Assignment 1

Jose D. Pague

September 2025

1 Part A Theory

1.1 Attention Mechanism

1.1.1 Derive the Scaled Dot-Product Attention formula step by step

The Scaled Dot-Product Attention formula is the core operation in the Transformer that determines how much focus each token should place on others in the sequence. The inputs include Queries (Q), Keys (K), and Values (V); these are the three main components that are used to determine which key parts of the input sequence should be emphasized. This result is essentially a weighted average of values that determines how strongly the query matches the related key, determining its correlation to the right corresponding key and output. The Query represents the actual information that is being sought, similar to a token, while the key is what information is available to be compared, similar to an index, and the Value is the actual content information that would be passed. Overall the equation

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

is defined by query (Q), key (K), value(), and queries and keys that stand for d_k, with the softmax used to gather the weights of the values. Q revolves in a matrix for the queries, and K and V for matrices for keys and values.

1.1.2 Explain why we divide by $\sqrt{d_k}$

The purpose of dividing by $\sqrt{d_k}$ is to prevent the dot products between queries and keys from becoming substantially larger as the dimension increases. Without this scaling, the softmax would likely produce very sharp probability distributions, leading to ineffective small gradients and unstable training. Dividing by $\sqrt{d_k}$ normalizes the variance of these scores and maintains the range to where the softmax would function effectively. This adjustment is an effective mitigation against unstable training and leads the Transformer to stable learning with a normalized variance through effective attention weights across different sizes.

1.1.3 What does the SoftMax achieve in this context?

The main role of the softmax function is to convert the raw similarity scores from the key values to a normalized probability distribution, which leads to meaningful weights for those values. The softmax takes the raw scores from $QK^T/\sqrt{d_k}$ to obtain the appropriate weights required. This softmax function leads to positive values between the ranges of 0 and 1, with weights summing to 1 for the probability distribution. The higher the score, the larger the weight, which leads to more focus on that token's value, and if the score is lower than the weight is smaller, which has less influence on the final output. This allows for the output to be a weighted average of V and decides how much focus should be on each token.

1.2 Multi-Head Attention

1.2.1 Why do we use multiple heads instead of one?

Multiple attention heads are used in the Transformer as it allows the model to capture a variety of different types of data between tokens at the same time. It allows the information to be averaged into one representation, which improves the detail of the data that would have potentially been missed if it were a single head. Multiple attention heads enable the model to examine different aspects, such as syntax and semantics, in greater detail, which enhances training and improves model performance on complex dependencies. The next benefit is that a multiple attention head approach has a similar computational cost to a single-head attention.

1.2.2 Explain mathematically how queries, keys, and values are projected per head.

For each attention head, the queries, keys, and values are first projected through lower-dimensional subspaces using separate learned weight matrices. Each head has its own learned linear projects:

$$Q_i = QW_i^Q, \quad K_i = KW_i^K, \quad V_i = VW_i^V$$

which demonstrates:

$$W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}, \quad W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}, \quad W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$$

The attention is then computed for each head separately, as shown:

$$\text{head}_i = \text{Attention}(Q_i, K_i, V_i)$$

The next steps are the concatenation of outputs after all the heads are computed and then projected back into another weight matrix:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

This allows for each head to properly learn and maintain an efficient computation.

1.2.3 What advantage does this give in terms of representational power?

The main benefit of using multiple heads is that it increases the representational power of the model through efficient capture of the different types of relationships in parallel. This allows for different representation subspaces from multiple heads instead of compressing all attention into one perspective and risking losing important details. Multiple heads can offer different positions in different aspects of input, giving a grand focus on details that a single head would likely not be able to capture (semantics, syntax, etc.). As each head can handle subspaces of the embedding, it creates a diverse representation, which, when combined, leads to a detailed representation of the sequence. The primary benefit of the model is capable of handling more complex dependencies in language and other sequential data.

1.3 Positional Encoding

1.3.1 Why is positional information necessary in Transformers?

Positional information is necessary in Transformers because, unlike RNNs or CNNs, the architecture does not demonstrate any sequential order. Attention treats tokens as sets, which are not under a structural order, which leads to the importance of word order being considered as in common natural language. This means positional encoding is critical to understand the word structure being implied and its relationships from the tokens, along with its sequence. Positional encoding is necessary to be added to the embedding of the model to utilize the content and position of the token. Fixed or learned, positional encoding is also to be considered. These can be found at the bottom of the encoder and decoder stacks. This positional encoding is beneficial as it can be designed in the same dimension as the embeddings and can be directly summed into the proper input representation. This is essential in order to maintain positional sequence structure for the Transformer.

1.3.2 Write down the sinusoidal encoding formulas and explain what property they provide

The sinusoidal positional encoding formulas are shown throughout the Transformer model as a continuous and predictable way to represent token positions. These are demonstrated through two distinct formulas, such as

$$PE(pos, 2i) = \sin\left(\frac{10000^{2i}}{d_{\text{model}}}\, pos\right), \quad PE(pos, 2i+1) = \cos\left(\frac{10000^{2i}}{d_{\text{model}}}\, pos\right)$$

with pos representing the position of the sequence, i the dimension index, and d_{model} the embedding size. Each dimension is associated with a sinusoid with a different frequency, which leads to a variety of ranges of periodic patterns. This can be shown through the changes in wavelengths that form a geometric progression, which leads to short and long-range dependencies as explained with

2π to $10000 \cdot 2\pi$. The PE_{pos} is correlated as a linear function helpful for relative positions, which can be a representation for fixed offsets k, PE_{pos+k}. Overall, the main benefit of this model is its ability to handle sequence lengths longer than those seen in training.

1.4 Feed-Forward Networks (MLP block)

1.4.1 What role does the two-layer MLP play in each Transformer block?

The role of this two-layer feed-forward neural network (MLP) is to process the output of the attention mechanism at every position with enhanced detail. The formula consists of two layers, shown below:

$$FFN(x) = \max(0,\, xW_1 + b_1)W_2 + b_2$$

with an additional ReLU activation in between in order to add non-linearity, adjust the network through two convolutions expanding the size of the d_{model} from 512 to d_{ff} 2048 before being shrunk again at the next layer. This expansion allows for a kernel size of 1 and a complex pattern opportunity, adjusts negative values through ReLu application to 0, while keeping positive values. This can be interpreted as a 1x1 convolution twice because independent processing occurs for each position without mixing between sequence positions. Overall, this helps with complex mappings learning integration for the model, compared to just simple linear relationships, and is independently applied to each token with the same parameters across positions for parallel computation.

1.4.2 Why do we apply it identically at every position?

The reason we apply this identically at every position as a feed-forward network is to ensure efficiency and consistency across the sequence. The importance of confirming that each token's representation is processed separately, but with the same weights for each position, is to effectively ensure each position is treated equally and without bias towards specific positions. This improves parallelism as all positions can be simultaneously computed. Additionally, this helps maintain an adequate and stable model size through consistent parameters. The only differences between those tokens are their content and positional encoding, but not their MLP weights.

1.5 Normalization and Residuals

1.5.1 What is LayerNorm (or RMSNorm) doing mathematically?

The purpose of the LayerNorm (or RMSNorm) is to stabilize the training by normalizing the activations within each sub-layer of the Transformer. This is done across the feature dimension of a token. For instance, in the encoder, it describes having a stack of N = 6, all identical layers with 2 sub-layers. The first is a multi-head and the second is a position-wise feed-forward network,

which then uses a residual connection towards the two sub layers. The Layer Norm consists of an equation $\text{LayerNorm}(x) = \frac{x-\mu}{\sigma} \cdot \gamma + \beta$ where x is the input vector for a individual token, μ is the mean of features and σ is the standard deviation, and $\gamma + \beta$ is the learning scale and shifting parameters. These are used to maintain tokens to have a mean of 0 while a variance of 1 before scaling or shifting. The ultimate outcome is a stable range activation to prevent vanishing gradients and improve the convergence. This is demonstrated in the paper, furthermore in the $\text{LayerNorm}\big(x + \text{Sublayer}(x)\big)$ which handles the normalization in its residual connection and maintains output dimensions of $d_{model} = 512$. These are critical for the encoder and decoder as the encoder relies on two sub layers, then layer normalization, while decoders follow a similar format, but with an additional third sub layer.

1.5.2 Why are residual connections important for training deep networks?

The importance of residual connections for Transformers is that they allow information to directory go through the network, enabling stable and fast training of deep architectures. They are critical for the encoder and decoder as mentioned in $\text{LayerNorm}(x) = \frac{x-\mu}{\sigma} \cdot \gamma + \beta$ which lead to dimensional outputs of $d_{model} = 512$ while for the decoder it follows a similar path with self-attention modification of the sub-layer to avoid subsequent positions from being attended to. This validates that positions less than i are being depended on for known outputs. This avoids backpropagation, vanishing gradients, helps to allow the network to continue forward, avoid degradation in performance, and faster convergence for improved training and stability. Overall, this is used for multi-head attention and feed-forward layers, and then LayerNorm, which allows multiple layers (encoders and decoders layers) to be possible without loss of information.

1.6 Paper results

1.6.1 What main tasks did the authors evaluate on?

The author evaluates the Transformers on benchmarks consisting of large-scale machine translation compared to sequence-to-sequence models that are typically found. The WMT 2014 English to German (EN-DE) translation task was evaluated on while also the WMT 2014 English-to-French (EN-FR) translation task as well. It was shown that the Transformer was able to outperform other models, such as ByteNet and MoE, with a score of 28.4 for BLEU and a 41.8 BLEU score for EN-FR, with an effective training rate that is 1/4 the time compared to others.

1.6.2 Compare Transformers to RNNs/LSTMs in terms of parallelism and performance.

The Transformers outperformed the RNNs and LSTMs in parallelism and performance through its attention-based architecture, which removes the re-

liance on sequential dependency. While RNNs handle token processing sequentially, they are faulty their less reliant on parallelization. Transformers, on the other hand, are able to process all tokens at the same time in parallel with self-attention. This improves GPU and TPU training time. Additionally, Transformers have a higher BLEU score than RNNs or LSTMs models and are more effective for long-range dependencies, making them a superior method for achieving faster training times and accuracy capability.

2 Part C - Computer Architecture & System Analysis

2.1 System Specification

2.1.1 What CPU, GPU (if any), and memory are available on your machine?

The CPU is an Intel Core i7-12700K, and the GPU is an NVIDIA RTX 3060 Ti with 8 GB of dedicated GDDR6 memory.

2.1.2 List the number of cores, base clock frequency, memory capacity, and memory type (e.g., LPDDR4, GDDR6).

The CPU has 8 Performance cores and 4 Efficiency cores (12 cores, 20 threads total), with a base frequency of 3.6 GHz. The system memory capacity is 32 GB DDR5, running at 4800 MHz. The GPU memory is 8 GB GDDR6.

2.2 Theoretical Peak Performance

2.2.1 Compute the theoretical FLOPs (floating-point operations per second) of your machine for FP32 and FP16.

For GPU: FLOPs = (#cores $\times$ operations per cycle $\times$ clock frequency).
$4864 \times 2 \times 1.665$ GHz $\approx$ **16.2** TFLOPs (FP32).
Memory bandwidth: 14 Gbps $\times(256/8)$ = **448 GB/s**.

For CPU: FLOPs = (#cores $\times$ SIMD width $\times$ clock frequency).
AVX2 is 256-bit is 8 FP32 lanes; with FMA $\Rightarrow$ 16 FLOPs/cycle/core.
$12 \times 16 \times 3.6$ GHz $\approx$ **0.691** TFLOPs (FP32).
No native FP16 SIMD on this CPU, so FP16 peak $\approx$ FP32 peak.

2.2.2 Compute the theoretical memory bandwidth from memory frequency $\times$ bus width $\times$ channels.

System RAM: 4800 MT/s $\times$ 16 B $\approx$ **76.8** GB/s.
GPU VRAM: 14 Gb/s $\times \frac{256}{8} \approx$ **448** GB/s.

2.3 Practical Performance Check

2.3.1 Use available tools (e.g., torch.cuda.get_device_properties, nvidia-smi, or lscpu) to measure actual peak FLOPs and bandwidth. Compare theoretical vs practical numbers — how close is real performance to the peak? Why might there be a gap?

Actual results are FP32 at 17.09 TFLOPS, while FP16 is 35.73 TFLOPS. While the theoretical FLOPS are 16.2, and for FP16, the theoretical FLOPS are ≈ 16.2 The main reason for this 2x theoretical gap is because of the Tensor Cores provide higher FP16 throughput compared to FP32, as the boost clocks went beyond 1.665 GHz.

2.4 Transformer Workload Mapping

2.4.1 Estimate the FLOPs for one forward pass of your Transformer model. (Hint: major cost comes from QK^T matmuls, softmax, and MLP matmuls.)

Assume L=4, h=4, d=256, T=128 (batch size B does not affect per-sequence FLOPs).

Per layer (dominant terms):

$$\text{FLOPs}_{\text{layer}} \approx (4+16)Td^2 + 4T^2d = 20\,Td^2 + 4\,T^2d.$$

Plugging in:

$$20 \cdot 128 \cdot 256^2 = 167{,}772{,}160, \qquad 4 \cdot 128^2 \cdot 256 = 16{,}777{,}216,$$

$$\Rightarrow \text{ per layer} = 184{,}549{,}376 \text{ FLOPs}.$$

Total (all L=4 layers):

$$\boxed{\text{FLOPs}_{\text{forward}} \approx 4 \times 184{,}549{,}376 = 738{,}197{,}504 \approx 0.738 \text{ GFLOPs}}$$

2.4.2 Estimate the memory footprint (weights + activations) for your model.

The model has about 3.22M parameters. This corresponds to roughly 12.9 MB in FP32 (or 6.4 MB in FP16).

During training, activation memory dominates. For batch size $B = 64$, sequence length $T = 128$, and hidden size $d = 256$, the activations require about 370 MB in FP32 (or 185 MB in FP16).

2.4.3 Based on your calculations, is your workload more compute-bound or memory-bound? Justify.

The workload is primarily compute-bound on the GPU, as the model requires about 0.74 GFLOPs per forward pass but only about 13 MB of parameter memory

2.5 Inference vs Training Efficiency

2.5.1 Training requires forward + backward + gradient updates. How much more FLOP-intensive is training compared to inference?

It is nearly double the work of the forward and weight gradient computations.

2.5.2 Given your system, which stage (training or inference) achieves higher hardware utilization? Why?

On the RTX 3060 Ti, higher utilization occurs when larger batches and full sequence matmuls (attention/MLP) are used as the GPUs are being used; therefore, training requires higher hardware utilization.

2.6 Scaling Considerations

2.6.1 If you double the hidden size or number of layers, how do FLOPs and memory scale? Would your system still fit the model in memory? If not, what optimizations would you consider (quantization, gradient checkpointing, etc.)?

Flops would scale by 4x because of the hidden size. Layers would double in size. The RTX 3060 Ti would be fine to handle this change, but as it continues to grow, it will begin to face stress tests in memory and training time. Some ways to optimize performance are a reduction in training time and cutting memory use.

2.7 Conclusion

In conclusion, the transformer proved effective in training as it outperformed traditional RNNs and CNNs while capturing richer informational data types through multi-head computation with the same computational cost as a single head attention. This enables an improved representation power and effective training, overall greater descriptive response, while improving model performance against complex dependencies. This project was closer to a compute-bound than memory-bound, as FLOP utilization dominates in this training performance shown in this system. The hardware in real-time performance was lower than the theoretical performance because of hardware overheads and limits on its own efficiency. However, scalability is possible with optimization in computational and memory demands, such as in reduction in training time and memory use.

To see GitHub Repository:

`https://github.com/josedavp/Assignment_1_EE5453_Jose.git`

The overall AI usage (reflection and raw queries), code, logs, outputs, README, and training screenshot can be found in the GitHub link.