch sequence.

Convolution layer in ViT.
The ViT architecture does not use convolutional layers for hierarchical feature extraction. However:

- **Patch Embedding**. ViT uses a linear projection to embed image patches. Each patch (e.g., $16 \times 16 \times 3$) is flattened into a vector and multiplied by a learnable weight matrix to produce the embedding.
- **Implementation detail**. This linear projection can be efficiently implemented as a convolutional operation with kernel size 16×16, stride 16, and output channels equal to the embedding dimension. This is purely an implementation optimization, and it conceptually remains a linear projection, not a convolution for feature extraction.

After the initial patch embedding, ViT uses only Transformer layers (self-attention and feed-forward networks) for all feature extraction. In summary, while the patch embedding can be implemented as a convolution for efficiency, ViT does not rely on convolutional operations for feature extraction, which is the defining characteristic of CNNs.

Problem b

Solution

Role of positional embeddings in Vision Transformer. Positional embeddings in ViT serve to provide spatial information about where each patch is located in the original image. Since the Transformer architecture is inherently permutation-invariant (self-attention treats the input as an unordered set), positional embeddings are necessary to encode the spatial relationships between patches. Without positional embeddings, the model would have no way to distinguish between different spatial arrangements of the same patches.

The positional embeddings are added to the patch embeddings before being fed into the Transformer encoder, allowing the model to learn spatial relationships and understand the 2D structure of the image.

Differences from positional encodings in the original Transformer.

1. **Learned vs. fixed**.
   - **Original Transformer**. Uses fixed sinusoidal positional encodings that are predetermined mathematical functions (e.g., $PE_{(pos,2i)} = \sin(pos/10000^{2i/d})$). These are not learned during training.
   - **ViT**. Uses learned positional embeddings that are trained parameters initialized randomly and optimized during training. The model learns the optimal way to encode spatial positions.
2. **Dimensionality**.
   - **Original Transformer**. Encodes 1D sequential positions (position in a sequence of tokens).
   - **ViT**. Encodes 2D spatial positions (row and column in the image grid of patches). Although the patches are treated as a 1D sequence, the positional embeddings can capture 2D spatial relationships.
3. **Flexibility**.
   - **Original Transformer**. Fixed encodings can generalize to sequences longer than those seen during training due to their mathematical definition.
   - **ViT**. Learned embeddings are specific to the number of patches seen during training. For different input resolutions, the positional embeddings may need to be interpolated.

Problem c

Solution

The ViT generates the final classification output through the following process:

1. **Prepend a learnable [CLS] token**. Before feeding the patch embeddings into the Transformer encoder, a special learnable classification token (called [CLS] token) is prepended to the sequence of patch embeddings. This token does not correspond to any image patch but serves as a learnable placeholder for aggregating global image information. The input sequence thus becomes: $[\text{CLS}; \text{patch}_1; \text{patch}_2; \ldots; \text{patch}_N]$, where N is the number of patches.
2. **Add positional embeddings**. Positional embeddings (including one for the [CLS] token) are added to all tokens in the sequence to provide spatial information.
3. **Process through Transformer's encoder**. The entire sequence passes through multiple Transformer encoder layers. Through self-attention, the [CLS] token can attend to all patch tokens and aggregate information from the entire image. The Transformer learns to encode global image representations into the [CLS] token's output representation.
4. **Extract [CLS] token representation**. After processing through all Transformer layers, only the output representation corresponding to the [CLS] token (the first position in the sequence) is used for classification. This representation contains the aggregated global information about the entire image.
5. **Classification head**. The [CLS] token's output representation is passed through a classification head, which is typically a simple feed-forward neural network (often just a single linear layer or an MLP with one hidden layer). This head maps the [CLS] representation to class logits for

each class in the classification task. Mathematically: $y = \text{MLP}(z_{\text{CLS}})$, where z_{CLS} is the output of the Transformer encoder at the [CLS] token position.

6. **Final prediction**. The class logits are passed through a softmax function (during inference) to produce class probabilities, and the class with the highest probability is selected as the final prediction.

Problem d

Solution

ViT and CNNs exhibit significantly different performance characteristics depending on the size of the training dataset:

1. **Small datasets (e.g., ImageNet-1k with 1.3M images)**.
   - **CNNs outperform ViT**. When pre-trained on mid-sized datasets such as ImageNet without strong regularization, ViT models yield modest accuracies of a few percentage points below ResNets of comparable size.
   - For example, ViT-B/32 performs much worse than ResNet50 on small subsets (9M images), despite having comparable computational cost.
   - Even with regularization, larger ViT models (e.g., ViT-Large) underperform compared to smaller ViT-Base models when trained only on ImageNet.
2. **Medium datasets (e.g., ImageNet-21k with 14M images)**.
   - Performance gap narrows: ViT-Base and ViT-Large achieve similar performance.
   - CNNs still maintain a slight edge in some cases.
   - When fine-tuned on the ImageNet-21k dataset, ViT models close much of the performance gap with CNNs.
3. **Large datasets (e.g., JFT-300M with 300M images)**.
   - **ViT surpasses CNNs:** With JFT-300M, ViT sees the full benefit of larger models and overtakes ResNets in performance.
   - ViT-H/14 and ViT-L/16 pre-trained on JFT-300M outperform state-of-the-art CNNs like BiT (ResNet) and Noisy Student (EfficientNet) on multiple benchmarks.
   - Larger ViT variants (L, H) significantly outperform smaller ones as dataset size increases.

Explanation for this trend.

- **Inductive biases in CNNs**. CNNs have strong image-specific inductive biases such as locality, two-dimensional neighborhood structure, and translation equivariance baked into each layer throughout the model. These biases are beneficial when training data is limited, as they guide the model toward solutions that work well for images without requiring extensive data to learn these patterns.

  ViTs lack some of the inductive biases inherent to CNNs, and therefore do not generalize well when trained on insufficient amounts of data. ViT must learn spatial relationships and image structure from scratch through self-attention.

- **Data-driven learning at scale**. Large-scale training trumps inductive bias. When sufficient data is available, ViT can learn the relevant patterns directly from data, which is not only sufficient but actually beneficial. The flexibility of learning representations without hard-coded biases allows ViT to discover more optimal patterns for the specific task when given enough training examples.

- **Overfitting on small datasets**. ViTs overfit more than ResNets with comparable computational cost on smaller datasets. This reinforces the intuition that convolutional inductive bias is useful for smaller datasets, but for larger ones, learning patterns directly from data provides better results.

In summary, CNNs are more data-efficient due to their built-in visual inductive biases, while ViT requires large-scale pre-training to reach its full potential, at which point it can match or exceed CNN performance.