ATML - Summer Work

Assignment 1 - Transformers for Vision and Language

28 June 2025

Guidelines

Transformers have revolutionized both natural language processing (NLP) and computer vision by relying on self-attention mechanisms rather than strictly sequential or grid-structured processing. In this assignment, you will explore the dot-product self-attention at the heart of transformers, apply transformers to language and vision tasks, and investigate advanced attention variants. The assignment is modular, progressing from fundamental concepts to practical implementations and then to cutting-edge attention mechanisms.

No submission needed, but think about these why you should do it:

- You want to explore how transformers work internally and practiced with them on both text and image data.
- Consider how the dot-product attention mechanism is a unifying theme across modalities – in NLP it helps connect words in a sentence, in vision it connects parts of an image.
- The advanced mechanisms show that as researchers and engineers, we continuously seek ways to make this powerful mechanism more efficient (FlashAttention, sparse patterns) or more scalable (GQA) for large applications.
- Understanding these will prepare you to implement and innovate on transformers in your future projects

Reading List for first 3 Tasks:

Before you start attempting the Tasks, make sure you go through the following reading list with reasonable level of understanding:

1. Understanding Deep Learning book, available at: https://udlbook.github.io/udlbook/. I have taken most of the content from its Chapter 12 on Transformers.
2. The Illustrated Transformer, available at: https://jalammar.github.io/illustrated-transformer/. This is a very good resource on understanding Transformers.
3. Exploring Visual Attention in Transformer Models: https://medium.com/@nivonl/exploring-visual-attention-in-transformer-models-ab538c06083a#:~:text=Generating%20Attention%20Maps
4. HuggingFace Model outputs, available at:: https://huggingface.co/docs/transformers/main_classes/output

Task 1: Dot-Product Self-Attention Fundamentals

This part does not require coding – it tests your mathematical understanding of self-attention. Let us try to look at and interpret the formula for self-attention and answer conceptual questions to demonstrate understanding of how attention operates.

1. Recall that in a transformer, each input token (word or patch embedding) is first projected into three vectors: a query q, a key k, and a value v. For a set of N input tokens with embeddings $x_1, x_2, \ldots, x_N$, let Q, K, V denote the matrices of their query, key, and value vectors (each of size d_q, d_k, d_v respectively for each token). The scaled dot-product self-attention output for the nth token is given by a weighted sum of all value vectors, where weights are obtained from query-key dot products. Write down the formula for the scaled dot-product self-attention. Starting from token x_n's query q_n and all keys $k_1, \ldots, k_N$, the attention weight a_{mn} from token m (as key) to n (as query) is:

$$a_{mn} = \frac{\exp\left(\frac{q_n^T k_j}{\sqrt{d_k}}\right)}{\sum_{j=1}^{N} \exp\left(\frac{q_n^T k_m}{\sqrt{d_k}}\right)}$$

Make sense of each component of this formula: the dot product $q_n \cdot k_m$, the scaling by $\sqrt{d_k}$, and the softmax normalization. Why do we divide by $\sqrt{d_k}$ before applying softmax? Consider what would happen to the gradient if d_k is large and we did not scale the dot products.

2. Show that for each query q_n, the attention weights form a probability distribution over the N positions. In other words, argue (mathematically) that $a_{1n} + a_{2n} + \cdots + a_{Nn} = 1$ and $a_{mn} \geq 0$ for all m. What is the significance of this property in terms of how information is "routed" in the self-attention mechanism?

3. Interpretation: In your own words, understand what the dot-product attention is computing for each output token z_n. What does a high value of the dot product $q_n^T k_m$ signify about tokens x_n and x_m? Why do we use a softmax on these dot products before weighting the values? Provide a brief example (conceptual or numeric) illustrating how attention can make token x_n "pay attention" to another token x_m that is relevant. For instance, you might consider a sentence where $x_n =$ "it" and $x_m =$ "animal" – the attention mechanism can learn to assign a higher weight to the value of "animal" when processing "it", effectively incorporating information about "animal" into "it"'s output.

4. Scaling and Stability: Using the formula and your explanation from part (a), discuss the effect of the scaling factor $\frac{1}{\sqrt{d_k}}$ on training stability. What problem does it mitigate when d_k is large? Relate your answer to the behavior of the softmax function for extreme input values.

Task 2: Transformers in NLP – Implementation and Experiments

In this task, you will apply transformer models to a natural language problem. There are two parts: (2.1) implementing and experimenting with a toy transformer to solidify understanding of attention through code, and (2.2) using a pre-trained transformer (via HuggingFace) to analyze real attention patterns in language data. All code can be run on CPU (no GPU required) by using small models and datasets.

1. **Implementing a Mini-Transformer on a Synthetic Task:** To appreciate how transformers can "learn" a task, you will create a small synthetic dataset and train a minimal transformer model on it. This could be a sequence-to-sequence or sequence classification task that requires the model to use attention to succeed. Possible examples:

   Sequence reversal: Input a random sequence of symbols (e.g. characters or numbers) and train the model to output the sequence in reverse order. This task forces the model to attend to tokens across the entire sequence.

   Target token identification: Input a sequence of symbols with a special marker, and the model must output the symbol that was marked (testing if it can learn to attend to the "target" token anywhere in the input).

   Parity or counting task: The input is a sequence of bits or numbers, and the model outputs something like the parity (even/odd count of a certain symbol) – requiring global aggregation of information via attention.

   Choose one such task (or another of your design) and proceed with the following steps:

   (a) Data Generation: Define a procedure to generate synthetic training and test data for the task. Ensure the sequences are of manageable length (e.g. 5 to 10 tokens) and keep the overall dataset small (e.g. a few thousand examples or fewer) so that training is feasible on CPU.

   (b) Model Implementation: Implement a simple transformer model appropriate for the task. You can use PyTorch or another framework. The model should include at least: an embedding layer for input tokens, one or two self-attention layers, and maybe a small feed-forward network, plus an output layer. (For sequence-to-sequence tasks, you might use an encoder-decoder structure or predict outputs autoregressively. For simplicity, you can also formulate the task as a single-output prediction if that fits, e.g. parity.) If using multi-head attention, use a small number of heads (e.g. 2 or 4) and a small hidden size. You may also implement a single-head attention for simplicity, since the focus is on dot-product attention mechanism. Feel free to reuse or adapt open-source code for basic transformer components (with citation) if needed.

   (c) Training: Train the model on the synthetic dataset until it achieves a reasonable performance on the task (e.g. near-perfect accuracy on the test set of the reversal or identification task). Since the dataset is simple, a transformer with a few thousand parameters should be able to learn it. Monitor the training loss and accuracy. If training is slow, restrict the dataset size or number of epochs to something that completes in a few minutes on CPU.

(d) Attention Visualization: After training, inspect the learned attention patterns. For a given test example, extract the attention weight matrix from the model's self-attention layer(s). You may need to modify your model to return attention weights (similar to how HuggingFace models can return attention when configured). Visualize the attention matrix for one layer or head – for example, create a heatmap where the x-axis is "query position" and y-axis is "key position", and the cell color indicates the attention weight a_{mn}. Does the pattern align with what you expect for the task? (E.g. for sequence reversal, you might expect an attention head that connects positions i and $N - i + 1$ strongly, since the output token at position i should attend to the input token at position $N - i + 1$.)

(e) Analysis Questions (2.1): Discuss the results of your experiment. Did the model learn the task successfully? What do the attention patterns tell you about how it learned to solve the task? For instance, if multiple heads are used, do you observe different heads focusing on different aspects (or perhaps some heads just learn an identity mapping while others do the core task)? Relate your findings to the theoretical expectations from Task 1. If the attention pattern is highly interpretable (e.g. mostly diagonal or off-diagonal), explain why that makes sense. If it's not, speculate what the model might be doing.

If you face difficulty training from scratch, you may instead experiment with a pre-trained small transformer (e.g. a distilled GPT-2 or similar) on a simplified version of the task by fine-tuning, as an alternative. However, implementing one from scratch is highly encouraged for learning.

2. **Analyzing a Pre-trained Language Transformer's Attention:** Transformers pre-trained on language (like BERT or GPT-2) exhibit interesting attention behaviors. In this part, you will use HuggingFace Transformers to analyze attention in a real-world text example. No training is required here; you will use an existing model in inference mode.

   (a) Choose a Model: Use a HuggingFace model such as BertModel or BERT-base-uncased (for encoder attention) or a GPT-2 (for decoder attention). For simplicity, we recommend an encoder like BERT since its self-attention is easier to visualize for a full input sequence. Load the model with output_attentions=True so that it returns attention weights. Also prepare a tokenizer to convert text to model inputs.

   (b) Select an Input Sentence: Craft or choose a sentence (or a short paragraph) that might have interesting attention dynamics. For example, a sentence with pronouns or ambiguous references (similar to the earlier example: "The animal didn't cross the street because it was too tired."), or a sentence with multiple clauses. Tokenize this input and run it through the model to obtain attention matrices.

   (c) Examine Attention Weights: The model will output attention tensors for each layer and head (for BERT, shape is [num_layers, batch_size, num_heads, seq_len, seq_len]). Focus on a subset: for instance, pick one of the middle or last layers and examine all its heads, or pick a specific head that is known to have a certain behavior (research has shown some BERT heads focus on punctuation, some on next-word, some on syntax, etc.). Plot a heatmap of the attention matrix for one or two representative heads. You can label the axes with token words to make it interpretable.

   (d) Interpretation: Identify patterns in the plotted attention. For example, do you see a head that attends strongly to the [CLS] token (perhaps gathering summary info), or a head that attends primarily to the immediate previous token (which might be capturing local context)? Does a pronoun like "it" attend to the noun it refers to (e.g. "animal") as hoped? Document any notable observations. If multiple heads/layers were examined, how do their patterns differ? You might observe that lower layers focus more on local relationships (adjacent words or common phrases), while higher layers might capture more global or semantic connections.

   (e) Analysis Questions: What do the attention distributions reveal about how the model processes the sentence? Give specific examples from your visualization – e.g., "In layer 8 head 3, the word 'it' had most of its attention weight (0.6) on 'animal', indicating the model is likely linking the pronoun to the noun". Discuss whether the observed behavior aligns with your expectations of how a transformer should encode language structure. Also, comment on the variability between heads: are some heads "specialized" (such as attending to punctuation or to the first token) as has been reported in literature? Include any supporting citations or references if you use known findings to compare.

Task 3: Transformers for Vision – Vision Transformer (ViT)

Transformers are not limited to text; Vision Transformers (ViTs) apply self-attention to image patches. In this task, you will explore a pre-trained Vision Transformer model for image classification, and investigate how it uses

attention on visual data. In this task, you should understand how ViT represents images as patches and how to interpret the attention weights to understand model decisions. You will have practiced using transformer models for vision and visualizing their inner workings.

1. Using a Pre-trained ViT for Image Classification: Choose a pre-trained ViT model from HuggingFace, such as google/vit-base-patch16-224, which was pre-trained on ImageNet. Use the corresponding image processor (feature extractor) to prepare input images. Select 1–3 images to test the model on – these could be from a standard dataset (e.g. an ImageNet sample) or any images of your choice (ensure they are appropriately sized, e.g. 224×224 pixels, or resize them with the feature extractor). Run the images through the ViT model to get predicted class labels. Record the top-1 prediction (the model's guess for the object in the image) and whether it seems reasonable.

2. Visualizing Patch Attention: One advantage of transformers in vision is the ability to visualize attention maps over the image patches, which can serve as a form of model interpretability (analogous to saliency maps in CNNs). We will create an attention-based visualization for the ViT's output. Do the following:

   - Configure the model to output attention weights. In HuggingFace's ViTModel or ViTForImageClassification, you can pass output_attentions=True when calling the model (similar to the NLP case).
   - Focus on the class token's attention in the last layer. In ViTs, a special learnable "[CLS]" token is appended to the sequence of patch embeddings, and its output is used for classification. The attention weights from this CLS token to all patch tokens in the final layer indicate which patches were most influential for the classification decision. Extract the attention matrix from the last transformer layer. This will have shape (batch_size, num_heads, seq_len, seq_len) – where seq_len = N_patches + 1 (the +1 is the class token).
   - You may aggregate the heads for simplicity: for example, take the average over all heads' attention matrices in the last layer, or even just use one head if it appears to focus well. Locate the row (or column, depending on implementation) in that matrix corresponding to the CLS token's attention to other tokens. This will give a vector of size equal to the number of patches, representing how much attention the class token gave to each patch.
   - Reshape this attention vector back into the 2D spatial arrangement of patches. For instance, if the image was 224×224 with 16×16 patches, there are $(224/16)^2 = 14 \times 14 = 196$ patches. You can form a 14×14 map of the attention values.
   - Visualize the attention map: You can upsample this patch attention map to the image resolution and overlay it on the image to see which regions are highlighted. (This is similar to creating a heatmap overlay on the image.) Use a library (like matplotlib or PIL) to create a semi-transparent red overlay where intensity corresponds to attention weight.

3. Analyze the Attention Map: Include the produced attention-map-overlaid image. Describe what you observe: Did the ViT focus on the regions of the image that correspond to the predicted class object? For example, if the image was of a dog and the model predicted "dog", do the highlighted patches outline the dog's figure? This provides insight into whether the model is looking at the correct features or if it might be focusing on background or other cues. Compare this attention-based explanation to typical CNN attention (if you are familiar, e.g. convolutional networks often use CAM or Grad-CAM – you don't need to implement those, just conceptually compare). Discuss the advantages of transformers having built-in attention for interpretability. If you notice any peculiar behavior (e.g. the model attended to an odd region), note that as well.

4. Beyond classification: If you are interested and time permits, you could repeat the above for a different ViT task, such as image captioning or detection using a transformer-based model (like DETR for object detection which uses cross-attention). This is optional; the primary goal is to analyze the attention in the classification ViT.

Optional: Task 4: Flash, Sparse, and Grouped-Query Attention

Thus far, we focused on the standard dense dot-product self-attention. In this final task, we explore three advanced variants that address some limitations of vanilla attention, especially in the context of efficiency and scalability. These are FlashAttention, Sparse Attention, and Grouped-Query Attention (GQA). For each subtask, you will investigate the mechanism and, where feasible, perform a small experiment or analysis. The primary aim is to understand what each technique does and why it matters, rather than to implement them from scratch (though simple simulations are encouraged if instructive).

Following is a separate reading list for this task:

- Longformer documentation: https://huggingface.co/docs/transformers/model_doc/longformer
- Brief Review — Longformer: The Long-Document Transformer: https://sh-tsang.medium.com/brief-review-longformer-the-long-document-transformer-8ab204d56613#:~:text=Brief%20Review%20%E2%80%94%20Longformer%3A%20The,a%20task%20motivated%20global%20attention
- What is grouped query attention (GQA)? https://www.ibm.com/think/topics/grouped-query-attention#:~:text=In%20autoregressive%20LLMs%20used%20for,given%20attention%20weights%20approaching%200
- ELI5: FlashAttention https://gordicaleksa.medium.com/eli5-flash-attention-5c44017022ad#:~:text=,%E2%80%94%20its%20outputs%20are%20the
- Flash Attention https://www.hopsworks.ai/dictionary/flash-attention
- Fast and memory-efficient exact attention with io-awareness." Advances in neural information processing systems 35 (2022): 16344-16359. https://proceedings.neurips.cc/paper_files/paper/2022/hash/67d57c32e20fd0a7a302cb81d36e40d5-Abstract-Conference.html

1. Sparse Attention – Reducing Complexity by Limiting Connections: One approach to handle long sequences is sparsifying the attention matrix. Instead of every token attending to every other (N^2 interactions), each token attends only to a subset of tokens (say $k \ll N$ of them). This yields approximate attention (not exact, since some interactions are dropped) but can drastically cut computational cost to O(N·k) which can be linear in N if k is constant or grows slowly.

   (a) Describe two sparse attention patterns that have been used in transformer variants:

   - Local (Sliding Window) Attention: Each token attends only to tokens within a fixed window of size w around it (for example $w = 5$ to left and right). This is used in models like Longformer (along with some global tokens), and in image transformers like Swin Transformer (where attention is restricted to patches in a local window). Discuss how this limits complexity to $O(N \cdot w)$ and why it might work well for data like images or text where local context is most relevant. What is a potential drawback? (Answer: truly long-range dependencies beyond the window cannot be directly captured in one layer, though shifting windows or multiple layers can indirectly connect distant tokens.)
   - Strided or Dilated Attention / Global Tokens: Another pattern is to have a subset of tokens attend globally. For example, BigBird and Longformer introduce a few global tokens (or allow some tokens to attend to all others at intervals) which attend to all positions, ensuring connectivity across the sequence with fewer links. Another approach is random sparse attention or dilated patterns where each token attends to a scattered fixed set of other tokens (some at long distances). These provide a balance between locality and coverage. Briefly mention one such approach (e.g., BigBird combines local + random + global attention to theoretically cover all pairs of tokens across layers) and how it maintains performance on tasks while cutting complexity to linear.

   (b) Complexity Analysis: Suppose a transformer uses a fixed attention span of w (window) for each token. If the sequence length is N, what is the complexity of attention per layer in big-O notation? Compute an example: say $N = 1000$ and $w = 50$ – how many fewer attention score computations is that compared to full attention? (Full would be $N^2 = 10^6$; sparse would be $N \times 50 \approx 50,000$, a $20\times$ reduction.) Discuss how this affects memory and speed, especially for long texts or documents (e.g., processing a document of length 10k, full attention is 100 million operations, which is infeasible, whereas a sparse approach could be linear in 10k).

(c) Illustration: It can be insightful to see the difference between full and sparse attention matrices. Write a small script to generate an $N \times N$ attention mask for full attention (which is just all ones except self maybe) and for a sparse pattern (like a banded matrix with width w). Visualize these matrices for, say, $N = 30, w = 5$ (a smaller case) as images where black/white indicates whether a query attends to a key. You should see a full matrix filled for dense, and a banded structure for local sparse, maybe with a few dots elsewhere for any global connections if you add. Include this visualization if possible. It should resemble Figure 12.15 in the textbook, where (a) dense attention is an all-to-all matrix, and (b) a sparse pattern covers only some entries.

(d) Trade-offs: Discuss when using sparse attention is appropriate despite losing some connectivity. For instance, in tasks like long document classification or certain image analyses, local patterns might suffice and the speed gain is crucial. However, for tasks requiring precise long-range reasoning (like certain algorithmic or multi-hop tasks), sparse attention might hurt accuracy unless carefully designed. Also mention that often models compensate by stacking layers: even if one layer is local, multiple layers with shifting windows (as in Swin Transformer) or random connections can eventually propagate information globally.

(e) Finally, note any specific model example: "Longformer's combination of sliding window + a few global tokens allows it to achieve near full-attention performance on QA tasks with sequence lengths of several thousand, while running in linear time.

2. Grouped-Query Attention (GQA) – Balancing Multi-Head and Multi-Query: Multi-head self-attention (MHA) uses separate projections for queries, keys, and values across H heads, which is powerful but memory-intensive. Multi-query attention (MQA) is an efficiency tweak where all heads share the same keys and values, essentially reducing key/value heads to 1 while keeping H query projections. MQA greatly reduces memory and compute in the decoder (especially for large LMs generating text) but can degrade quality because it restricts the model's expressiveness. Grouped-Query Attention (GQA) is a recent innovation that generalizes these: it partitions the H heads into G groups, each group sharing one key and value projection. This means we have G sets of key/value instead of H (if $G < H$, it's fewer; if $G = 1$, it's MQA; if $G = H$, it's standard MHA).

(a) Explain GQA with an example: Suppose a transformer layer has $H = 8$ attention heads. In standard MHA, there are 8 independent W^Q, W^K, W^V matrices (one per head). In MQA, there would be 8 query matrices but 1 shared W^K and W^V for all heads. Now consider GQA with $G = 4$ groups: we will have 8 query matrices (one per head as usual), but only 4 distinct key matrices and 4 distinct value matrices. That means each group of 2 heads shares the same keys and values. Describe how this setup reduces the number of parameters and memory for keys/values compared to full MHA. (Compute: full MHA has 8 sets of K,V; GQA with 4 groups has 4 sets, which is a 2× reduction; MQA has 1 set, an 8× reduction. GQA is in-between.) Also mention how at inference time this reduces the size of key/value caches (important for LLMs generating token by token).

(b) Overview of grouped-query method: Multi-head (left) vs. Grouped-Query (center) vs. Multi-query (right). In multi-head attention, each of the H heads has its own key and value vectors (shown as different colored bars). In multi-query attention, one key and value are shared across all heads. Grouped-query attention finds a middle ground: keys/values are shared within groups of heads (in this illustration, 3 groups are shown). This allows fewer key/value projections (saving memory) while retaining more diversity than MQA.

(c) Why does GQA maintain accuracy better than MQA? Use the above understanding to answer: MQA can be seen as an extreme case where all heads see identical key/value information, so essentially the heads only differ in their query projection. This can hurt model quality because the ability to focus on different types of information is limited – if one head attends strongly to a particular token, all heads are forced to use that same key/value for that token. GQA, by using a moderate number of groups (e.g. $G = 8$ for a model with $H = 16$ heads, so 2 heads per group), still allows some independence. Each group's heads can specialize together on a subset of information distinct from other groups, preserving modeling power closer to full MHA. Summarize this trade-off in terms of speed vs. quality: GQA aims for faster inference (almost as fast as MQA) with minimal loss in performance (almost as accurate as MHA).

(d) Critical thinking: In what situations would you consider using GQA in a model? Think about very large language models deployed in production for generative tasks – inference speed is critical. Many such models (e.g., some versions of GPT-3, Llama2) actually use MQA (grouping all heads' keys/values) to cut down memory and latency at some cost to perplexity. GQA provides a flexible alternative: one could

train a model with MHA and convert to GQA (some research shows you can merge heads post-hoc) to get a speedup. Discuss any possible challenges: e.g. does grouping heads require retraining or can it be applied to a pre-trained model? (It has been shown that one can "merge" learned heads into groups with minimal fine-tuning, which is promising for retrofitting existing models.) Also consider the generality: could grouped-key or grouped-value ideas be applied to other parts of the model?

3. FlashAttention – Fast Memory-Efficient Attention: FlashAttention is an algorithmic improvement that reorders and fuses the operations of attention to minimize memory usage and optimize speed on hardware. FlashAttention still computes the exact same attention result as standard dense attention (no approximation), but does so more efficiently by utilizing the GPU memory hierarchy.

   (a) Understand in your own words how FlashAttention achieves higher efficiency. You may refer to these key ideas:

      i. Avoiding the explicit $N \times N$ attention matrix: FlashAttention uses tiling of the sequence and computes attention in blocks so that the full attention matrix is never materialized in memory.

      ii. Kernel fusion: Multiple steps (calculating QK^T, applying softmax, weighting V) are fused into one GPU kernel, reducing memory read/write overhead.

      iii. IO-awareness: The algorithm is designed to reduce costly data transfers between GPU high-bandwidth memory (HBM) and on-chip SRAM by keeping data on-chip as much as possible during computation.

      iv. In your explanation, see how the time complexity in practice is improved. (The theoretical compute is still $O(N^2)$ for N tokens, but because memory access is the bottleneck for large N, FlashAttention can achieve near-linear scaling for throughput by being IO-efficient.) Also note that FlashAttention is exact – it returns the same attention output as the naive implementation, just faster and using less memory.

   (b) Mini-Experiment: If you have access to the flash-attn library or PyTorch's efficient attention functions, try a small comparison on a random tensor to verify correctness and speed. For example:

      i. Generate random Q, K, V tensors of shape (batch=1, seq_len=512, dim=64).

      ii. Compute attention output in two ways: (1) the standard way (e.g. using PyTorch nn.Softmax on QK^T), and (2) using an available FlashAttention function (if available, e.g. flash_attn from the library or nn.functional.scaled_dot_product_attention in PyTorch 2.0 which is similar in spirit).

      iii. Verify that the outputs are nearly identical (differences only due to numerical precision). Measure execution time for each on CPU (and GPU if available, though GPU is not required). FlashAttention should be much faster on GPU for large sequences (as reported, e.g. 2–3× speedups for long sequence, and use significantly less memory (because it doesn't allocate the full attention matrix). On CPU, improvements might be less noticeable or not applicable, since FlashAttention is primarily tailored for GPU memory hierarchy.

   (c) Use-cases and Impact: In what scenarios is FlashAttention most beneficial? Consider training of very long sequences or very large models (like GPT-style language models with sequence length 1k–2k or more). FlashAttention enables such models to run faster and fit in memory where standard attention might be a bottleneck. Briefly mention any known results, e.g., "FlashAttention allowed training BERT-large with sequence length 512, 15% faster than the previous best, and GPT-2 with length 1k 3× faster than baseline." This demonstrates that the main gains are in speed and scalability without sacrificing accuracy (since it's exact).