Quality Estimation for Machine Translation Using BERT and ParaCrawl Dataset

Project Overview

This project aims to estimate the quality of machine translation between English and another language (e.g., Gaelic) without the need for human evaluation. We use the ParaCrawl dataset to train a BERT-based model that predicts a quality score for translation pairs. The model architecture is based on **BERT** (Bidirectional Encoder Representations from Transformers), which provides contextual embeddings for both source and target sentences.

The project focuses on training a neural network to predict the fluency and adequacy of translations and includes mechanisms for efficient memory management during training on GPU.

Project Files and Structure

- **qemodel.py**: Defines the BERT-based Quality Estimation model.
- **train.py**: Main training loop to fine-tune the BERT model and predict translation quality.
- **utils.py**: Contains utility functions like downloading dataset and parsing the qz file.
- **requirements.txt**: Python package dependencies for running the project.
- **README.md**: Detailed project documentation, including setup and usage instructions.

Key Components and Workflow

1. Dataset

- **ParaCrawl Dataset**: We use bilingual translation pairs (English-to-target language) from the ParaCrawl dataset. This dataset consists of aligned

sentence pairs extracted from web sources, and it is used to train a model that estimates the quality of translations.

- **Download & Preprocessing**: We download the dataset, tokenize the text using **BERT Tokenizer**, and convert the data into input IDs and attention masks that can be fed into BERT. The dataset is divided into training and testing sets.

Code:

```
source_texts, target_texts = load_paracrawl_data()  # Download and extract sentences
tokenizer = BertTokenizer.from_pretrained('bert-base-multilingual-cased')
dataset = ParaCrawlDataset(source_texts, target_texts, tokenizer)
```

2. Model Architecture

- **BERT-based Model**: The model uses a pre-trained multilingual BERT (bert-base-multilingual-cased) to encode both the source and target sentences. The output from the BERT model is passed to a linear layer (regressor) that predicts a single quality score for each translation pair.
- **CLS Token Output**: We use the pooled output of the [CLS] token, which is a contextual representation of the entire sentence pair, to feed into the regression layer.

Code:

```
class QEModel(nn.Module):
  def __init__(self, bert_model_name):
    super(QEModel, self).__init__()
    self.bert = BertModel.from_pretrained(bert_model_name)
    self.regressor = nn.Linear(self.bert.config.hidden_size, 1)

  def forward(self, input_ids, attention_mask)
    outputs = self.bert(input_ids=input_ids, attention_mask=attention_mask)
    pooled_output = outputs.pooler_output
    score = self.regressor(pooled_output)
    return score
```

3. Training Loop

- **Loss Function**: We use **Mean Squared Error (MSE)** as the loss function since we're predicting a continuous quality score.
- **Optimizer**: We use **AdamW** optimizer with weight decay to fine-tune the pre-trained BERT model and learn the parameters of the regression layer.
- **Batch Training**: The training is done in batches (batch size of 16). For each batch, we move the data to the GPU, perform a forward pass through the model, compute the loss, backpropagate the gradients, and update the model's parameters.
- **CUDA Memory Management**: To ensure efficient GPU memory usage, we clean up unused CUDA memory and delete unused tensors after each batch to prevent out-of-memory errors.

Code:

```
for epoch in range(epochs):

    model.train()
    total_loss = 0

    for batch in data_loader:
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        scores = batch['score'].to(device)

        optimizer.zero_grad()
        outputs = model(input_ids=input_ids, attention_mask=attention_mask).squeeze()
        loss = criterion(outputs, scores)
        loss.backward()
        optimizer.step()
```

4. Evaluation

- **Evaluation Loop**: After training, the model is evaluated on a test set. During evaluation, we disable gradient computation using torch.no_grad() to reduce memory usage and increase inference speed.

- **Prediction and Metrics**: For each batch in the test set, we predict the translation quality score and compute the Mean Squared Error (MSE) between predicted and true scores.

Code:

```
model.eval()
predictions, true_scores = [], []

with torch.no_grad():
   for batch in data_loader:
      input_ids = batch['input_ids'].to(device)
      attention_mask = batch['attention_mask'].to(device)
      scores = batch['score'].to(device)

      outputs = model(input_ids=input_ids, attention_mask=attention_mask).squeeze()
      predictions.extend(outputs.cpu().numpy())
      true_scores.extend(scores.cpu().numpy())

      del input_ids, attention_mask, scores, outputs
      torch.cuda.empty_cache()

mse = mean_squared_error(true_scores, predictions)
print(f"Test Mean Squared Error: {mse:.4f}")
```

4. Results

After running the code for the first 100,000 data values, the final loss of the model was around 5e-4. With the test loss being 0.05. With this I can say that the model that was made can work pretty well in distingusing the quality of the tranlation automatically.

How to Run the Project

1. Setup Environment

- Install the required dependencies by running:

```
pip install -r requirements.txt
```

2. Download and Preprocess Data

- Download the ParaCrawl dataset and preprocess it by running:

```
python dataset_downloader.py
```

3. Train the Model

- Start training the BERT-based quality estimation model by running:

```
python train.py
```

4. Evaluate the Model

- After training is complete, the model will be evaluated on the test set. The Mean Squared Error (MSE) will be printed to the console.

Project Notes

1. **GPU Utilization**: The project is optimized to run on a GPU. If a GPU is not available, the model will fall back to the CPU, though this will slow down training significantly.
2. **Memory Management**: We have incorporated memory cleanup functions (`torch.cuda.empty_cache()`) to prevent out-of-memory errors when working with large datasets and models on GPU.
3. **Hyperparameter Tuning**: Feel free to experiment with different hyperparameters like learning rate, batch size, and the number of epochs for better results.

Conclusion

This project demonstrates the use of a pre-trained BERT model for the task of **Quality Estimation** in **Machine Translation**. By utilizing the ParaCrawl dataset and leveraging GPU for efficient training, we build a model that can predict translation quality without human annotations. This approach can be extended to other language pairs or adapted for specific domains of machine translation.