Deep Learning in Practice – Professor Spyros Gidaris

Homework 1 Technical Report - Language Decoder Only Model

Zhongke Sun

1. Introduction

This coding assignment explores the practice of a decoder-only transformer architecture using PyTorch. The implementation consists of constructing the fundamental elements of a transformer: embedding layers, positional encodings, self-attention mechanisms, and feed-forward networks. These components are combined to create a decoder-only architecture capable of processing text data. The model is then trained on the TinyStories dataset, a collection of short, human-like text samples, using a GPT-2 tokenizer. During the training process, I used some methods to optimize the results, such as dynamic learning rate and setting the early-stopping mechanism.

2. Import Libraries

In this coding assignment, I need to use *PyTorch*, *Math*, *Pandas*, *Seaborn*, *Matplotlib*. I also activated my GPU for faster training in Part 5.

3. Basic Transformer Class

3.1. Embedding Matrix

The embedding layer is a key component in transformer architectures, enabling the conversion of input tokens into dense, high-dimensional vectors. I used *nn.Embedding* to instantiate an embedding layer with a vocabulary size of 100 and an embedding dimension of 512, which maps each token in the vocabulary to a 512 dimensional vector.

The embedding matrix, representing the weights of the embedding layer. The first three rows of the embedding matrix are printed as the following:

```
Embedding Layer: Embedding(100, 512)
Shape of Embedding Matrix:torch.Size([100, 512])
First 3 Rows of Embedding Matrix:
tensor([[-1.1258, -1.1524, -0.2506,  ..., -1.6989,  1.3094, -1.6613],
        [-0.5461, -0.6302, -0.6347,  ...,  0.5374,  1.0826, -1.7105],
        [-1.0841, -0.1287, -0.6811,  ..., -0.0363,  0.0981,  0.9636]],
       grad_fn=<SliceBackward0>)
```

Figure 1. First Three Rows of Embedding Matrix

A sequence of tokens [0, 1, 2] was passed through the embedding layer. The resulting embedding vectors were printed and compared with the corresponding rows of the embedding matrix. The comparison confirmed that the output embeddings for tokens [0, 1, 2] matched exactly with the first three rows of the embedding matrix.

```
 Print embedding Vectors for Tokens 0, 1, 2:
tensor([[-1.1258, -1.1524, -0.2506,  ..., -1.6989,  1.3094, -1.6613],
        [-0.5461, -0.6302, -0.6347,  ...,  0.5374,  1.0826, -1.7105],
        [-1.0841, -0.1287, -0.6811,  ..., -0.0363,  0.0981,  0.9636]],
       grad_fn=<EmbeddingBackward0>)

Compare with the first 3 rows of the embedding matrix:
True
```

Figure 2. Comparison with the Corresponding Rows of the Embedding Matrix

3.2. Positional Encoding

Positional encoding enables the model to capture the order of tokens in a sequence. Unlike recurrent models, transformers process input sequences in parallel, and thus, positional information must be explicitly incorporated. This is achieved through sinusoidal functions that encode the position of each token along the embedding dimensions.

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \tag{1}$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \tag{2}$$

Where pos represents the position of the token in the sequence, and i represents the dimension index in the embedding vector.

3.2.1. Execution

A positional encoding matrix of shape (*max_seq_len*, *embed_model_dim*) was created, where *max_seq_len* is the maximum sequence length, and *embed_model_dim* is the embedding dimension. Sinusoidal and cosine values were computed for each position and dimension based on the equations above.

Then the positional encoding matrix was stored as an *nn.Parameter* to ensure it does not require gradients. During the forward pass, positional encodings corresponding to the sequence length were sliced and added to the input embeddings.

Sample input data of zeros with shape (*batch_size*, *sequence_length*, *embedding_dimension*) was passed through the *PositionalEmbedding* layer. The output data confirmed that positional information was added to the embeddings.

3.2.2. Result

A tensor of zeros representing sample input data with shape (*batch_size=2*, *sequence_length=10*, *embedding_dimension=16*).

```
Max_seq_len: 10
Sample Input Data:
tensor([[[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]],

        [[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]]])
```

Figure 3. Sample Input Data

The output data showed that sinusoidal and cosine values were correctly added to the embeddings. For instance, at position 0, all embedding dimensions had values corresponding to sinusoidal and cosine functions:

```
Output Data with Positional Encoding:
tensor([[[ 0.0000e+00,  1.0000e+00,  0.0000e+00,  1.0000e+00,  0.0000e+00,
           1.0000e+00,  0.0000e+00,  1.0000e+00,  0.0000e+00,  1.0000e+00,
           0.0000e+00,  1.0000e+00,  0.0000e+00,  1.0000e+00,  0.0000e+00,
           1.0000e+00],
         [ 8.4147e-01,  5.4030e-01,  9.9833e-02,  9.9500e-01,  9.9998e-03,
           9.9995e-01,  1.0000e-03,  1.0000e+00,  1.0000e-04,  1.0000e+00,
           1.0000e-05,  1.0000e+00,  1.0000e-06,  1.0000e+00,  1.0000e-07,
           1.0000e+00],
         [ 9.0930e-01, -4.1615e-01,  1.9867e-01,  9.8007e-01,  1.9999e-02,
           9.9980e-01,  2.0000e-03,  1.0000e+00,  2.0000e-04,  1.0000e+00,
           2.0000e-05,  1.0000e+00,  2.0000e-06,  1.0000e+00,  2.0000e-07,
           1.0000e+00],
         [ 1.4112e-01, -9.8999e-01,  2.9552e-01,  9.5534e-01,  2.9996e-02,
           9.9955e-01,  3.0000e-03,  1.0000e+00,  3.0000e-04,  1.0000e+00,
           3.0000e-05,  1.0000e+00,  3.0000e-06,  1.0000e+00,  3.0000e-07,
           1.0000e+00],
         [-7.5680e-01, -6.5364e-01,  3.8942e-01,  9.2106e-01,  3.9989e-02,
           9.9920e-01,  4.0000e-03,  9.9999e-01,  4.0000e-04,  1.0000e+00,
           4.0000e-05,  1.0000e+00,  4.0000e-06,  1.0000e+00,  4.0000e-07,
           1.0000e+00],
         [-9.5892e-01,  2.8366e-01,  4.7943e-01,  8.7758e-01,  4.9979e-02,
           9.9875e-01,  5.0000e-03,  9.9999e-01,  5.0000e-04,  1.0000e+00,
           5.0000e-05,  1.0000e+00,  5.0000e-06,  1.0000e+00,  5.0000e-07,
           1.0000e+00],
         [-2.7942e-01,  9.6017e-01,  5.6464e-01,  8.2534e-01,  5.9964e-02,
           9.9820e-01,  6.0000e-03,  9.9998e-01,  6.0000e-04,  1.0000e+00,
           6.0000e-05,  1.0000e+00,  6.0000e-06,  1.0000e+00,  6.0000e-07,
           1.0000e+00],
         [ 6.5699e-01,  7.5390e-01,  6.4422e-01,  7.6484e-01,  6.9943e-02,
           9.9755e-01,  6.9999e-03,  9.9998e-01,  7.0000e-04,  1.0000e+00,
           7.0000e-05,  1.0000e+00,  7.0000e-06,  1.0000e+00,  7.0000e-07,
           1.0000e+00],
         [ 9.8936e-01, -1.4550e-01,  7.1736e-01,  6.9671e-01,  7.9915e-02,
           9.9680e-01,  7.9999e-03,  9.9997e-01,  8.0000e-04,  1.0000e+00,
           8.0000e-05,  1.0000e+00,  8.0000e-06,  1.0000e+00,  8.0000e-07,
           1.0000e+00],
         [ 4.1212e-01, -9.1113e-01,  7.8333e-01,  6.2161e-01,  8.9879e-02,
           9.9595e-01,  8.9999e-03,  9.9996e-01,  9.0000e-04,  1.0000e+00,
           9.0000e-05,  1.0000e+00,  9.0000e-06,  1.0000e+00,  9.0000e-07,
           1.0000e+00]],
```

Figure 4. Part of Output Data with Positional Encoding

3.3. Self-Attention

Self-attention allows the model to dynamically weigh the importance of different tokens in a sequence when encoding or decoding information. This mechanism computes attention scores between all tokens in the input sequence, enabling the model to capture dependencies and contextual relationships effectively.

In this section, I implemented a multi-head self-attention mechanism. Multi-head attention divides the embedding space into smaller dimensions, computes self-attention independently for each head, and then combines the results. This allows the model to attend to different aspects of the input simultaneously.

3.3.1. Execution

In the *MultiHeadAttention* class, linear projection matrices were initialized for computing Query (Q), Key (K), and Value (V) vectors from the input embeddings. An output projection matrix was also defined to combine the results of the attention heads.

Then input embeddings are projected to Q, K, and V matrices using the learned linear transformations. For a batch size of 32, sequence length of 10, and embedding dimension of 512, the Q, K, and V matrices have shapes (32, 10, 512). The Q, K, and V matrices are reshaped to separate the heads. For 8 heads, each head operates on embeddings of size 64 (512/8), resulting in shapes (32, 10, 8, 64).

Furthermore, attention scores are computed as the scaled dot product of the Q and K matrices:

$$\text{Attention Scores} = \frac{Q \cdot K^T}{\sqrt{d_{head}}} \tag{3}$$

The attention weights are multiplied with the V matrix to compute the output for each head. The outputs from all heads are concatenated and passed through the output projection matrix to restore the original embedding dimension.

During decoding, a causal mask is applied to ensure that each token only attends to itself and previous tokens, preventing the model from "peeking" at future tokens.

3.3.2. Result

Input dimensions: (*batch_size=2*, *seq_length=10*, *embed_dim=512*)

Output dimensions after multi-head attention: (*batch_size=2*, *seq_length=10*, *embed_dim=512*)

```
tensor([-0.0003,  0.0237, -0.0364], requires_grad=True)
tensor([ 0.0141, -0.0128,  0.0270])
```

Figure 5. Weights from Query_matrix and Output from Attention

The executed result shows that the query, key, and value projections were computed as expected. And proper aggregation of information across all heads.

4. Decoder-only Architecture

The decoder-only transformer architecture focuses on processing sequences for autoregressive tasks, such as text generation, where the output is generated one token at a time. Unlike an encoder-decoder architecture, the decoder-only model leverages causal masking to ensure that future tokens in the sequence do not influence predictions for the current token.

The implementation mainly relies on *TransformerBlock* and *TransformerDecoderOnly*.

4.1. TransformerBlock

Each TransformerBlock represents a building block of the decoder, comprising the following components:

- **Multi-Head Self-Attention**: Captures dependencies between tokens in the sequence while applying a causal mask to prevent attention to future tokens.
- **Layer Normalization and Residual Connection**: Stabilizes training and enhances gradient flow.
- **Feed-Forward Network**: Increases the model's capacity by introducing non-linearity with two linear layers and a ReLU activation.
- **Dropout Layers**: Prevents overfitting by randomly zeroing out some connections.

4.1.1. Execution

Firstly, I applied self-attention (calculate attention) to the input sequence, using the causal mask. Then I added a residual connection and normalized. The output was passed through the feed-forward network. Finally, I added another residual connection and normalized to produce the final block output.

4.1.2. Result

A random input tensor with dimensions (*batch_size=32*, *seq_length=4*, *embed_dim=512*) was passed through a single TransformerBlock with 8 heads and an expansion factor of 4. The output shape matched the input shape, demonstrating correctness.

```
TransformerBlock test passed!
```

Figure 6. TransformerBlock test passed!

4.2. TransformerDecoderOnly

TransformerDecoderOnly stacks multiple *TransformerBlock* layers and incorporates embedding for tokens and positional information.

4.2.1. Execution

- **Masking**: Generates a causal mask for the sequence to ensure unidirectional attention.
- **Embedding**: Converts input tokens into dense vectors using word embeddings and positional encodings.
- **Layer Stacking**: Sequentially passes the input through multiple *TransformerBlock* layers.
- **Normalization**: Apply layer normalization to the output.
- **Output Projection**: Applies a final linear layer to project the decoder's output to the vocabulary size, enabling predictions for the next token.

4.2.2. Result

The *TransformerDecoderOnly* class was tested by initializing a model with 2 layers, 8 attention heads, and embedding dimensions of 512. A random sequence of token indices was passed through the model. The output shape matched (*batch_size*, *seq_length*, *vocab_size*).

```
Max_seq_len: 10
TransformerDecoderOnly passed the full forward pass test!
```

Figure 7. TransformerDecoderOnly passed the full forward pass test!

5. Train and Test our Decoder-only architecture

The decoder-only architecture was trained and tested on the *TinyStories* dataset using the GPT-2 tokenizer. The objective was to fine-tune a lightweight language model capable of generating coherent and creative English text. The training setup included defining model hyperparameters, loading and tokenizing the dataset, and configuring the training loop to optimize the model parameters.

5.1. Model Hyperparameters

The following hyperparameters were defined for the training process:

Table 1. Model Hyperparameters

Hyperparameters	Value
Batch Size	128
Max Sequence Length	64
Embedding Dimension	512
Number of Layers	6
Number of Attention Heads	8
Dropout Rate	0.1
Learning Rate	3×10^{-4}
Maximum Iterations	5000
Evaluation Iterations	500

5.2. Building Dataset and Dataloader

The *TinyStories* dataset, a collection of short, human-like narratives, was loaded using the *datasets* library. The first 12,000 stories were extracted for training and validation.

```
Print the first 1000 characters from the TinyStories:
One day, a little girl named Lily found a needle in her room. She knew it was difficult to play with it becau

Lily went to her mom and said, "Mom, I found this needle. Can you share it with me and sew my shirt?" Her mon

Together, they shared the needle and sewed the button on Lily's shirt. It was not difficult for them because
Once upon a time, there was a little car named Beep. Beep loved to go fast and play in the sun. Beep was a he

One day, Beep was driving in the park when he saw a big tree. The tree had many leaves that were falling. B
```

Figure 8. Print the First 1000 Characters from the TinyStories

The GPT-2 tokenizer from the *tiktoken* library was used to encode the text into token IDs, which are numerical representations of words or subwords. The tokenized data was stored as a PyTorch tensor for efficient processing.

```
Data after the the tokenization process (fist 100 indices):
tensor([ 3198,  1110,    11,   257,  1310,  2576,  3706, 20037,  1043,   257,
        17598,   287,   607,  2119,    13,  1375,  2993,   340,   373,  2408,
          284,   711,   351,   340,   780,   340,   373,  7786,    13, 20037,
         2227,   284,  2648,   262, 17598,   351,   607,  1995,    11,   523,
          673,   714, 34249,   257,  4936,   319,   607, 10147,    13,   198,
          198,    43,   813,  1816,   284,   607,  1995,   290,   531,    11,
          366, 29252,    11,   314,  1043,   428, 17598,    13,  1680,   345,
         2648,   340,   351,   502,   290, 34249,   616, 10147,  1701,  2332,
         1995, 13541,   290,   531,    11,   366,  5297,    11, 20037,    11,
          356,   460,  2648,   262, 17598,   290,  4259,   534, 10147,   526])
```

Figure 9. Data after the Tokenization Process (First 100 Indices)

The tokenized data was split into training and validation sets, with 90% of the data allocated for training and the remaining 10% for validation.

```
Train data shape torch.Size([2352219]), validation data shape torch.Size([261358])
```

Figure 10. Train & Validation Data Shape after Splitting

5.3. Create GPT Model And Its Optimizer

The decoder-only model was initialized using the defined hyperparameters, and an AdamW optimizer was configured for training.

```
Max_seq_len: 64
TransformerDecoderOnly(
  (word_embedding): Embedding(50257, 512)
  (position_embedding): PositionalEmbedding()
  (layers): ModuleList(
    (0-5): 6 x TransformerBlock(
      (attention): MultiHeadAttention(
        (query_matrix): Linear(in_features=512, out_features=512, bias=False)
        (key_matrix): Linear(in_features=512, out_features=512, bias=False)
        (value_matrix): Linear(in_features=512, out_features=512, bias=False)
        (output_projection): Linear(in_features=512, out_features=512, bias=True)
      )
      (norm1): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
      (norm2): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
      (feed_forward): Sequential(
        (0): Linear(in_features=512, out_features=2048, bias=True)
        (1): ReLU()
        (2): Linear(in_features=2048, out_features=512, bias=True)
      )
      (dropout1): Dropout(p=0.2, inplace=False)
      (dropout2): Dropout(p=0.2, inplace=False)
    )
  )
  (norm_out): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
  (fc_out): Linear(in_features=512, out_features=50257, bias=True)
)
```

Figure 11. Model Summary

The AdamW optimizer was chosen for training due to its efficiency and suitability for transformer-based models. The optimizer was initialized with the defined learning rate (3×10^{-4}).

Additionally, the number of trainable parameters in the model's key components was computed to better understand its complexity.

```
The number of trainable parameters in the input embedding layer: 25731584
The number of trainable parameters in all the transformer blocks: 18905088
The number of trainable parameters in the final linear layer of the model: 25781841
The total number of trainable parameters: 70419537
```

Figure 12. The Number of Trainable Parameters Summary

The total parameter count confirms that the model's complexity is consistent with its configuration.

5.4. Define the Loss Function

To optimize the decoder-only transformer model, the **cross-entropy loss** is implemented as the criterion for training here. This loss function measures the difference between the predicted logits and the target labels, ensuring the model learns to assign high probabilities to the correct tokens.

The cross-entropy loss is defined as:

$$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c} \cdot \log(p_{i,c}) \tag{4}$$

- N: Total number of samples (tokens).
- C: Number of classes (vocabulary size).
- $y_{i,c}$: Binary indicator (1 if class c is the correct label for token i, else 0).
- $p_{i,c}$: Probability assigned to class c by the model for token i.

Here:

- $y_{i,c}$: The ground truth token indices.
- $p_{i,c}$: Derived from the logits after applying softmax.

5.5. Train the GPT Model And Its Optimizer

In order to minimize training and validation loss, the training loop consists critical mechanisms for model optimization, including periodic evaluation, early stopping, and learning rate scheduling.

5.5.1. Learning Rate Scheduler

A cosine annealing scheduler is used to dynamically adjust the learning rate over the course of training. This ensures better convergence by gradually reducing the learning rate as training progresses.

5.5.2. Early Stopping

Training halts early if there is no improvement in validation loss for 5 consecutive evaluation intervals. This prevents overfitting and unnecessary computations.

The training process involves:

- Forward Pass: Input is passed through the model to compute logits.
- Loss Calculation: Cross-entropy loss is computed using the predicted logits and target labels.
- Backward Pass: Gradients are calculated and propagated backward to update model parameters.
- Learning Rate Update: Scheduler adjusts the learning rate after every iteration.
- Evaluation: Periodically evaluates the model on the validation set to monitor performance.

The result is as the following:

```
  0%|          | 0/5000 [00:00<?, ?it/s]
step 0: train loss 11.0375, val loss 11.0335
 10%|▉         | 502/5000 [01:38<2:58:27,  2.38s/it]
step 500: train loss 2.9695, val loss 3.0569
 20%|██        | 1002/5000 [03:04<2:38:57,  2.39s/it]
step 1000: train loss 2.5653, val loss 2.7616
 30%|███       | 1502/5000 [04:31<2:18:04,  2.37s/it]
step 1500: train loss 2.3334, val loss 2.6233
 40%|████      | 2002/5000 [05:57<1:58:41,  2.38s/it]
step 2000: train loss 2.1637, val loss 2.5527
 50%|█████     | 2502/5000 [07:23<1:38:52,  2.37s/it]
step 2500: train loss 2.0143, val loss 2.5310
 60%|██████    | 3002/5000 [08:50<1:19:19,  2.38s/it]
step 3000: train loss 1.8986, val loss 2.5216
 70%|███████   | 3502/5000 [10:17<59:30,  2.38s/it]
step 3500: train loss 1.7835, val loss 2.5251
 80%|████████  | 4002/5000 [11:43<39:25,  2.37s/it]
step 4000: train loss 1.6933, val loss 2.5465
 90%|█████████ | 4502/5000 [13:10<19:45,  2.38s/it]
step 4500: train loss 1.6029, val loss 2.5688
100%|██████████| 5000/5000 [14:36<00:00,  5.70it/s]
step 4999: train loss 1.5247, val loss 2.6006
```

Figure 13. The Training Process

The training process was executed successfully with validation loss stabilizing at 2.5216. The use of a learning rate scheduler and early stopping ensured efficient training.

5.6. Generate Stories Using the Trained Model

Eventually, the trained transformer decoder-only model is utilized to generate text. The model takes a context as input and predicts the next tokens iteratively, constructing coherent and creative text sequences. The output demonstrates the model's capability to capture patterns and generate stories based on the TinyStories dataset.

```
And so, one day, the fairy waved her wand and waved her wand. The fairy felt very proud of herself for becom

The fairy smiled and twirled around with pride, stars jumped out together. With the chirp, the fairy would re
Once upon a time, there was a swimming in the water. It was so beautiful that it almost fell from the ocean.

The happy sea knew it was time for a dive to sail. The seagull was happy that he got to please and skillfully

Later that day, the seahorse played catch their own water in the bathtub. The fish had a big stick that touch
```

Figure 14. Generated Story