Task 1. Dataset Acquisition - Your first task is to find a suitable text dataset. (1 points)

1) Choose your dataset and provide a brief description

The dataset selected is Star Wars - Thrawn Trilogy 01: Heir to the Empire by Timothy Zahn. It is a text-rich dataset containing the full transcription of the first book in the *Thrawn Trilogy* set in the *Star Wars* universe.

Dataset Description:

This dataset is part of the *Star Wars* Expanded Universe and focuses on the New Republic's battle against a new Imperial threat, Grand Admiral Thrawn. It is suitable for language modeling tasks as it contains rich, narrative text with dialogue and descriptions.

Dataset Details:

- Source: Hugging Face datasets library
- Name: myothiha/starwars
- Features: The dataset contains a single feature: text.
- Split: Train: 7,860 rows, Validation: 8,101 rows and Test: 9,236 rows

Task 2. Model Training - Incorporate the chosen dataset into our existing code framework. Train a language model that can understand the context and style of the text.

1) Detail the steps taken to preprocess the text data. (1 points)

Tokenization:

In this step, the text data is broken down into smaller units (tokens). For this task, the torchtext tokenizer with the 'basic_english' option is used. It splits the text into tokens, removing punctuation and normalizing case.

- Tokenizer: The torchtext.data.utils.get_tokenizer('basic_english') function is used to create a tokenizer.
- Function to tokenize: A lambda function (tokenize_data) is applied to the dataset where each example's text is tokenized into a list of tokens. The function removes the original text column and replaces it with a new tokens column.

Numericalization (Turning Text into Numbers):

After the text is broken into smaller pieces (tokens), those pieces (words) need to be changed into numbers so the model can understand them.

- Building a Vocabulary:
    - A vocabulary is a list of all the words (or tokens) in the dataset.
    - This list is created by looking at all the words in the training data and including only the words that appear at least 3 times. Words that appear less often are ignored.

- Special words like <unk> (representing "unknown word") and <eos> (marking "end of sentence") are added to the vocabulary.

- Setting a Default for Unknown Words:
    - If the model encounters a word it hasn't seen (one not in the vocabulary), it will use the <unk> token instead.
- Adding Special Tokens:
    - The <unk> token is inserted at index 0, and the <eos> token is placed at index 1. These tokens help the model handle cases like unknown words or knowing when a sentence ends.

Batch Preparation (Grouping Data for Training):

Once the text is converted into numbers, the data needs to be organized into smaller groups (batches) for training.

- Token Conversion:
    - Each word in the text is replaced with its corresponding number from the vocabulary.
- Making Batches:
    - The data is split into batches, which are groups of numbers. Each batch contains a set of words (now numbers) that will be processed together by the model.
    - The batches are arranged so that each one has the same number of words. Any leftover words that don't fit into a full batch are discarded.

Each batch contains numbers representing the words from the text. These batches are used by the model for training.

2) Describe the model architecture and the training process. (1 points)

The architecture of the language model is built using an LSTM (Long Short-Term Memory) network, which is particularly effective for processing sequences of data, such as text.

breakdown of the model:

1. Embedding Layer:
    - Purpose: Converts the words (tokens) into dense vectors of fixed size (embeddings). This helps in capturing the semantic meaning of words in the context of the data.
    - Implementation: nn.Embedding(vocab_size, emb_dim) where vocab_size is the size of the vocabulary and emb_dim is the dimensionality of the word embeddings.
2. LSTM Layer:

- Purpose: The LSTM processes the embedded word vectors sequentially. It learns to capture long-range dependencies in the text by maintaining a memory cell across timesteps.
- Implementation: nn.LSTM(emb_dim, hid_dim, num_layers=num_layers, dropout=dropout_rate, batch_first=True) where:
    - emb_dim: The dimensionality of the input embeddings.
    - hid_dim: The number of hidden units in the LSTM.
    - num_layers: The number of stacked LSTM layers.
    - dropout_rate: The probability of dropout applied during training to prevent overfitting.
    - batch_first=True: Ensures that input data is expected in the format (batch_size, seq_len).

3. Dropout Layer:
    - Purpose: Applied after both the embedding and LSTM layers to regularize the model and prevent overfitting.
    - Implementation: nn.Dropout(dropout_rate) where dropout_rate is the probability of setting some weights to zero during training.
4. Fully Connected (FC) Layer:
    - Purpose: The LSTM output is passed through a fully connected layer to produce a prediction for the next word in the sequence. This output has a shape matching the vocabulary size, as the model is trying to predict the probability of each word in the vocabulary being the next word.
    - Implementation: nn.Linear(hid_dim, vocab_size) where hid_dim is the number of hidden units and vocab_size is the size of the vocabulary.
5. Weight Initialization:
    - The model uses a custom weight initialization strategy for better convergence. The embedding layer's weights are initialized within a specific range, while the weights in the LSTM and fully connected layers are also initialized uniformly.

Training Process:

The training process involves the following key steps:

1. Model Setup:
    - The model is created with the specified vocabulary size, embedding dimension, hidden layer dimension, number of LSTM layers, and dropout rate.
    - The optimizer used is Adam with a learning rate (lr = 1e-3), and the loss function used is CrossEntropyLoss, which is suitable for classification tasks (predicting the next word from the vocabulary).

2. Batch Creation:

    - The dataset is split into batches. Each batch contains a sequence of tokens (word indices), and the target for each token is the next token in the sequence (the "next word" prediction).
    - A function get_batch is used to extract sequences of tokens and their corresponding target tokens.

3. Training Loop:

    - The training loop iterates over the entire dataset for a predefined number of epochs (n_epochs = 50).
    - For each batch:
        - The optimizer is reset (optimizer.zero_grad()).
        - The hidden states of the LSTM are detached from the previous batch to prevent backpropagating through the entire history of the model's computations.
        - The model's predictions are computed for the input sequence.
        - The loss is calculated by comparing the predicted next word with the actual next word (target).
        - The loss is backpropagated, and the optimizer updates the model's weights accordingly.
        - Gradient clipping (torch.nn.utils.clip_grad_norm_) is used to prevent exploding gradients by limiting the size of the gradients during backpropagation.

4. Validation:

    - After each epoch of training, the model is evaluated on a validation dataset to check its performance. The validation loss is calculated similarly to the training loss.
    - If the validation loss improves, the model's parameters are saved (torch.save(model.state_dict(), 'best-val-lstm_lm.pt')).

5. Learning Rate Adjustment:

    - A ReduceLROnPlateau scheduler is used to decrease the learning rate if the validation loss doesn't improve after a certain number of epochs. This helps in fine-tuning the model and can lead to better convergence.

6. Perplexity Calculation:

    - Perplexity is used as a measure of the model's performance. It is calculated by exponentiating the loss (e.g., math.exp(train_loss)). Lower perplexity indicates better performance.

Task 3. Text Generation - Web Application Development - Develop a simple web application that demonstrates the capabilities of your language model.

1) The application should include an input box where users can type in a text prompt.

2) Based on the input, the model should generate and display a continuation of the text. For example, if the input is " Harry Potter is" , the model might generate " a wizard in the world of Hogwarts.

3) Provide documentation on how the web application interfaces with the language model.

Web application

The web application allows users to interact with a trained LSTM-based language model to generate text based on a user-provided prompt. Users can input a starting text (a "prompt") and specify a "temperature" value to control the creativity or randomness of the generated text. The model responds by generating a sequence of text that extends or completes the input prompt.