Tweet Sentiment using RNN and Transformer

Team Members:

1. Khushal Mandavia kqm5921@psu.edu
2. Harsh Ladani hal5240@psu.edu

1. Introduction

Sentiment analysis, a pivotal aspect of natural language processing (NLP), involves the computational identification and categorization of opinions expressed in a piece of text.

This project aims to leverage the Sentiment140 dataset, a rich repository of annotated tweets from a Kaggle competition, to understand public sentiment dynamics. The dataset's comprehensive nature, encompassing a wide array of sentiments, makes it ideal for this study. The sentiment could either be a negative one or a positive one.

2. Task & Dataset & Preprocessing

The main task for our Final Project was to identify the sentiment of a certain text and we decided to do so for tweets. The models can identify whether a tweet is tweeted with a positive or a negative sentiment.

The Sentiment140 dataset comprises tweets annotated for sentiment, offering a real-world glimpse into public opinion. There are 1.6 million data points in the dataset. The dataset contains the following columns:

1. **target**: the polarity of the tweet (*0* = negative, *4* = positive)
2. **ids**: The id of the tweet (*2087*)
3. **date**: the date of the tweet (*Sat May 16 23:58:44 UTC 2009*)
4. **flag**: The query (*lyx*). If there is no query, then this value is NO_QUERY.
5. **user**: the user that tweeted (*robotickilldozr*)
6. **text**: the text of the tweet (*Lyx is cool*)

Feature Engineering

Data preprocessing is a critical step, encompassing tokenization, stemming, lemmatization, and the removal of stop words. These steps refine the dataset, enhancing the model's ability to learn from textual nuances. For this particular dataset since the tweets could have hashtags, @'s and URL links which do not assist in showing any sentiment it was important to remove either those tags, @'s or any links composed in the tweet so that the model only has pure English to judge the sentiment from. Since, the columns are either have a 0 or a 4, 0 being negative and 4 being positive and we only wanted a binary dependent variable we turned each 4 into a 1 so that we only have 0's and 1's.

Also, from the given columns not all columns are required or important for the model to be accurate. The two most important columns are the target column and the text column. The text column is the tweet and is the independent column while the target columns is the dependent column associated with the sentiment of the tweet.

The dataset was also not quite random since the first half were all tweets with negative sentiments while the other half was all negative sentiments. Thus it was important to randomize the dataset so that it leaves no biases for the model.

Data size Reduction

The Transformer model turned out to be quite computational heavy to process 1.6 million datapoints and thus we had to reduce the size of our data from 1.6 million to just 2000 for the transformer model

Train, Test and Validation Split

We split our dataset into train, validation and test sets. 80% of the overall data was split for training purposes and 20% for testing. Out of the 80% of training data further 20% was split as validation set.

Dataset and Data loaders

For both the dataset we converted the pandas data frame to a dataset supported by pytorch, the rnn_twitter_dataset.py for the RNN model, the bert_twitter_dataset.py for the transformer model. In both the dataset the tweets are tokenized from the bert-base-uncased tokenizer from hugging face and the dataset returns the tweet_text, input_ids, attention_mask, length and the labels.

Then, for both the models we convert the dataset into batches using the DataLoader from pytorch. For the RNN model we have a larger batch size of 16000 since RNN's are not parallelized as Transformers and do not require larger memory and for the Transformer model the batch size was only 900 since a larger batch size would've required a larger memory which made the kernel crash in our case and thus 900 was the optimum size.

3. Deep Learning Systems Implementation & Architectures

RNN LSTM Model

For the RNN model we have used the Long Short-Term Memory model (LSTM).

The LSTM model boasts a multi-layered architecture, with each layer fine-tuned to capture temporal relationships in textual data.

The choice of LSTM stems from its proven efficacy in handling long-term dependencies, a common challenge in text-based datasets.

The RNN LSTM model we've designed is tailored for sentiment analysis, a task crucial in the field of NLP. This model is structured to effectively process sequential text data, learning to capture the nuances and context essential for accurately gauging sentiment.

Key Components of the Model

1. Embedding Layer (nn.Embedding):
    - This layer is the entry point for our input data. It transforms each token in the vocabulary into a high-dimensional space, where semantically similar words are represented closely.
    - Key Parameters:
        - vocab_size: Total number of unique tokens in our vocabulary.
        - embedding_dim: The size of each embedding vector.
2. LSTM Layer (nn.LSTM):
    - The core of our model, LSTM (Long Short-Term Memory) layers are adept at handling sequences with varying lengths and learning long-term dependencies.
    - Key Features:
        - Handles embedding_dim as input size.
        - hidden_dim: Defines the size of the LSTM's hidden state.
        - num_layers: The count of LSTM layers stacked together.
        - Optional bidirectional setting, which we've set to False.
        - Includes dropout for regularization.

        - batch_first=True ensures batch size is the first dimension of input and output.

3. Linear and Dropout Layers (nn.Linear, nn.Dropout):
    - After processing the data through LSTM layers, the model employs a linear layer for final output generation and a dropout layer for regularization.
    - Linear Layer:
        - Transforms the LSTM output to the desired output dimension.
    - Dropout Layer:
        - Helps in reducing overfitting by randomly zeroing parts of the input with a probability defined by the dropout parameter.

Forward Method Implementation

In the forward method, the data flow through the model is defined. It involves sorting the sequences, packing, and processing them through the LSTM layer. The hidden state obtained is then passed through dropout and linear layers to produce the final output.

Hyperparameters and Their Significance

- Hyperparameter Tuning:
    - The model's performance largely hinges on the optimal selection of hyperparameters.
    - Important Hyperparameters:
        - vocab_size: Derived from the tokenizer's vocabulary.
        - embedding_dim: Set at 12, influencing the embedding layer's output.
        - hidden_dim: The size of LSTM hidden layers, set at 8.
        - output_dim: Typically 1 for binary classification tasks.
        - n_layers: The number of LSTM layers, which is 2 in our model.
        - birectional: Determines if the LSTM is bidirectional, false in our case
        - drouput: Set at 0.5, controlling the dropout layer's effect.

Model Instantiation and Training

The model, once instantiated with these parameters, is ready for training. The effectiveness of training will depend on how well these parameters are tuned to our specific dataset and task requirements.

In conclusion, our RNN LSTM model embodies a well-thought-out architecture suitable for sentiment analysis, leveraging the strengths of LSTM for sequential data processing. With appropriate hyperparameter tuning and training, it should be adept at capturing the subtleties in sentiment expressed in tweets.

BERT Transformer Model

BERT's transformative architecture, featuring multiple layers of attention mechanisms, excels in contextual understanding.

Its selection for this project is backed by its prowess in capturing nuanced meanings in complex sentence structures.

Our model, BertForSentimentAnalysis, harnesses the power of the DistilBERT model, a streamlined version of the original BERT (Bidirectional Encoder Representations from Transformers). DistilBERT maintains most of BERT's performance while being lighter and faster.

Key Components of the Model

1. BERT Layer (DistilBertModel.from_pretrained):
    - The backbone of our model, this pre-trained BERT layer, processes the input data.
    - Key Features:
        - Initialized with weights from 'distilbert-base-uncased', a pre-trained model.
        - Option to freeze BERT layers, preventing them from updating during training.
2. Classification Layers:
    - After processing by BERT, the output is fed into a series of classification layers to determine sentiment.
    - Structure:
        - A linear layer reduces the dimension from BERT's output size to an intermediary size (8 in our case).

        - Batch normalization and LeakyReLU activation are applied for stabilization and non-linearity.
        - A dropout layer for regularization, followed by another linear layer to map to the final output dimension (1 for binary classification).

Forward Method Implementation

In the forward method:

- Input data is passed through the BERT layer with an attention mask.
- The last hidden state from BERT's output is extracted.
- This output is then passed through the sequential classification layers to produce the final logits.

Hyperparameters and Training Details

- Freezing BERT Layers:
    - This is an optional step but can significantly speed up training and is useful when fine-tuning on smaller datasets.
    - When layers are frozen, their weights are not updated during backpropagation.
- Classifier Parameters:
    - The classifier uses a smaller dimension (8) to bridge the gap between BERT's high-dimensional output and the binary classification task.
    - The combination of batch normalization, LeakyReLU, and dropout helps in regularizing the model and ensuring a stable learning process.

Model Instantiation and Usage

In machine learning models utilizing BERT (Bidirectional Encoder Representations from Transformers), the freeze_bert=True setting is crucial. It locks the pre-trained BERT layers, allowing the focus to shift entirely to optimizing the new classification layers. This approach ensures that during the forward pass, BERT's initial processing of inputs forms the basis for subsequent sentiment classification. By freezing the pre-trained layers, the model efficiently harnesses the robust foundational knowledge of BERT, while the trainable classification layers are fine-tuned to specific sentiment analysis tasks, achieving a balance between leveraging pre-existing strengths and adapting to new challenges.

4. Training Details

Each model was meticulously trained, with hyperparameters like batch size, learning rate, and epoch count finely adjusted for optimal performance.

Training challenges, such as managing computational resources and preventing overfitting, were systematically addressed.

RNN LSTM Training

The training process is divided into two main functions: train for training the model and evaluate for validating its performance. Both functions iterate over batches of data, but train updates the model weights while evaluate does not, ensuring the validation process does not affect the model state.

The ‘train’ Function

- Model State: Sets the model to training mode, enabling dropout layers and batch normalization layers to work in train mode.
- Batch Processing: Iterates over each batch in the data_loader.
    - Data Preparation: Extracts input IDs, attention masks, lengths, and labels from each batch.
    - Prediction: Passes the data through the model to get predictions.
    - Loss Computation: Calculates the loss using criterion, which is nn.BCEWithLogitsLoss in this case, suitable for binary classification tasks.
    - Accuracy Calculation: Computes the binary accuracy of the predictions.
    - Backpropagation: Performs a backward pass to calculate gradients.
    - Optimizer Step: Updates the model parameters based on the calculated gradients.
- Epoch Performance: Aggregates the loss and accuracy for each epoch.

The ‘evaluate’ Function

- Model State: Sets the model to evaluation mode, which turns off dropout and batch normalization layers.
- Batch Processing: Similar to train but wrapped in torch.no_grad() to prevent gradient calculations, which saves memory and computations.

- Epoch Performance: Computes and returns the average loss and accuracy over the validation dataset.

Binary Accuracy Function

- Accuracy Calculation: Converts the model's logits to binary predictions (0 or 1) using a sigmoid function and rounding. The accuracy is the proportion of correct predictions.

Optimizer and Criterion

- Optimizer: Adam optimizer with a learning rate of 0.5. Adam is a popular choice due to its efficiency and adaptive learning rate features.
- Loss Function: BCEWithLogitsLoss combines a sigmoid layer and the binary cross-entropy loss in one single class, which is more numerically stable than using a plain Sigmoid followed by a BCELoss.

Training Loop

- Epochs: The model is trained for a total of 5 epochs. Each epoch involves a full pass over the training dataset and a subsequent pass over the validation dataset.
- Training and Validation: In each epoch, the model is trained on the training dataset and evaluated on the validation dataset, allowing us to monitor the model's performance and check for overfitting.

Conclusion

Our training setup was meant to be robust, incorporating essential aspects such as loss calculation, accuracy metrics, and optimization steps. The training and evaluation functions are well-designed to provide a clear picture of the model's performance as it learns from the data. The choice of loss function, optimizer, and binary accuracy calculation align well with the demands of binary sentiment classification tasks.

BERT Transformer Training

The training routine is structured to optimize the BERT model for sentiment analysis, with a focus on efficient gradient accumulation and accuracy measurement.

The ‘train’ Function

- Model State: Sets the model to training mode, enabling specific layers like dropout to function in the training context.
- Gradient Accumulation:

    - Implemented to accumulate gradients over several forward passes, enhancing training efficiency, especially when working with limited memory resources.
    - Controlled by accumulation_steps, set to 4 in this case.
- Batch Processing:
    - Data Preparation: Retrieves input IDs, attention masks, and labels from the data loader.
    - Loss Calculation: Scales the loss by the accumulation steps, which is then accumulated over multiple batches.
    - Optimizer Step: The optimizer updates the model parameters after accumulating gradients over the specified number of steps.
- Accuracy Computation: Uses the binary_accuracy function to calculate the accuracy for each batch.
- Epoch Performance: Calculates the average loss and accuracy across all batches in the data loader.

The ‘evaluate’ Function

- Model State: Sets the model to evaluation mode, disabling layers like dropout.
- No Gradient Tracking: Utilizes torch.no_grad() to prevent gradient computation, optimizing memory usage and computation.
- Batch Processing:
    - Similar to the train function, but without the gradient accumulation and optimizer steps.
    - Computes loss and accuracy for each batch.
- Epoch Performance: Aggregates the total loss and accuracy over the entire validation dataset.

Binary Accuracy Function

- Calculates accuracy by comparing the rounded sigmoid predictions with the actual labels, a standard approach for binary classification tasks.

Optimizer and Loss Function

- Optimizer: Uses the Adam optimizer, known for its efficiency and adaptive learning rates, with a learning rate of 0.5.

- Loss Function: Employs BCEWithLogitsLoss, which combines a sigmoid layer with binary cross-entropy loss, ideal for binary classification.

Training Loop

- Epochs: The model is trained for 2 epochs, with each epoch involving training on the entire dataset and then evaluating on the validation dataset.
- Monitoring Training and Validation Metrics: Keeps track of training and validation loss and accuracy, allowing for monitoring of the model's performance and generalization capability.

Conclusion

Our training setup for the BERT Transformer model is well-crafted, with gradient accumulation enhancing training efficiency and binary accuracy providing a clear metric for performance evaluation. The use of BCEWithLogitsLoss and the Adam optimizer is well-aligned with the demands of the task. This approach should facilitate effective training of the BERT model for sentiment analysis, particularly in environments with computational constraints.

5. Results, Observations, and Conclusions

Our primary metric for assessing performance was binary accuracy, calculated using a custom function. This approach provided a clear and direct measure of each model's ability to correctly classify sentiments.

Binary Accuracy Function:

The binary_accuracy function plays a crucial role in evaluating the models. It converts the raw prediction logits to binary outcomes (0 or 1) using a sigmoid function and rounding off. The accuracy is then determined by comparing these predictions to the actual labels.

RNN LSTM Model Results

- Initial Observations: The RNN model initially required approximately 45 minutes to complete 5 epochs of training. This duration highlighted the need for optimization.
- Hyperparameter Adjustment: By carefully adjusting the hyperparameters—some reduced, others increased—we struck an optimal balance between accuracy and training time.

- Final Performance: Post optimization, the RNN model's training time was significantly reduced to about 10 minutes, achieving a test accuracy of **50.11%.** This outcome demonstrates the effectiveness of hyperparameter tuning in enhancing both performance and efficiency.

BERT Transformer Model Results

- Computational Challenges: The BERT model, while slightly more accurate, presented substantial computational demands. Its initial setup required extensive processing time, making it impractical for our available resources.

- Data and Batch Size Reduction: To manage these demands, we reduced the dataset size from 1.6 million to 2,000 entries and the batch size to 900. This adjustment was necessary due to the Transformer's parallelized nature, which consumes considerable memory.

- Post-Adjustment Observations: Despite reducing the dataset size and batch size, the BERT model still took close to an hour for training. It achieved a test accuracy of **50.55%**, marginally higher than the RNN model.

- Hardware Limitations: We noted that the BERT model's performance could potentially be improved with access to accelerated GPU hardware. This would allow for processing larger datasets and batch sizes, likely enhancing accuracy.

Conclusion and Learnings

- Trade-Offs in Model Selection: Our experience underscores the trade-offs between model complexity and computational efficiency. While the BERT model showed a slight edge in accuracy, its heavy computational demands limited its practicality in our setup.

- Hyperparameter Tuning: The significant improvement in the RNN model's training time post hyperparameter optimization highlights the critical role of tuning in machine learning.

- Hardware Considerations: This project also shed light on the importance of hardware in training deep learning models. Accelerated hardware, such as GPUs, can be pivotal in leveraging the full potential of more complex models like BERT.

In summary, our exploration into sentiment analysis using RNN LSTM and BERT Transformer models provided valuable insights into the balance between accuracy, computational efficiency, and the importance of hardware capabilities in deep learning tasks.

6. Challenges and Solutions

In our sentiment analysis project, the training and debugging of the RNN LSTM and BERT Transformer models presented significant challenges. The RNN LSTM's training was time-consuming, hindering swift debugging and adjustments. However, the greater challenge was with the BERT Transformer model. Its high accuracy was offset by substantial computational demands, often leading to kernel failures. To manage this, we reduced the dataset size dramatically from 1.6 million to 2,000 entries and the batch size from 16,000 to 900. This scaling down, while necessary, made the hyperparameter tuning process arduous and time-intensive. The experience underscored the complexities of balancing model accuracy with computational efficiency in deep learning.

Solutions Implemented

1. PyTorch Profiling:
    - We employed PyTorch Profiling to identify bottlenecks in the model training process.
    - This tool helped us pinpoint parts of the model that were particularly time-consuming, allowing us to understand and address the inefficiencies.
2. Hyperparameter Optimization and Data Management:
    - We experimented with and optimized various hyperparameters to find a balance between computational efficiency and model accuracy.
    - Adjusting the dataset and batch size for the Transformer model was a crucial step in managing its computational demands.
3. Gradient Accumulation for the Transformer Model:
    - To further optimize the training process of the Transformer model, we implemented gradient accumulation.

    - This technique allowed us to effectively train the model with a smaller batch size by accumulating gradients over several forward passes before performing a backpropagation step.

7. Conclusion

In conclusion, our sentiment analysis project, leveraging both RNN LSTM and BERT Transformer models, provided valuable insights into the practicalities of implementing deep learning techniques for natural language processing. While the RNN LSTM model demonstrated reasonable performance with manageable computational demands, the BERT Transformer model, despite its slightly higher accuracy, posed significant computational challenges. The project highlighted the importance of careful hyperparameter tuning, efficient data management, and the necessity of balancing computational resources with model performance. This experience not only deepened our understanding of the intricacies involved in training sophisticated NLP models but also underscored the trade-offs that must be considered when selecting and optimizing models for real-world applications.

8. References

Citation: Go, A., Bhayani, R. and Huang, L., 2009. Twitter sentiment classification using distant supervision. *CS224N Project Report, Stanford, 1(2009), p.12*.