BBM409 : Introduction to Machine Learning Lab.
Fall 2024
Instructor: Assoc. Prof. Dr. Hacer Yalım Keleş
TA: Sümeyye Meryem Taşyürek

Assignement 4: LSTM for Sentiment Analysis

Due date: Friday, 27-12-2024, 11:59 PM.

The objective of this assignment is to cultivate your comprehension and familiarity with Recurrent Neural Networks (RNN) concept with a focus on Long Short-Term Memory (LSTM) networks and their application in sequence-based tasks, specifically sentiment analysis of text data.

1 Implementation of a LSTM Network

In this section of the assignment, you will implement an LSTM-based model from scratch for the task of sentiment analysis using the deep learning framework **PyTorch**.
Your objective is to construct an LSTM architecture using fundamental components such as embedding layers, LSTM layers, and fully connected layers for binary classification. You will train this model on the provided subset of **IMDB Movie Reviews Dataset**, a benchmark dataset for sentiment analysis which contains 10,000 labeled reviews. Reviews are labeled as either positive or negative, with a balanced class distribution. You will split this dataset into training, validation and testing subsets.

Recurrent Neural Networks (RNNs) are inherently designed for sequential data, making them suitable for tasks like sentiment analysis. RNNs are often used to process language, for example, to map from one sequence of words to another sequence of words. However, RNNs take vectors as input, so to get them to process words, we need some way of turning words into numerical vectors. One way to do this is with a one-hot encoding, where each word is represented as a high-dimensional sparse vector. However, this approach can be inefficient and lacks semantic information about the relationships between words.

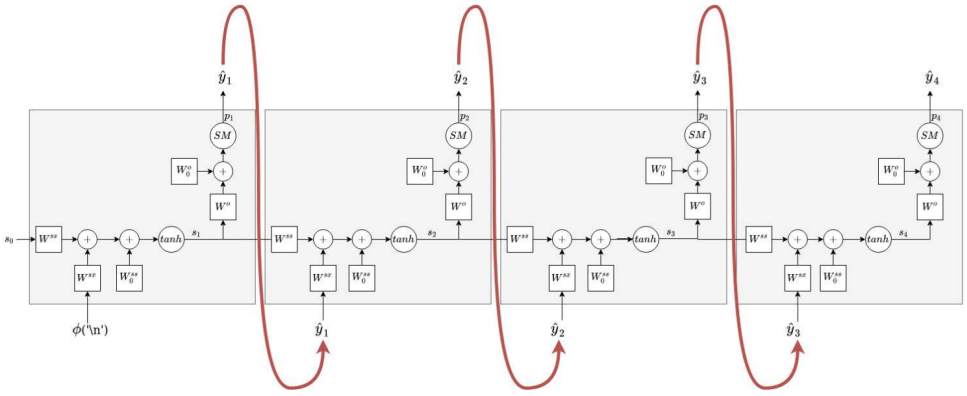

Figure 1: A simple RNN architecture

Instead, more informative and lower-dimensional representations of words, called **word embeddings**, are commonly used. Word embeddings assign each word a vector such that words appearing in similar sentence contexts have embeddings close in vector space. One popular technique for producing word embeddings is called **Word2Vec**. Word2Vec uses a machine learning approach to learn an embedding weight matrix of size [number of words x embedding size]. This matrix is trained such that the embeddings capture semantic relationships between words, enabling models to better understand language context.

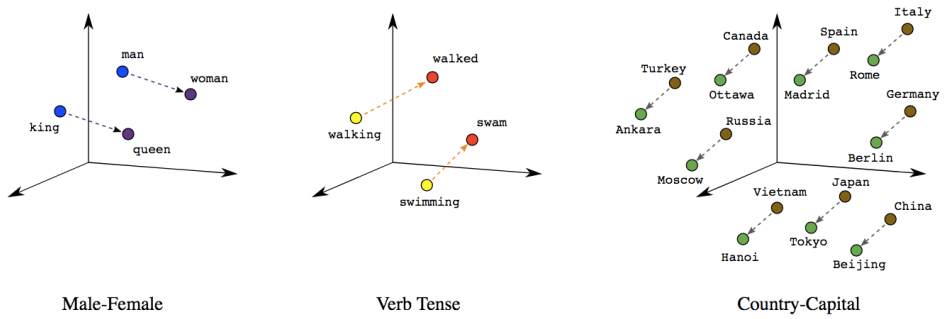

Figure 2: Linear Relationships between Words.

Vanilla RNNs often struggle to capture long-term dependencies due to issues like the vanishing gradient problem. **LSTM networks**, a specialized form of RNNs, overcome these limitations by introducing gates to manage the flow of information:

- Forget Gate: Decides what information to discard from the previous state.
- Input Gate: Determines which new information to store in the state.
- Output Gate: Produces the next hidden state based on the cell state.

Given an input sequence of words represented as numerical vectors using Word2Vec embeddings, LSTMs process the sequence step by step, retaining context information in their hidden states. This ability to capture both short-term and long-term dependencies enables LSTMs to excel at sentiment analysis tasks, where understanding the context of words is crucial.
In this assignment, you will preprocess the dataset by tokenizing text, converting words into vectors using Word2Vec embeddings, and padding sequences for uniform input length. Then, you will build, train, and evaluate an LSTM model for sentiment classification. The assignment is structured to guide you through the fundamental steps of preparing data, constructing the model, training it, and analyzing its performance.

Steps to Follow

- **Load the IMDB Dataset.** The dataset is provided in CSV format. Use a Python library such as pandas to load the CSV file. Perform exploratory data analysis (EDA) to understand the dataset.
- You can consider to use a subset of this dataset, if your computational resources are limited. It's up to you to decide how many samples you will use considering the trade-off between running time and accuracy. Explain your choices and make comments on your observations.

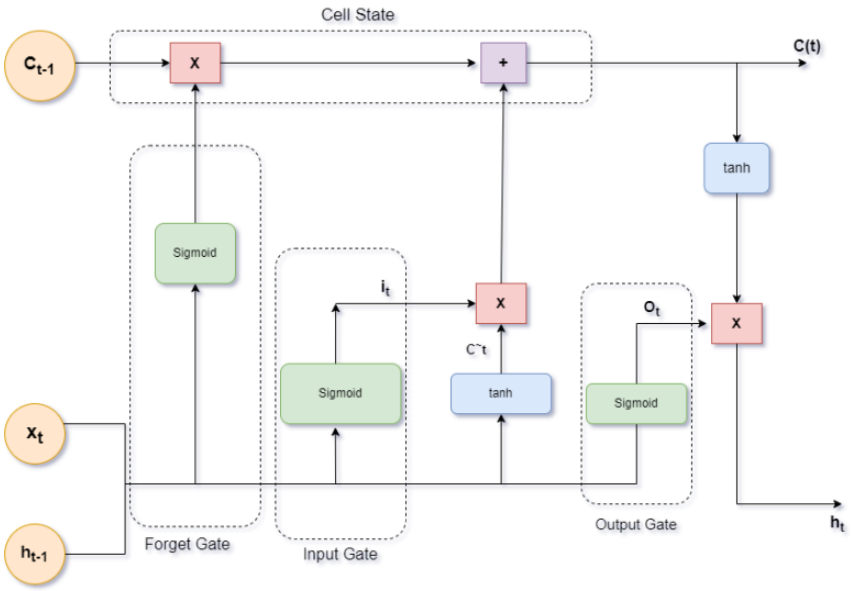

Figure 3: LSTM Architecture

- Preprocess the text data by converting text to lowercase, removing special characters, punctuation, and extra spaces. Tokenize reviews into sequences of words using a tokenizer (e.g., from nltk or torchtext). Represent the sentiments in a meaningful numerical form.
- **Use Word2Vec embeddings** to represent words as dense vectors. Load pre-trained Word2Vec embeddings and map words in the reviews to their corresponding vectors. You can make use of gensim.downloader to load word vectors such as "fasttext-wiki-news-subwords-300".
- **Visualize word vectors** to understand how they work. Start by choosing a manageable number of words from the vocabulary, ideally 100–500. Focus on words that are frequent in the dataset or relevant to the task, such as sentiment-rich terms like "good," "bad," "amazing," or "terrible." This ensures that the visualization is both clear and meaningful.
- For the selected words, retrieve their corresponding vector representations from the embedding matrix or directly from the pre-trained Word2Vec model. Each word will have a dense vector of a fixed dimensionality (e.g., 300 for fasttext-wiki-news-subwords-300).
- **Use Principal Component Analysis (PCA)** to reduce the dimensionality of the word vectors from their original size (e.g., 300) to 2D or 3D. PCA simplifies the high-dimensional data while preserving as much variance as possible, making it easier to visualize.

- For 2D visualization, create a scatter plot and annotate each point with its corresponding word to provide context. For 3D visualization, use interactive plotting tools like matplotlib's Axes3D to explore the spatial relationships between words dynamically.
- **Examine the resulting visualization** for patterns. Look for clusters of similar words, such as synonyms or words with similar sentiment, and observe the distances between opposite terms. Demonstrate how word vectors capture semantic relationships.
- **Represent each token** in the dataset with its corresponding index from the vocabulary. This step prepares the dataset for embedding layers, which require numerical indices as input.
- Ensure all sequences have the same length by **padding** shorter sequences with zeros using PyTorch's utilities. Deciding the padding length is important. Do not let the padded zeros to be the majority of the sequences. Explain your choice.
- Create a matrix where each row corresponds to the vector representation of a word in the vocabulary.
- The **embedding matrix** will have dimensions [vocab_size, embedding_dim], where embedding_dim is the size of the word vectors (e.g., 300 for fasttext-wiki-news-subwords-300).
- Use PyTorch's nn.Embedding.from_pretrained method to load the pre-trained embedding matrix into the model.
- Decide whether to freeze the embedding weights or allow them to be fine-tuned during training and explain your choice.
- Combine the padded sequences and corresponding sentiment labels into a dataset structure compatible with PyTorch. Use PyTorch's Dataset and DataLoader classes to manage the dataset efficiently and create mini-batches for training.
- To ensure balanced training, **split the dataset** into three subsets: Training set (70%) to be used for training the LSTM model. Validation set (20%) to be used for monitoring model performance during training. Test set (10%): Used for final evaluation of the trained model.
- **Manually implement the LSTM cell** to understand its architecture and functionality. Implement forget, input, and output gates. Update the cell state and compute the hidden state for each time step. Explain your steps and usage of the gates.
- **Build an LSTM layer** by combining one or multiple LSTM cells to process a sequence. Iterate through the sequence and update the hidden and cell states at each step. You can make experiments with the number of LSTM cells in a layer.
- **Build an LSTM model**. Integrate the **embedding layer** into the model architecture. The embedding layer will convert input indices (from padded sequences) into dense word vectors.

- These dense vectors are then passed to the LSTM layer for sequential processing.
- Integrate the custom LSTM layer into a full model.
  - Embedding Layer: Use pre-trained Word2Vec embeddings.
  - Custom LSTM Layer: Process the input sequence to generate hidden states.
  - Dense Output Layer: Use a fully connected layer with a sigmoid activation for binary classification.
- Define **appropriate loss function** for binary classification. Choose an **optimizer** and set its parameters (e.g., learning rate). Explain your choice of loss function and optimization algorithm.
- Iterate over the training dataset in mini-batches. Implement forward pass, feed the inputs through the network to compute the predictions.
- Compute the loss between predictions and actual labels. Implement backward pass, compute gradients of the loss with respect to model parameters. Update model parameters using the optimizer based on computed gradients.
- Validate the model on the validation set periodically to monitor performance and plot the validation and training loss at the end.
- Repeat the training process for a suitable number of epochs (at least 50 epochs).
- You can conduct experiments with different learning rates (lr) and batch sizes to find best performing model. Keep track of performance metrics during these experiments to identify the optimal configuration. Select your best model with respect to validation accuracy. It is enough for you to visualize the accuracy and loss change of the best model across training and validation datasets. Mention about your learning rate and batch size selection process in your notebook/report.
- Test the best model on the test set to evaluate its performance. Compute metrics such as accuracy, precision, recall, and F1-score to assess classification performance.
- Explain and analyze your findings and results.
- Summarize the performance of the model on the test set. Comment on the results.
- Discuss any challenges encountered during training and potential areas for improvement.

NOTE: PyTorch builds a computation graph dynamically as operations are performed. Your custom `LSTMCell`, `LSTMLayer`, and `SentimentLSTM` classes will rely on PyTorch's built-in operations like `torch.cat`, `torch.sigmoid`, `torch.tanh`, and `nn.Linear`. These operations are differentiable, and their gradients are implemented internally by PyTorch.

Understanding the LSTM Architecture

Long Short-Term Memory (LSTM) networks are designed to address the vanishing gradient problem in standard RNNs, enabling them to capture long-term dependencies in sequences. Each LSTM cell consists of three main components called **gates**, which regulate the flow of information:

- **Forget Gate**: Decides which parts of the previous cell state c_{t-1} to forget. It is computed as:
$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$
where σ is the sigmoid activation function, W_f are learnable weights, and b_f is the bias.

- **Input Gate**: Determines which new information to store in the cell state. It is computed in two parts:
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{c}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$

- **Cell State Update**: Combines the contributions of the forget and input gates to update the cell state:
$$c_t = f_t \cdot c_{t-1} + i_t \cdot \tilde{c}_t$$

- **Output Gate**: Determines the output (hidden state) h_t based on the updated cell state:
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t \cdot \tanh(c_t)$$

Implementation Steps

You will create an `LSTMCell` class to compute the operations for a single time step. The following steps outline how to do this:

1. **Initialize Weights and Biases**:
   - Define the weight matrices W_f, W_i, W_c, W_o for the forget, input, candidate cell state, and output gates, respectively.
   - Define the biases b_f, b_i, b_c, b_o.
   - You can define these within the `nn.Linear` layers for each gate.

2. **Forward Pass**:
   - Concatenate the input x_t and the previous hidden state h_{t-1} along the feature dimension.
   - Compute each gate's activation (forget, input, and output) and the candidate cell state using their respective equations.
   - Update the cell state c_t based on the forget and input gates.
   - Compute the new hidden state h_t using the output gate and updated cell state.

3. **Combine for Sequence Processing (LSTMLayer)**: To handle sequences, loop over the input sequence (time steps) and pass each time step through the `LSTMCell`. Maintain and update the hidden state h_t and cell state c_t across time steps.

4. **Define the `SentimentLSTM`**:

   (a) Initialize an embedding layer using pre-trained embeddings (e.g., Word2Vec) with `nn.Embedding.from_pretrained`.

   (b) Use the `LSTMLayer` to process the embedded sequence.

   (c) Extract the final hidden state of the sequence (corresponding to the last time step).

   (d) Pass the final hidden state through a fully connected layer to produce the output logits.

   (e) Apply a sigmoid activation function to produce probabilities for binary classification.

Guidelines for Implementation

1. **Class Definition**: Define a class `LSTMCell` that inherits from `torch.nn.Module`.

2. **Initialization**: In the constructor (`__init__`), define the weights and biases for all gates using `torch.nn.Linear`. Combine the input size and hidden size to calculate the appropriate dimensions for weight matrices.

3. **Forward Method**: Implement the forward method to:

   - Concatenate the input and previous hidden state.
   - Compute the activations for each gate and update the cell and hidden states.
   - Return the updated hidden and cell states.

4. **Sequence Layer**: Use the `LSTMCell` within `LSTMLayer` class to process entire sequences. Iterate through the sequence dimension, updating the hidden and cell states at each time step.

Tips for Testing and Debugging

- Start by testing the `LSTMCell` on a single time step with dummy inputs. Verify the dimensions and outputs of each gate.
- Extend to processing sequences once the single-step implementation is correct.
- Compare the outputs of your custom LSTM with PyTorch's built-in `nn.LSTM` to ensure correctness.

2 What to Hand In

Your submitted solution should include the following:

- The filled-in Jupyter Notebook as both your source code and report. The notebook should include (1) markdown cells reporting your written answers alongside any relevant figures and images and (2) well-commented code cells with reproducible results.
- Additionally, your report should be self-contained, encompassing a concise summary of the problem statement and a detailed exposition of your implemented solution.
- Begin by providing a concise overview of the problem statement, detailing the objectives and scope of the assignment.
- Describe the dataset. Include information about the features, targets, and any preprocessing steps applied to the data.
- Explain the approach adopted for each part of the assignment. Describe the algorithms implemented.
- Explain the evaluation metrics utilized to assess the performance of your models and justify their relevance to the problem at hand.
- Reflect on the outcomes of your experiments.
- Feel free to include tables or figures to emphasize specific aspects of your solution and enhance clarity for the reader.
- **You are also expected to participate in the upcoming Kaggle challenge.** You will use Kaggle's submission system to evaluate the performance of your best-performing model. This step ensures that your model is tested on a hidden test set to reveal their true performance in an unbiased manner. The Kaggle submission system will compute and display the final scores based on your uploaded model. **This will contribute 5% of the grade, based on Kaggle test performance.** So you should save your best models to be uploadded into the Kaggle. Challenge details will be announced shortly. **Stay tuned!**

You should prepare a ZIP file named <student id>.zip containing the Jupyter notebook in ipynb format as well as its .py (Python file) version containing all your codes. Submit it to submit.cs.hacettepe.edu.tr, where you will be assigned a submission. The file hierarchy is up to you as long as your Jupyter notebook and Python file works fine.

Attention! Please note that training the models could potentially require a considerable amount of time, particularly depending on the performance capabilities of your computer. To ensure that you obtain the required results in a timely manner, it is advisable to start the assignment early. If you find yourself starting the assignment later than anticipated and your computer's CPU capacity is limited, it might be beneficial to implement the assignment using Google Colab, leveraging its GPU resources for faster computation.

3 Grading

- Implementing and visualizing word vectors (30%)
- Creating and integrating the embedding matrix (10%)
- Implementing and optimizing custom LSTM model (40%)
- Evaluation and analysis of custom LSTM model (20%)

Note: Preparing a good report is important as well as the correctness of your solutions! You should explain your choices and their effects to the results.

Academic Integrity

All work on assignments must be done individually unless stated otherwise. You are encouraged to discuss with your classmates about the given assignments, but these discussions should be carried out in an abstract way. That is, discussions related to a particular solution to a specific problem (either in actual code or in the pseudocode) will not be tolerated. In short, turning in someone else's work, in whole or in part, as your own will be considered as a violation of academic integrity. Please note that the former condition also holds for online/AI sources. Make use of them responsibly, refrain from generating the code you are asked to implement. Remember that we also have have access to such tools, making it easier to detect such cases.

References

[1] IMDB Dataset of 50K Movie Reviews
[2] A Guide to Word Embeddings