Assignment 5 — Solutions

Attention & Transformer Architecture

Part B — Transformers & LLMs · 21 questions with answers

Q1. What is a seq2seq model? What is the encoder–decoder architecture and how does it work?

Answer

A **seq2seq (sequence-to-sequence)** model converts an input sequence into an output sequence whose length may differ (e.g. English sentence → Hindi sentence).

Encoder–decoder architecture:

- **Encoder** reads the input sequence and compresses it into a representation (in classic RNN seq2seq, a single fixed-size **context vector** = final hidden state; in Transformers, a sequence of contextual vectors).
- **Decoder** generates the output **one token at a time**, conditioned on the encoder representation and the tokens it has already generated, starting from a <start> token until it produces <end>.

Training uses teacher forcing; inference uses greedy or beam search.

Q2. Applications, pros and cons of the encoder–decoder architecture.

Answer

Applications: machine translation, text summarization, question answering, chatbots/dialogue, speech-to-text (Whisper), image captioning (CNN/ViT encoder + text decoder), code generation, text-to-SQL.

Pros: handles variable-length input and output; general framework for many tasks; encoder and decoder can be different modalities.

Cons (classic RNN version): the fixed-length context vector is an **information bottleneck** – performance drops for long sentences; sequential processing is slow; exposure bias from teacher forcing. Attention and Transformers solved most of these.

Q3. What is attention and why was it needed (the bottleneck problem)? ★

Answer

Attention lets the model, while producing each output, **look at all input positions and focus on the most relevant ones** by computing a weighted sum of their representations, where the weights are learned relevance scores.

Why it was needed – the bottleneck problem: in RNN seq2seq the whole input sentence had to be squeezed into one fixed-size vector. Long sentences lost information, and translation quality dropped sharply with length. With attention (Bahdanau 2014), the decoder gets a **different context vector at every step**, built directly from all encoder states. It also handles alignment (which source word matches which target word) and gives some interpretability (attention maps).

Q4. What are the different types of attention (Bahdanau/additive, Luong/multiplicative, self, cross)?

Answer

- **Additive (Bahdanau) attention** – score = $v^T \tanh(W_1 h_dec + W_2 h_enc)$; uses a small feed-forward network.
- **Multiplicative (Luong) attention** – score = $h_dec^T W h_enc$ (or plain dot product); faster.
- **Scaled dot-product attention** – used in Transformers.
- **Self-attention** – Q, K, V all come from the same sequence.
- **Cross (encoder–decoder) attention** – Q from decoder, K/V from encoder.
- **Masked (causal) attention** – cannot look at future tokens.
- **Multi-head attention**, and efficient variants: MQA, GQA, sliding-window, sparse, Flash attention.
- Also **global vs local** and **soft vs hard** attention.

Q5. What is self-attention? Explain Query, Key and Value with an intuitive example. ★ · Asked: Google, NLP rounds

Answer

Self-attention lets every token in a sequence look at every other token in the **same** sequence and build a context-aware representation of itself.

Each token embedding x is projected into three vectors with learned matrices: **Q = xW_Q, K = xW_K, V = xW_V**.

- **Query** – "what am I looking for?"
- **Key** – "what do I contain / what can I be matched on?"
- **Value** – "the information I give if selected."

Intuition (library search): your query is the search text, keys are the book titles, values are the book contents. You compare the query with all keys, and read a weighted mix of the matching contents.

Example: "The animal didn't cross the street because **it** was tired." The query for "it" matches strongly with the key of "animal", so the new vector for "it" contains information from "animal".

Q6. What is Scaled Dot-Product Attention? Write the formula. Why do we divide by $\sqrt{d_k}$? ★

Answer

$\text{Attention}(Q, K, V) = \text{softmax}(Q \cdot K^T / \sqrt{d_k}) \cdot V$

Steps: (1) dot product of each query with all keys = similarity scores; (2) divide by $\sqrt{d_k}$; (3) (optional mask); (4) softmax → attention weights that sum to 1; (5) weighted sum of value vectors.

Why divide by $\sqrt{d_k}$: for vectors with d_k dimensions (unit-variance entries), the dot product has variance d_k, so with $d_k = 64$ the scores become large. Large values push softmax into regions where one weight ≈ 1 and others ≈ 0, making **gradients extremely small**. Scaling by $\sqrt{d_k}$ keeps the variance ≈ 1 and keeps training stable.

Q7. What is multi-head attention and why is it better than a single head? ★

Answer

Multi-head attention runs several attention operations ("heads") **in parallel**, each with its own W_Q, W_K, W_V projecting to a smaller dimension (d_model / h). Outputs of all heads are concatenated and projected with W_O:

MultiHead(Q,K,V) = Concat(head₁, …, head_h)·W_O

(BERT-base: 12 heads × 64 dims = 768.)

Why better than one head: each head can learn a **different type of relationship** – one head for syntax (subject–verb), one for coreference ("it" → "animal"), one for nearby words, etc. A single head would average all of these into one distribution. Total compute is about the same as one full-size head.

Q8. What is the difference between self-attention and cross-attention? How does cross-attention work in the decoder?

Answer

- **Self-attention:** Q, K and V all come from the **same sequence** – tokens attend to each other within a sentence (used in encoder and decoder).
- **Cross-attention: Q comes from one sequence (decoder)** and **K and V come from another (encoder output)**.

How cross-attention works in the decoder: after the masked self-attention sub-layer, each decoder position creates a query from its current representation and attends over all encoder outputs (keys/values). This lets the decoder decide which parts of the source sentence are relevant for the word it is generating now (e.g. when generating the French word, look at the corresponding English word). Cross-attention is also how text conditions image generation in diffusion models.

Q9. How do Transformers account for word order? Explain positional encoding using sine and cosine. ★ · Asked: NLP interview rounds

Answer

Self-attention is **permutation-invariant** – without extra information, "dog bites man" and "man bites dog" look identical. So Transformers **add a positional encoding vector to each token embedding** before the first layer.

Sinusoidal formula (original Transformer):

PE(pos, 2i) = sin(pos / 10000^(2i/d_model))

PE(pos, 2i+1) = cos(pos / 10000^(2i/d_model))

Even dimensions use sine, odd use cosine; each dimension pair has a different wavelength (from 2π to 10000·2π), like the hands of a clock running at different speeds.

Why sine/cosine: unique encoding for every position; values bounded in [−1, 1]; PE(pos + k) is a linear function of PE(pos), so the model can easily learn **relative positions**; no parameters; can extrapolate to longer sequences than seen in training.

Q10. If the input embedding dimension is 1024, what is the dimension of the positional encoding vector? Why?

Answer

1024. The positional encoding is **added element-wise** to the input embedding, so it must have exactly the same dimension as the embedding (d_model). For a sequence of length n, the PE matrix has shape (n, 1024).

Q11. What are residual (skip) connections and why are they needed?

Answer

Residual (skip) connections add a sub-layer's input to its output: **output = x + SubLayer(x)**. In a Transformer, each attention and feed-forward sub-layer is wrapped as LayerNorm(x + SubLayer(x)) (or x + SubLayer(LayerNorm(x))).

Why needed:

- Gradients can flow directly backward through the "+ x" path → avoids vanishing gradients and allows very deep stacks (96+ layers).
- Each layer only learns a **refinement (residual)** of the representation, which is easier to optimize.
- Preserves the original token/positional information through the layers.

Q12. What is layer normalization and why is it used in Transformers? Pre-LN vs Post-LN?

Answer

Layer normalization normalizes each token vector across its features: $\hat{x} = (x - \mu)/\sigma$, then $y = \gamma\hat{x} + \beta$ (learnable).

Why in Transformers: stabilizes activations and gradients, speeds up training, and – unlike BatchNorm – works independently of batch size and sequence length, and identically at training and inference.

- **Post-LN** (original paper): LayerNorm(x + SubLayer(x)). Can be unstable for deep models and needs learning-rate warm-up.
- **Pre-LN** (GPT-2, Llama and most modern LLMs): x + SubLayer(LayerNorm(x)). More stable gradients, easier to train deep models. Many LLMs use **RMSNorm** (no mean subtraction, cheaper).

Q13. What is the role of the feed-forward network in each Transformer block?

Answer

After attention, each Transformer block has a **position-wise feed-forward network** applied to each token independently:

$FFN(x) = W_2 \cdot \text{activation}(W_1x + b_1) + b_2$, with an inner dimension usually **4× d_model** (768 → 3072 → 768). Activation: ReLU (original), GELU (BERT/GPT), SwiGLU (Llama).

Role: attention **mixes information between tokens**; the FFN **processes each token's information** with non-linearity. Research suggests the FFN layers act as a **key–value memory that stores factual knowledge**. They hold about two-thirds of the model's parameters.

Q14. What is masked (causal) attention and why is it used in the decoder? ★

Answer

Masked (causal) self-attention prevents a position from attending to **future positions**. Before softmax, scores for future tokens are set to $-\infty$ using an upper-triangular mask, so their attention weights become 0.

Why: the decoder generates text left-to-right. During training we feed the whole target sentence at once (for parallelism), so without the mask the model could "cheat" by looking at the next word it is supposed to predict. The mask makes training match inference, where future tokens do not exist yet. All GPT-style decoder-only LLMs use causal masking. (A separate **padding mask** hides [PAD] tokens.)

Q15. What is the role of the final linear + softmax layer in the decoder?

Answer

The decoder's final hidden state for each position (dimension d_model) is passed through a **linear layer** that projects it to the **vocabulary size** (e.g. 768 → 50,257), producing **logits**. **Softmax** converts these logits into a **probability distribution over all vocabulary tokens**. The next token is chosen from this distribution (argmax for greedy, or sampling with temperature/top-k/top-p). During training, cross-entropy loss is computed between this distribution and the true next token.

Q16. What is teacher forcing and why is it significant? What is exposure bias?

Answer

Teacher forcing: during training, the decoder receives the **ground-truth previous token** as input at each step, instead of its own (possibly wrong) prediction.

Significance: faster and more stable convergence (errors don't compound during training), and in Transformers it allows the **whole target sequence to be processed in parallel** (with causal masking).

Exposure bias: at inference the model must use its own predictions, which it never saw during training. One early mistake can cause errors to snowball. Mitigations: **scheduled sampling** (gradually replace ground truth with model predictions), beam search, sequence-level training/RL fine-tuning.

Q17. Explain the complete Transformer architecture end-to-end (draw it). ★ ·

Asked: TCS, Google

Answer

Encoder (N = 6 identical layers):

1. Input tokens → token embeddings (× √d_model) + positional encodings.
2. Each layer: **multi-head self-attention** → Add & LayerNorm → **feed-forward network** → Add & LayerNorm.
3. Output: a contextual vector for every input token.

Decoder (N = 6 layers):

1. Shifted-right target tokens → embeddings + positional encodings.
2. Each layer: **masked multi-head self-attention** → Add & Norm → **cross-attention** (Q from decoder, K/V from encoder output) → Add & Norm → **FFN** → Add & Norm.

Output head: Linear → Softmax over vocabulary → next-token probabilities.

Original base model: d_model = 512, 8 heads, d_ff = 2048, dropout 0.1, Adam with warm-up. Training uses teacher forcing; inference is autoregressive (generate one token, append, repeat).

(Draw it: two stacks side by side, encoder output arrows feeding into every decoder layer's cross-attention.)

Q18. What is the time and memory complexity of self-attention with respect to sequence length? ★

Answer

For sequence length n and model dimension d:

- **Time:** $O(n^2 \cdot d)$ – every token computes a score with every other token (the n × n attention matrix).
- **Memory:** $O(n^2)$ for the attention matrix (per head, per layer), plus $O(n \cdot d)$ activations.

Doubling the context length makes attention **4× more expensive**. This is why long-context models use Flash Attention (reduces memory to O(n) by not storing the full matrix), sliding-window/sparse attention, and KV caching during generation. (RNNs are $O(n \cdot d^2)$ but sequential.)

Q19. Advantages of Transformers over traditional seq2seq (RNN) models.

Answer

- **Parallel training** across all positions (no sequential recurrence) → much faster on GPUs/TPUs.
- **Direct long-range connections** between any two tokens; no information bottleneck.
- **No vanishing gradients through time** (residual connections + attention).
- **Scales** extremely well with data and parameters → foundation models and transfer learning (pre-train once, fine-tune for many tasks).
- **Better accuracy** on translation, summarization, QA, etc.
- Attention weights give some **interpretability**.
- One architecture works for text, images (ViT), audio (Whisper) and multimodal models.

Q20. Implement scaled dot-product attention from scratch in NumPy or PyTorch. [CODE]

Answer

```
import torch, torch.nn.functional as F, math

def scaled_dot_product_attention(Q, K, V, mask=None):
    # Q, K, V: (batch, heads, seq_len, d_k)
    d_k = Q.size(-1)
    scores = Q @ K.transpose(-2, -1) / math.sqrt(d_k)      # (b, h, n, n)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, float("-inf"))
    weights = F.softmax(scores, dim=-1)
    return weights @ V, weights

# Example with a causal mask
b, h, n, d = 1, 2, 4, 8
Q, K, V = torch.randn(b, h, n, d), torch.randn(b, h, n, d), torch.randn(b, h, n, d)
causal = torch.tril(torch.ones(n, n))
out, w = scaled_dot_product_attention(Q, K, V, causal)
print(out.shape)        # torch.Size([1, 2, 4, 8])
print(w[0, 0])          # lower-triangular weights, each row sums to 1

# NumPy version
import numpy as np
def attention_np(Q, K, V):
    s = Q @ K.T / np.sqrt(K.shape[-1])
    s = np.exp(s - s.max(axis=-1, keepdims=True))
    return (s / s.sum(axis=-1, keepdims=True)) @ V
```

PyTorch 2.x also provides an optimized built-in: `F.scaled_dot_product_attention(Q, K, V, is_causal=True)`.

Q21. How do LLMs understand words with multiple meanings, e.g. "bank" of a river vs a financial "bank"? ★ · Asked: EPAM

Answer

Through **contextual embeddings built by self-attention**:

1. The token "bank" starts with the **same input embedding** in both sentences.
2. In every Transformer layer, self-attention lets "bank" **attend to the surrounding words** – "river", "water", "sat" in one sentence; "deposit", "loan", "money" in the other.
3. The weighted mix of those context vectors is added to the "bank" representation layer after layer, so by the upper layers the two "bank" vectors are **completely different** – one close to finance concepts, the other close to geography.
4. Multi-head attention lets different heads capture different cues (nearby words, syntax, topic).

This is the key advantage over static embeddings like Word2Vec, which give "bank" one fixed vector.