Homework 2 - Natural Language Processing

Classification using Pretrain BERT model

This code implements a fine-tuning approach for the BERT-base-uncased model for a multiple-choice question answering task. The goal is to predict the correct answer from four choices given a fact and a question stem. Below is a desccription of what is going on:

1. **Data Preparation and Encoding:**

   - The input data is assumed to be in JSONL format, where each line represents a question instance with fields like `"fact1"`, `"question"` (containing `"stem"` and a list of `"choices"` with `"text"` and `"label"`), and `"answerKey"`.
   - The `MultipleChoiceDataset` class handles the loading and preprocessing of this data. For each question, it constructs four input sequences, one for each choice, in the following format: `[CLS] <fact> <stem> <text of choice #i> [SEP]`.
   - The `BertTokenizer` from the `bert-base-uncased` checkpoint is used to tokenize these sequences. Special tokens (`[CLS]` and `[SEP]`) are added, and the sequences are padded or truncated to a `max_length` (set to 128 in this code). Attention masks are also generated to indicate the actual tokens versus padding tokens.
   - The correct answer label (A, B, C, or D) is converted to a numerical index (0, 1, 2, or 3) using a `label_map`.

2. **Model Architecture:**

   - The `BertForMultipleChoiceCustom` class defines the model. It utilizes the pre-trained `BertModel` from the `bert-base-uncased` checkpoint as the base encoder.
   - A single linear layer (`nn.Linear`) is added on top of the hidden state of the `[CLS]` token from the final layer of the BERT model. This linear layer maps the `[CLS]` embedding to a single output value (logit) for each choice.

3. **Classification Methodology:**

   - During the forward pass, for each input instance (with four choices), the four encoded sequences are passed through the `BertModel`.
   - The hidden state corresponding to the `[CLS]` token (the first token) from the last layer is extracted for each of the four choices. This `[CLS]` embedding is intended to represent the entire input sequence's contextualized information.
   - Each of these four `[CLS]` embeddings is then fed into the linear classification layer, producing a single logit score for each choice.
   - These logits are then used with a `CrossEntropyLoss` function during training to compare against the true label. The model learns to assign a higher logit score to the correct choice.

4. **Fine-tuning:**

   - The `BertForMultipleChoiceCustom` model is fine-tuned on the training dataset.

- The `AdamW` optimizer (from `torch.optim`) is used to update the model's parameters. A learning rate of 2e-5 is set.
- The training loop iterates for a specified number of epochs (3 in this code). In each epoch, the model is trained on batches of data. The loss is calculated, backpropagation is performed, and the optimizer updates the weights.
- After each epoch, the model's performance is evaluated on the validation dataset to monitor progress and potentially tune hyperparameters (though not explicitly done in this basic script). Accuracy is calculated as the percentage of correctly predicted answers.

5. **Inference:**

    - For inference on the validation and test sets, the same data preprocessing steps are applied to create the input sequences.
    - The fine-tuned model is used to predict the logits for each of the four choices.
    - The predicted answer is the choice with the highest logit score (obtained using `torch.argmax` along the choice dimension).
    - The accuracy is then calculated by comparing the predicted answers with the true labels.

Reported Accuracies

Zero-Shot Accuracy:

- **Validation Set:** 28%
- **Test Set:** 22%

Fine-tuned Accuracy:

- **Validation Set:** 65%
- **Test Set:** 64%

Limitations of the Approach and Possible Solutions

1. **Input Length Constraints:** BERT has a maximum input sequence length (typically 512 tokens). If the combined length of the fact, stem, and choice text exceeds this limit, the tokenizer will truncate the input, potentially leading to a loss of crucial information.

    - **Possible Solutions:** Explore different truncation strategies (e.g., prioritizing the choice text), or if the context is very long, consider techniques like sliding window (though this adds complexity to the classification).

2. **Computational Cost:** Fine-tuning large transformer models like BERT can be computationally expensive and time-consuming, especially with larger datasets and more training epochs.

    - **Possible Solutions:** Utilize GPUs for faster training, experiment with smaller batch sizes if memory is a constraint, or explore more efficient fine-tuning techniques if the task allows.

3. **Hyperparameter Tuning:** The hyperparameters (learning rate, batch size, number of epochs, maximum sequence length) are set to fixed values in this script. These values might not be optimal for the specific dataset.

- **Possible Solutions:** Implement a hyperparameter search strategy (e.g., grid search or random search) using the validation set to find a better configuration.

This approach provides a basic yet effective method for fine-tuning BERT for multiple-choice question answering by treating each choice as a separate input and using the `[CLS]` token embedding for classification. The reported accuracies will indicate the effectiveness of this strategy on the given dataset.

Generation of a Multiple Choice Letter (A, B, C, or D) using custom Decoder-only Transformer Architecture

This code implements a generative approach to answer multiple-choice questions from the OpenBookQA dataset. The core idea is to train the custom Transformer model (in HW1) to generate the correct answer key (A, B, C, or D) given the fact, question stem, and answer choices.

1. **Data Preparation and Tokenization:**

- The `tokenize_data` function reads the OpenBookQA JSONL files. For each question, it constructs a sequence that includes:
  - A `[START]` token to mark the beginning.
  - The `<fact1>` enclosed in angle brackets.
  - The `<question stem>` enclosed in angle brackets.
  - Each answer choice, prefixed with its label (`[A]`, `[B]`, `[C]`, `[D]`) and enclosed in angle brackets.
  - An `[ANSWER]` token to indicate the start of the target answer.
  - The correct `answerKey` (A, B, C, or D) enclosed in angle brackets.

- The `encode_sequences` function uses the `GPT2TokenizerFast` to convert these text sequences into sequences of token IDs. Special tokens `[START]`, `[A]`, `[B]`, `[C]`, `[D]`, and `[ANSWER]` are added to the tokenizer's vocabulary to be treated as distinct units.

- The `QADataset` class prepares the data for training by creating input-target pairs of fixed `seq_length`. For each sequence, the input is all tokens except the last, and the target is the input sequence shifted by one position (standard language modeling objective).

2. **Model Architecture:**

- The code utilizes a custom `Transformer` model defined in a separate `hw1.py` file.
- The `load_pretrained_model` function attempts to load weights from a pre-trained model (specified by `model_path`). It handles the vocabulary expansion needed due to the addition of the special tokens. The embeddings and output layers are initialized with the pre-trained weights for the original vocabulary and randomly initialized for the new special tokens.

3. **Pre-training:**

- The model was pretrained on the wiki103 dataset as define in `hw1.py`

4. **Fine-tuning:**

- The `main` function loads the pre-trained model and fine-tunes it on the OpenBookQA training data.
- The `AdamW` optimizer is used with a learning rate of 5e-5.

- The `CrossEntropyLoss` is the primary loss function, ignoring padding tokens.
- A `focused_loss` function is implemented, which combines the standard language modeling loss with a higher weight on the loss at the position of the answer token (`[ANSWER]`). This encourages the model to specifically learn to predict the correct answer key.
- The training loop iterates for a specified number of epochs (20 in this code). During each epoch, the model processes the training data in batches, calculates the loss, performs backpropagation, and updates the model's weights. The average loss is printed for each epoch.

5. **Evaluation:**

- The `evaluate_model` function calculates the accuracy on a given dataset (validation or test). It iterates through the data, makes predictions, and checks if the token predicted immediately after the `[ANSWER]` token matches the true answer key token.
- The `verify_tokenization` function prints the token IDs and decoded forms of the special tokens to ensure they are correctly processed by the tokenizer.

7. **Limitations of the Generative Approach:**

- **Generation Complexity:** Generating coherent and accurate text, especially for a specific task like question answering, can be challenging. The model might generate unexpected or irrelevant tokens after the `[ANSWER]` token.
- **Vocabulary Bias:** The pre-trained vocabulary might not optimally represent all the nuances of the OpenBookQA dataset, even with the addition of special tokens.

8. **Possible Solutions:**

- **Larger Pre-trained Models:** Utilizing larger and more capable pre-trained models could improve the generation quality and task understanding.
- **Multi-task Learning:** Training the model on multiple related tasks could improve its ability to understand and generate relevant information.

Fine-tuned Accuracy:

- **Validation Set:** 25%
- **Test Set:** 27%

I realize that this model is just guessing, but according to the loss the model was actually learning. The issue is even if I evaluate it for longer it still doesn't improve the accuracy. This was a puzzling issue and I couldn't solve it within the time constraints.

Logs

```
Epoch 1, Avg Loss: 3.3675
Epoch 2, Avg Loss: 2.1510
Epoch 3, Avg Loss: 2.0227
Epoch 4, Avg Loss: 1.9654
Epoch 5, Avg Loss: 1.9270
Epoch 6, Avg Loss: 1.8930
Epoch 7, Avg Loss: 1.8316
Epoch 8, Avg Loss: 1.7028
Epoch 9, Avg Loss: 1.5223
Epoch 10, Avg Loss: 1.3773
Epoch 11, Avg Loss: 1.2256
Epoch 12, Avg Loss: 1.1038
Epoch 13, Avg Loss: 1.0636
Epoch 14, Avg Loss: 0.9734
Epoch 15, Avg Loss: 0.9253
Epoch 16, Avg Loss: 0.8787
Epoch 17, Avg Loss: 0.8480
Epoch 18, Avg Loss: 0.8171
Epoch 19, Avg Loss: 0.8094
Epoch 20, Avg Loss: 0.7808
```

Generation of Actual Text Answer

This code implements a generative approach to answer questions from the OpenBookQA dataset by training a Transformer model to directly generate the text answer.

1. **Data Preparation and Tokenization:**

- The `tokenize_data` function processes the OpenBookQA JSONL files. For each question, it constructs a sequence containing:
  - `[START]` token.
  - `<fact1>` enclosed in angle brackets.
  - `<question stem>` enclosed in angle brackets.
  - Each answer choice, prefixed with its label (`[A]`, `[B]`, `[C]`, `[D]`) and enclosed in angle brackets.
  - `[ANSWER]` token to mark the beginning of the answer.
  - `<answerText>` (the actual text answer) enclosed in angle brackets. If the `answerText` field is missing, it defaults to "N/A".
- The `encode_sequences` function uses the `GPT2TokenizerFast` to convert these text sequences into token IDs. Special tokens `[START]` and `[ANSWER]` are added to the tokenizer's vocabulary, along with the choice labels if not already present.
- The `QADataset` class prepares input-target pairs for training with a fixed `seq_length`. The input is all tokens except the last, and the target is the input shifted by one position (standard language modeling).

2. **Model Architecture:**

- The `load_pretrained_model` function loads weights from a pre-trained Transformer model. The vocabulary is expanded to include the new special tokens, with embeddings and output weights for these tokens initialized randomly.

3. **Pre-training:**

- The model is trained on the wiki103 dataset as defined in `hw1.py`

4. **Fine-tuning:**

- The `main` function fine-tunes the pre-trained model on the OpenBookQA training data.
- The `AdamW` optimizer is used with a learning rate of 5e-5.
- The `CrossEntropyLoss` is the primary loss function, ignoring padding tokens.
- A `focused_loss` function is implemented to emphasize the generation of the correct answer. It calculates the standard language modeling loss and combines it with a loss calculated specifically on the tokens following the `[ANSWER]` token, giving more weight to the accurate generation of the answer text.

5. **Evaluation:**

- The `evaluate_model` function evaluates the model's generative performance using several metrics:
  - **BLEU (Bilingual Evaluation Understudy):** Measures the n-gram overlap between the generated and ground truth answers.
  - **ROUGE (Recall-Oriented Understudy for Gisting Evaluation):** Measures the overlap of n-grams, word sequences, and word pairs between the generated and reference texts.
  - **BERTscore:** Computes a similarity score between the generated and reference sentences using pre-trained BERT embeddings, capturing semantic similarity better than n-gram based metrics.
- The function iterates through the validation and test datasets, generates answers conditioned on the input sequence up to the `[ANSWER]` token, and then calculates these metrics by comparing the generated text with the `answerText` from the dataset.

6. **Limitations of the Generative Approach:**

- **Sensitivity to Training Data:** The quality and diversity of the `answerText` in the training data heavily influence the model's ability to generate correct and coherent answers.
- **Computational Cost:** Training generative Transformer models can be computationally expensive.

7. **Possible Solutions:**

- **Data Augmentation:** Augmenting the training data with paraphrased questions and answers could improve the model's robustness.
- **More Sophisticated Decoding Strategies:** Experiment with different decoding strategies (e.g., beam search with length penalties) to improve the quality of the generated answers.
- **Larger Pre-trained Models:** Using larger and more capable pre-trained models could lead to better generative performance.

8. **Performance of Generative vs. Classification Approach:**

- **Generative Approach (this code):** Directly generates the text answer. Performance is evaluated using metrics like BLEU, ROUGE, and BERTscore, which assess the similarity between the generated and ground truth text.
- **Classification Approach (from previous context):** Predicts the correct answer choice from a given set of options. Performance is typically evaluated using accuracy.

Results

Average BLEU score: 0.0041 Average ROUGE scores: {'rouge1': 0.04208231164594359, 'rouge2': 0.0, 'rougeL': 0.03996190007753424} Average BERTscore F1: 0.8057

Validation BLEU: 0.0037 Validation ROUGE: {'rouge1': 0.04641229124214067, 'rouge2': 0.0, 'rougeL': 0.04399413648023591} Validation BERTscore F1: 0.8027

Test BLEU: 0.0041 Test ROUGE: {'rouge1': 0.04208231164594359, 'rouge2': 0.0, 'rougeL': 0.03996190007753424} Test BERTscore F1: 0.8057

These results are pretty terrible, but I couldn't figure out where my model was going wrong. My answer predicitons were mainly only 1-4 letters, and I didn'y know why. The loss of the model was decreasing so it was learning something, I am just not sure wha. I speculate it was my generation function that was messing up, but I did not have a lot of time to fix it as I wanted to get this turned in on time