Assignment 3

Deep Learning KU, WS 2024/25

Team Members		
Last name	First name	Matriculation Number
Hinum-Wagner	Jakob	01430617
Nozic	Naida	12336462

Tasks

0.0.1 Task a)

Get familiar with the dataset and briefly analyze its structure. What is returned when we fetch a batch of data from this dataset using a dataloader? Describe what the block_size parameter controls.

0.1 Solution a)

1. Dataset Structure and Character Frequency

The dataset structure and character frequency provide key insights into the content and diversity of the data. The following code was used to analyze the dataset:

```
import collections

# Count the frequency of each character
char_counts = collections.Counter(dataset.data.tolist())
most_common_chars = char_counts.most_common(10)
least_common_chars = char_counts.most_common()[-10:]

print("\nMost common characters:")
for char, count in most_common_chars:
    print(f"'{dataset.int_to_string[char]}': {count}")

print("\nLeast common characters:")
for char, count in least_common_chars:
    print(f"'{dataset.int_to_string[char]}': {count}")

# Plot character frequency distribution
plt.figure(figsize=(10, 5))
plt.bar([dataset.int_to_string[char] for char, _ in most_common_chars],
        [count for _, count in most_common_chars])
plt.title("Top 10 Most Common Characters")
plt.xlabel("Character")
plt.ylabel("Frequency")
plt.show()
```

- **Vocabulary Analysis:**
  - The dataset includes 169 unique characters. This is a relatively small vocabulary size, suggesting that the dataset primarily consists of standard ASCII characters with limited inclusion of extended symbols or special characters.
  - Examples of unique characters: `['\t', '\n', ' ', '!', '"', '#', '$', '%', '&', "'"]`. These characters include whitespace, punctuation, and common symbols, reflecting a typical text corpus.
- **Most Common Characters:**

- The top 10 most frequent characters include spaces and vowels, aligning with the statistical properties of English text. Spaces dominate, making up a significant fraction of the dataset, which is expected as they separate words.

- **Least Common Characters:**
  - Rare characters appear only once in the dataset . These could be typos, stylistic elements, or characters from infrequent language-specific text.

- **Frequency Distribution:**
  - The bar chart visualizes the dominance of certain characters, especially spaces and vowels. This imbalance could have implications for the model's training, potentially biasing it toward over-represented characters.

2. Batch Output Analysis

The `DataLoader` returns batches containing input (`x`) and target (`y`) tensors. The following code was used to fetch a batch and analyze its structure:

```
from torch.utils.data import DataLoader

# Instantiate the dataloader
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

# Fetch one batch
for batch in dataloader:
    x, y = batch
    print(f"Shape of input (x): {x.shape}")
    print(f"Shape of target (y): {y.shape}")

    # Decode and display a subset of sequences
    for i in range(3):
        print(f"\nDecoded Input (x[{i}]):", dataset.decode(x[i]))
        print(f"Decoded Target (y[{i}]):", dataset.decode(y[i]))
    break
```

- **Shape and Structure of Tensors:**
  - **Input Tensor (`x`):**
    - Shape: `(batch_size, block_size)`, where `batch_size=10` and `block_size=1000`.
    - Each row represents a sequence of `block_size` characters from the dataset.
  - **Target Tensor (`y`):**
    - Shape: Same as input: `(batch_size, block_size)`.
    - Each row in `y` is the input sequence shifted by one character, effectively serving as the ground truth for predicting the next character in the sequence.

- **Examples from the Batch:**
  - **Input and Target Alignment:**

    - The decoded input and target sequences are nearly identical, except for the target being shifted by one character. This setup is typical for sequence modeling tasks like language modeling, where the model learns to predict the next character.
  - **Real-World Contexts:**
    - Examples such as `"Naruto: The Lost Story..."` and `"Dolittle..."` reveal that the dataset includes diverse text snippets, potentially sourced from movie descriptions or synopses. This diversity introduces varied vocabulary and sentence structures, providing a rich training corpus.
  - **Text Continuity:**
    - Input and target sequences maintain text continuity across the `block_size`. This ensures that dependencies within and across sequences are preserved, enabling the model to learn meaningful language patterns.

3. Analysis of the Block Size Parameter

The `block_size` parameter, set to 1000 in this example, defines the length of the sequences returned by the `DataLoader`. The following code highlights its role:

```
block_size = 1000
print("\nBlock Size Impact Analysis:")
print(f"Block size controls the sequence length for input and target.")
print(f"Current block size: {block_size}")
print(f"Each input and target pair is {block_size} characters long.")
```

- **Impact on Input and Target Data:**
  - Larger block sizes allow the model to process longer contexts, which is critical for tasks requiring an understanding of global text structure (e.g., document-level summaries or long-form content generation).
  - However, longer block sizes increase memory requirements and computational complexity, potentially slowing down training.
- **Information Content:**
  - Each sequence of 1000 characters represents a relatively large portion of text, sufficient to capture sentence-level and paragraph-level dependencies.
  - For example, a block of 1000 characters may contain multiple sentences, providing the model with richer contextual information compared to smaller block sizes.
- **Generalization and Overfitting:**
  - A very large block size could lead the model to memorize specific sequences instead of generalizing patterns, especially if the dataset is not sufficiently diverse.
  - A smaller block size may fail to capture long-term dependencies, limiting the model's capacity for complex language tasks.

4. Insights from Shapes and Values in Batch Data

- **Shape Analysis:**
  - The shapes of input and target tensors (`torch.Size([10, 1000])`) align with expectations for a batch size of 10 and block size of 1000.
  - This fixed shape ensures consistency, which is critical for feeding data into deep learning models.
- **Data Representation:**
  - Input (`x`) and target (`y`) tensors contain encoded integers representing characters in the dataset. The integers are mapped back to characters using a decoder function for interpretability.
- **Diversity in Batches:**
  - Even within a single batch, the text sequences vary widely, demonstrating that the data shuffling mechanism works effectively. This helps avoid bias during training and ensures robust model performance.

5. Practical Implications for Modeling

- **Sequence Modeling:**
  - The alignment between input and target makes this dataset ideal for next-character prediction tasks, such as training a recurrent neural network (RNN), Transformer, or any sequence-to-sequence model.
- **Diversity and Vocabulary:**
  - The relatively small vocabulary size and the presence of rare characters could influence how embeddings are initialized and trained, with frequent characters having a larger impact on gradients.
- **Optimization Challenges:**
  - The high frequency of spaces and certain vowels could lead to skewed training, where the model prioritizes predicting common characters over rarer ones. Techniques like character weighting or loss adjustment may mitigate this.

Summary

What is returned when we fetch a batch of data from this dataset using a dataloader?
When fetching a batch of data from the dataset using a dataloader, the following components are returned:

- **Input Tensor (x):** A tensor of shape `(batch_size, block_size)` representing the input sequences. Each row contains a sequence of characters encoded as integers, with a length equal to `block_size`.
- **Target Tensor (y):** A tensor of the same shape `(batch_size, block_size)`, representing the target sequences. Each row is a one-character-forward-shifted version of the corresponding input sequence, making it suitable for next-character prediction tasks.

For example, if the `block_size` is set to 1000 and `batch_size` is set to 10:

- Input (`x`) contains 10 sequences, each 1000 characters long.
- Target (`y`) contains corresponding shifted sequences for next-character prediction.

Describe what the `block_size` parameter controls.
The `block_size` parameter controls the sequence length of both the input and target tensors in each batch. It specifies the number of consecutive characters included in each sequence. A detailed description of its role is as follows:

- **Context Length:** It determines how much contextual information the model receives for each input sequence. A larger `block_size` allows the model to learn longer-range dependencies in the text.
- **Input and Target Pairing:** The `block_size` directly defines the length of the sequences for both input (`x`) and target (`y`), ensuring they match.
- **Training Complexity:** Larger values of `block_size` increase the computational and memory requirements, but they enable the model to learn more complex patterns in the data.

In this analysis, the `block_size` was set to 1000, meaning each input and target sequence consisted of 1000 characters. This ensures sufficient context for learning dependencies within and across sentences while balancing computational demands.

Task b)

Using only PyTorch primitives[2], 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. Explain in your report 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?

Solution b)

1. Implementation of `CausalSelfAttention`

The `CausalSelfAttention` class implements multi-head scaled dot-product self-attention with a causal mask. This mechanism ensures that each token in a sequence attends only to itself and preceding tokens, preserving the autoregressive property required for tasks like language modeling.

Initialization: Configuration Setup:
The class receives a configuration object (`GPTConfig`) that specifies the following parameters:

- `vocab_size`: Total number of unique tokens in the vocabulary.
- `block_size`: Length of input sequences.
- `n_block`: Number of Transformer blocks.

- `n_head`: Number of attention heads.
- `n_embd`: Dimensionality of the embeddings.
- `dropout`: Dropout rate used for regularization.
- `bias`: Boolean flag indicating whether bias terms are included in the linear layers.

Component Initialization:

- A linear layer projects input embeddings into query (Q), key (K), and value (V) matrices for all attention heads simultaneously. The output dimension is $3 \times$ `embedding dimension`.
- Another linear layer is used to project the concatenated output from all heads back into the original embedding space.
- A dropout layer is included after the final projection for regularization, helping to reduce overfitting during training.

Forward Pass: The forward pass of the `CausalSelfAttention` class is divided into several key steps:

Input Transformation:

- The input tensor x, with shape (B, T, C) (batch size, sequence length, embedding dimension), is passed through the linear layer (`qkv_map`) to produce a combined tensor for Q, K, and V. This tensor has shape $(B, T, 3 \times C)$.

Splitting and Reshaping:

- The combined tensor is split into Q, K, and V, each with shape (B, T, C).
- These tensors are reshaped and transposed to separate the multiple attention heads, resulting in tensors with shape $(B, \texttt{n\_head}, T, \texttt{head dimension})$, where

$$\texttt{head dimension} = \frac{\texttt{embedding dimension}}{\texttt{number of heads}}.$$

Scaled Dot-Product Attention:

- The attention scores are computed using the formula

$$\texttt{attn\_scores} = \frac{Q \cdot K^T}{\sqrt{d_k}},$$

  where d_k is the head dimension. Scaling by $\sqrt{d_k}$ helps stabilize gradients by preventing the softmax function from saturating.

Causal Masking:

- A causal mask (lower triangular matrix) is applied to the attention scores, ensuring each token can only attend to itself and preceding tokens. Future tokens are masked with $-\infty$.

Softmax Normalization:

- The masked attention scores are passed through a softmax function, yielding attention weights normalized across the sequence length.

Weighted Summation:

- The attention weights are used to compute a weighted sum of V, producing the attention output for each head.

Final Projection:

- The outputs from all heads are concatenated and projected back to the original embedding space using the second linear layer.
- Dropout is applied to the final output for regularization.

2. Explanation of Q, K, and V

In the self-attention mechanism:

Query (Q):

- Represents the "question" each token in the sequence asks about its relationship to other tokens.
- Derived from the token's embedding via a linear transformation.

Key (K):

- Represents the "answer" or reference provided by each token.
- Computed in a similar manner to Q but serves as a comparison basis.

Value (V):

- Represents the actual content carried by each token.
- Used to compute the weighted sum based on attention weights.

Interaction of Q, K, and V:

- Q and K are compared using the dot product to calculate similarity scores.
- These scores are scaled, masked, and normalized using softmax to produce attention weights.
- The attention weights are then applied to V to compute a weighted sum, which becomes the final output of the self-attention mechanism.

3. Dimensionality of Q, K, and V

In a multi-head attention mechanism, the dimensions of Q, K, and V are typically:

$$Q, K, V : (B, \texttt{n\_head}, T, \texttt{head dimension}),$$

where

$$\texttt{head dimension} = \frac{\texttt{embedding dimension}}{\texttt{number of heads}}.$$

4. Importance of the Causal Mask

The causal mask ensures that tokens in the sequence attend only to themselves and preceding tokens, preserving the autoregressive property. Without the causal mask:

- Tokens could attend to future tokens, violating the causality constraint.
- Information leakage would occur during training, as tokens would have access to data they should not yet "see."
- The model would become unsuitable for autoregressive tasks such as language modeling and text generation.

5. Addressing TODOs

Generating Q, K, and V:

- The output of the `qkv_map` layer was split into Q, K, and V, each of shape (B, T, C).
- These were reshaped and transposed to handle multiple attention heads, producing tensors with shape $(B, \texttt{n\_head}, T, \texttt{head dimension})$.

Computing Attention Scores:

- The dot product of Q and K^T was calculated and scaled by $\sqrt{d_k}$.

Applying the Causal Mask:

- A lower triangular mask was applied to attention scores to restrict attention to valid positions.

Weighted Summation of V:

- The softmax-normalized attention weights were multiplied with V, producing the weighted output for each head.

Final Projection:

- The concatenated outputs from all heads were projected back to the original embedding space using a linear layer.

6. Testing the Implementation

To validate the implementation of the `CausalSelfAttention` class, the following steps were performed:

- A synthetic batch of input embeddings was generated with the following properties:
  - Batch size $(B) = 2$
  - Sequence length $(T) = 8$
  - Embedding dimension $(C) = 12$
- The output of the `CausalSelfAttention` class was compared to a precomputed reference output using `torch.allclose`.
- The implementation was verified to match the expected behavior, ensuring that all components—masking, attention computation, and projection—were functioning correctly.

Details of the Test Code:

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
config = GPTConfig(
    vocab_size=10,
    block_size=8,
    n_block=6,
    n_head=6,
    n_embd=12,
    dropout=0.0,
    bias=True
)

torch.manual_seed(1337)
if torch.cuda.is_available():
    torch.cuda.manual_seed(1337)

x = torch.randn(2, 8, 12).to(device)
attention = CausalSelfAttention(config).to(device)
att_out = attention(x)

# Read expected output from a precomputed file
att_out_expected = torch.load('CausalSelfAttention_out.pt', map_location=device)

# Verify outputs
assert torch.allclose(att_out, att_out_expected, atol=1e-7), "Outputs do not match!"
print("Test passed successfully!")
```

The test confirmed that:

- The attention weights were correctly computed, respecting the causal masking constraint.
- The projection layer correctly combined the outputs from all attention heads.
- The final output matched the expected precomputed values, validating the correctness of the implementation.

7. Summary of Findings and Insights

CausalSelfAttention Mechanism:

- The `CausalSelfAttention` class uses scaled dot-product self-attention with multiple heads, enabling it to focus on different parts of the input simultaneously.
- A causal mask ensures that tokens attend only to preceding tokens, preserving the autoregressive property.

Key Components:

- **Q, K, and V:**
  - Derived from the input embeddings through a shared linear transformation.

  - Interaction between Q and K computes similarity scores, while V provides the content for weighted summation.

- **Causal Mask:**
  - Critical for ensuring that tokens do not attend to future positions in the sequence.
  - Prevents information leakage and maintains the validity of autoregressive predictions.

Dimensionality:

- Multi-head attention splits the embedding dimension across multiple heads, allowing parallel computation.
- The dimensions of Q, K, and V are consistent across all heads, facilitating efficient attention computation.

Testing and Verification:

- The implementation was rigorously tested against precomputed outputs, ensuring that all components of the attention mechanism functioned as expected.
- The correctness of masking, scaling, normalization, and projection was verified.

This implementation successfully adheres to the assignment requirements, providing a robust and efficient realization of causally masked, multi-head scaled dot-product self-attention.

8. Detailed Dimensional Analysis

Below are the exact dimensions based on the provided `GPTConfig` and the test code:

Configuration Parameters:

- Batch size $(B) = 2$
- Sequence length $(T) = 8$
- Embedding dimension $(C) = 12$
- Number of heads $(H) = 6$
- `head dimension` $= \frac{C}{H} = \frac{12}{6} = 2$

Step-by-Step Dimensions

1. *Input Tensor* (x)
   - Shape: $(B, T, C) = (2, 8, 12)$
2. *Linear Layer Output* (QKV)
   - The `qkv_map` projects x into a single tensor containing Q, K, and V for all heads:
   - Shape: $(B, T, 3 \times C) = (2, 8, 36)$
3. *Splitting into Q, K, and V*
   - After splitting, each tensor Q, K, and V has:

- Shape: $(B, T, C) = (2, 8, 12)$

4. *Reshaping and Transposing for Multi-Head Attention*
   - Each of Q, K, and V is reshaped and transposed to separate the attention heads:
   - Shape: $(B, H, T, d_k) = (2, 6, 8, 2)$
5. *Attention Score Computation* $(Q \cdot K^T)$
   - Attention scores are computed by taking the dot product of Q and K^T:
   - Shapes: $(B, H, T, d_k) \times (B, H, d_k, T) \to (B, H, T, T)$
   - Result: $(2, 6, 8, 8)$
6. *Causal Mask Application*
   - The causal mask is broadcasted to match the shape of the attention scores:
   - Mask shape: $(1, 1, T, T) = (1, 1, 8, 8)$
7. *Attention Weights (Softmax Output)*
   - The masked attention scores are passed through a softmax function:
   - Shape remains: $(B, H, T, T) = (2, 6, 8, 8)$
8. *Weighted Summation* $(\text{attn\_weights} \cdot V)$
   - The attention weights are used to compute the weighted sum of V:
   - Shapes: $(B, H, T, T) \times (B, H, T, d_k) \to (B, H, T, d_k)$
   - Result: $(2, 6, 8, 2)$
9. *Concatenation of Heads*
   - The outputs from all heads are concatenated along the head dimension:
   - Shape: $(B, T, H \times d_k) = (2, 8, 12)$
10. *Final Projected Output*
    - The concatenated tensor is passed through the final projection layer to return to the original embedding dimension:
    - Shape: $(B, T, C) = (2, 8, 12)$

Final Dimensions Summary:

- *Input Tensor* (x)*:* $(B, T, C) = (2, 8, 12)$
- *Linear Layer Output* (QKV)*:* $(B, T, 3 \times C) = (2, 8, 36)$
- *Split Q, K, and V:* $(B, T, C) = (2, 8, 12)$
- *Reshape for Heads:* $(B, H, T, d_k) = (2, 6, 8, 2)$
- *Attention Scores* $(Q \cdot K^T)$*:* $(B, H, T, T) = (2, 6, 8, 8)$
- *Attention Weights (Softmax Output):* $(B, H, T, T) = (2, 6, 8, 8)$
- *Weighted Summation Output:* $(B, H, T, d_k) = (2, 6, 8, 2)$
- *Concatenated Output:* $(B, T, H \times d_k) = (2, 8, 12)$
- *Final Projected Output:* $(B, T, C) = (2, 8, 12)$

0.2 Task c)

Implement the MLP class that follows each attention block. Follow the TODOs in the code. For the GELU activation function you can use nn.GELU3 from PyTorch.

0.3 Solution c)

MLP Class Overview

Summary of TODO #1: Implement the MLP

- **First Linear Layer (Snippet)**
  `self.fc1 = nn.Linear(config.n_embd, 4 * config.n_embd, bias=config.bias)`
  Expands the input dimensionality from `n_embd` to $4 \times$ `n_embd`. This is critical because it increases the representational capacity of the network, allowing it to capture more complex interactions before projecting back down.

- **GELU Activation (Snippet)**
  `self.gelu = nn.GELU()`
  Applies the GELU (Gaussian Error Linear Unit) function to introduce non-linearity. By using GELU instead of ReLU or other functions, the MLP can benefit from smoother gradient flow, which often leads to more stable training.

- **Second Linear Layer (Snippet)**
  `self.fc2 = nn.Linear(4 * config.n_embd, config.n_embd, bias=config.bias)`
  Projects the data back down to `n_embd`. This ensures that the output dimension matches the original embedding size, maintaining compatibility with subsequent blocks (e.g., attention layers).

- **Dropout (Snippet)**
  `self.dropout = nn.Dropout(config.dropout)`
  Applies dropout to reduce overfitting, with rate controlled by `config.dropout`. During training, dropout randomly zeroes some of the output features, preventing the network from relying too heavily on particular nodes.

Summary of TODO #2: Implement the Forward Pass

- **Step 1 (Snippet)**
  `x = self.fc1(x)`
  Expands the input from `n_embd` to $4 \times$ `n_embd`. This is where the network first increases its dimensionality, offering a larger feature space for the subsequent activation function to operate on.

- **Step 2 (Snippet)**
  `x = self.gelu(x)`
  Introduces non-linearity via the GELU activation. This step helps the MLP learn complex, non-linear relationships. Unlike standard ReLU, GELU can retain small negative values, potentially improving convergence.

- **Step 3 (Snippet)**
  `x = self.fc2(x)`

Projects the expanded representation back to `n_embd`. By returning to the original embedding size, the model can seamlessly integrate this output with subsequent network components (such as another attention block).

- **Step 4 (Snippet)**
  `x = self.dropout(x)`
  Randomly zeros some output features to mitigate overfitting. The dropout probability is specified by `config.dropout`, ensuring a tunable balance between regularization and model capacity.

- **Step 5**
  Returns the final output `x`, concluding the MLP forward pass. This final tensor matches the original embedding dimension, enabling consistent flow through the rest of the model architecture.

Key Observations:

- The MLP component acts as a feed-forward sub-network, considerably enriching the representational capacity between attention layers. By expanding to $4 \times$ `n_embd`, the model can capture a wider range of features before compressing them back down.

- Allowing bias to be switched on or off (`config.bias`) offers flexibility; some models may prefer bias-free layers for certain normalization setups, while others may benefit from the additional parameters.

- The dropout rate (`config.dropout`) and use of GELU can be tuned for specific tasks, potentially improving generalization and reducing convergence difficulties.

- Overall, this MLP block seamlessly integrates into the Transformer-like architecture, providing a robust mechanism to expand, activate, and then reduce dimensionality in tandem with the attention mechanism.

0.4 Task d)

A Block module which consists of CausalSelfAttention and MLP modules (as well as LayerNorm and residual connections) is already provided for you. Carefully read the code of the GPT class and complete the forward pass.

0.5 Solution d)

GPT Forward Pass with CausalSelfAttention and MLP Blocks

Block Module Overview (Code Snippets)

```
class Block(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.layernorm_1 = nn.LayerNorm(config.n_embd, bias=config.bias)
        self.attention = CausalSelfAttention(config)
        self.layernorm_2 = nn.LayerNorm(config.n_embd, bias=config.bias)
        self.mlp = MLP(config)
```

```
def forward(self, x):
    x = x + self.attention(self.layernorm_1(x))
    x = x + self.mlp(self.layernorm_2(x))
    return x
```

Each `Block` contains:

- A **LayerNorm** operation (twice), normalizing the embeddings before passing them to the attention and MLP sub-layers.
- The **CausalSelfAttention** module, enabling attention over the current and preceding tokens.
- The **MLP** module, providing a feed-forward expansion and projection.
- **Residual Connections**, added by returning `x + ...` after each sub-layer.

Key TODOs for the GPT Forward Pass

```
# DONE: Implement the forward pass of the GPT model
# 1) Embed the tokens and positions using the embedding layers self.transformer.embed_token
#    and self.transformer.embed_position.
# 2) Add the token embeddings and position embeddings together and pass the result
#    through the dropout layer.
# 3) Pass the result through all the transformer blocks.
# 4) Apply layer normalization.
# 5) Obtain the logits by projecting the result to the vocabulary space using the head layer
```

Below is an outline showing how these steps can typically be pieced together, referencing essential snippets rather than complete code:

- **Token + Position Embeddings (Snippet)**

  ```
  token_emb = self.transformer.embed_token(idx)        # shape: (B, T, n_embd)
  position_emb = self.transformer.embed_position(pos)   # shape: (B, T, n_embd)
  x = token_emb + position_emb
  x = self.transformer.dropout(x)
  ```

  In practice, `idx` would be the input token indices, and `pos` the positional indices. After summation, dropout is applied to help regularize the model.

- **Passing Through Blocks (Snippet)**

  ```
  for block in self.transformer.blocks:
      x = block(x)
  ```

  Each `Block` contains `CausalSelfAttention` and an `MLP` with LayerNorm and residual connections, as shown above.

- **Final Layer Normalization (Snippet)**

  ```
  x = self.transformer.ln_f(x)
  ```

A final LayerNorm can be applied after all blocks, aligning with standard Transformer architectures.

- **Projection to Vocabulary (Snippet)**

```
logits = self.lm_head(x)
```

This final projection maps the hidden representations to vocabulary-sized logits, enabling next-token predictions.

How the Steps Integrate

1. **Token + Position Embeddings:** Convert each token index to a dense embedding, then add a learned positional embedding.
2. **Dropout:** Immediately apply dropout to the summed embeddings to mitigate overfitting.
3. **Transformer Blocks:** Sequentially pass the embeddings through several `Block` modules. Each block:
   - Normalizes the input with `layernorm_1` and applies `CausalSelfAttention`.
   - Adds the attention output back to the input (*residual connection*).
   - Normalizes again with `layernorm_2` and applies the `MLP`.
   - Adds the MLP output back (*another residual connection*).
4. **Final Normalization:** Normalize the output of the last block.
5. **Logits Projection:** Feed the normalized representations to `self.lm_head`, which projects to the vocabulary dimension for token prediction.

By following these snippets, you can complete the forward pass of the GPT model in accordance with the provided TODOs. This includes properly embedding inputs, stacking multiple blocks containing `CausalSelfAttention` and `MLP`, and finally generating logits for the next-token predictions.

Task e)

The GPT model $f_\theta(x_{<t})$ not only outputs the logits z_t for the probability mass function $p_\theta(x_t|x_{<t})$, but also the logits for all distributions $\{p_\theta(x_k|x_{<k})\}_{k=2}^{t}$. Recall that given the logits z_t, we can compute the corresponding probability mass function as

$$p_\theta(x_t|x_{<t}) = \text{softmax}(z_t)_{x_t}.$$

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

$$p_\theta^\tau(x_k|x_{<k}) = \text{softmax}(z_k/\tau)_{x_k}.$$

Implement the `sample` method in the GPT class, which receives a temperature τ and a starting sequence $(x_1, \ldots, x_t)$ and autoregressively samples $x_{t+i} \sim p_\theta^\tau(x_{t+i}|x_{<t+i})$ for $i = 1, \ldots, \text{max\_new\_tokens}$. Can we condition on arbitrarily long sequences $x_{<t+i}$?

Solution e)

The sample method from the GPT model is implemented as given below:

```
@torch.no_grad()
def sample(self, idx, max_new_tokens, temperature=1.0):
    for _ in range(max_new_tokens):
      # Crop the input sequence if it exceeds the block size
      idx_input = idx if idx.shape[1] <=
      self.config.block_size else idx[:, -self.config.block_size:]

      # Forward pass: Get logits from the model
      logits, _ = self.forward(idx_input)

      # Extract logits for the last token in the sequence
      logits = logits[:, -1, :]

      # Apply temperature scaling
      logits = logits / temperature

      # Convert logits to probabilities
      probs = F.softmax(logits, dim=-1)

      # Sample next token from the probability distribution
      next_token = torch.multinomial(probs, num_samples=1)

      # Append the sampled token to the sequence
      idx = torch.cat((idx, next_token), dim=1)

    return idx
```

The method generates a sequence of tokens autoregressively by sampling one token at a time up to the maximum of *max_new_tokens*. We start with the input sequence *idx*, which is cropped to fulfill the size constraint. Then we conduct a forward pass in order to calculate the logits for the next token. The logits are then used to calculate probabilities with the SoftMax function, scaled by a temperature parameter to control randomness. At this point we can sample the next token by using the calculated probabilities with multinomial sampling. Every new predicted token is added to the input sequence and we do this until *max_new_tokens*.

The input sequence $x_{<t+1}$ cannot be arbitrarily long because the GPT model is designed with a fixed context window size, defined by its block size. When the length of the input sequence exceeds the block size, the model truncates it to include only the most recent block size tokens. This was done under this portion of the code:

```
idx_input = idx if idx.shape[1] <=
self.config.block_size else idx[:, -self.config.block_size:]
```

Task f)

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

Solution f)

The model was trained using the default hyperparameters as they have showed satisfactory results. Further hyperparameter tuning will be done in the following tasks. We have generated a plot on figure 1 that showcases the training and validation losses over iterations, with the default starting point hyperparameters. Additionally, we have plotted the training loss values after every 10 iterations, as they are done in the given code. This was done to be able to see the precise evolution of the training losses. On the other hand, we have plotted the validation loss values after every 500 iterations, once again as they are done in the given code. The figure 1 highlights the points at which the validation loss is logged. The last calculated validation loss has the value of 1.2632 at the iteration 4500, while the training loss was 1.2553.

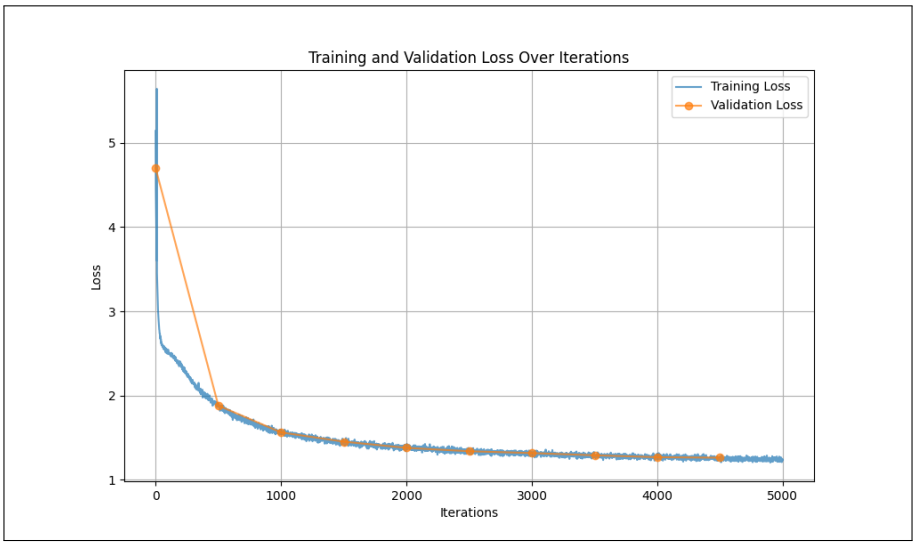

Figure 1: Evolution of train and validation losses with default hyperparameters

By observing the training dynamics, we can notice that both the training and validation losses decrease steadily. In fact, both losses are very close in values. This indicates that the model is not overfitting or underfitting. The model is also stable, with no sudden spikes or oscillations in the loss evolution on the plot. At the end, we can state that the model has converged due to the fact that the losses have stopped decreasing across iterations and we have reached a plateau.

Task g)

Generate samples from your trained model using temperatures $\tau \in \{1.5, 1.0, 0.8, 0.5, 0.1, 0.0001\}$. Include representative samples for each temperature in your report and comment how influences the generated text.

Solution g)

Generated sample for temperature 1.5:
Jesumbize, Rachelgie of Flyucks Kôur, tech, nortio voicei0. Nine to hit Darvescga Queen, atturing trique?!" Hats M'Ry's Fabl: After the Joter, Pava estable RPRP's work to right-swacastip butcher, a specialisterman witching, buttetiope universe, known and himself, visionn, hives valaged a girlin's wealthy studie" Clask Ci(Gor]²Teod: So 'Kijōz: Show Omaz: Through as Cinderella La Sovamboria, the Gale F. Wise: Super Johnji discovers 'Silver from gnokus, Sonion, mos iningle tomboyi's greatest clan p

Generated sample for temperature 1.0:
Marchus lifestylle, 10011. Celebis: A flying vassion: A fairy lover and mystery boss are satisfying the government's fresh boyfriend on the green son. The Red muder, she is illucted to help from part as a tracks at with his Green Gulf, his lifelong sister Tren with Billy about the Aisia Warked Chatter (Lany, hat Bolletten)'s, but certaix unsults raws that could be a real to life with the grhndy pet and romances again her from Hollywood vacationing king in citizens. Calm Show Academy: A post-meet

Generated sample for temperature 0.8:
Down Nutterwalker and the Green Nights: The story of the American struck behind and daughter. The Porn Flesh: A Nin-glanded operative Vasican grows can she loves. In the death of his past, if he is no unsured by the Dragon, Block inspires the evil Marvel against him: Although to kidnap a sister in the heart of Madasmas and her son Little series. Against the customers and her father's any married to their differences both spokers to better season and former July Charlotta invasion. What makes out

Generated sample for temperature 0.5:
The Adventure: The Last Movie: A group of teenage games to be a clone from a support paradise and a murder are more for mines. The story is a man who called to the band of Christ Bronzell, a life with a mysterious man who has no one them to its the lives of a police and in the wood doctor who have been to survive. The Gunsen in the Merch: A group of downfalls from the survival sexual family murders and the perfect are the terrorized death of her sister's and to be commander who has been more tha

Generated sample for temperature 0.1:
The Man and the Secret of the Man with a small town where he believes that he has to become a secret from the countryside and the story of the most powerful people and the country of the survivors of the most powerful powers who have been forced to save the monsters of the countryside and make a state and has been forced to control the story of the survivors and the beautiful serial killers. The Bank: A young girl was a superhero and his family friend and his family and his family and the subjec

Generated sample for temperature 0.0001:
The Man and the British School find themselves and the search for a survivor of the story of the countryside and the story of the survivors of the story and the story of the countryside and the story of the survivors of the story and the story of the countryside and the story of the survivors of the story and the story of the countryside and the story of the survivors of the story and the story of the countryside and the story of the survivors of the story and the story of the countryside and th

Interpretation of how the temperature influences the text
We can notice that the temperature significantly affects the diversity and the randomness of the generated text.

- Temperature 1.5: The text sample is quite chaotic and does not make much logical sense. There are many misspellings and incorrect word combinations. From this we can conclude that higher temperature values output text with a higher level of randomness.
- Temperature 1.0: From the logical point of view, the text with the temperature 1.0 makes more sense and retains some level of coherence. It is also relatively creative with unusual ideas, but with less misspellings and grammar mistakes compared to the temperature 1.5. However, it does still remain quite confusing and needs improvement in its structure.
- Temperature 0.8: This text has a better balance between randomness and structure. Ideas flow logically and there is still a little bit of creativity and diversity in the sample. It avoids overly repetitive or generic text.
- Temperature 0.5: Logic an structure is very good and compared to the previous temperature of 0.8, this sample is slightly less creativity and has reduced diversity. Therefore, due to the lower level or randomness, the text is more predictable, reducing the risk of nonsensical outputs.
- Temperature 0.1: The repetitiveness is more prominent in this generated text. It is quite simple and there are similar patterns used within it. Therefore, at very low temperatures, the model relies on the most probable outputs, leading to repetitive and monotonous text.
- Temperature 0.0001: The text is completely repetitive and seems like it is using only the most probable phrases and words, without adding any new content or originality. Therefore, this text is the most predictable and quite robotic.

Task g)

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

Solution g)

We began the task with hyperparameter tuning, where we experimented with the following parameter values:

Learning rate: [1e-2, 1e-3],
Weight decay: [1e-2, 1e-1],
Grad clip: [1.0, 0.5],
Dropout: Initially 0.0, and then 0.2 with the best detected model
Batch size: 128

For testing all possible combinations between these values, we have used a smaller model due to computational and time constraints.

Number of blocks: 4,
Number of heads: 4,
Embedding dimension: 128,
Block size: 128

More specifically, we have tested out 8 types of models listed in the below tables 1-8. All models used the smaller model parameters from above (n_blocks: 4, n_heads: 4, n_embd: 128, block_size: 128). Initially we have tested out the first 7 types of models that had dropout of 0.0, that lasted almost an hour to execute. Afterwards, we have taken the best model from the first seven (which was model 4), in order to test if the loss decreases with increasing the droput to 0.2. This was done in model 8 and we have noticed that the results did not improve. The results showed that the optimal loss was of value 1.2433 and was obtained from model 4.

Model 0
learning_rate: 0.01
batch_size: 128
weight_decay: 0.01
grad_clip: 1.0
dropout: 0.0
best validation loss: 2.5349

Table 1: Model 0

Model 1
learning_rate: 0.01
batch_size: 128
weight_decay: 0.01
grad_clip: 0.5
dropout: 0.0
best validation loss: 2.4356

Table 2: Model 1

Model 2
learning_rate: 0.01
batch_size: 128
weight_decay: 0.1
grad_clip: 1.0
dropout: 0.0
best validation loss: 1.5279

Table 3: Model 2

Model 3
learning_rate: 0.01
batch_size: 128
weight_decay: 0.1
grad_clip: 0.5
dropout: 0.0
best validation loss: 1.4817

Table 4: Model 3

Model 4
learning_rate: 0.001
batch_size: 128
weight_decay: 0.01
grad_clip: 1.0
dropout: 0.0
best validation loss: 1.2433

Table 5: Model 4

Model 5
learning_rate: 0.001
batch_size: 128
weight_decay: 0.01
grad_clip: 0.5
dropout: 0.0
best validation loss: 1.24505

Table 6: Model 5

Model 6
learning_rate: 0.001 batch_size: 128 weight_decay: 0.1 grad_clip: 1.0 dropout: 0.0
best validation loss: 1.2534

Table 7: Model 6

Model 7
learning_rate: 0.001 batch_size: 128 weight_decay: 0.1 grad_clip: 0.5 dropout: 0.0
best validation loss: 1.2451

Table 8: Model 7

Model 8
learning_rate: 0.001 batch_size: 128 weight_decay: 0.01 grad_clip: 1.0 dropout: 0.2
best validation loss: 1.399

Table 9: Model 8

After we have determined the best hyperparameters, we have experimented with increasing the model size of the best model. The new parameters that were used are given below. The validation loss obtained from the best model is of value 0.5877. The training took 60 minutes.

Number of blocks: 8,
Number of heads: 8,
Embedding dimension: 256,
Block size: 256,
Learning rate: 0.001,
Batch size: 128,
Grad clip: 1.0,
Dropout: 0.0

The plot 2 showcases the training and validation loss over iterations for this larger model.

Figure 2: Evolution of train and validation losses of the best model

We have also generated two samples with the best model with temperature 0.8, that balances the level of randomness and structure in the text.

Text 1 with temperature 0.8:
Jesus Christmas: The official story of the Legendary Last Valley of the Destruction on the American president and is released on the Navy warden who has gone refuge with the Lawre Bank and the world is about to believe in an undetectable book. The Last Dark: A successful, five-week penguins but first sight will have to go up in over as different sides every day. The Stolen Prince: A young farm boy migrates his groom and leads a single mom shooting in a modern student and stepbring boxer with a p

Text 2 with temperature 0.8:
The World of Metropolis: Mnike is a brilliant drug pug, a woman seeks redemption and let her for lost her beloved powers. She stars to worried love for someone she thinks she is about to believe. This film was exposed to an act of defeating the allied monses including the painting ring of triple makes a stand as he his cousin fantasies and inspires him to his house that he is the only daunting reality would she can be anything. Police Academy: Home of the Red Riding of Ozzy: When an airline coup

Compared to the initial baseline model with the default hyperparameters, the new larger model has a significantly lower validation loss. The first model had 1.26, while the best larger model has the loss of 0.588. If we compare the generated sample from both the models with temperature 0.8, the new larger model produces text that exhibits more coherence and structure, while still maintaining diversity. This is firstly caused by larger embedding sizes that allow the model to represent tokens with greater nuance and capture more semantic information. Therefore, the text is more contextually aware and coherent. The larger number of blocks increase the model's capacity to learn complex patterns and relationships in the data. By settting the block size to 256, the model should also be able to generate text with longer dependencies. Lastly, the number of heads support the model to capture more fine-grained relationships, leading to better handling

of ambiguous or nuanced contexts.

Lastly, regarding the computational requirements, we have used Google Colab with T4 GPU set as the runtime. For the complete training done under task h), we needed around 2 hours.

Furthermore, if needed, we have provided below 8 plots of the 8 models that were generated during hyperparameter search.

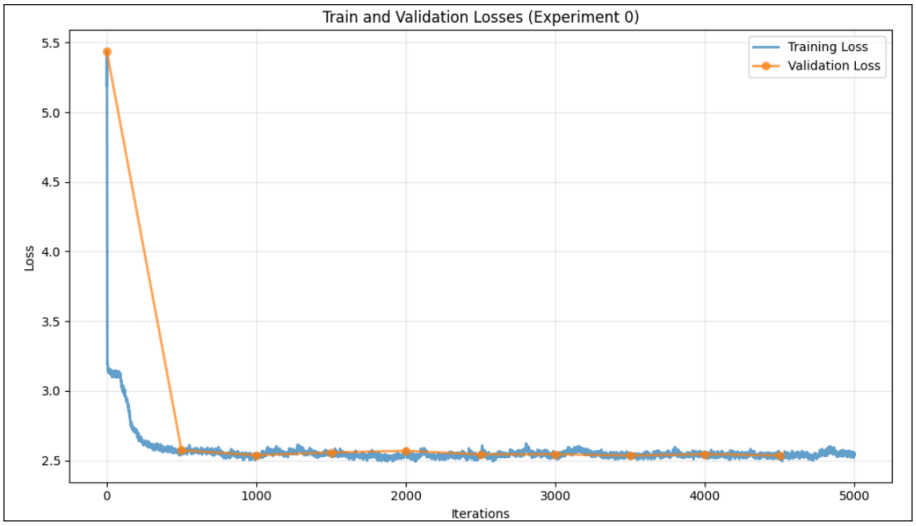

Figure 3: Evolution of train and validation losses of the Model 0

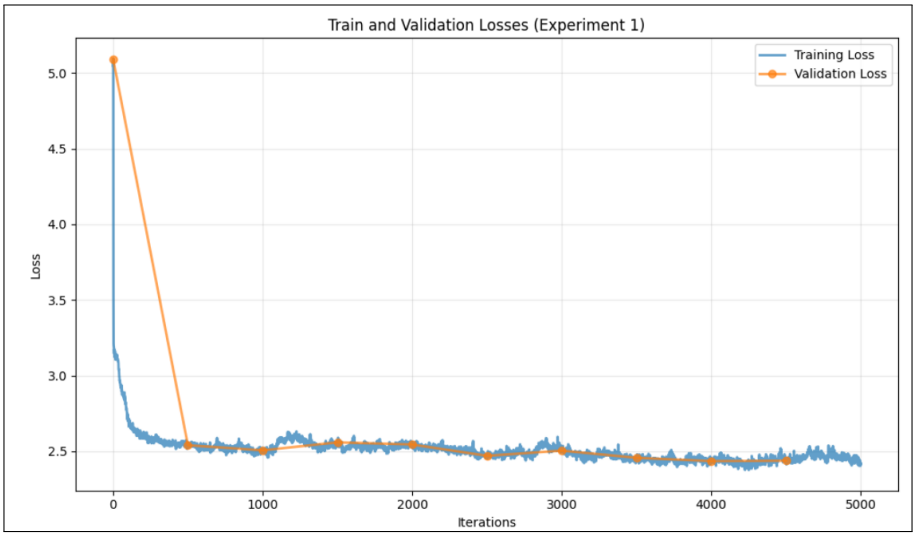

Figure 4: Evolution of train and validation losses of the Model 1

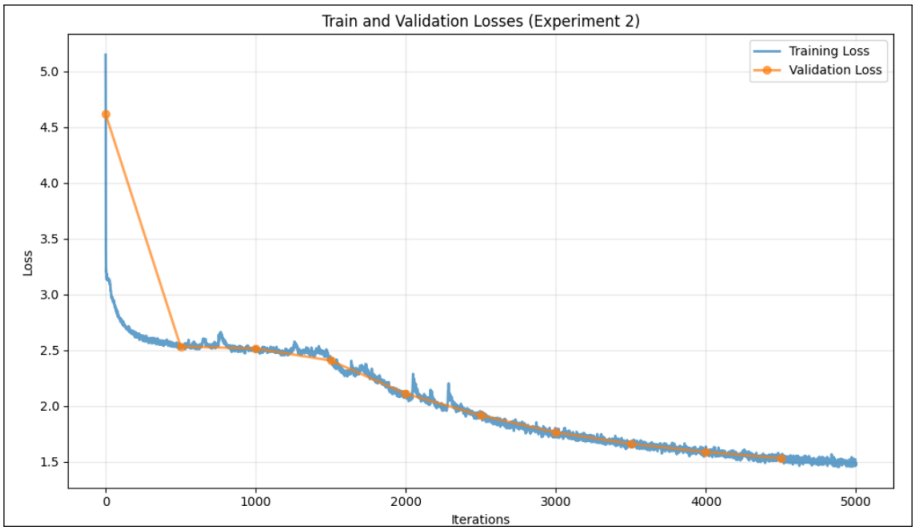

Figure 5: Evolution of train and validation losses of the Model 2

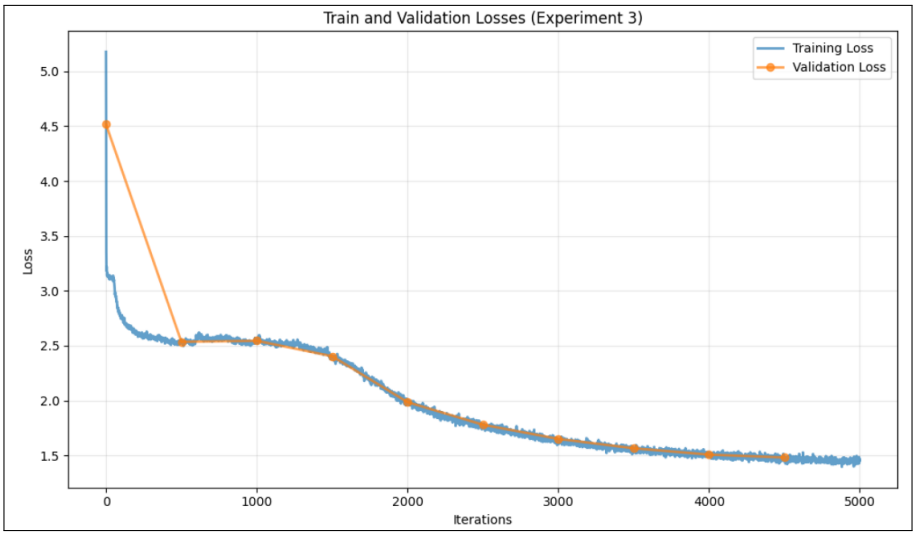

Figure 6: Evolution of train and validation losses of the Model 3

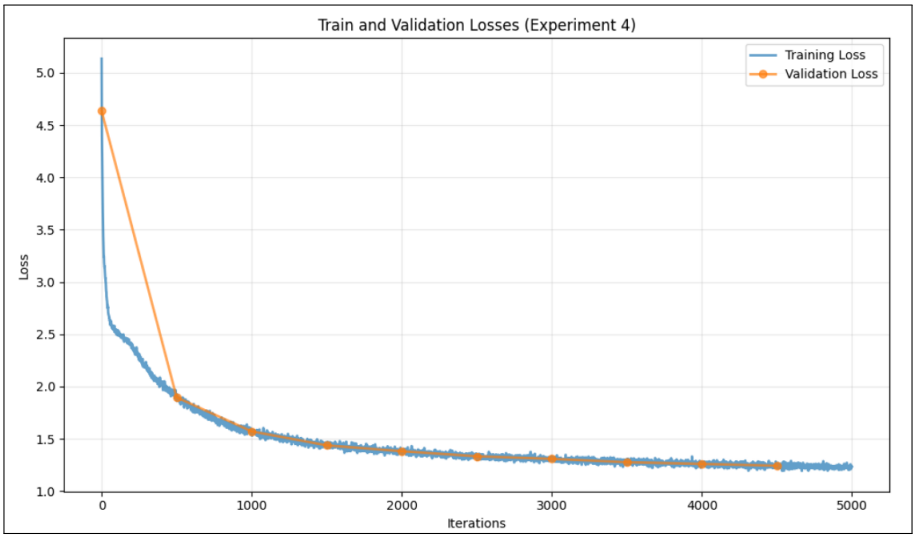

Figure 7: Evolution of train and validation losses of the Model 4

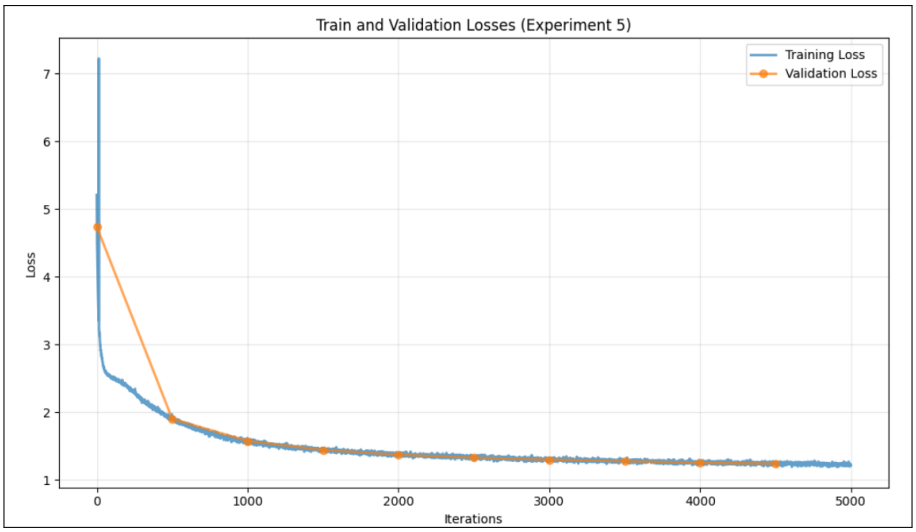

Figure 8: Evolution of train and validation losses of the Model 5

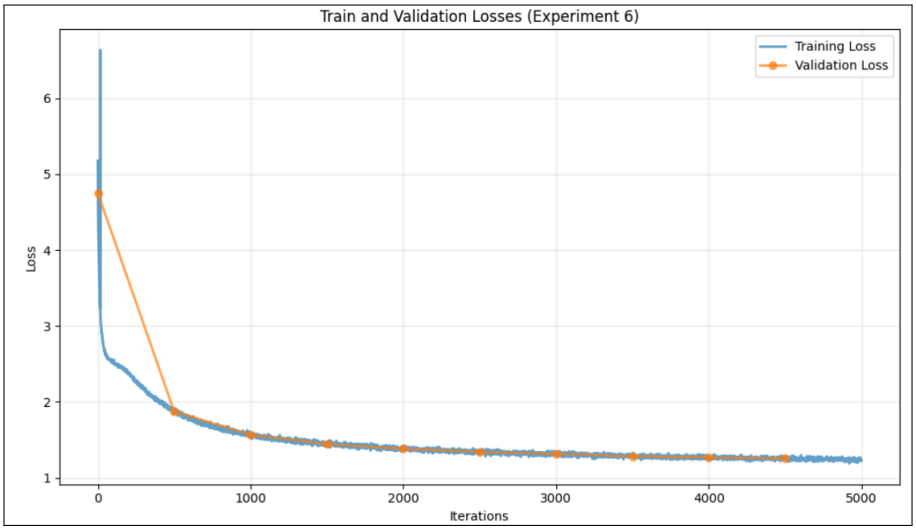

Figure 9: Evolution of train and validation losses of the Model 6

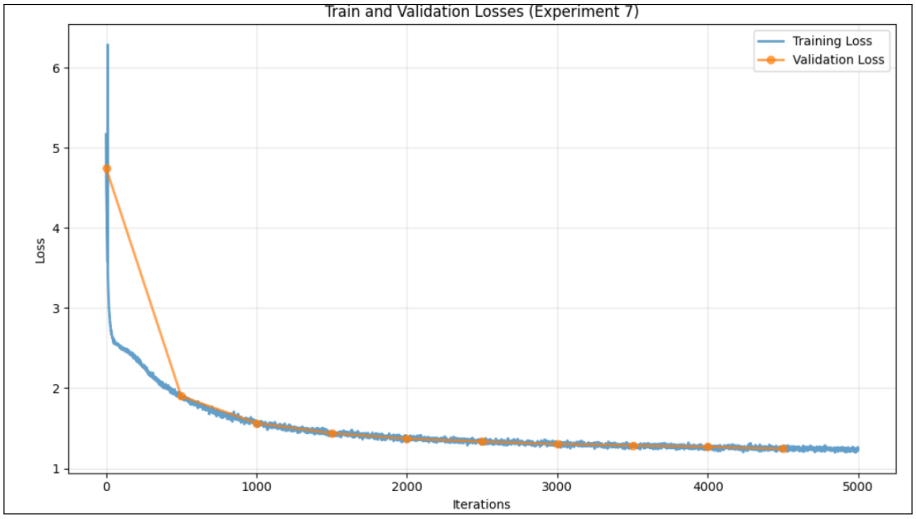

Figure 10: Evolution of train and validation losses of the Model 7

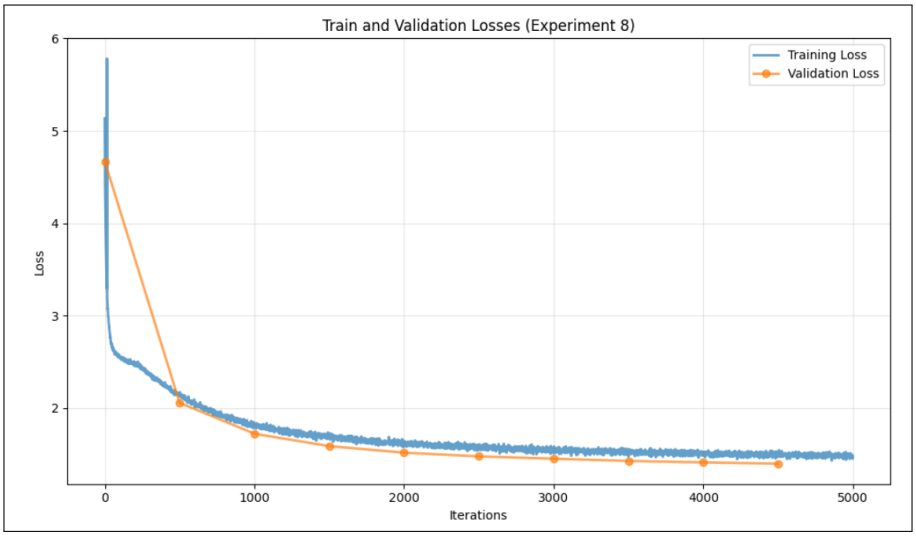

Figure 11: Evolution of train and validation losses of the Model 8