Research Project Overview: Developing a 2D Contextual Transformer for Spreadsheet Analysis

Concept and Motivation

1. **Inspiration from Transformers**:
    - Transformers process sequences in parallel and bidirectionally, making them faster and more efficient than traditional RNNs. However, they primarily work in a 1D structure, suited for language tasks (left-to-right or right-to-left processing).
    - Spreadsheets present a unique **2D contextual structure**: each cell's content depends on the context provided by both rows and columns. I hypothesized that building a model with 2D contextual understanding would enhance spreadsheet comprehension, particularly for complex tasks, while also improving training efficiency compared to traditional language models.

Task Definition and Initial Scope

2. **Defining the Supervised Task and Vision for Financial Consultancy**:
    - **Long-Term Vision**: The goal is to develop an AI model that can analyze various types of spreadsheets similarly to a financial consultant. The envisioned model would take in multiple types of financial sheets (e.g., balance sheets, P&L statements, expense reports, inventory lists) and generate insights on a company's financial health. For example:
        - Given a company's financial spreadsheets, the model could analyze the P&L statement and balance sheet, providing comments on profitability, liquidity, or debt levels. It could then reference specific cells or rows in expense or inventory sheets to justify its insights, effectively acting as a comprehensive, spreadsheet-based financial consultant.
    - **Initial Scope**: To start, we focused on a simpler task—predicting **metadata** (e.g., bold, italic, underline, alignment, font size) for each cell based on its content. This framed the problem as a **supervised learning task** with 17 metadata types.
    - **Current Task**: For initial experimentation, the task was simplified to a binary classification of **boldness** (predicting if a cell is bold: 1 or non-bold: 0).

Dataset Curation and Parsing

3. **Dataset Expansion Using a Custom Web Parser**:
    - **Initial Dataset**: We began with the **Enron dataset**, containing 622 spreadsheets in .xls, .xlsx, and .csv formats. However, many files were outdated, lacked sufficient formatting metadata, or caused parsing errors, limiting their utility for training.
    - **Custom Web Parser Development**: To expand the dataset, I built a parser using **BeautifulSoup** and **asyncio** with async and ClientSession() to scrape .xls and .xlsx files from **data.gov**. This parser enabled asynchronous downloads with **5 concurrent requests**, significantly increasing download speed.
    - **Validation Criteria**: Files were validated based on type (.xls and .xlsx only, excluding .csv), response time (skipping URLs with response times >1 second), size (limiting files to under 2MB), and parsing compatibility. Files were parsed immediately post-download, and any incompatible files were removed to ensure consistency in the dataset. The final dataset consisted of **800 training files, 100 validation files, and 100 test files**, with subsets (teeny, micro, tiny, small, medium, big) for structured experimentation.

Data Representation and Preprocessing

4. **Building Vocabulary and Word Embeddings**:
    - Developed a custom vocabulary class to tokenize spreadsheet content, incorporating special tokens (<cls>, <eos>, <unk>, and <pad>) for sequence management. We then created embeddings for each token using the **GloVe-50** model, producing a **50-dimensional vector** for each token.
5. **Data Structuring for Model Input**:
    - Each spreadsheet was structured as a **PyTorch tensor**. For computational feasibility, each spreadsheet was limited to **100 rows by 100 columns**.
    - **Tokenizing Cell Content**: Cell content was tokenized with a custom vanilla tokenizer, padded/truncated to 32 tokens. The resulting tensor, x_tok, represented each spreadsheet's content as a **100x100x32 tensor**.
    - **Metadata Extraction**: Metadata for each cell was extracted into a **100x100x17 tensor** (y_tok), with 17 types of metadata (e.g., boldness at position 6, where 1 = bold, 0 = non-bold).
6. **Parsing and Preprocessing Pipelines**:

- We utilized **pandas, numpy, openpyxl, xlrd**, and **csv** libraries to parse each spreadsheet. Each file's content and metadata were extracted into x_tok and y_tok tensors, and any files that couldn't be parsed were automatically excluded.

7. **Batch Loading and Parallel Processing**:
   - A custom dataloader, SpreadsheetDataLoader, was developed to handle multiple x_tok and y_tok tensors along with file paths. For faster loading, parallel processing was enabled using the **Parallel** and **joblib** libraries, with os.cpu_count() // 4 as the number of jobs for optimal CPU utilization.

Model Implementation and Training

8. **Approach to Model Architecture**:
   - **Approach A**: Each cell's content was processed with context from all surrounding cells (global context).
   - **Approach B**: We focused on **row and column context** only, aligning with the hypothesis that a cell's content is primarily influenced by values within its row and column.
   - The TestRNN model explained below was implemented based on **Approach A**.
9. **Unified Step-by-Step Algorithm for Forward Pass (Including Global and Local Context)**:
   - **Example Scenario**: Let's consider a batch size of 8, where each spreadsheet is represented by a 8 x 100 x 100 x 32 tensor. We assume a vocabulary of 50,000 unique tokens, each mapped to a 50-dimensional vector, resulting in an embedding matrix of 50000 x 50.
   - **Forward Pass Algorithm**:
     -
       1. **Obtaining Global Context for Each Cell**:
       - a. To begin, we need to calculate the global context for each cell, which involves setting up a tensor H_local with dimensions batch x cells x hidden_dim = 8 x 10000 x 100. This tensor will store a 100-dimensional hidden state for each cell, capturing its overall context.

- b. We loop over each cell in the 8 x 100 x 100 structure using a 1D indexing approach, making it easier to iterate over each cell individually.
    - i. Inside this loop, we retrieve the 32 tokens for the current cell across all batches, resulting in a 8 x 32 tensor. This tensor represents the cell's content across each batch.
    - ii. We pass this tensor through an embedding layer, which converts each token to a 50-dimensional vector, producing a 8 x 32 x 50 tensor.
    - iii. After applying dropout (rate 0.05), we pass this tensor through the RNN layer, which outputs z (hidden state for each token) and h (final hidden state for each layer).
    - iv. Since we are focusing on capturing the overall context of the cell, we select h, which is stored at position [1] in the output. From h, we extract the hidden state from the last RNN layer using h[-1], giving a 8 x 100 tensor representing the final hidden state across batches.
    - v. We store this hidden state in H_local, ensuring each cell location contains a global summary.
- c. Once we have the hidden states for all cells in H_local, we calculate the **total context** by summing hidden states along the cell dimension, resulting in a 8 x 100 tensor.
- d. To get the global context for each cell excluding itself, we subtract each cell's hidden state from this total context, creating a 8 x 10000 x 100 tensor representing the global context for each cell.
- e. This final tensor is returned from cell_hs, containing global context vectors for each cell across all batches.

-

2. **Preparing to Compute Predictions**:

- With the global context tensor (H_global) ready, we define S_cube, a tensor of shape batch x rows x cols = 8 x 100 x 100, initialized with zeros. This will store the logits for each cell after combining global and local contexts.

-

3. **Loop to Combine Global and Local Context for Each Cell**:

- We then iterate over each cell using 1D indexing to combine the global and local contexts.
    - i. **Calculating Local Context**: For each cell, we repeat the embedding, dropout, and RNN steps to generate z and h. This time, we select z (position [0] in the output), which gives us the hidden state for each token in the cell's sequence.
        - To focus on the final token's context, we select z[:, -1, :], resulting in a 8 x 100 tensor representing the last token's hidden state, or local context.
    - ii. In parallel, we retrieve the global context for the cell from H_global, which is also a 8 x 100 tensor.
    - iii. We concatenate the local and global context tensors along the hidden dimension to form a 8 x 200 tensor, capturing both inside and outside cell information.
    - iv. Passing this tensor through the linear layer (_pred) yields logits, reshaped to 8 x 1 and then flattened to 8, for easy placement in S_cube.
    - v. Store the logits for the current cell in S_cube.

-

4. **Final Output**:

- After looping through all cells, S_cube contains logits for each cell in a 8 x 100 x 100 structure. This tensor is returned, representing the predictions for each cell across batches.

10. **Training Loop and Loss Calculation**:

The training process for the model follows a structured loop aimed at maximizing performance while minimizing overfitting. A key consideration was the **computational constraints** associated with the current model architecture, which significantly impacted training time. Even with access to a 40GB GPU, each epoch took around 13 minutes to train on 800 files with a batch size of 32. Consequently, each hyperparameter adjustment affected the overall training time, posing challenges for rapid experimentation and iteration. This section provides a detailed, step-by-step explanation of the training process, with specific attention to handling the class imbalance within our dataset.

- **Data Loader and Initial Configuration**:
    - We begin by setting up data loaders for both the training and validation datasets, where the training loader shuffles data randomly, and the validation loader maintains a fixed order. The batch_size, learning rate (lr), and DEVICE for training are defined here. To handle the substantial imbalance in bold versus non-bold cells (with non-bold cells vastly outnumbering bold cells), we calculate a positive weight (pos_weight). This weight is passed to the **BCEWithLogitsLoss** function to penalize the model more heavily for misclassifying bold cells, thus improving its sensitivity to the minority class. Specifically, pos_weight is set as the ratio of non-bold cells to bold cells across the training dataset.
- **Loop Configuration and Main Training Loop**:
    - The training loop begins with key hyperparameters such as the number of epochs (max_epochs), learning rate (lr), patience for early stopping, and intermediate save intervals (save_int). If intermediate saving is enabled (save_int > 0), the model's state is saved at specified intervals for later reference and analysis. Additionally, we initialize variables to track the best model performance (best_avgtrloss, best_valperp, etc.) and nimp_ctr, which counts the consecutive epochs with no improvement in validation loss. The early stopping criterion (patience) stops the training if the model doesn't improve over a specified number of epochs. The main training loop iterates over epochs until reaching max_epochs or triggering early stopping. At the start of each epoch, training mode is enabled for the model, and metrics are reset (curr_trloss and curr_valloss) to accumulate the losses for the current epoch.
- **Training Phase**:

  - In the training phase, each batch from the training loader is processed. The model performs a forward pass on the input tensor x_tok, which represents cell content. The logits output is compared with the target tensor y_tok (metadata for each cell) using **BCEWithLogitsLoss**. This function takes into account the previously defined pos_weight, which penalizes the model more heavily for misclassifying bold cells, balancing the loss across classes. The computed loss is accumulated to curr_trloss, which tracks the total training loss for the epoch. To update model weights, we perform backpropagation by calling loss.backward(). The clip_grad_norm_ function is applied to prevent gradient explosion, limiting gradients to a maximum value defined by mu. The optimizer (Adagrad in this case) then updates the model parameters, and the loss variable is cleared from memory after each batch to optimize computational resources.

- **Validation Phase**:
  - After training on all batches in an epoch, the model is switched to evaluation mode. Each batch from the validation loader is processed without backpropagation (using torch.no_grad()), and the validation loss is calculated in the same way as the training loss. The validation loss for each batch is accumulated into curr_valloss to track the total validation loss over the epoch. After calculating the average training and validation losses for the epoch, we compute the model's **perplexity** for both training and validation datasets. The best model is updated if the current validation perplexity is lower than the previous best. If not, nimp_ctr is incremented. Once nimp_ctr exceeds patience, the model triggers early stopping, halting further training to prevent overfitting.

- **Logging and Model Saving**:
  - After each epoch, the training and validation losses and perplexities are logged. If save_int > 0 and the current epoch is a multiple of save_int, the model's state is saved to the specified directory. At the end of training (or upon early stopping), a final message logs the best epoch, along with the corresponding training and validation losses and perplexities. If save_int > 0, the final model is saved in the designated directory.

- **Return**:

    - Once training is complete, the function returns the best-performing model, which has the lowest validation perplexity. This final model can be further evaluated on test data or used in downstream applications.

11. **Evaluation of Model Performance:**

- After training, the model's performance was assessed using two inference functions to evaluate metrics such as accuracy, precision, recall, F1-score, along with the non-bold to bold cell ratio and confusion matrix.
- **Individual File Evaluation:**
    - The first inference function takes a single file from the dataloader, passes it through the trained model, and retrieves the predicted bold/non-bold classification for each cell in the 100x100 grid. The initial output from the model consists of logits for each cell's bold/not-bold prediction. These logits are passed through a sigmoid function to obtain probabilities for all 100x100 cells. Using a threshold of 0.5, each cell's probability is converted into a binary prediction: cells with a probability greater than 0.5 are classified as bold (1), while those with a probability of 0.5 or less are classified as non-bold (0). This final predicted grid is compared to the actual grid to compute accuracy, precision, recall, and F1-score, averaged across all cells in the file. Additionally, the function provides the ratio of non-bold to bold cells in both the predicted and actual grids, and displays a confusion matrix showing true/false positives and negatives for the file.
- **Batch Evaluation Across All Files:**
    - The second function performs batch evaluation over the entire dataset or a specified subset (up to 200 files). It processes each file in the dataloader, applies the same sigmoid thresholding to convert logits into binary predictions, and aggregates the predicted metrics across all sheets. This function calculates the average performance metrics across all files, reports the overall non-bold to bold ratio, and presents a cumulative confusion matrix representing the aggregated results.

12. **Discussion of Results**

Due to lengthy training times, we initially trained the RNN model (Approach A) on a smaller micro dataset (50-6-6 train-val-test split) with 6 epochs, a patience of 2, batch size of 32, mu of 0.25, and a learning rate of 5.6e-5. Evaluations across all loaders (train, validation, and test) consistently showed poor results, with close to 0% accuracy, precision, recall, and F1-score. Confusion matrices revealed that the model predicted every cell as bold, resulting in only true and false positives (TP/FP) and no true or false negatives (TN/FN). This "predict-everything-as-bold" approach likely stemmed from the limited dataset size, low epoch count, and insufficient handling of class imbalance, alongside a simple model architecture, leading to minimal learning and poor generalization. However, when trained on a larger dataset of 800 files for 20 epochs, the model began recognizing basic patterns: it started predicting the entire rectangular area containing text as bold, regardless of whether individual cells within that area were empty. Although still oversimplified, this behavior suggests that with more data and training, the model may be beginning to identify areas of content as likely bold. This indicates the potential for improvement with further data and tuning. Notably, the cross model from Approach B performed even worse.

13. **Current Work and Next Steps**

To address the limitations observed in bold cell prediction, we are exploring two primary approaches: simplifying model architecture and utilizing an advanced model developed by my professor, the SAFFU model.

1. Simpler Model Architectures:
Based on my professor's feedback that the RNN model might be overly complex and inefficient to train, we are experimenting with simpler models to establish a strong baseline that outperforms random prediction. These simplified approaches include:

- Summing token embeddings within each cell and passing them directly through a feed-forward layer for bold/not-bold classification.
- Averaging token embeddings for each cell before feeding them into a linear layer for prediction.
- Testing basic models like Bag of Words to see if they yield effective results with reduced training time and complexity.

These simpler models aim to be less computationally demanding, allowing for quicker iterations and more efficient training to determine if they can handle basic distinctions in cell boldness.

2. Using the SAFFU Model:
We are also adapting the Self-Attentive Feed-Forward Unit (SAFFU) model, designed by my professor, to our task. SAFFU introduces efficiency by computing explicit solutions in the feed-forward layers, reducing computational cost and potentially enhancing model generalization. The model leverages a custom byte-pair encoding (BPE) tokenizer, which significantly minimizes vocabulary size, further aiding in training efficiency. Unlike the RNN approach, the SAFFU model aims to capture context through optimized self-attention mechanisms rather than recurrent connections, potentially providing a more refined context for each cell.

Since SAFFU is optimized for quicker and more stable training with smaller datasets, the hope is that this model will offer more precise predictions with reduced complexity, making it a promising candidate for the bold cell prediction task. Given access to its code and the BPE tokenizer, adapting SAFFU to our needs could bring significant improvements in training speed and predictive accuracy.