Deep Learning for Artificial Intelligence Engineering

Assignment 3

Generative Pre-Trained Transformer

Student:
Amer Delić (01331672)

Lecturers:
Ozan Özdenizci
Simon Hitzginger

Institute of Machine Learning and Neural Computation
Graz University of Technology

Graz, January 2026

Contents

List of Figures

1 Generative Pre-trained Transformer

The main goal and detailed description of this assignment are provided in the attached assignment sheet. This report focuses on presenting the implementation, experimentation, and results corresponding to those tasks.

1.1 Dataset analysis

The dataset consists of movie titles and overviews, which are concatenated into a single long string of characters. Each character is treated as a token and mapped to a unique integer ID. The entire text is then encoded as a tensor of integers.
A mini-batch fetched from the dataloader yields a pair of tensors (x, y), where x is a sequence of `block_size` characters and y is the same sequence shifted one character forward. This setup allows the model to learn to predict the next character given the previous ones. The `block_size` parameter controls the length of input sequences used during training. In Transformer models, the computational complexity of self-attention scales quadratically with sequence length. Therefore, increasing `block_size` allows the model to capture longer dependencies but also significantly increases memory and compute requirements.

1.2 Self-attention

The (causal) self-attention mechanism - represented by `CausalSelfAttention` class - relies on computing query ($\mathbf{Q}$), key ($\mathbf{K}$), and value ($\mathbf{V}$) matrices, followed by masked, scaled dot-product attention across multiple heads. The following subsections describe the implementation steps using PyTorch primitives.

1.2.1 Q,K,V

The input tensor x has shape (B, T, C), where B is the batch size, T is the sequence length, and C is the embedding dimension. First, a linear layer `qkv_map` is applied to project the input into a single tensor of shape $(B, T, 3C)$, which contains the concatenated query ($\mathbf{Q}$), key ($\mathbf{K}$), and value ($\mathbf{V}$) matrices. Using the `split` method, this combined tensor is then split into three separate tensors - $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$, each with shape (B, T, C).

```
# 1. Linear projection into Q, K, V separately
qkv = self.qkv_map(x)  # (B, T, 3*C)
Q, K, V = qkv.split(C, dim=2)  # each has shape (B, T, C)
```

Each of these tensors is reshaped into multi-head format $(B, T, n_{\text{head}}, d_k)$. By applying the `view` method, the embedding dimension C is divided into n_{head} attention heads, each of size $d_k = C/n_{\text{head}}$. After reshaping, the tensors are transposed to the format $(B, n_{\text{head}}, T, d_k)$ by

using the `transpose()` method. This transformation enables each attention head to process its own slice of the embedding space independently.

```
# 2. Reshape into multi-head format
# split the embedding dimension into n_head × d_k
Q = Q.view(B, T, self.n_head, d_k).transpose(1, 2)  # (B, n_head, T, d_k)
K = K.view(B, T, self.n_head, d_k).transpose(1, 2)  # (B, n_head, T, d_k)
V = V.view(B, T, self.n_head, d_k).transpose(1, 2)  # (B, n_head, T, d_k)
```

As shown above, in the context of self-attention, each token is mapped to a query **q**, key **k**, and value **v** vector through learned linear projections. These vectors are then stacked across the entire sequence and form the matrices **Q**, **K**, and **V**. The query **q** represents the current token's request for information ("what the token is looking for"), the key **k** represents the content of each token that can be attended to ("what each token offers"), and the value **v** contains the actual information that will be aggregated based on attention scores.
Attention weights are computed by taking the dot product between each query and all keys, producing a similarity score for each token pair, resulting in a so-called attention pattern. These scores are scaled and passed through a softmax function to form a probability distribution - the attention weights. Finally, these weights are used to compute a weighted sum over the value vectors, producing the output for each token.
In multi-head attention, this process is repeated in parallel across multiple subspaces of the embedding dimension, allowing the model to capture different types of relationships between tokens.

1.2.2 Attention weights

As described in the previous subsection, once the matrices **Q**, **K**, and **V** are obtained, the attention scores are computed by taking the dot product between **Q** and the transposed **K**, which indicates how strongly each token should attend to every other token. In PyTorch, the matrix multiplication is performed using the `@` operator, which applies batched matrix multiplication across all heads and all elements in the batch.
The keys are transposed using `transpose(-2, -1)` in order to swap the last two dimensions of **K**, changing its shape from $(B, n_{\text{head}}, T, d_k)$ to $(B, n_{\text{head}}, d_k, T)$. This ensures that the inner dimensions match for matrix multiplication, allowing each query vector of size d_k to be multiplied with all key vectors across the sequence length T.
These raw attention scores are then scaled by $\sqrt{d_k}$ to maintain numerical stability and prevent excessively large gradients during training.

```
# Compute attention scores
attn_scores = (Q @ K.transpose(-2, -1)) / math.sqrt(d_k)  # (B, n_head, T, T)
```

After computing the raw attention scores, a causal mask is applied to ensure that each token can only attend to itself and to previous tokens in the sequence. This is essential for autoregressive models, where information from future positions must not influence the current prediction. Without the causal mask, each token would be able to attend to future positions in the sequence, causing information leakage and breaking the autoregressive property required for next-token prediction.
A lower-triangular matrix of ones is created using `torch.tril`, producing a mask of shape (T, T) in which all entries above the main diagonal are zero. These zero entries correspond to positions that should not be attended to. The attention score matrix is then modified using `masked_fill`, which replaces all invalid (future) positions with $-\infty$. When the softmax function is applied, these values result in zeros, effectively preventing any attention to future tokens.

```
# Apply causal mask
mask = torch.tril(torch.ones(T, T, device=x.device))
attn_scores = attn_scores.masked_fill(mask == 0, float('-inf'))
```

After masking, the attention scores are converted into a probability distribution using the softmax function `F.softmax` from `torch.nn.functional` module. The function is applied along the last dimension so that the weights for each query token sum to one. This produces the final attention weights, which determine how strongly each token contributes to the output. Dropout is then applied to these weights using `F.dropout`, also from `torch.nn.functional` module, as a form of regularization, preventing overfitting by randomly dropping some attention connections. The argument `training=self.training` ensures that dropout is active only when the model is in training mode, and automatically disabled during evaluation.

```
# Softmax + dropout
attn_weights = F.softmax(attn_scores, dim=-1)  # (B, n_head, T, T)
attn_weights = F.dropout(attn_weights, p=self.dropout, training=self.training)
```

Finally, the attention weights are used to aggregate information from the value vectors. This is performed through a batched matrix multiplication between the attention weight matrix and the value matrix, again using the `@` operator. For each token, the attention weights determine how strongly each value vector contributes to the final representation. The result is a weighted sum of values, producing an output tensor of shape $(B, n_{\text{head}}, T, d_k)$ that contains the aggregated contextual information for each token.

```
# Weighted sum of values
aggregated_vals = attn_weights @ V  # (B, n_head, T, d_k)
```

1.3 MLP

After the attention block, an MLP is implemented that expands the token representation, applies a nonlinear activation, and then projects it back to the original embedding dimension. The MLP consists of a first linear layer, a GELU activation, a second linear layer, and a final dropout layer. The first linear layer expands the representation to $4 \times n_{\text{embd}}$, a widening factor commonly used in Transformer architectures because it increases the model's capacity to learn richer nonlinear transformations while keeping the computational cost manageable. The GELU activation provides smoother and more expressive nonlinear behavior than ReLU, which has been shown to improve performance in large language models. The final dropout layer randomly zeroes a fraction of activations during training, helping prevent overfitting and improving generalization.

1.4 Layer normalization

In the Block module, Layer Normalization is applied before both the attention and MLP components. Specifically, the input is first normalized using `LayerNorm`, then passed through the corresponding submodule, and finally added back to the original input via a residual connection. It normalizes each token independently by computing the mean and variance across its feature dimension, which helps stabilize training. Unlike Batch Normalization, which uses statistics across the whole batch and can be sensitive to batch size, Layer Normalization works the same regardless of batch shape and is therefore more reliable for sequence models and autoregressive transformers.

1.5 Forward pass

The forward pass of the GPT model begins by converting the input token indices into continuous vector representations using a token embedding layer. In parallel, a position embedding layer provides a learned positional encoding for each position in the sequence. These two embeddings are added element-wise and passed through a dropout layer.
The resulting sequence representation is then processed by a stack of Transformer blocks. Each block applies Layer Normalization, Causal Self-Attention, and an MLP module, combined with residual connections that stabilize training and preserve information flow. After all blocks have been applied, a final Layer Normalization is used to normalize the output.
Finally, the model projects the hidden states at every sequence position to the vocabulary dimension using a linear output layer. This produces logits of shape $(B, T, \text{vocab\_size})$, where each vector represents the unnormalized scores for predicting the next token at that position. All layers involved in this process are predefined in the model's constructor and reused directly during the forward pass.

1.6 Training and Convergence

The model was trained using the AdamW optimizer with weight decay for regularization. During training, the objective is to minimize the cross-entropy loss between the predicted logits and the ground-truth next tokens. Training was performed for a fixed number of iterations, and both training and validation losses were periodically evaluated.
The training loss was computed on mini-batches sampled from the training set, while the validation loss was estimated using a held-out validation set with dropout disabled to obtain a more stable estimate of the model's generalization performance.

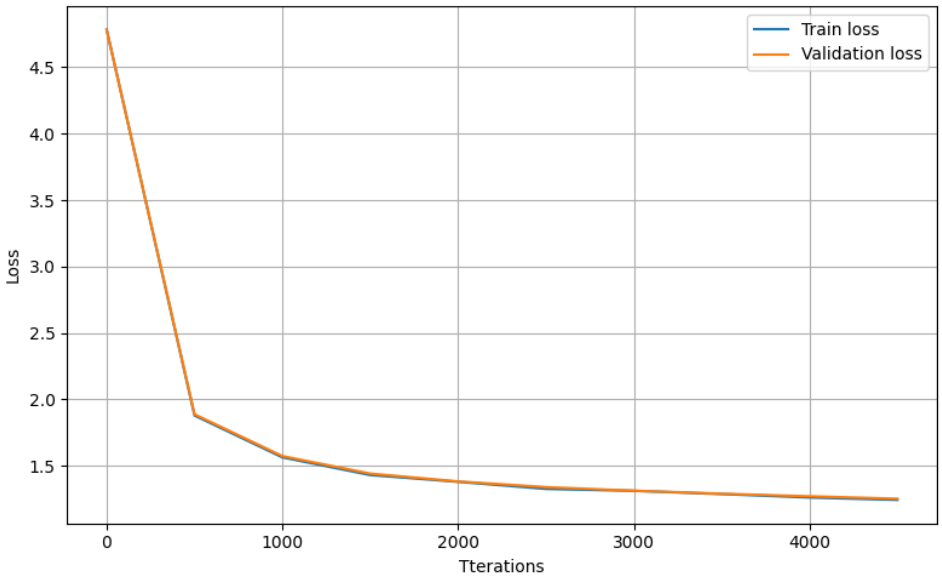

Figure 1.1: Training and validation loss as a function of training iterations.

As shown in the figure, the training loss decreases steadily over the course of training, indicating that the model is able to effectively fit the training data. The validation loss follows a similar trend and stabilizes after an initial decrease, suggesting that the model generalizes reasonably well and does not exhibit severe overfitting.

The gap between the training and validation loss remains limited, which indicates that the chosen model capacity and regularization settings are appropriate for the dataset. Minor fluctuations in the validation loss are expected due to the stochastic nature of mini-batch training and the relatively small validation sample used during evaluation.

1.7 Sampling

The `sample` method generates new tokens autoregressively. It starts from an initial sequence of token indices `idx` and iteratively appends `max_new_tokens` sampled tokens. In each iteration, the current sequence is optionally cropped to the last `block_size` tokens to respect the model's context window.
This cropped sequence, denoted as `idx_input`, is passed through the model, which returns logits of shape $(B, T, \text{vocab\_size})$.

```
# Forward pass
logits = self(idx_input)  # (B, T, vocab_size)
```

Since the model predicts a distribution for every position in the sequence, only the logits at the final position are relevant for the next token - these are extracted as `logits[:, -1, :]` and divided by the temperature parameter τ, which controls the sharpness of the resulting distribution.

```
# Extract logits for the last position and scale
logits = logits[:, -1, :] / temperature  # (B, vocab_size)
```

For $\tau < 1$, the logits are effectively amplified, which makes the softmax distribution sharper. Bigger differences between logits lead to bigger differences in probabilities, so the model becomes more confident and tends to choose high-probability tokens more often. In the limit $\tau \to 0$, the softmax distribution approaches an argmax, resulting in an almost deterministic selection of the token with the highest logit. In contrast, for $\tau > 1$, the logits are compressed, which makes the softmax distribution flatter. Because the differences between logits become smaller, the probabilities also become more even, so the model behaves more exploratively and lower-probability tokens receive a higher chance of being sampled. For very large τ the distribution approaches something close to uniform distribution over the vocabulary.

After temperature scaling, a softmax is applied to obtain a probability distribution over the vocabulary. Here, once again the softmax function `F.softmax` from `torch.nn.functional` module is used and applied along the last dimension.

```
# Convert scaled logits into a probability distribution
probs = F.softmax(logits, dim=-1) # (B, vocab_size)
```

The next token index is then drawn from this distribution using multinomial sampling, which is appropriate because it allows sampling proportionally to the model's predicted probabilities rather than always selecting the most likely token.

```
# Sample next token
next_token = torch.multinomial(probs, num_samples=1)
```

This is done using the `torch.multinomial` function, which interprets each row of the probability tensor as a categorical distribution and samples an index proportionally to its

probability. This means that tokens with higher probability are more likely to be selected, but lower-probability tokens may still be chosen depending on the temperature. This introduces controlled stochasticity into the generation process and enables the model to produce more diverse and less deterministic outputs.

1.8 Effect of Temperature on Generated Text

To evaluate the qualitative behavior of the trained model, text samples were generated using different temperature values $\tau \in \{1.5, 1.0, 0.8, 0.5, 0.1, 0.0001\}$. For each temperature, multiple samples were produced using the same trained model while varying only the sampling temperature.
For higher temperatures (e.g., $\tau = 1.5$), the generated text exhibits increased randomness and diversity. While this can lead to more creative outputs, it also introduces grammatical inconsistencies, incoherent phrases, and abrupt topic shifts. This behavior is expected, as higher temperatures flatten the softmax distribution and assign higher probability mass to lower-likelihood tokens.
At moderate temperatures (e.g., $\tau = 1.0$ and $\tau = 0.8$), the model produces more coherent and structured text. Sentences tend to follow plausible syntactic patterns, and local consistency improves while still retaining some variability in word choice and phrasing.
For low temperatures (e.g., $\tau = 0.5$ and $\tau = 0.1$), the model becomes increasingly deterministic. The generated text is more repetitive and conservative, often reusing common phrases and patterns observed during training. While grammatical correctness improves, diversity is significantly reduced.
In the extreme case $\tau = 0.0001$, the sampling process approaches a greedy decoding strategy. The model almost always selects the token with the highest probability, resulting in highly repetitive and predictable outputs with little variation across samples.
Additionally, generation was performed using different initial prompts of varying lengths. Longer prompts generally lead to more contextually consistent continuations, while very short prompts provide limited conditioning and result in more generic or less coherent outputs. This highlights the importance of sufficient context for autoregressive language modeling.

1.9 Model Scaling and Computational Considerations

In addition to the baseline model, larger transformer configurations were considered in order to study the effect of increased model capacity. These configurations differed in the number of transformer blocks, the embedding dimensionality, and the number of attention heads, which directly influence the total number of trainable parameters and the computational complexity of the model.
Due to computational constraints, a full comparative training of multiple large-scale models was not performed. Instead, the impact of model scaling is discussed qualitatively based on architectural considerations and known properties of transformer-based language models.
Increasing the number of blocks and the embedding dimensionality increases the expressive power of the model and enables it to capture more complex patterns and longer-range dependencies in the data. However, this also leads to a significant increase in memory usage and training time. In particular, the computational cost of the self-attention mechanism scales quadratically with the sequence length and linearly with the number of parameters.
Larger models generally require smaller learning rates and more careful regularization to ensure stable training. Without sufficient training data or computational resources, increasing model size may result in diminishing returns or even degraded generalization performance.
Overall, model scaling presents a trade-off between performance and computational cost. While larger architectures are theoretically capable of achieving lower training and validation

loss, practical limitations such as available hardware and training time must be taken into account when selecting an appropriate model configuration.

Appendix

Deep Learning for AIE (708.002) WS25
Assignment 3

Generative Pre-trained Transformer

Ozan Özdenizci, Simon Hitzginger
Institute of Machine Learning and Neural Computation, TU Graz
oezdenizci@tugraz.at, simon.hitzginger@tugraz.at

Points to achieve:	25 pts
Assignment Issued:	18.12.2025 16:00
Deadline:	29.01.2026 16:00 (**no extensions**)
Hand-in procedure:	You **can** work in groups of **at most two people**. **Exactly one** team member uploads two files to TeachCenter: **The report (.pdf)** and **the code skeleton solution as .ipynb**. **Do not upload a folder. Do not zip the files.** The first page of the report **must** indicate the group partner.
Assignment interviews:	You might be invited for an assignment interview.
Plagiarism:	If detected, 0 points for all parties involved.

General Remarks

- All results (i.e., plots, results of computations) should be included in your PDF report – the Jupyter Notebook file should only serve as a way to reproduce your results.
- In the `CausalSelfAttention`, `MLP`, and `GPT` classes, you are *only* allowed to add code to the respective `TODO` sections. Do not change the method signatures.
- Commented out code will not be executed or graded.
- Make sure all of your plots have labeled axes and are clearly described in your report.
- For all tasks, implement the required code and answer the associated questions in the report. Where specified, **include the corresponding code snippets in the report and provide an explanation**. A template illustrating the expected format will be made available.

Autoregressive Language Modeling

Autoregressive language models aim to model the probability distribution over sequences of discrete *tokens*. Although tokens are usually sub-word units, in this assignment we will consider each *character* to be a token. Given a sequence $\mathbf{x} = (x_1, \ldots, x_t)^\top$, we decompose the joint probability $p_\theta(\mathbf{x})$ using the chain rule of probability:

$$p_\theta(\mathbf{x}) = p_\theta(x_1) \prod_{k=2}^{t} p_\theta(x_k | \mathbf{x}_{<k}), \tag{1}$$

where $\mathbf{x}_{<k} = (x_1, \ldots, x_{k-1})^\top$ represents all tokens before position k. This decomposition allows us to model the probability of each token conditioned on all previous tokens in the sequence. Generative Pre-trained Transformers (GPTs) essentially use Transformer-based neural networks to model

$p_\theta(x_t|\mathbf{x}_{<t})$. Transformers utilize the self-attention mechanisms to capture long-range dependencies between tokens, making them particularly effective for generating coherent text by sampling one token at a time from these conditional distributions.

Task details:

a) (1 pts) : Get familiar with the dataset and briefly analyze its structure. What is returned when we fetch a mini-batch from this dataset using a dataloader? Describe what the `block_size` parameter controls and how it generally relates to the computational complexity of Transformers.

b) (8 pts) : Using only `PyTorch` primitives[1], implement the `CausalSelfAttention` class for multiple attention heads. This class will receive a batch of sequences of embeddings and performs causally masked, multi-head scaled dot-product self-attention.

Follow the `TODOs` in the code. In your report, include a code listing of your implementation of `CausalSelfAttention` and explain how it works in your own words. Explain what Q, K, and V represent in this context and how they interact to produce attention weights. Write down the dimensionality of Q, K, and V in the multi-head attention setting. What would happen if we would *not* apply the causal mask?

c) (2 pts) : Implement the `MLP` class that follows each attention block. Follow the `TODOs` in the code. For the `GELU` activation function you can use `nn.GELU`[2] from `PyTorch`.

d) (1 pts) : A `Block` module which consists of `CausalSelfAttention` and `MLP` modules (as well as `LayerNorm` and residual connections) is already provided for you. Explain in your report how the Layer Normalization works, where is the normalization performed, and how it differs from Batch Normalization in terms of its advantages/disadvantages?

e) (2 pts) : Carefully read the code of the `GPT` class, complete the forward pass and explain.

f) (2 pts) : Train the model using appropriate hyperparameters. The default hyperparameters should serve as a good starting point, but feel free to change them if you think this is necessary. Create a plot showing the training and validation loss over iterations. Analyze the training dynamics and comment on the model's convergence behavior.

g) (4 pts) The `GPT` model outputs logits $\mathbf{z}_k$ for all conditional distributions $p_\theta(x_k \mid \mathbf{x}_{<k})$. Given logits $\mathbf{z}_k$, we obtain:

$$p_\theta(x_k \mid \mathbf{x}_{<k}) = softmax(\mathbf{z}_k)_{x_k}.$$

Given a temperature value $\tau > 0$, we define:

$$p_\theta^\tau(x_k \mid \mathbf{x}_{<k}) = softmax(\mathbf{z}_k/\tau)_{x_k}.$$

Implement the `sample` method in the `GPT` class, which takes a temperature τ and a starting sequence $(x_1, \ldots, x_t)$, and samples $x_{t+i} \sim p_\theta^\tau(x_{t+i} \mid \mathbf{x}_{<t+i})$ for $i = 1, \ldots,$ `max_new_tokens`. In your report, include a code listing of your implementation, explain how it works, and describe the effect of the temperature τ on the softmax.

h) (2 pts) Generate samples from your trained model using temperature values $\tau \in \{1.5, 1.0, 0.8, 0.5, 0.1, 0.0001\}$. Include representative samples for each temperature in your report and comment

[1] i.e., do *not* use `PyTorch` functions that directly compute the attention mechanism

[2] https://pytorch.org/docs/stable/generated/torch.nn.GELU.html

on the generated output: How well does the model perform, and what kinds of patterns or errors do you observe? Discuss in particular how the temperature τ influences the nature of the generated text.

Additionally, try generating text from different starting prompts. Provide a few examples where you condition the model on initial substrings of different length and discuss how well it autocompletes the text.

i) (3 pts) Experiment with increasing the model size (number of blocks, embedding dimension, number of heads, etc.) and block size. Try at least 5 different architectures. Find appropriate training hyperparameters (learning rate, batch size, etc.) for each. Report the model, the number of parameters and your choice of hyperparameters in your report. Create a plot showing the training and validation loss over iterations for the models. Generate samples with an appropriate value of τ and compare the performance with the baseline model. Briefly comment on the computational requirements (i.e., how long a training run took on which hardware).