Reinvent the wheel: LSTM

Adam Láníček
xlanic04@stud.fit.vutbr.cz

Martin Dočekal
idocekal@fit.vutbr.cz

Abstract

The aim of this work is to implement the LSTM neural network using PyTorch & NumPy primitives and to consequently test it in the domain of sentiment analysis – namely determining whether a IMDB movie review exhibits a positive or negative sentiment. For this purpose, 25000 training reviews were firstly preprocessed, fed into a PyTorch embedding layer initialized with 50-dimensional pre-trained word embeddings. These embedded sequences consequently entered the custom LSTM layer and the result was sent into the custom fully connected layer producing prediction score. Using the pre-trained word embeddings, the model arrived at 88% accuracy in predicting the sentiment of another 25000 reviews after just one epoch of training.

1 Task Definition

Data preprocessing. The used dataset accessed using Tensorflow's helper function consisted of 50000 movie reviews, equally divided into training and test sets, where each review was assigned positive (labeled as 1) and negative sentiment (labeled as 0). Mean review length was 235, providing an indication a sensible resulting embedding sequence length used in training. In order to speed up the training process, pretrained embedding of words from the review vocabulary were looked up in the *GloVe dataset* (Pennington et al., 2014).

LSTM layer analysis & implementation. The chain-like nature of Long Short Term Memory layer (further referred to as LSTM) reveals that this neural network architecture is intimately related to sequences and lists (Olah, 2015). Moreover, the ability to preserve long term information within a sequence, for example compared to their vanilla RNN predecessors, notoriously suffering from vanishing gradients problem, renders LSTM to be a good candidate for the problem at hand.

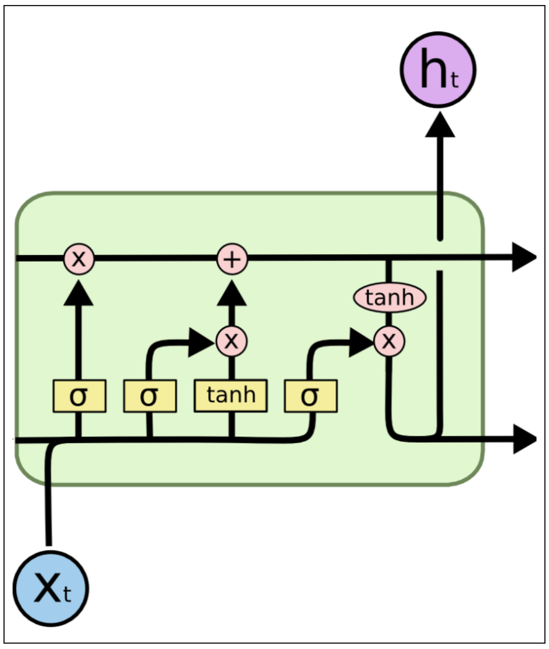

Figure 1: LSTM cell (Olah, 2015)

While both of the most popular machine learning frameworks, Tensorflow and Pytorch, offer high-level APIs to take advantage of LSTM layers in user-defined networks. The aim of this task work, however, is to implement a LSTM layer, consisting of multiple LSTM cells, using only NumPy and PyTorch primitives.

LSTM forward pass implementation. Compared to fully connected layer, a forward pass through a LSTM cell is rather a complex one, as depicted in Figure 1. LSTM cell consists of multiple *gates* that, in simple terms, determine of how much information is added or removed from the (long term) cell state propagated from the previous cell based on the current input and the output of the last cell. There are three gates in total (Olah, 2015), commonly referred to as:

1. *Forget gate* that considers the current data input x_t and h_{t-1} and for each elements within

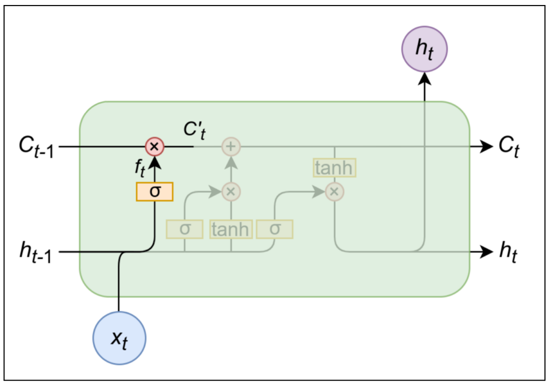

Figure 2: Forget gate (Esposito, 2020)

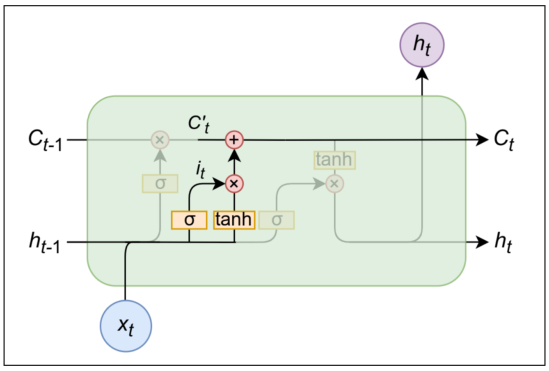

Figure 3: Input gate and update of the long-term memory (Esposito, 2020)

the vectors determines what part of the hidden state is to be preserved and what should to be forgotten. This decision is based on sigmoid function that outputs values between 0 (corresponding to dropping the previous information altogether) and 1 (corresponding to preserving all the information from previous cells) as formalized by Equation 1.

$$\begin{aligned} f_t &= \sigma(U_f x_t + V_f h_{t-1} + b_f) \\ c_t^{'} &= f_t . c_{t-1} \end{aligned} \tag{1}$$

2. *Input gate* within which the input x_t information is combined with the hidden state h_{t-1} and consequently added to the candidate solution $c_t^{'}$ provided by the forget gate as formally described by Equation 2. Studying the Figure 3 depicting the input gate and the consequent updating process of the cell state, attentive reader is able to see that after this step the cell's state (long term memory) is not updated anymore - it is the solution for long term memory c_t passed on to the next cell.

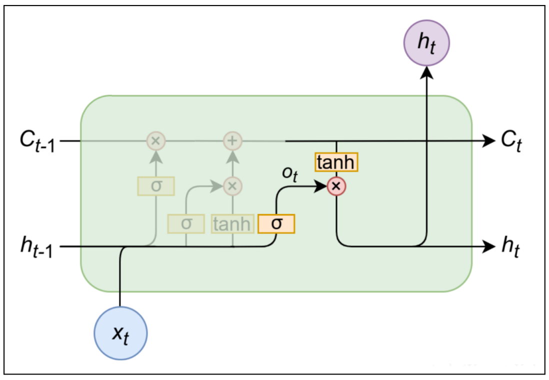

Figure 4: Output gate and hidden output of the cell (Esposito, 2020)

$$\begin{aligned} i_t &= \sigma(U_i X_t + V_i H_{t-1} + b_i) \\ C_t^{+} &= \tanh(U_c X_t + V_c H_{t-1} + b_c) \\ c_t &= c_t^{'} + i_t . c_t^{+} \end{aligned} \tag{2}$$

3. *Output gate* generates the second output of the LSTM cell, which is the hidden state h_t. Once again, as Equation 3 formally puts it, output gate takes into account the input x_t along with previous hidden state value h_{t-1} and calculates sigmoid value. This information is then merged with the long term memory solution (as the output of the input gate), generating a new value of hidden state to be either plugged into a next cell or taken as the final output of a LSTM layer.

$$\begin{aligned} o_t &= \sigma(U_o X_t + V_o H_{t-1} + b_o) \\ C_t^{+} &= \tanh(U_c X_t + V_c H_{t-1} + b_c) \\ h_t &= o_t . \tanh(C_t) \end{aligned} \tag{3}$$

Training & testing of the model. Last step was to utilize the implemented LSTM and fully connected layers and assemble a neural network for training, utilizing the 25000 labeled reviews of the IMDB dataset for this purpose. The remaining 25000 reviews were used for testing, as more thoroughly discussed in the next section.

2 Method

2.1 Data preprocessing

IMDB movie review dataset containing 50k labeled movie reviews was fetched using Tensorflow library API and split in half into training and test sets. Additionally, another API provided corresponding vocabulary consisting of 88588 words in total. To decrease computational complexity, vocabulary used during the training process was limited to 10k most frequently occuring words. To further reduce training time, embeddings for this limited vocabulary were looked up in the GloVe dataset (Pennington et al., 2014), containing 50 dimensional pretrained embeddings for 400k words. Unknown words, representing both words absent in the pretrained embeddings as well as those that were denoted as unknown because of the vocabulary size limit (10000), were embedded as tensor of average values in each of 50 dimensions.

Review sequences in the training dataset were on average 237 words long, thus to capture reasonable share of sequences to their full length sequence length of 300 words was chosen. Shorter sequences were padded using special padding word, embedded as a 50 dimensional tensor of zeros.

2.2 LSTM layer and forward pass implementation

As a baseline for the CustomLSTM class abstracting the LSTM layer served *torch.nn.Module* providing access to all standard features of PyTorch library neural network layers. Parameters (tensors) of corresponding sizes required for the LSTM forward pass computation used in equations 1, 2 and 3 were defined as instance parameters of the class. Weights were initialized using the Xavier normal distribution method (Glorot and Bengio, 2010) and implemented in the PyTorch library, biases were initialized as tensors containing only zeros.

Forward pass was implemented in instance method

```
def forward(self, x,
    init_states=None)
```

where:

- *x* is the sequence input of dimensions *(batch size, sequence size, embedding dimensions)*

- *init_states* is a tuple containing the h_{t-1} and c_{t-1} LSTM cell inputs as discussed in Task Definition section or as depicted any of the Figures 2, 3 or 4

The *forward* method processes its arguments in the following manner:

1. If previous cell outputs were not provided, h_{t-1} and c_{t-1} are initialized to zero.

2. The core of the method is represented by the loop over each word within the provided sequence, or, more precisely, multiple sequences, if *batch size > 1*, which is mostly the case in practice. One iteration represents one LSTM cell, for which the output values for all three *gates* as well as h_t and c_t output are calculated using the implementations corresponding to Equations 1, 2 or 3.

3. Last step is to extract the last element of the hidden sequence (h_t output of the last word in a sequence) and return it for further processing.

2.3 LSTM output processing & backpropagation of error

After completing the forward pass of LSTM layer, a tensor with dimensions *(batch_size, 2)* is obtained and consequently fed into a custom *MyFullyConnected* (linear) layer. Initialization of weights and biases of this layer followed the their initialization scheme within torch.nn.Linear module as referenced by PyTorch documentation[1]. *MyFullyConnected* layer that performs standard operation:

$$y = xw^T + b \tag{4}$$

Outputs of linear layer are confronted with the ground truth labels using PyTorch's implementation *torch.nn.CrossEntropyLoss*, producing value of the cross-entropy loss function for the given batch.

A natural next step to conclude a training round of LSTM network is to utilize the backpropagation algorithm (Kelley, 1960) to adjust the weights throughout the model in a way that decreases the total classification error using gradient calculus. Unfortunately, due to time constraints, I was not able to finish its implementation and therefore took advantage of the PyTorch's *autograd engine* that is able to calculate the gradients of the inputs, based on an automatically inferred computational graph from both *CustomLSTM* and *MyFullyConnected* layers (Paszke et al., 2017). After that, a weight update in the favourable direction is performed by the Adam optimizer.

[1] https://pytorch.org/docs/stable/generated/torch.nn.Linear.html

3 Experimental Setup

Training and testing of the LSTM network described in the previous section was performed in the Google Colab Pro environment, providing Python Jupyter notebooks runtime. With the aim of taking advantage of the available Tesla P100 GPU, numerous attempts were made to run the neural network training on it. However, approximately 4 out of 5 runs led to a crash of Colab runtime for reasons still unknown, partially due to the lower ability to debug CUDA issues and resulting crashes of the Google Colab environment. That said, CPU training as well as training using the TPU accelerators ran without any issues. Both models mentioned in the next section were trained using the TPU accelerators.

The aim of the experiments was to analyze the differences between these two different approaches:

1. Initializing the PyTorch embedding layer with the pretrained weights from the GloVe dataset with the hope of arriving at minimal loss and maximal prediction accuracy as quickly as possible

2. Let PyTorch library pseudo-randomly initialize the embeddings and analyze how much it affects the model loss & accuracy evolution

Both models were trained for 10 epochs with the learning rate of 0.005, each epoch taking approximately 20 minutes in Google Colab Pro environment. Training iterations were run with 64 reviews at once (i.e. batch size was set to 64) with the help of *torch.utils.data.DataLoader* API. During training, model accuracy was continuously evaluated after every tenth batch by running the inference on all of the 25000 reviews from the test set.

4 Results and Analysis

Table 1 shows the evolution of loss and accuracy criteria during the first 5 epochs of training. Not surprisingly, there is a significant difference between the count of training iterations required to arrive at the maximum possible precision, which lied at 89% in case of model taking advantage of pretrained embeddings. The pace of reaching the maximum accuracy levels, and even the accuracy of 83% at the end of first epoch, is really impressive, confirming the hypothesis that pretrained embeddings should speed up the training significantly and save a lot of computational resources as a result. The accuracy of vanilla variant of LSTM did not pass 84% mark during the 10 epochs, remaining constant from the fifth epoch onwards.

Epoch number	Loss	Accuracy
1	0.485	0.83
	0.519	0.74
2	0.154	0.84
	0.369	0.81
3	0.071	0.87
	0.383	0.81
4	0.025	0.89
	0.634	0.81
5	0.036	0.89
	0.244	0.84

Table 1: Training evolution of a LSTM network with and without using pretrained embeddings during the first 5 epochs

5 Conclusion

The aim of this work was to implement a custom LSTM network from scratch and evaluate its performance in the domain of sentiment analysis. LSTM cell was analyzed and its forward pass implemented. Despite not having implemented the backward pass[2] by hand due to time reasons, I was able to gain significant insight both into LSTM working as well as into PyTorch and working with custom neural network layers in general. The benefits of using pretrained embeddings for embedding layer initialization were confirmed and the model taking advantage of them was able to predict the sentiment of a movie review with the accuracy of 89%.

In general I have really been impressed with the accuracy of a neural network containing only one LSTM and one fully connected layer, confirming that LSTMs are really powerful for sequences prediction. It would be interesting to conduct more experiments in the future, especially after refactoring the code to make the network trainable on GPU. That is still a challenge that remains to be faced.

References

Piero Esposito. 2020. Building a lstm by hand on pytorch. https://towardsdatascience.com/building-a-lstm-by-hand-on-pytorch-59c02a4ec091. Accessed: 2021-12-29.

[2]LSTM variant of backpropagation algorithm

Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy. PMLR.

Henry J Kelley. 1960. Gradient theory of optimal flight paths. *Ars Journal*, 30(10):947–954.

Christopher Olah. 2015. Understanding lstm networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/. Accessed: 2021-12-29.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543.