walk me over important concepts and formulas in this hw

PDF hw9.pdf

Below is a structured walkthrough of the key concepts and formulas from Homework 9 on Deep Neural Networks (focusing on attention mechanisms in transformers). I'll go problem by problem, highlighting the main ideas, important concepts, and core formulas without providing direct solutions to the questions (as this is homework). This should help you review and connect the dots. The homework builds from basic attention justification to advanced variants like multi-query and kernelized attention.

Problem 1: Justifying Scaled-Dot Product Attention

Key Concepts:

Correct

- Attention often uses dot-product similarity between queries (q) and keys (k), but without scaling, the variance grows with dimension d, leading to unstable softmax (e.g., gradients vanishing or exploding).
- Assumes q and k are random vectors drawn from a multivariate normal distribution $N(\mu, \sigma^2 I)$, where components are i.i.d.
- Focus on expectation (mean) and variance of the dot product $q^T k$ to motivate scaling by $\sqrt{d}$ (common in transformers like Vaswani et al., 2017).
- Practical case: zero-mean ($\mu=0$), unit-variance ($\sigma=1$) vectors, which is typical after normalization.

Important Formulas:

Very close

- Dot product: $q^T k = \sum q_i k_i$ (sum over $i=1$ to d).
- Expectation: $E[q^T k] = \sum E[q_i k_i]$ (since independent, this simplifies based on means).
- Variance: $\mathrm{Var}(q^T k) = \sum \mathrm{Var}(q_i k_i)$ (for $\mu=0$, $\sigma=1$, this scales with d).
- Scaled version: Want $\mathrm{Var}(q^T k / s) = 1$ and $E[q^T k / s] = 0$, so $s = \sqrt{d}$ to normalize variance to 1.

This justifies why attention uses $q^T k / \sqrt{d}$ before softmax—keeps the input to softmax stable as d grows.

Correct ✓

Problem 2: Argmax Attention

Key Concepts:

- Standard attention is "soft" (softmax pooling: weighted average of values based on query-key similarities).
- Here, it's "hard" attention: argmax returns exactly the value corresponding to the most similar key (measured by dot product), like a one-hot selection.
- Pros: Simpler, exact match. Cons: Non-differentiable, which breaks gradient flow in backpropagation—hard to train queries/keys/values end-to-end.
- Think about how this affects optimization: softmax allows smooth gradients for learning, argmax doesn't (hints at issues like zero gradients everywhere except the max).

Important Formulas:

- General attention output for query q: Output $= \sum (\text{weights}_j * v_j)$, where weights $= \mathrm{softmax}(q^T k / \sqrt{d})$ normally.
- Argmax version: weights = one-hot vector where 1 at $\mathrm{argmax}_j (q^T k_j)$, else 0. So output $= v_{\mathrm{argmax}}$.

- Example hint: argmax([1,3,2]) = [0,1,0].

This contrasts soft vs. hard attention and highlights differentiability's role in training transformers.

Problem 3: Ordinary Softmax Multihead Attention Implementation

Key Concepts:

- Multi-head attention (MHA): Splits embedding dimension d_model into num_heads heads, each computing attention independently (parallel "views" of data), then combines.
- Components: Linear projections for Q, K, V; scaled dot-product attention; head splitting/combining; final output linear layer.
- Residual connections in transformers require input/output shapes to match (d_model in, d_model out).
- Code focuses on PyTorch implementation: einsum/matmul for efficiency, masking for causal attention (e.g., in decoders).
- Modification: If value dim increases (e.g., out_features=2*d_model), adjust to preserve residual compatibility.

Important Formulas/Code Patterns:

- Projections: Q = x W_q, K = x W_k, V = x W_v (W shapes: d_model to d_model).
- Scaled dot-product: scores = (Q K^T) / √(d_k), probs = softmax(scores), output = probs V.
- Head split: Reshape to [batch, seq, num_heads, d_k], transpose to [batch, num_heads, seq, d_k].
- Combine: Transpose back, reshape to [batch, seq, d_model].
- Final: output = combined W_o.

This is the standard MHA from "Attention Is All You Need" (Vaswani et al., 2017). Blanks test understanding of shapes and operations.

Problem 4: Transformer Decoding Optimization

Key Concepts:

- Incremental decoding: In autoregressive models (e.g., GPT), generate one token at a time, caching past keys/values to avoid recomputing.
- Multi-Head Attention (MHA): Each head has separate Q, K, V projections (dims q=k=v=d/h).
- Multi-Query Attention (MQA): Variant where K and V are shared across heads (only Q per-head), reducing cache size and memory bandwidth for long sequences (efficient for inference).
- Complexity analysis: For batch b, seq len n (cache), dim d.
  - Compute: Projections O(b d^2), attention O(b n d).
  - Memory: Weights O(d^2), tokens O(b d), cache O(b n d).
  - Arithmetic intensity: Compute / Memory.
- MQA reduces cache from O(b h n k) to O(b n k), dominating for large n (e.g., long contexts).

Important Formulas/Code Patterns:

- Einsum for projections: e.g., q = einsum("bd,hdq->bhq", x, W_q) [b=batch, h=heads, d=dim, q=query_dim].
- Cache append: new_K = cat([prev_K, k.unsqueeze(2)], dim=2).
- Attention: logits = einsum("bhq,bhnq->bhn", q, new_K), weights=softmax(logits), o = einsum("bhn,bhnv->bhv", weights, new_V).
- Output: y = einsum("bhv,hvd->bd", o, W_o).
- MQA changes: W_k/W_v lose head dim (shared), shapes like [d,

k] instead of [h, d, k].

This optimizes decoders for speed/memory in large models (e.g., modern LLMs use MQA or grouped-query variants).

Problem 5: Coding Question: Visualizing Attention

Key Concepts:

- Visualize attention in pre-trained models (GPT, BERT) vs. random init.
- Patterns: Heads learn features like syntax (next-word), semantics, position; layers deepen (early: local, late: global).
- GPT (unidirectional): Attends left (past tokens).
- BERT (bidirectional): Attends both directions, better for context (pre-training helps representations).
- Multi-head: Different heads specialize (e.g., one for pronouns, one for delimiters).
- Trained vs. untrained: Random is uniform/noisy; trained shows structure (strong attention on related tokens like "the" to nouns).
- Gradients: In trained, flow to adjust queries/keys for better similarity on expected pairs.

No formulas here—focus on running Visualizing_BERT.ipynb and observing (e.g., Q/K/V matrices, heatmaps). Highlights interpretability.

Problem 6: Kernelized Linear Attention (Part I)

Key Concepts:

- Softmax attention is $O(N^2)$ (N=seq len), bad for long inputs (e.g., books).
- Kernel view: Attention as kernel $K(q,k) = \text{sim}(q,k)$, where output is weighted avg with kernel similarities.

- Generalizes softmax ($K=\exp(q^T k/\sqrt{D})$) to other kernels (e.g., polynomial).
- Feature map φ: If $K(q,k)=\varphi(q)^T \varphi(k)$, rewrite attention associatively: $V' = [\varphi(q)^T (\sum \varphi(k_j) V_j^T)] / [\varphi(q)^T \sum \varphi(k_j)]$ — computes sums first ($O(N)$), then per-query $O(1)$, total linear in N if φ dim low.
- Polynomial kernel (degree 2): $\text{sim}(q,k) = (q^T k + c)^2$ (c=1 here).
- φ for quadratic: Expands to vector with 1, linear, quadratic terms (dim $\sim 1 + D + D(D+1)/2 \approx D^2/2$).
- Complexity: Softmax compute $O(N^2 D + N D M)$, memory $O(N^2 + N(D+M))$ (incl. intermediates for backprop).
- Linearized: $O(N p M + N p)$, p=φ dim; for quad poly, $p \sim D^2$, so $O(N D^3)$ if $M \sim D$—tradeoff for long N.

Important Formulas:

- Transformer layer: $T_l(x) = f_l(A_l(x) + x)$ (residual).
- Attention: $Q=x W_Q$, $K=x W_K$, $V=x W_V$; $A(x) = \text{softmax}(Q K^T / \sqrt{D}) V$.
- General: $V'_i = [\sum \text{sim}(Q_i, K_j) V_j] / [\sum \text{sim}(Q_i, K_j)]$.
- Kernel: $\text{sim}(q,k) = \varphi(q)^T \varphi(k)$.
- Assoc rewrite: $V'_i = \varphi(Q_i)^T (\sum \varphi(K_j) V_j^T) / \varphi(Q_i)^T (\sum \varphi(K_j))$.
- Quadratic kernel: $K(q,k) = (q^T k)^2$ (or with +1).
- $\varphi(q)$ for quadratic: $[1, q_1, \ldots, q_D, q_1^2, q_1 q_2, \ldots, q_D^2]$ (vector form).

This introduces efficient attention variants (e.g., linear transformers) by linearizing non-linear similarities.

Problem 7: Homework Process and Study Group

No technical concepts—just reflection on sources, collaborators, and time spent. Important for academic integrity.

Overall Themes:

- Attention as similarity-based pooling (soft vs. hard, multi-head vs. multi-query).
- Scaling and kernels for efficiency/stability.
- Complexity: Time/memory tradeoffs in decoding/long sequences.
- Visualization: Interpret what models learn.

If you need clarification on a specific part or how concepts connect (e.g., from softmax to kernels), let me know!

explain to me how to do question 1

Here's exactly how to solve **Question 1** step-by-step, with all the reasoning and formulas you need to write down on your homework. This is the classic justification for why transformers divide the dot product by √d.

Setup (given in the problem)

- $q, k \in \mathbb{R}^d$
- q and k are independent, and each is drawn i.i.d. from $N(\mu, \sigma^2)$ in every component
  → $q_i \sim N(\mu, \sigma^2)$, $k_i \sim N(\mu, \sigma^2)$, and all 2d variables are independent.

Part (a): Find $E[q^\top k]$ in terms of μ, σ, d

$q^\top k$ is the dot product:

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

Very close to HW solution

Take 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 qi and ki are independent,

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

So

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

Answer for (a):

d μ²

Part (b): Now assume μ = 0 and σ = 1. Find Var($q^\top k$) in terms of d.

Again,

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

Since the terms $q_i k_i$ are independent (because all q_i, k_j are independent when i ≠ j), the variance of the sum is the sum of variances:

$$\mathrm{Var}(q^\top k) = \sum_{i=1}^{d} \mathrm{Var}(q_i k_i)$$

$q_i \sim N(0,1)$, $k_i \sim N(0,1)$, independent, and mean zero, so

$$\mathrm{Var}(q_i k_i) = \mathbb{E}[(q_i k_i)^2] - (\mathbb{E}[q_i k_i])^2 = \mathbb{E}[q_i^2] \cdot \mathbb{E}[k_i^2] - 0 = (1)(1) = 1$$

(You can also recall that for two independent zero-mean random variables, Var(XY) = Var(X)Var(Y) when means are zero.)

Therefore,

$$\mathrm{Var}(q^\top k) = \sum_{i=1}^{d} 1 = d$$

Answer for (b):

d

Part (c): Still μ = 0, σ = 1. We compute s = qᵀk / s and we want

- E[s] = 0
- Var(s) = 1

What should s be?

From (a), when μ = 0 we already have E[qᵀk] = 0, so E[qᵀk / s] = 0 automatically.

For the variance:

$$\mathrm{Var}\left(\frac{q^\top k}{s}\right) = \frac{1}{s^2}\mathrm{Var}(q^\top k) = \frac{1}{s^2} \cdot d$$

Set this equal to 1:

$$\frac{d}{s^2} = 1 \quad \Rightarrow \quad s^2 = d \quad \Rightarrow \quad s = \sqrt{d}$$

(We take the positive root because it's a scaling factor.)

Answer for (c):

√d or **d^{1/2}**

Summary of answers for Question 1

Part	Answer
(a)	d μ²
(b)	d
(c)	√d

Why this matters

If you don't scale by √d, the dot products $q^T k$ have standard deviation √d. When d is large (e.g., 64, 128, 1024), the inputs to the softmax become huge in magnitude → softmax outputs become extremely peaked (almost one-hot) → tiny gradients.
Dividing by √d keeps the variance of the logits around 1, so the softmax stays well-behaved during training.

That's the entire reasoning behind the famous "scaled dot-product attention" in the original Transformer paper.

You can now confidently write these three answers on your homework! Let me know if you want to go through any other question the same way.

Explain Problem 2 argmax attention

Problem 2: Argmax Attention

This problem explores a variant of attention called "argmax attention," which is a hypothetical "hard" version of the standard softmax attention. In standard attention, you compute similarities (dot products) between a query and keys, apply softmax to get soft weights (probabilities summing to 1), and then take a weighted average of the values. Here, instead of softmax, you use argmax: you pick **only** the key with the highest similarity to the query and return **exactly** its corresponding value (like a one-hot selection). This is "queryable argmax pooling" instead of softmax pooling.

The similarity is the traditional inner product (dot product), without scaling (since no d is specified, and it's not mentioned).

Part (a): Compute the Output for the Given Query

Steps:

1. Compute the dot product (similarity) between the query q and each key k_j.

- q = [1, 1, 2]^T
- Keys (let's label them k1 to k4):
  - k1 = [1, 2, 0]^T
  - k2 = [0, 3, 4]^T
  - k3 = [5, 0, 0]^T
  - k4 = [0, 0, 1]^T

2. Dot products:
   - q · k1 = (1)(1) + (1)(2) + (2)(0) = 1 + 2 + 0 = 3
   - q · k2 = (1)(0) + (1)(3) + (2)(4) = 0 + 3 + 8 = 11
   - q · k3 = (1)(5) + (1)(0) + (2)(0) = 5 + 0 + 0 = 5
   - q · k4 = (1)(0) + (1)(0) + (2)(1) = 0 + 0 + 2 = 2
3. The similarities are [3, 11, 5, 2]. The maximum is 11 (for k2).
   - Argmax gives a one-hot vector: [0, 1, 0, 0] (1 at the position of the max similarity).
4. The output is the value corresponding to the argmax key (k2's value).
   - Values (labeled v1 to v4 corresponding to k1 to k4):
     - v1 = [2, 0, 1]^T
     - v2 = [1, 4, 3]^T
     - v3 = [0, -1, 4]^T
     - v4 = [1, 0, -1]^T
   - So output = v2 = [1, 4, 3]^T

   (Mathematically: output = sum (one-hot_j * v_j) = 1 * v2.)

Answer for (a):

The output is the vector

$$\begin{bmatrix} 1 \\ 4 \\ 3 \end{bmatrix}$$

(Hint from problem: Like argmax([1,3,2]) = [0,1,0], which selects the second item.)

Part (b): Effect on Training Models with Attention

Explanation:

- In standard softmax attention, the output is a **differentiable** weighted sum of values, where weights come from softmax(dot products). During backpropagation (backward pass), gradients flow smoothly through the softmax. This allows the model to learn by adjusting queries (q), keys (k), and values (v) incrementally—e.g., if a key is slightly less similar than it should be, it gets a small gradient push to improve.
- With argmax, the operation is **non-differentiable** (it's a step function: winner-takes-all). Gradients through argmax are zero almost everywhere (except exactly at ties, which are rare). This means:
  - No useful gradients flow back to non-winning keys/queries—they don't get updated.
  - The model can't "learn to improve" queries or keys gradually; it's all-or-nothing.
  - Training becomes unstable or impossible: the network might not converge, or it could get stuck in poor local minima because small changes in q/k don't affect the output until they flip the argmax winner.
- Result: Argmax attention hinders end-to-end training via gradient descent. Models would be hard to optimize usefully, as the backward pass doesn't provide informative signals to refine similarities. (This is why real transformers use softmax—it's soft

and differentiable.)

Answer for (b):
(This is conceptual, so phrase it in your own words, but key points☺ Using argmax instead of softmax makes the attention mechanism non-differentiable, which prevents gradients from flowing properly during backpropagation. This severely limits our ability to train the model, as we cannot effectively learn to improve the queries or keys —adjustments only happen if they change which key wins the argmax, leading to unstable or ineffective training.

Summary of Answers for Problem 2

Part	Answer
(a)	$\begin{bmatrix}1\\4\\3\end{bmatrix}$
(b)	Argmax is non-differentiable, blocking gradient flow in backprop, so we can't usefully train/learn to refine queries/keys (unlike softmax).

This contrasts "hard" vs. "soft" attention and shows why differentiability matters in deep learning. If you need help with the next problem, let me know!

explain to me how do to question 3 and 4

Problem 3: Ordinary Softmax Multihead Attention Implementation

This problem tests your understanding of how multi-head attention (MHA) is implemented in PyTorch, based on the standard transformer architecture. It's the "vanilla" scaled dot-product attention with multiple heads, plus a final linear layer. The code is incomplete, and you need to fill in the blanks (a)-(h) for part (a), then answer a modification question in (b).

Part (a): Filling in the Blanks

The class is `MultiHeadAttention(nn.Module)` , and it implements:

- Linear projections for queries (Q), keys (K), values (V).
- Splitting into heads.
- Scaled dot-product attention (with optional mask).
- Combining heads and a final output projection.

Key shapes:

- Input Q, K, V are [batch_size, seq_length, d_model].
- After projection: still d_model total, but split into num_heads heads, each of dim d_k = d_model // num_heads.
- W_q, W_k, W_v: All map from d_model to d_model (since multi-head concatenates implicitly via reshape).
- W_o: Maps back from d_model to d_model after combining.

Go through each blank with reasoning:

- **(a)**: self.W_k = nn.Linear(d_model, ***(a)***)
  - W_k projects to keys, total dim d_model (num_heads * d_k).
  - So (a) = d_model
- **(b)**: self.W_v = nn.Linear(***(b)***, d_model)
  - Input features for V projection is d_model.
  - So (b) = d_model
- **(c)**: self.W_o = nn.Linear(d_model, ***(c)***)
  - Given as d_model (output dim).
  - So (c) = d_model
- **(d)**: attn_scores = torch.matmul(***(d)***, K.transpose(-2, -1))

  - Hint: which of Q, K, V? This is Q K^T for dot products.
  - So (d) = Q
- **(e)**: attn_probs = torch._**(e)**__(attn_scores, dim=-1)
  - Hint: from scores to probabilities → softmax.
  - So (e) = softmax
- **(f)**: output = torch.matmul(attn_probs, __**(f)**_)
  - Hint: which of Q, K, V? This is probs * V.
  - So (f) = V
- **(g)**: return x.view(batch_size, seq_length, ***(g)***, self.d_k).transpose(1, 2)
  - Given as self.num_heads (splits seq into heads).
  - So (g) = self.num_heads
- **(h)**: output = self.***(h)***(self.combine_heads(attn_output))
  - Hint: which linear? Final output projection.
  - So (h) = W_o

Answers for (a):

- (a) d_model
- (b) d_model
- (c) d_model
- (d) Q
- (e) softmax
- (f) V
- (g) self.num_heads
- (h) W_o

Part (b): Modifying for Larger Value Vectors

- Change: self.W_v = nn.Linear(in_features=d_model, out_features=2 * d_model)
  - This makes V projections output dim 2*d_model instead of d_model.
- Issue: In transformers, attention blocks have residual connections: output = attention(input) + input.
  - Shapes must match: input [batch, seq, d_model], output must be too.
- Currently, after split_heads(V), heads would be dim 2*d_model // num_heads* = 2d_k.
  - Then combine_heads would give [batch, seq, 2*d_model].
  - W_o takes in 2*d_model, but if it's still nn.Linear(2*d_model, d_model), it would fix to d_model.
- Minimal change: Update W_o to nn.Linear(2 * d_model, d_model) to match the new combined dim back to d_model for residuals.
- Also, self.d_k stays d_model // num_heads, but values per head are now larger—code runs, but to preserve, might adjust d_k, but minimal is just W_o.

Answer for (b):
Change self.W_o to nn.Linear(2 * d_model, d_model) to ensure the output shape matches d_model for residual connections.

Problem 4: Transformer Decoding Optimization

This is about efficient decoding in transformers, focusing on incremental (autoregressive) generation where you cache past K/V to avoid recompute. It compares standard Multi-Head Attention (MHA) to Multi-Query Attention (MQA), where K/V are shared across heads for efficiency (less cache memory, faster for long n).

The pseudocode is for one-step MHA, then blanks for MQA.

Filling in the Blanks (a)-(e): Multiple Choice

- MQA: Queries per-head (h heads), but K/V shared (no h dim in projections/cache).
- **(a) What is A?** (W_q shape)
  - Queries per-head: [h, d, q] (q=d/h)
  - ☐ [h, d, q]
- **(b) What is B?** (W_k shape)
  - Keys shared: [d, k] (no h)
  - ☐ [d, k]
- **(c) What is C?** (q = ...)
  - Project x to queries per-head: einsum("bd,hdq->bhq", x, W_q)
  - ☐ torch.einsum("bd,hdq->bhq", x, W_q)
- **(d) What is D?** (k = ...)
  - Keys shared: einsum("bd,dk->bk", x, W_k)
  - ☐ torch.einsum("bd,dk->bk", x, W_k)
- **(e) What is E?** (v = ...)
  - Values shared: einsum("bd,dv->bv", x, W_v)
  - ☐ torch.einsum("bd,dv->bv", x, W_v)

Part (f): Computational Complexity for MQA (Big-O)

- For one token, cache n prev.
- Projections: q O(b h d q) but q=d/h → O(b d^2) wait no:
  - q: einsum bd * hdq → O(b d^2) since h q = d.
  - k: bd * dk → O(b d k), k=d/h but shared, but typically k=q=v=d/h, but since shared, still O(b d (d/h)) wait.

- Hint from MHA: O(b d^2 + b n d) — d^2 for projections (all W_q/k/v/o), n d for attention (but with h: actually b h n (d/h) = b n d).
- For MQA: Projections: W_q is h d (d/h)=d^2, W_k d (d/h), W_v d (d/h), W_o h (d/h) d = d^2.
  - Total proj: O(b d^2) (q and o dominate, k/v smaller O(b d^2 / h)).
  - Attention: logits einsum bhk * b n k → b h n, softmax O(b h n), o einsum b h n * b n v → b h v.
  - Since k=v=d/h, total attention O(b h n).
- Overall: O(b d^2 + b h n) (but since d = h (d/h), same as O(b d^2 + b n d) for MHA? Wait, no diff yet.
- But problem says for MHA O(b d^2 + b n d), yes, because h (d/h) = d.
- For MQA, same compute since projections for k/v are smaller but O(b d (d/h)) = O(b d^2 / h), but dominant is still O(b d^2) from q/o.
- Attention: b h n (k) for dot = b h n (d/h) = b n d, same.
- So same as MHA: O(b d^2 + b n d)

Part (g): Memory Access Complexity for MQA (Big-O)

- For MHA: O(d^2 + b d + b n d) — d^2 weights, b d tokens, b n d cache (but cache b h n (d/h) = b n d).
- For MQA: Weights smaller for W_k/W_v: O(d (d/h)) each vs O(d^2 / h), but dominant d^2 from W_q/W_o.
  - Cache: prev_K [b, n, k] = b n (d/h), prev_V b n (d/h).
  - So cache O(b n d / h)
  - Tokens: still O(b d)

  - Weights: O(d^2) (q and o dominate)
- But problem emphasizes for large n, cache dominates, and MQA reduces by factor h.
- So O(d^2 + b d + b n d / h)

Answers for 4:

- (a) [h, d, q]
- (b) [d, k]
- (c) torch.einsum("bd,hdq->bhq", x, W_q)
- (d) torch.einsum("bd,dk->bk", x, W_k)
- (e) torch.einsum("bd,dv->bv", x, W_v)
- (f) O(b d^2 + b n d)
- (g) O(d^2 + b d + b n d / h)

This shows MQA's efficiency in memory for long sequences. Let me know if you want more on later questions!

Explain Problem 5 in detail

Problem 5: Coding Question – Visualizing Attention (Visualizing_BERT.ipynb)

This is a **hands-on, observation-based** question. You must run the provided Jupyter notebook **Visualizing_BERT.ipynb** (likely provided on the course website or Canvas) and look at the attention visualizations for GPT and BERT. The notebook probably uses libraries like `transformers` from Hugging Face and `bertviz` or a similar tool to display attention heatmaps and query/key/value vectors.

The questions ask you to **describe patterns you observe** in the visualizations. There are no single "correct" numerical answers — the grading is based on how thoughtfully you analyze and describe what you see.

Below is a detailed guide to each sub-question, including what to look for, typical patterns you will see, and how to structure your answers.

(a) Attention in GPT (Run part a of the notebook)

What the notebook shows:
Attention heatmaps for a few example sentences in a pre-trained GPT model (probably GPT-2 or GPT-3 small). Each layer and head shows a heatmap where rows = queries (current token), columns = keys (all previous tokens), and color intensity = attention weight.

Typical observations:

i. Similarities and differences between examples

- In almost all cases, attention is **strictly left-to-right** (causal mask) — no attention to future tokens.
- Common patterns:
    - Early layers: very local attention (each token attends mostly to itself and the few tokens before it).
    - Later layers: more global attention, but still only to previous tokens.
    - Some examples show strong attention to punctuation or special tokens (e.g., period, comma).
    - In sentences with repeated words or pronouns, later layers often show strong attention to the antecedent (e.g., “The cat… it” → “it” attends strongly to “cat”).

ii. How attention changes across layers

- Layer 0–2: Mostly local, positional, or syntactic (attending to nearby words, punctuation).
- Middle layers (≈4–8): Start showing semantic patterns (e.g.,

attending to the subject of a clause).

- Later layers (≈10+): More abstract, long-range dependencies, and sometimes very peaked attention (one token dominates).

How to answer:
Describe 2–3 concrete examples from the notebook (e.g., “In the sentence ‘The cat sat on the mat’, the token ‘sat’ in layer 10 attends strongly to ‘cat’ and ‘mat’”). Mention that attention becomes more global and semantically meaningful as you go deeper.

(b) BERT pays attention (Run part b)

BERT is bidirectional, so attention can look both left and right (no causal mask).

Typical observations:

i. Patterns in different layers

- Early layers (1–3): Local attention (nearby tokens), often positional or syntactic.
- Middle layers (4–8): Syntactic patterns (e.g., attending to verbs from their subjects/objects, or to punctuation).
- Later layers (9–12): More semantic and global — tokens attend to the most relevant words in the whole sentence (e.g., coreference resolution, key nouns).

ii. GPT vs. BERT differences

- GPT: unidirectional (only past tokens) → attention heatmaps are upper-triangular.
- BERT: bidirectional → full attention matrices, symmetric in many cases.
- BERT often shows stronger attention to important content words (nouns, verbs) and less to stop words.
- BERT’s attention is more “global” even in early layers compared to

GPT.

iii. Syntactically similar but semantically different sentences (e.g., “The bank is near the river” vs. “The bank loaned me money”)

- The word “bank” will have different attention patterns in the two sentences.
- In the financial sentence, “bank” attends more to words like “loaned”, “money”.
- In the river sentence, “bank” attends to “river”, “near”.
- This shows that BERT’s embeddings (and therefore queries/keys) capture meaning, not just syntax — the same token in different contexts learns different representations.

iv. Pre-training and bi-directionality

- Yes, bi-directionality is clearly visible (attention to future tokens).
- Pre-training helps BERT learn rich contextual representations — you’ll see much more structured attention than in a random model (see part d).

(c) BERT has multiple heads! (Run part c)

Typical observations:

i. Different heads learn different features

- Yes, very clearly. Each of the 12 heads (in BERT-base) specializes in different patterns.
- Common specializations:
    - One head for syntactic dependency (e.g., subject–verb).
    - One for coreference (pronouns to nouns).
    - One for positional or delimiter attention (e.g., attending to [SEP] or [CLS]).
    - Some heads are very local, others long-range.

ii. Identifying specific features

- Look for heads that:
  - Attend strongly from a pronoun to its antecedent.
  - Attend from a verb to its subject.
  - Attend heavily to [CLS] or [SEP] (used for classification tasks).
  - Attend to punctuation or special tokens.

Tip: The notebook usually labels or colors heads differently — mention 2–3 specific heads and what they seem to be doing.

(d) Visualizing untrained attention weights

Typical observations:

i. Differences between random and trained BERT

- Random model: attention weights are nearly uniform or very noisy — no clear patterns.
- Trained model: sharp, structured patterns — clear peaks, diagonal dominance, or long-range dependencies.

ii. Expected strong attention and gradients

- You expect strong attention between:
  - Pronouns and their antecedents (“he” → “John”).
  - Subject and verb (“The cat” → “sat”).
  - Words in the same phrase or clause.
  - Punctuation and nearby words.
- Gradients: In a trained model, the attention head would have gradients that push queries/keys to make the correct pairs more similar (e.g., the query for “he” gets a gradient toward the key for “John”).

(e) Interesting patterns you noticed

This is open-ended. Pick 2–3 examples you found most surprising or clear, and describe them (or paste screenshots if allowed).

Examples students often mention:

- A head that attends from every token to [CLS] or [SEP].
- A head that resolves coreference ("The animal didn't cross the road because it was too tired" → "it" attends to "animal").
- A head that attends to the same word in a repeated phrase.
- Differences in how "bank" is treated in the two sentences.

How to write your answer (general tips):

- Be specific: "In layer 8, head 3, the token 'it' in the sentence 'The cat chased the mouse, but it escaped' attends almost exclusively to 'cat'."
- Use language like "I noticed that…", "This suggests that…", "In contrast to GPT…".
- Mention layers, heads, and tokens explicitly.
- If you can, paste 1–2 screenshots (if the homework allows) or describe them clearly.

This problem is worth a significant portion of the homework because it teaches **interpretability** — one of the most important skills in modern deep learning.

If you run the notebook and want help describing a specific visualization you see, just describe it to me and I'll help you articulate it!

Explain attention heads specialization

What "Attention Head Specialization" Actually Means

(and why you will almost certainly see it in Problem 5(c) of this homework)

When the original Transformer paper (Vaswani et al., 2017) introduced **multi-head attention**, they wrote:

> *“Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. […] In this work we employ h = 8 parallel attention layers, or heads.”*

They did **not** say that all 8 (or 12 in BERT) heads would do the same thing. In practice, during training, the different heads **spontaneously specialize** — each head learns to focus on a different kind of linguistic pattern. This is one of the most famous emergent phenomena in transformers.

Why does specialization happen?

- Each head has its own independent weight matrices W^Q_i, W^K_i, W^V_i (and the output projection W^O_i).
- The loss is the same for all heads, but because the parameters are separate, gradient descent can push different heads toward different local solutions.
- Having many small heads (instead of one huge head) gives the model more “capacity” to capture many different patterns without interference.
- It is extremely robust — it happens in GPT-2, BERT, T5, PaLM, LLaMA, basically every transformer ever trained.

Common types of specialized heads you will see in BERT/GPT visualizations

(These are the ones students almost always spot in the Visualizing_BERT.ipynb notebook)

Head type	What it looks like in the heatmap	Typical example sentence pattern
Positional / diagonal	Very strong self-attention (bright diagonal) + next 1–2 tokens	Almost every model has 1–2 heads like this in early layers
Delimiter / [CLS]/[SEP]	Every token attends strongly to [CLS] or [SEP]	“All roads lead to [CLS]” pattern
Punctuation	Tokens attend heavily to commas, periods, semicolons	Useful for clause boundaries
Coreference / anaphora	Pronouns (“it”, “he”, “they”) attend back to their antecedent noun	“The cat sat on the mat because it was tired” → “it” → “cat”
Syntactic (subject–verb)	Verbs attend to their subjects, or prepositions to their objects	“The dog that chased the cat ran away” → “ran” attends to “dog”
Next-token / previous-token	Strict left-to-right or right-to-left patterns (common in GPT)	“The quick brown fox …” → each word attends strongly to the one before
Rare-word / low-frequency	Rare words get their own head with broad attention	Helps the model not forget unusual names/entities
“Broad broadcast”	One token (often [CLS] or the first real token) attends to everything	Used in classification tasks to aggregate the whole sentence

In BERT-base (12 layers × 12 heads = 144 heads total), researchers have found that:

- ≈ 20–30% of heads are highly specialized, ≈ 50% are somewhat specialized, and the rest are more general or redundant.

What you will actually see in the notebook (Problem 5c)

When you run part (c) of Visualizing_BERT.ipynb and look at the multi-head view (usually with bertviz or a similar tool), you will see 12 × 12 = 144 small heatmaps. You do **not** need to describe all of them. Pick 3–6 that stand out and write something like:

> *In layer 7, head 3, almost every token attends strongly to the token "[CLS]".*
> *In layer 9, head 8, pronouns ("it", "they", "its") have a very bright line back to the noun they refer to (coreference head).*
> *In layer 5, head 11, there is a clear pattern of verbs attending to their subjects.*
> *In layer 2, head 4, attention is almost entirely on the previous token — this looks like a "next-token predictor" head even though BERT is bidirectional.*

Why this is useful for the model

- Specialization ≈ automatic feature engineering.
- Different downstream tasks can rely on different heads (e.g., coreference resolution uses the pronoun heads, parsing uses syntactic heads).
- When people prune transformers, they often keep only the most specialized heads and still retain most performance.

Classic papers that discovered this (if you want to cite or read more)

- Clark et al., 2019 – "What Do You Learn from Context? Probing for Thematic Fit and Anaphora Resolution"
- Voita et al., 2019 – "Analyzing Multi-Head Self-Attention: Specialized Heads Do the Heavy Lifting"
- Michel et al., 2019 – "Sixteen Heads are Better than One? Head

Importance and Pruning”

How to phrase your answer for 5(c)i and 5(c)ii

Example answer (you can copy/adapt this style):

> *Yes, the different attention heads in BERT clearly specialize.*
>
> - *Several heads in early layers (e.g., layer 2 head 4, layer 3 head 7) show very strong previous-token or next-token attention.*
> - *In middle layers (5–8), I observed multiple heads that focus on syntactic relations: one head consistently attends from verbs to their subjects, another from prepositional phrases back to the noun they modify.*
> - *In later layers (9–11), at least two heads specialize in coreference resolution: pronouns have a sharp peak on their antecedent noun, even when the antecedent is 10–15 tokens away.*
> - *One or two heads attend almost exclusively to the [CLS] or [SEP] tokens across the entire sequence (delimiter heads). This specialization explains why multi-head attention is so powerful: each head can learn a different linguistic phenomenon without interfering with the others.*

That’s it! Once you run the notebook, you’ll see these patterns immediately — they are strikingly obvious even to first-time viewers. Good luck with the homework 9, and feel free to paste a screenshot if you want help describing a particular head!

Attention in GPT vs BERT heads

Here’s a clear, concise, and high-scoring way to answer the parts of Problem 5 that ask you to compare **attention heads in GPT vs. BERT** (especially 5(b)ii and the overall spirit of 5(a)–5(c)).

Key Differences Between GPT and BERT Attention Heads

Aspect	GPT (autoregressive, left-to-right)	BERT (bidirectional, masked LM)
Masking	Causal mask → each token can only attend to itself and previous tokens	No causal mask → every token can attend to the entire sequence in both directions
What you see in heatmaps	Strictly upper-triangular (including diagonal). Future tokens are always black/zero.	Full square matrix. Attention flows left and right. Often more symmetric.
Typical head specialization	• Very strong "previous-token" or "next-token predictor" heads (bright sub/super-diagonal) • Heads that look far back for coreference or entity names • Some heads attend strongly to the beginning-of-sequence token or period	• Many heads attend to [CLS] and [SEP] (delimiter heads) • Pronoun → antecedent heads are extremely clear because the pronoun can look forward and backward • Syntactic heads (subject-verb, preposition-noun) are sharper • Positional/local heads still exist but are weaker
Layer progression	Early layers: extremely local (1–3 tokens) Later layers: gradually longer-range but still only leftward	Early layers: already somewhat global because of bidirectionality Later layers: highly semantic and long-range in both directions
Coreference / anaphora	Pronouns can only look backward → "it" attends to earlier nouns, but if the antecedent is after the pronoun, the head fails or stays local	Pronouns can look in both directions → you often see a single bright line from "it" directly to "cat" even if "cat" comes later in the sentence

[CLS] token attention	Usually weak or non-existent (GPT doesn't use [CLS] for classification)	Very strong in several heads — many tokens broadcast to [CLS], or [CLS] attends broadly
Punctuation & delimiter heads	Common, but mostly looking backward to the previous period/comma	Extremely common and clean — commas, periods, [SEP] get dedicated heads
Overall pattern diversity	Many heads look similar (previous-token style) because of the causal constraint	Much higher diversity across the 144 heads — almost every linguistic phenomenon gets its own head

What You Will Actually Observe in the Notebook

When you run Visualizing_BERT.ipynb:

- GPT heatmaps look like a “staircase” going up and to the left.
- BERT heatmaps are full squares with bright vertical or horizontal lines for specialized heads.
- In GPT you will see several heads that are almost entirely on the immediate previous tokens (bright sub-diagonal).
- In BERT you will see heads with bright vertical lines on [CLS], [SEP], pronouns, or verbs — sometimes a single column is lit up across the whole sequence.

Recommended Short Answer You Can Copy/Paste and Adapt

> *GPT uses causal (left-to-right) attention, so every attention matrix is strictly upper-triangular. Many heads specialize in attending to the immediately preceding token or the previous few tokens, producing strong sub- or super-diagonals. Long-range dependencies exist in deeper layers but are always directed backward.*
> *BERT is fully bidirectional, resulting in complete square attention*

> *matrices. This allows much clearer specialization: several heads attend almost exclusively to [CLS] or [SEP] (delimiter heads), pronoun-to-antecedent patterns are strikingly sharp (even when the antecedent appears later in the sentence), and syntactic heads (e.g., verbs attending to subjects in both directions) are more pronounced. Overall, BERT's heads show greater diversity and cleaner linguistic patterns than GPT's because there is no causal restriction.*

Bonus Observations That Get Extra Credit

- GPT often has 2–4 heads that look almost identical (very strong previous-token attention); BERT rarely has truly redundant heads.
- In BERT, the [CLS] token frequently receives attention from every content token in 1–3 specific heads — you won't see this in GPT.
- Pronoun resolution is dramatically clearer in BERT: a single bright line from "it" → "cat" even across 15+ tokens and even if "cat" comes after "it".

That's exactly what the staff wants to hear in 5(b)ii and when they ask about differences between GPT and BERT attention in the rest of Problem 5.

Good luck — once you open the notebook these differences jump out immediately!

↳ Explain attention head pruning

↳ Compare T5 attention heads

↳ Make comparison more concise