ZPJa - 2023

Reinventing the wheel
LSTM from scratch

Michal Glos (xglosm01)

January 4, 2024

1 Project proposal

Task: The project will focus on developing a Long Short-Term Memory (LSTM) neural network from scratch to perform sentiment analysis on dataset $\mathbf{D}$ consisting of review texts $\mathbf{t}$ and the customer ratings $\mathbf{r}$. The LSTM model M with learnt parameters $\mathbf{\Theta}$ will approximate the ground truth ratings $\mathbf{r}$ with predicted ratings $\mathbf{r_{pred}}$, given the review texts:

$$\mathbf{r_{pred}} = M(\mathbf{t}, \mathbf{\Theta})$$

Solution: A **LSTM** neural network module will be implemented using **Python3**. Simple primitives will be sourced either from **pytorch** or **cupy** module to retain the GPU support. Also a dataset management module will be implemented, responsible for dataset download, processing and standardizing.

Data: The dataset management module will implement support of at least 2 of the following datasets:

- Amazon dataset: 142.8 million Amazon reviews sorted into categories[1]
- Yelp dataset: 6,990,280 reviews[2]
- Trip Advisor dataset: 50 millions of hotel reviews from Trip Adviser[3]

Experiments: Several experiments will be executed, focusing on: LSTM evaluation on test dataset, tuning the LSTM parameters for different datasets, test bidirectional LSTM architecrute, exposing different review entry attributes to LSTM input (e.g. reviewer age, gender, etc ...)

Edit: Experiments involving reviewer attributes in the model could not be conducted due to the anonymization of the data, which excludes information about the review authors. The only available attribute was the verification status of the user in the Amazon dataset, which will be included in the experiments despite its limited information content.

[1] Amazon DS: https://cseweb.ucsd.edu/~jmcauley/datasets/amazon_v2/
[2] Yelp DS: https://www.yelp.com/dataset
[3] Trip Advisor DS: https://github.com/Diego999/HotelRec

2 Problem description

The problem of predicting ratings from textual reviews using LSTM and a multi-layered perceptron (MLP) can be mathematically defined by establishing: the model's components, the nature of the input and output, and the process by which the model transforms the input into the output. This problem falls under sequence regression with variable-length inputs.

Predicting a numerical rating r from a set of all ratings R for a given textual review t from a set of all reviews T is formulated as:

$$V = \{w \mid w \in t \wedge t \in T\}$$

$$\forall t \in T : t = (w_1, w_2, \ldots, w_n)$$

$$\forall w \in t : w \in V$$

where V represents the vocabulary of all words in the set T, and each word w belongs to the vocabulary V. Each sequence t can have a unique length $|t|$.

2.1 Data preprocessing

Each sentence must first undergo tokenization to normalize the language using an appropriate algorithm:

$$(token_1, token_2, \ldots, token_m) = tokenize((w_1, w_2, \ldots, w_n))$$

Then, T_{token} constructed, which is a list of all tokenized review texts. Such data are then used for training a Word2Vec [3] model in order to generate token vectors representations of the same vector size. For the purpose of this project, the Word2Vec model is considered a black box, which once is trained, provides token representations for further text processing.

$$T_{token} = \{Word2Vec(tokenize(t)) | \forall t \in T\}$$

$$Word2Vec(tokenize(t)) = (\vec{e_1}, \vec{e_2}, \ldots, \vec{e_n})$$

The final step in preparing the dataset for the $LSTM$ model involves creating a set of 2-element tuples (t_{emb}, r). A function $\mathrm{F} : T \to R$ ($R = \{1, 2, 3, 4, 5\}$) is used to assign the rating value to each of the textual reviews t, hence the tuple is created. The dataset (a set of 2-element tuples) is constructed:

$$\text{Dataset} = \{(Word2Vec(tokenize(t)), \mathrm{F}(t)) \mid t \in T\}$$

2.2 LSTM and MLP models

The model can be defined as a function $\mathrm{F} : T_e \times \boldsymbol{\Theta} \to \mathcal{R}$, where $\boldsymbol{\Theta}$ represents the set of all possible parameters for the given model. Since the model comprises two distinct entities — LSTM and MLP — the full model equation is:

$$\mathrm{F} : T_e \times \boldsymbol{\Theta}_{\mathbf{LSTM}} \times \boldsymbol{\Theta}_{\mathbf{MLP}} \to \mathcal{R}$$

This particular model is a regression model, so the function the model approximates have the range on the whole set $\mathcal{R}$. Because only integers could be assigned to the reviews, the models output will be rounded to the nearest integer. The values will then be clipped, so the final range of the function will be the same as the set $R = \{1, 2, 3, 4, 5\}$. Such process would only be used during evaluations. During the training process, it is more efficient to take advantage of the smoother loss function, when working with the real numbers (raw model outputs) instead.

The $LSTM$ model is a variant of RNN (recurrent neural network), therefore it processes the input sequentially and recursively. The main $LSTM$ advantage is its memory mechanism, where the cell itself decides on how much information from previous hidden states to forget and how much information to add in there. Therefore, the model does not work only with its input and learned parameters, it also needs its hidden and cell states from previous iterations. For visualisation, see 2.

The trained parameters $\mathbf{\Theta_{LSTM}}$ consist of 4 pairs of weights and biases for 4 gates, which are in practice single linear layers with sigmoid or hyperbolic tangens activation functions. For visual representation of the LSTM cell itself, see the illustration below (1).

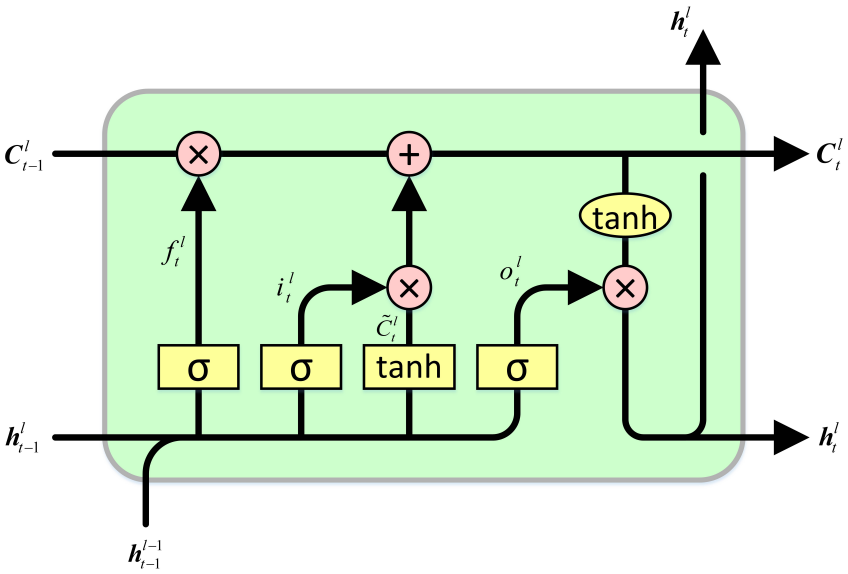

Figure 1: Graphical representation of the LSTM cell (before unrolling due to sequence processing). The output gate is denoted as o_t^l, the input gate as i_t^l and the forget gate f_t^l. The candidate gate $\widetilde{C}_t^l$ produces candidate state, which is transformed into the cell state C_t^l and then into the hidden state h_t^l. This schema nicely visualizes the dependency on the cell and hidden states from the previous timesteps. Historic values os the C_{t-1}^l and h_{t-1}^l, as well as the cells input h_{t-1}^{l-1}, which could be either a token embedding or when the cell is further in the model architecture it would be the previous layer output. This image was taken from here[5].

As apparent on the schema, each gate has a pair of trainable parameters. It's the layer weights, which are matrices of shape $(|h_{t-1}^{l-1}| + |h_{t-1}^l|, |h_{t-1}^l|)$ and a bias, which is a vector of size $|h_{t-1}^l|$. We can denote those as:

- Forget gate parameters - weights W^f and bias b_f
- Input gate parameters - weights W^i and bias b_i
- Output gate parameters - weights W^o and bias b_o
- Candidate gate parameters - weights W^C and bias b_C

After we established the notation and parameter names, we can mathematically denote the rules of a single LSTM step in following equations [2]:

$$
\begin{aligned}
f_t &= \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \\
i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\
o_t &= \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \\
\tilde{c}_t &= \tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

Where $[a, b]$ represents operation of concatenation of vector a with vector b. The input x_t is the token embedding, but could also represent h_{t-1}^{l-1} as in the figure 1. LSTM cell could have several layers - the first layers input is the token embeddings, the further layers inputs are always the previous layers outputs. The σ symbol denotes soft-max function.

When processing the first embedded token of the sequence, there are no states from previous steps h_{t-1} and $ct-1$. They could either be arbitrary set or learnt in the same manner as the LSTM model parameters are. Either way, the initial states will be denoted with timestep set to 0 - c_0, h_0.

By recurrently applying this set of equations on the input sequence unitl the sequence is exhausted, we obtain a sequence of hidden states. From a higher level, this could be written as:

$$\mathbf{h} = LSTM(\mathbf{x}, h_0, c_0)$$

Where $\mathbf{x}$ is the whole input sequence, $LSTM$ a high level abstraction of LSTM model and $\mathbf{h}$ is the sequence of extracted features of the input sequence $\mathbf{x}$. Such sequence has to be aggregated into a single vector for it to fit into the MLP. Specifically, functions such as mean, max-pooling, sum or picking the last value after processing the whole sequence are commonly used. Let's denote the function (whatever the function really is) as *aggregate*:

$$h_x = aggregate(\mathbf{h})$$

As the last step for predicting the rating of a review, a simple two-layered perceptron with σ (sigmoid) activation after the first layer and none at the output, which is suitable for the task of regression. The parameters of $\mathbf{\Theta_{MLP}}$ could be divided into weights and biases - W and b:

- First layer parameters - W_1 and b_1
- Second layer parameters - W_2 and b_2

The final step could finally be denoted as:

$$r_{raw} = \sigma(h_x \times W_1 + b_1) \times W_2 + b_2$$

$$r_{pred} = \lfloor r_{raw} \rceil$$

where $\lfloor x \rceil$ is rounding a number to the nearest integer.

2.3 LSTM and MLP learning

The parameters $\mathbf{\Theta}$, encompassing both $\mathbf{\Theta_{LSTM}}$ and $\mathbf{\Theta_{MLP}}$, are optimized using the backpropagation algorithm. The process hinges on a differentiable objective function J. For this project, Mean Squared Error (MSE) is chosen for its suitability in regression tasks:

$$J = MSE(\mathbf{y^{gt}}, \mathbf{y^{pred}}) = \sum_{i=0}^{n} (y_i^{gt} - y_i^{pred})^2$$

where $\mathbf{y^{pred}}$ contains the raw predicted values r_{raw}, ensuring more precise gradients compared to the rounded predictions r_{pred}. The vector $\mathbf{y^{gt}}$ represents the ground truth ratings.

The differentiated form of the MSE function, necessary for backpropagation, is given by:

$$\frac{\partial J}{\partial y^{pred}} = -2 \sum_{i=0}^{n} (y_i^{gt} - y_i^{pred})$$

Consequently, the gradients with respect to the weights and biases of the MLP can be computed:

$$\frac{\partial J}{\partial W_2} = a_1^T \times \frac{\partial J}{\partial y^{pred}}$$

$$\frac{\partial J}{\partial b_2} = \frac{\partial J}{\partial y^{pred}}$$

$$\frac{\partial J}{\partial W_1} = \frac{\partial J}{\partial y^{pred}} \times W_2^T \times \sigma'(z_1)$$

$$\frac{\partial J}{\partial b_1} = \frac{\partial J}{\partial y^{pred}} \times W_2^T$$

Here, z_i represents the output of the ith MLP layer before activation, and a_i is the output after activation. $\sigma'(z)$ is the derivative of the sigmoid function with respect to its input z. The MLP parameters are then updated by subtracting these gradients, scaled by a predetermined learning rate - gradient descent.

To compute the gradient of $\frac{\partial J}{\partial h_x}$ for backpropagation through the LSTM layer, we use:

$$\frac{\partial J}{\partial h_x} = \frac{\partial J}{\partial y^{pred}} \times W_2^T \times \sigma'(a_1) \times W_1^T$$

Training the LSTM involves the use of backpropagation through time (BPTT) due to its recurrent nature. Given a LSTM with parameters Θ_{LSTM}, the goal is to minimize the loss function, typically Mean Squared Error (MSE), for the regression problems. The LSTM processes an input sequence $T_e = (\vec{e_1}, \vec{e_2}, ..., \vec{e_n})$ and produces a sequence of hidden states $\mathbf{h} = (h_1, h_2, ..., h_T)$.

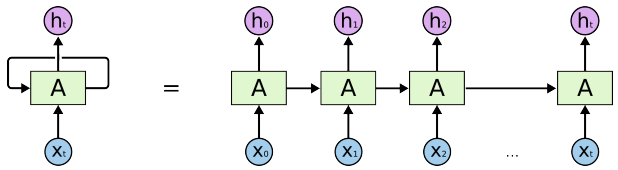

Figure 2: Unrolled RNN into a feedforward computational graph.

In order to continue computing gradients for the LSTM, we have to propagate gradients from the aggregated value h_x to the vector or hidden states $\mathbf{h}$. If picking the last vector is the chosen aggregation function, the chained gradients from the MLP could just be assigned to the last step of the unrolled LSTM. Chained gradients for non-last timestamps would be set to zeros.

$$\frac{\partial J}{\partial h_n} = \frac{\partial J}{\partial h_x}$$

The total value of the hidden state gradients have two components. The first (let's say primary) were the chained derivatives propagated directly from the MLP model itself through the aggregation function. The other one is accumulated from the next timesteps, where they flowed-in as the primary gradients and using chaine rule, they could be propagated through the datalink passing cell states from timesteps to the sequence beggining. So the "pick the last" hidden state aggregation function will have the primary component only in the last timestep (as every other aggregation function), but all the other timesteps will comprise only of the secondary component.

When the summing aggregation function is used, the gradients get distributed to all the timesteps, causing all the gradients from non-last timestep to comprise of both primary and secondary gradient. The secondary gradient is denoted as ϵ, $\epsilon_n = 0$:

$$\frac{\partial J}{\partial h_t} = \frac{\partial J}{\partial hx} + \epsilon_t$$

and finally, when the mean operator is used, the gradients get normalized by the sequence length

$$\frac{\partial J}{\partial h_t} = \frac{1}{N}\frac{\partial J}{\partial hx} + \epsilon_t$$

where N is the length of the currently processed sequence.

2.3.1 Backpropagation Through Time (BPTT) for Gradient Computation

In the context of training the LSTM, Backpropagation Through Time (BPTT) is utilized to compute the gradients of the loss function with respect to the network's parameters. For an LSTM with parameters $\mathbf{\Theta_{LSTM}}$, this involves calculating the gradients across each time step of the input sequence.

The BPTT process begins by computing the gradient of the loss function at the final output of the LSTM and then propagating this gradient backward through each cell and time step. The equations for these computations are as follows [6]:

$$\frac{\partial J}{\partial W} = \sum_{t=1}^{T} \frac{\partial J}{\partial h_t}\frac{\partial h_t}{\partial W}$$

$$\frac{\partial J}{\partial b} = \sum_{t=1}^{T} \frac{\partial J}{\partial h_t}\frac{\partial h_t}{\partial b}$$

Here, W and b represent the weights and biases of the LSTM's gates, and $\frac{\partial J}{\partial h_t}$ is the gradient of the loss function with respect to the hidden state at timestep t.

The gradients for the gates at each time step are computed using the following equations:

$$\delta c_t = \frac{\partial J}{\partial h_t} \odot \tanh'(c_t) \odot o_t + \frac{\partial J}{\partial c_t}$$
$$\delta \tilde{c_t} = \delta c_t \odot i_t \odot \tanh'(\tilde{c_t})$$
$$\delta f_t = \delta \tilde{c_t} \odot c_{t-1} \odot \sigma'(f_t)$$
$$\delta o_t = \tanh(c_t) \odot \frac{\partial J}{\partial h_t} \odot \sigma'(o_t)$$
$$\delta i_t = \delta c_t \odot \tilde{c_t} \odot \sigma'(i_t)$$

Where $\odot$ represents element-wise multiplication of matrices. The gradient of the loss function with respect to the previous cell state is given by:

$$\frac{\partial J}{\partial c_{t-1}} = \delta c_t \odot f_t \tag{1}$$

Finally, the gradients for each set of parameters are accumulated over the sequence:

$$\frac{\partial J}{\partial W_f} = \sum_{t=1}^{T} \delta f_t \cdot [h_{t-1}, x_t]^T$$
$$\frac{\partial J}{\partial W_i} = \sum_{t=1}^{T} \delta i_t \cdot [h_{t-1}, x_t]^T$$
$$\frac{\partial J}{\partial W_o} = \sum_{t=1}^{T} \delta o_t \cdot [h_{t-1}, x_t]^T$$
$$\frac{\partial J}{\partial W_c} = \sum_{t=1}^{T} \delta \tilde{c_t} \cdot [h_{t-1}, x_t]^T$$
$$\frac{\partial J}{\partial b_f} = \sum_{t=1}^{T} \delta f_t$$
$$\frac{\partial J}{\partial b_i} = \sum_{t=1}^{T} \delta i_t$$
$$\frac{\partial J}{\partial b_o} = \sum_{t=1}^{T} \delta o_t$$
$$\frac{\partial J}{\partial b_c} = \sum_{t=1}^{T} \delta \tilde{c_t}$$

To enhance the smoothness of the gradient flow, gradients are propagated through time through cell state and also through the hidden state. The cell state propagating to the previous timestep could be computed by the equation 1. The hidden state gradient equation (equation 2) result is concatanation of 2 vectors. First is gradient with respect to the input, which is useful when the input could also be trained, such as with the mulit-layered LSTM networks. The other component is the gradient of the hidden state from the previous timestep, which will be used in previous timestep gradient updates.

$$[\frac{\partial J}{\partial h_{t-1}}, \frac{\partial J}{\partial \vec{e_t}}] = \delta i_t \times W_i^T + \delta o_t \times W_o^T + \delta f_t \times W_f^T + \delta \tilde{c_t} \times W_{\tilde{c}}^T \tag{2}$$

where $[a, c]$ stands for concatenation of vector a with vector b. Sizes of both vectors are given by the model parameters, so the division point is implicated. Equations 1 and 2 in timestep 0 could be used to flow the gradients into the initial states (hidden and cell), causing them to be trained and carry some additional information to marginally enhance the accuracy.

$$\frac{\partial J}{\partial h_0}; \frac{\partial J}{\partial c_0}$$

All gradients with respect to weights, biases or initial states are scaled by learning rate and subtracted from the parameters to perform gradient descent.

3 Implementation details

The whole software was implemented in programming language **Python3**. Several key modules were used, such as **PyTorch**, **NumPy**, **nltk** and **gensim**. Software supports computation on both - a GPU and a CPU. All computations and datasets are implemented with PyTorch tensors, always with autograd option turned off, even for the LSTM bacpass. The best way of handling the training process is to load the dataset into RAM (cpu) and the model itself onto GPU (cuda). More entries of the dataset could be loaded into the bigger memory, while the training is performed on much faster *CUDA* device, while the transfer delay starts to be neglectable around the batch size of 64, and its effect fades with increasing batch sizes. The implementation also allows to use *HalfTensor*, *Tensor* and *DoubleTensor*, which are **PyTorch** tensors of 16-bit, 32-bit and 64-bit floats respectively. This functionality works with the model and the dataset and could be exploited to load 2 times more the data when 16-bit float is used, instead of the common 32-bit float. The software could either be used from CLI or as modules in another python source code or jupyter notebook. The whole source code could be found in `./src` directory of the project.

For running the program, python version 3.8 is recommended. Ideally, a Anaconda or VirtualEnv environment would be created with custom python version and all dependencies from `requirements.txt` would be installed.

The implementation could be divided into 3 main parts: The datasets, models and other (CLI interface, python environment and Makefile).

3.1 Datasets

In the project, four datasets are utilized: two dummy datasets (`SummingFloatDataset`, `SummingIntDataset`) for basic numerical operations, and two real-world review datasets (`AmazonDataset`[4] and `TripAdvisorDataset`[1]). Each of these inherits from the `DatasetBase` class, which sets the foundation for dataset functionality.

The `DatasetBase` class is integral, providing essential capabilities such as:

- Embedding textual data via a trained Word2Vec model.
- Data subdivision and balancing - `DatasetBase` supports the division of data into training, testing, and evaluation subsets. Additionally, it incorporates an EasyDataset object, compatible with `torch.utils.data.Dataset`. Into this class, the train, test and eval partitions are loaded for further use. Such compatibility unlocks all the possible potential of the PyTorch standard dataset.
- Efficient loading of large datasets in diverse formats (gzip, zip), leveraging multi-processing for increased speed.
- Methods for downloading datasets from the internet, complete with progress indicators.

Inherited datasets extend these functionalities. They mostly handle dataset-specific preprocessing and download, eventually the dummy dataset generation. The Amazon dataset is categorized into 29 product categories, requireing manually selecting the categories in the `./src/datasets/amznDS.py` source code.

Each dataset class incorporates a custom argparse parser for CLI interaction, ensuring almost full controll even from the CLI.

In addition to real-world datasets, the project incorporates two dummy datasets, `SummingIntDataset` and `SummingFloatDataset`, derived from the `DatasetBase` class. These datasets are specifically designed for model behavior verification and play a role in testing and validating the LSTM model.

SummingIntDataset

- Generates sequences of random integers, with the target (`y_gt`) being the sum of these integers.
- The dataset allows for adjusting the range of integer values (`min_val` and `max_val`), sequence length (`seq_len`), and vector size (`vector_size`), which is basically equivalent to embedding size parameter, offering flexibility for various testing scenarios.

SummingFloatDataset

- Similar to `SummingIntDataset`, but it generates sequences of random floating-point numbers, sampled from uniform distribution on interval $\langle 0; 1 \rangle$.
- The dataset creates sequences of floats with a specified length and vector size, similar to `SummingIntDataset`, but without specifying minimum or maximum values.
- The target value is the rounded sum of the sequence elements, which might be more challenging to the model abilities compared to `SummingIntDataset`

Amazon dataset

The `AmazonDataset` class, derived from `DatasetBase`, implements data download and preprocessing.

- Upon initialization, the class checks for the presence of dataset fragments in the specified `data_folder`. If any fragment is missing, it triggers a download process using URLs from a predefined list of resources to download the particular fragment.
- Once downloaded, the dataset fragments are loaded and reviews are tokenized. The `load_gzip_json` method is used to read gzip-compressed JSON files, extracting necessary attributes.
- The class embeds tokens using a $Word2Vec$ model (embedding size could be set by parameter `embedding_size`). This process converts text reviews into fixed-length vector representations.
- The dataset can be balanced first with trashing data with already collected rating and if it was not enough, with oversampling (balanced parameter) and is divided into training, testing, and evaluation subsets based on the `data_ratios` tuple.
- It offers configuration of embedding size, inclusion of the verified review bit into the embeddings and control over the maximal length of sequences (`max_seq_len`).

TripAdvisor dataset

The TripAdvisorDataset manages TripAdvisor hotel reviews dataset, which is quite extensive and could have not been fit whole into 32 GB of RAM, so only a fraction of the dataset could be loaded.

- If the dataset is not already downloaded in the `data_folder`, it is fetched from a given URL (`DOWNLOAD_URL`).
- The class loads the dataset from a zip file using the `load_zip` method, loading only specific attributes of interest to save on memory, and then processes it through tokenization and embedding similar to AmazonDataset.
- This class also allows for configuration of the embedding size, the ratio of data division for training, testing, and evaluation (`data_ratios`), and an option to balance the dataset with trashing redundant reviews with particular rating or if it was not enough - oversampling (`balanced` parameter).
- After embedding, the data are split into X and y components, shuffled, and divided into training, test, and evaluation sets according to the specified ratios.

3.2 Models

LSTMCell and BiLSTMCell

The LSTMCell class implements a full LSTM cell that handles heterogeneous sequences in a batch with zero padding. It includes several key features:

- Initialization: The class is initialized with input and hidden state sizes. It also defines device and data type for torch tensors.
- Gates Initialization: It initializes four gates (forget, input, output, candidate) with Xavier uniform initialization for weights and zeros for biases.

- Forward Step: The `forward_step` method performs a single LSTM operation, taking into account the input, previous cell state, and hidden state. This is implemented manually with tensor operations, no autograd used.
- Backward Step: `backward_step` is responsible for one step of backpropagation, updating the weights and biases based on the gradients. This is also implemented manually with tensor operations, no autograd used.
- Backpropagation Through Time (BPTT): The class supports BPTT for gradient computation across sequences.

The `BiLSTMCell` class is an extension that implements a bidirectional LSTM cell. It consists of two `LSTMCell` instances, one processing the input sequence as-is and the other processing the reversed sequence. The outputs of both cells are concatenated to form the final output.

Regressor

The Regressor class is a simple two-layer neural network designed to process LSTM features into a predicted value. Key aspects include:

- Network Structure: It consists of two linear layers. The first layer uses a sigmoid activation function, while the second layer has no activation function, making it suitable for regression tasks.
- Backward Pass: Implements a custom backward pass method for updating the network's weights and biases based on the gradients from the loss function.

LSTM

The `LSTM` class combines the functionalities of `LSTMCell` (or `BiLSTMCell`) and `Regressor` to create a complete model for sequence regression tasks. It includes:

- Model Configuration: Allows setting various parameters like input size, hidden state sizes, aggregation mode bag (sum), mean, or last, and bidirectionality.
- Forward Pass: Combines LSTM cell(s) and the regressor to process input sequences and output predicted values.
- Training and Evaluation: Implements methods for training the model on a dataset, evaluating its performance, handling checkpoints and loading the most accurate from checkpoints to evaluate the model. Also implements learning rate decay as a product of set learning rate and e^a, where a goes from 0 to a value set in parameter in the train method (by default -5).

3.3 Others

- Makefile - for removing python cache and compressing the source code into an archive - `make zip`.
- `lstm-cli.py` - Provides an option to controll the source code from CLI, rather then *jupyter notebook*, which is preffered.
- `arguments.py` - file implements class `CCLIParser`, which creates custom CLI argument parser dynamically with respect to specific dataset being used.
- `requirements.txt` - list of python dependencies.
- There will also be some *Jupyter Notebooks* showcasing some examples and code for plotting the model training process.

4 Experiments

This section describes a series of experiments designed to evaluate the performance of our LSTM model. The primary focus of these experiments is to assess the models accuracy in star rating predictions, its convergence stability, and its overall behavior under various configurations. We conduct these experiments using several datasets, including both dummy datasets for foundational analysis and real-world datasets for further model tuning. It's essential to note that the goal of these experiments is not to benchmark against contemporary models but to provide a deeper understanding and insight into this specific model implementation behavior for given input data. All experiments were executed on an Intel i7 machine with 32 GB of RAM and with an external GPU RTX 2060 connected via a Thunderbolt 3.

4.1 Experiments on dummy datasets

Following preliminary adjustments, a series of experiments was initiated to assess the model's behavior on a larger dummy datasets, containing one million data entries. The whole dataset is always divided by the ratio of $90 : 5 : 5$ into the training, testing and evaluation subsets. These tests aimed to evaluate the model's convergence, stability, and accuracy and for other information about the training process.

As the first experiment, several models were trained on the 2 dummy datasets `SummingFloatDataset` and `SummingIntDataset`. All the models were trained for 1 million steps, 250 reviews in each batch. Sequences within these datasets comprised of eight vectors, each vector containing eight elements generated from uniform distributions from the intervals $\langle 0, 7\rangle$ and $\langle 0, 1\rangle$ for integers and floats respectively. Learning rates were set to 0.01, 0.001, and 0.0001 to observe different paces of the training process and it's stability (no lr decay employed here). The LSTM's hidden state size was set to 64, and the MLP's hidden layer was set to 8. Unidirectional architecture along with mean aggregation function were used. Model was tested each 1000 steps on testing dataset.

During the training process, the loss and accuracy is measured periodically on the training and testing dataset. It was observed that the smoothed training and testing loss and accuracy closely resembles each other. This indicates there is no (or at least very minimal) overfitting of the model, given the data extent compared to the relatively small LSTM model. For the clarity purpose, the testing values will be shown on the plot for better readability, omitting the training part, which is identical, just with added noise (no overfitting observed).

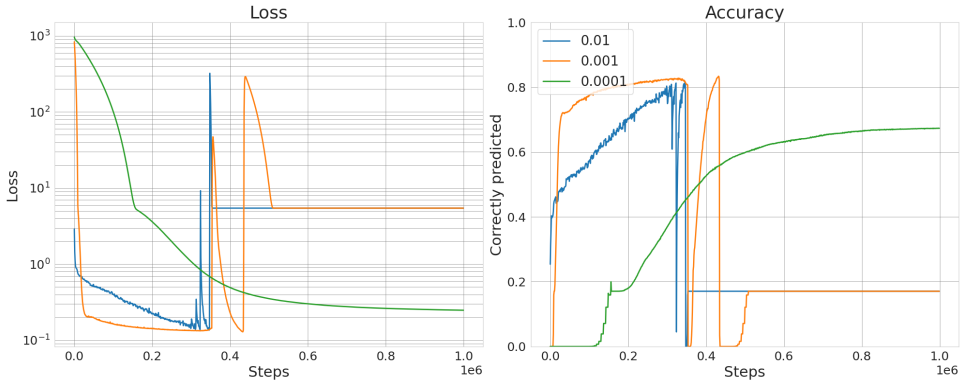

Figure 3: Dynamics of training on the SummingFloatDataset, utilizing various learning rates to probe the model stability and learning abilities. The left graph plots the testing loss, while the right shows the models testing accuracy. The x-axis enumerates the training steps, and the legend shows the learning rates.

At higher learning rates, the LSTM exhibits swift initial improvements, which subsequently give way to performance degradation, signaling learning instability. In contrast, reduced learning rates result in a consistent, albeit slower, enhancement in model performance. Though the model trained with higher learning rate would still wield the best accuracy before imploding due to it's unstable training process. This dichotomy highlights the delicate equilibrium between learning rate and model stability. Future experiments will, therefore, incorporate learning rates from the lower spectrum examined, though the value will yet be tested on the real-world datasets, for which this is just an suboptimal initial value. For later experiments, exponential learning rate is implemented.

4.2 Experiments with TripAdvisor and Amazon Datasets

Next experiments were conducted on both - the TripAdvisor and Amazon dataset, though only the Amazon results will be shown in the plots. This decision was done to increase the clarity of the plots and because the trends in the measured metrics were consistent across the hyperparameters and the datasets used.

The experiments were methodically designed and executed on a balanced dataset comprising of a 1 million unique reviews in the Trip Advisor and half a million in the Amazon datasets. Tokens were embedded into 32 long vectors of floats. Sequences were truncated at 80 tokens, aligning with typical review lengths to ensure enough information is carried into the model and the whole batch does not have to use large amount of zero paddings.

4.2.1 Batch Size Optimization

Determining the optimal batch size for training the LSTM model was a core aspect of the experiments, because it was observed such parameter has big influence in the final accuracy of the model. The learning rate was set to a 0.0001 to facilitate gradual yet consistent learning, employing also the learning rate decay (learning rate decays exponentionally - the learning rate is scaled by $e^{-\text{coeff}}$ at the final step, while the coeff increases in linear manner during the training process). The decay configuration: coeff $= 5$. The LSTM's hidden state size was set 256 to allow for a more complex representation, while the MLP's hidden layer size was set to 32. The model operated in unidirectional mode with mean function for aggregation. Standard parameter configuration from before is always retained, unless it's explicitly stated otherwise.

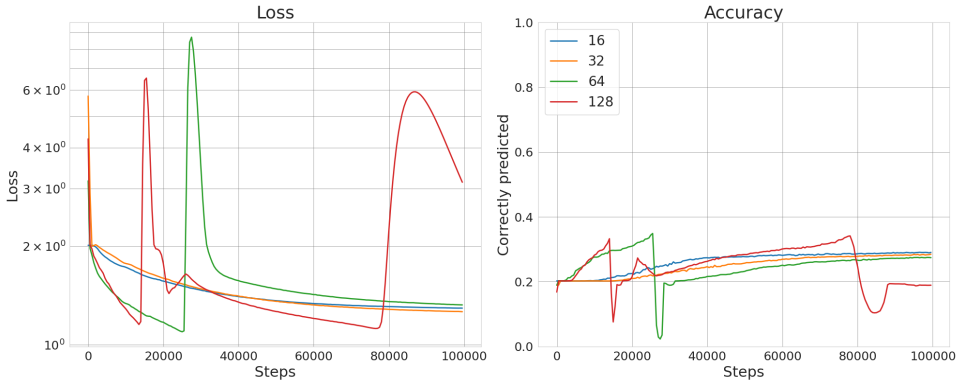

Figure 4: Assessment of the LSTM model's training behavior on the TripAdvisor dataset across variable batch sizes. The chart axes are consistent with those outlined in prior illustrations, delineating training steps on the x-axis and loss and accuracy on the y-axis.

In the experimental run, the bigger batches seem to peak in the beggining of the training and in general - achieve better results. This is a tradeoff bought out with the model emergent instability and tendency to explode. Though still not the kind of instability where the model weights explode to infinity and invalidates itself as a whole (though this happens with higher value of learning rate). As a direct result of this experiment, batches of 64 will be used for futher experiments.

4.2.2 Exploration of Model Dimensions

Several experiments were undertaken to determine the effect of the model size - the LSTM cell hidden state and the MLP hidden layer - on the model overall accuracy. It's important to determine the bottleneck of the model - whether it lies in the data representation, model memory capabilities or just the model architecture. The configuration was retained from the previous runs.

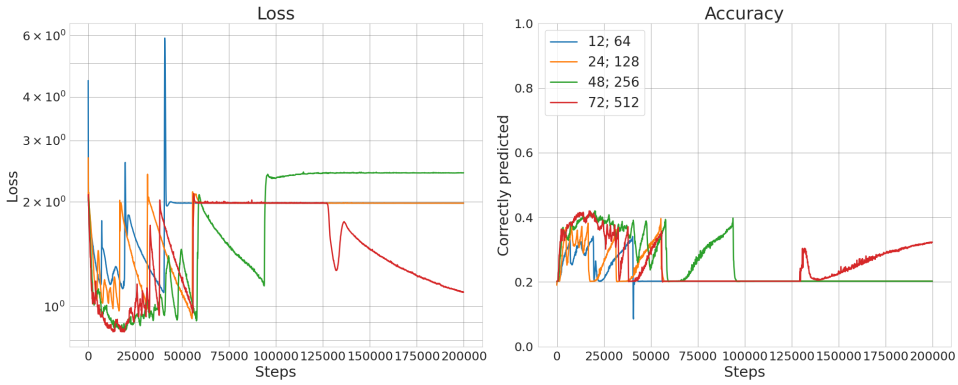

Figure 5: Training procedure visualization, showcasing the influence of varying MLP hidden layer sizes and LSTM hidden state dimensions, denoted in the legend in that order. Though the plot might be a little confusing to look at due to its instability and the count of showcased models.

The examination revealed that the LSTM configurations with hidden state sizes of 256 and 512 yielded the most favorable results or that the increase in model size cause increase in the model accuracy. Further investigation indicated that the MLP dimension plays a minimal role in influencing model performance, provided it remains above a critical threshold. Below this threshold, there is a noticeable decline in accuracy, emphasizing the necessity for a minimum model complexity.

With a benefit of hindsight, erroneous thought process was identified above. While increasing the model accuracy was not observed with the increase of the MLP size only, in further experiments, it was found out that the critical MLP size treshold where it becomes the bottleneck positively correlates with the LSTM hidden state size. So increasing the LSTM hidden state size would create a bottleneck out of the MLP, which size was not adjusted. Further experiments of the MLP hidden layer size were conducted in the subsection 4.2.5.

4.2.3 Aggregation Mechanisms

We investigated three primary aggregation techniques: mean, sum, and last-sequence-element. The aim was to determine the impact of these aggregation methods on the model's learning dynamics. Because of the broad scope of the project and extensive implementation, other aggregation functions were not implemented.

Model with LSTM of size 512 and MLP of hidden layer size equal to 72. All the other configuration stayed unchanged.

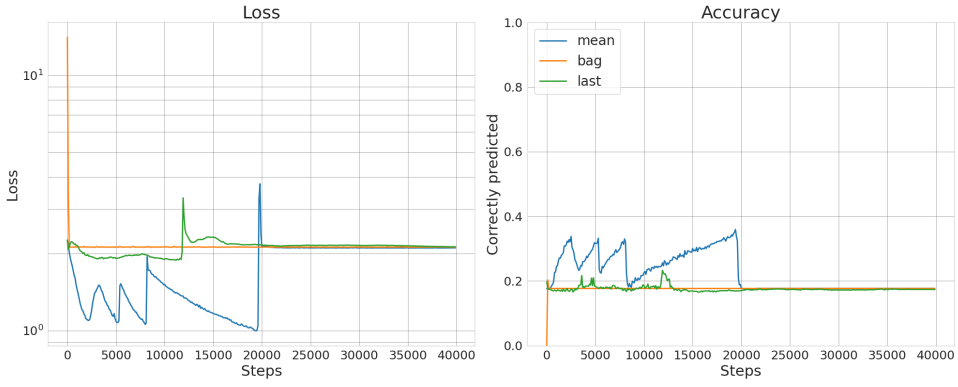

Figure 6: Depiction of the model's training trajectory employing various aggregation functions. The plot configuration is identical to previous setups.

The comparative analysis led to the conclusion that mean aggregation outperformed its counterparts, emerging as the sole method capable of delivering at least low level of prediction accuracy - which is around 40%. The alternative functions underperformed dramatically and their accuracy reached random

model choice models with roughly 20% of accuracy. The underperforming models proved no sense of information extraction, that lead to the models predicting expected rating value - 3. Hence only mean will be considered as viable aggregation function.

4.2.4 "Best Performance" Experiments

Following the calibration of initial parameters, a comprehensive evaluation was conducted to discern the impact of bidirectional processing and varying model dimensions. This assessment was premised on the hypothesis that bidirectionality and increased model complexity might enhance the LSTM's ability to capture nuanced sequential patterns in review data, though bidirectional LSTM cell has half the parameters compared to the unidirectional architecture. Twelve models were created, each trained on distinct partitions of the TripAdvisor dataset. The primary subset encompassed one million reviews, each encoded into a 32-dimensional vector space, while the secondary subset incorporated two million reviews represented in a 16-dimensional vector space. These models were tested with three different LSTM cell sizes: 256, 512, and 1024 units and the MLP hidden layer size was set to 32.

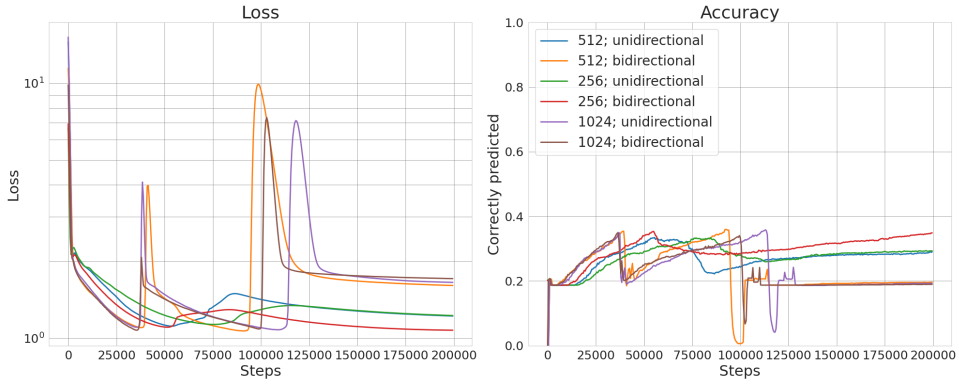

Figure 7: Loss and accuracy trajectories for LSTM models on the million reviews TripAdvisor dataset.

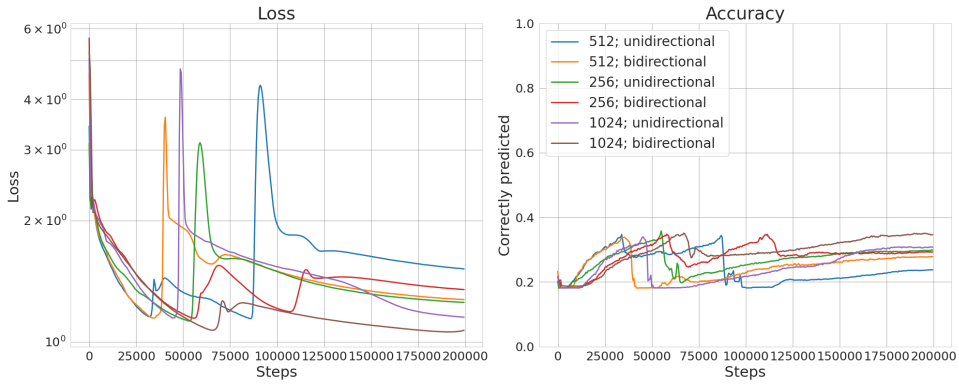

Figure 8: Loss and accuracy trajectories for LSTM models on the 2 million reviews TripAdvisor dataset.

The results are not very evident, but a little increase in accuracy and decrease in loss could be slightly correlated with the architecture and size of the LSTM model, though not in some gamechanging manner.

The same experiments were conducted on variants of the Amazon dataset, differentiated by an attribute indicating user verification. Usually roughly $\frac{2}{3}$ of the reviews come from verified user. Each dataset comprised half a million reviews, transformed into 32-dimensional vectors. In the subset without user verification, all 32 bits are used for Word2Vec embeddings, while the other subset has the verification bit appended to each tokens 31-bit embeddings. Models were configured identically to the experiments conjducted before on the TripAdvisor datasets.

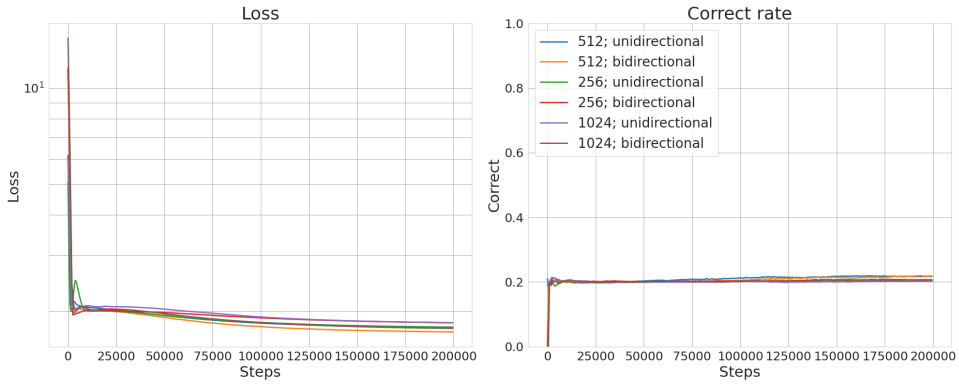

Figure 9: Loss and accuracy plots for LSTM models trained on the Amazon dataset containing the verification bits.

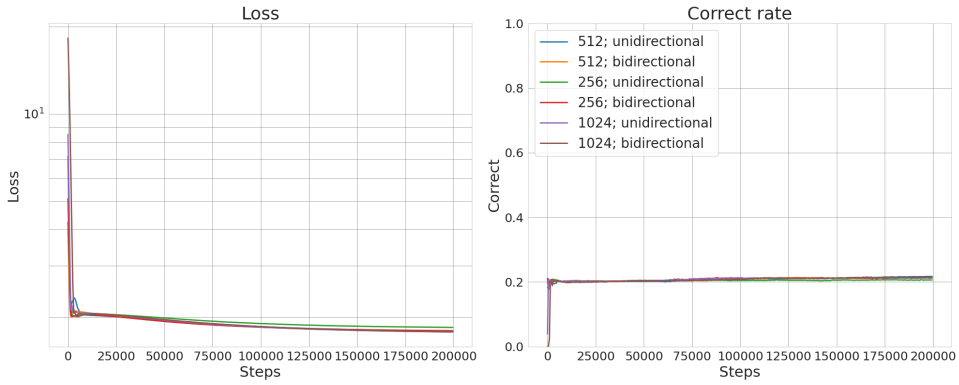

Figure 10: Loss and accuracy plots for LSTM models trained on the Amazon dataset not containing the verification bits.

Contrary to my beliefs of conducting a final experiments and obtaining best-accuracy models, the experimental results for the Amazon dataset indicated a significant regression in performance, compared to the previous experimental runs. The Trip Advisor dataset also experienced some performance drop, though not so significant. Amazon was always outcompeted on the accuracy compared to the Trip Advisor dataset, though reaching over 30 % was common. This was particularly perplexing given the consistency in hyperparameter settings across experiments - an LSTM size of 796, a bidirectional architecture, and a mean aggregation function.

The results obtained, while expected to top the experiments before, are really underwelming. A little underpeforming model on the Trip Advisor dataset will be examined in further subsections for the MLP hidden size parameter change response. I suspect the models bottle-neck resides in there, because we tested the LSTM size along with the MLP hidden size - always increading both linearly. Employing the bigger MLP hidden size, I believe we will be able to reach much better levelf of accuracy, see next subsection.

Regarding the Amazon, the results are much worse then in previous experiments and the reason for it will be examined in subsection 4.2.6. Some kind of error in need of fixing might have been encountered, so closer examination is neccessary.

This caught me by surprise, so other experiments will yet have to be conducted. The goal of the experiments is to reach accuracy of 50% or more on the Trip Advisor dataset, which will certainly be chellanging.

4.2.5 Further imporvements to accuracy

A set of experiments was undertaken to identify the optimal MLP hidden layer size in relation to the LSTM hidden state size. Initial experiments varied the MLP size while fixing the LSTM layer size to 796 units. Subsequent experiment explored the performance of a substantially larger model with a bidirectional LSTM layer size of 2048 and an MLP size of 1024 to examine the effects of increased model capacity over an extended number of training steps.

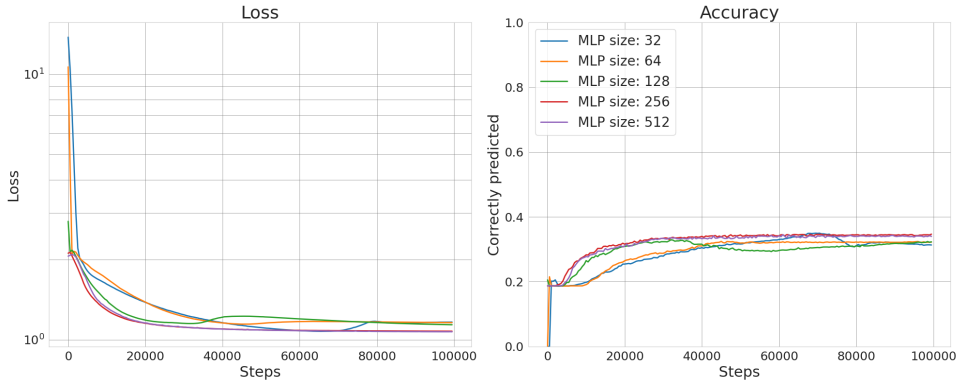

Figure 11: Comparative loss and accuracy performance for varying MLP hidden layer sizes with a fixed LSTM size of 796. It is evident that increasing the MLP size improves model stability and accuracy up to a point, beyond which the gains plateau. Though the inflexion point is (as shown in experiments before) affected by the LSTM hidden state size.

The initial findings suggest that larger MLP sizes generally correlate with improved stability and accuracy, as indicated by smoother loss curves and higher, more stable accuracy rates. However, the benefits appear to diminish beyond a certain threshold.

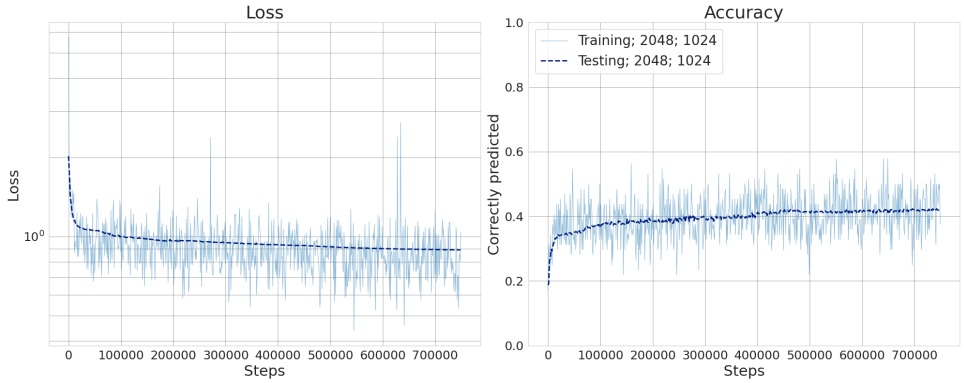

Figure 12: Extended training performance of a large-scale bidirectional LSTM model with LSTM size 2048 and MLP size 1024. The loss stabilization and correct rate suggest that the model benefits from the increased complexity, but the extended training time unveils a potential overfitting issue, as seen in the divergence of training and testing correct rates at the end of the training process. Such discrepancy though is minimal and could barely be seen, but it still is a first instance of overfitting in our experiments.

In the extended training experiment, the larger model size initially shows promising declines in loss and increases in correct rate, though the trend does not persist through the whole process. Best evaluation accuracy was achieved with the evaluation dataset - 42.3%.

These experiments underline the intricate balance between model size and generalization capabilities. While larger models have a higher capacity for learning complex patterns, they also pose a higher risk of overfitting, especially when trained for an extended period.

Other experiments have been done to examine the effect of changing the embedding size while shrinking the dataset size itself. Those 2 parameters have to be balanced out, because of the RAM limitations.

I also implemented the datasets to be capable of existing in whatever pytorch version of float tensors, so using `float16`, we can fit double the data into the same sized dataset. In total, 4 runs were executed with 1000000 reviews and 64-bit embeddings, 500 000 reviews and 128-bit embeddings, 250 000 reviews and 256-bit embeddings and finally 125 000 reviews with 512-bit embeddings. The model used was a unidirectional model with 1024 LSTM hidden state size and 256 MLP hidden layer size, with learning rate of 0.0001 and batch size used of 64. Other configuration was identical to the previous one.

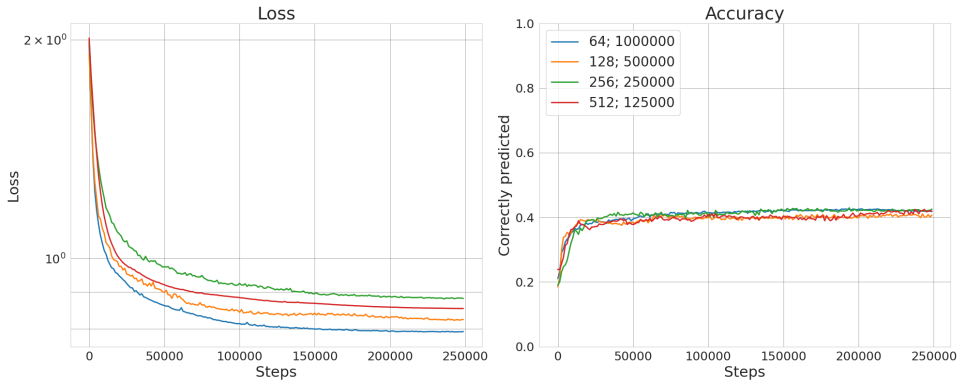

Figure 13: Training progression of the experiments showcasing little to no effect on the model accuracy. The plot layout is identical to all the previous.

The results obtained are not at all statistically significant, for such results we would have to repeat the experiments for several times. But the training process does not show any signs of accuracy decrease or increase with the dataset configuration change. Though no numerical observations could be made, the idea of the model not overfitting for any of the given datasets implies there is no need to increase the review count and sacrifice the embedding representation vector size.

Much larger model will be used for final training session, for which the 250 000 reviews with 256 embeddings was chosen, for balancing the tradeoff inbetween the total sample size and the token embedding size. The model was configured to: 3072 long LSTM hidden state and 4096 long MLP hidden layer. All the other configuration would stay the same as in the experiments before.

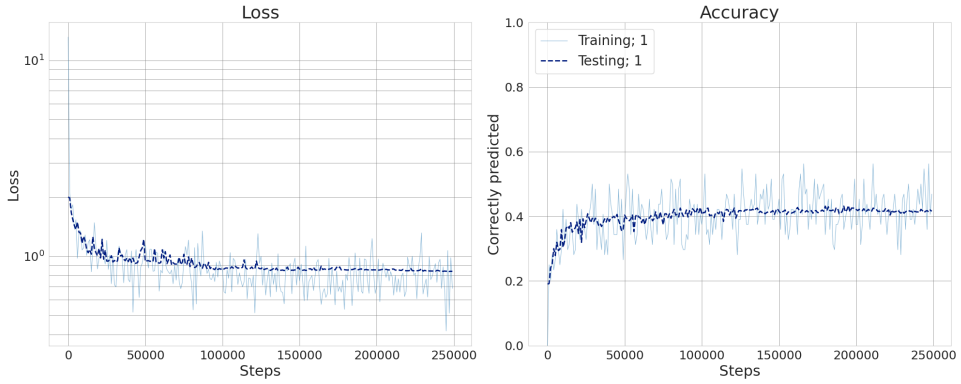

Figure 14: Visualization of the training process of our final experiment. The plot layout is identical to all the previous ones.

Resutls of out best behaving model showed accuracy of 43.33%, which is quite substantial, given the complexity of data and simplicity of the model used. In the plot, we can notice that as well as the long run before, the training and testing loss and accuracy started diverging at the end and the model was no more getting more accurate, overfitting ... Sadly, I do not have enough time to run more experiments because of the computational costs of such runs and considering the overfitting, the model might even benefit from lower-granularity representation of token embeddings with more reviews. Perhaps, more experiments will be conducted just to feed my curiosity and their results will be uploaded in the repository `github.com/michal-glos-brq/LSTM-from-scratch`.

4.2.6 Amazon Dataset Tuning

Regarding the Amazon dataset experiments - learning from the previous experiments on Trip Advisor dataset results, model configuration and the dataset preprocessing was tweaked to reflect those observations. Amazon dataset was loaded with 500 000 review texts and 256-bit token embeddings. The model used have 2048 parameters int the LSTM cell, 1024 in the MLP hidden layer, was bidirectional and employed mean as the aggregation function. The model was trained in batches of 128 per 150 000 training steps.

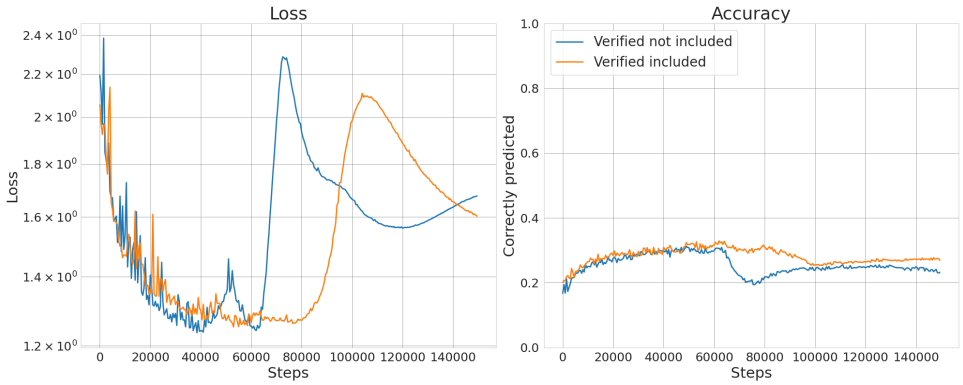

Figure 15: Visualization of tainingn process on the Amazon dataset with verified bit included or not. The plot layout is identical to the previous plots.

The best performing model with the validation bit present achieved 32.9% accuracy on the evaluation dataset and without the verification bit, the best accuracy measured was 31.2%. Though the results might also not be statistically significant, the possibility of increase in accuracy when the verified bit is appended to the embeddings is real. The fact the verified reviews are quite prevalent in the dataset might actually indicate some sort of division of the review styles, affecting the later classification process, thus increasing the accuracy. This is all a speculation though and would need further investigation either the data itself or run statistically significant experiments with these models to determine the true impact of the attribute itself.

4.3 Torch datatypes

A test run was conducted to observe possible increases or decreases in model accuracy when different datatypes will be used for the model matrices. Datatypes of float16, float32 and float64 were used and were tested on the hyperparameter configuration of: 0.0001 learning rate, batches of 64, unidirectional architecture and mean was used ad aggregating function. This experiment was conducted on the TripAdvisor dataset only (with one million reviews and token embeddings of size 32). Training was conducted in several configurations for 50000 steps:

- LSTM size: 128; MLP size: 32; Datatype: float64;
- LSTM size: 256; MLP size: 64; Datatype: float32;
- LSTM size: 128; MLP size: 32; Datatype: float32;
- LSTM size: 512; MLP size: 128; Datatype: float16;
- LSTM size: 128; MLP size: 32; Datatype: float16;

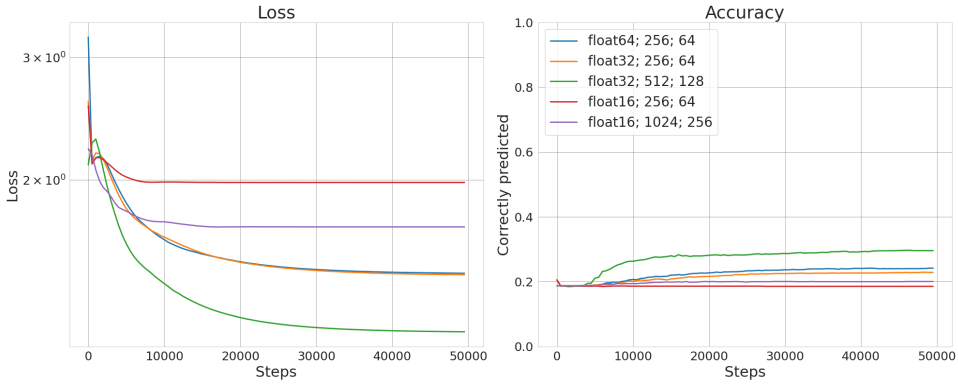

Figure 16: Training progress of different datatypes of the model parameters. The plot layout is identical to the standard layout used throughout this project.

The results are quite interesting. The idea behind the testing was trying to have the same sized models, just using different datatypes. The other tested configuration was set to have the same amount of bits in the LSTM hidden layer and the MLP hidden layer, which lead to some interesting results.

An interesting observation is that the model utilizing 64-bit floats achieved almost the same accuracy and loss as the model using 32-bit floats. In contrast, the 16-bit float model appears to have learned very little. This outcome might be attributed to the pseudo-optimal size of the model parameters being 32 bits. There are likely valid reasons why 32-bit floats have become the standard; one possible explanation is that 64-bit floats do not provide a significant increase in information and granularity, while 16-bit floats are considerably constrained in both granularity and value range.

In this experiment, where memory sizes were held constant for the LSTM hidden state and MLP hidden layer, the 32-bit float model also demonstrated superior performance. This supports the notion that 32-bit floats strike a favorable balance between precision and the datatype size. The 64-bit model had fewer parameters, its performance lagged as anticipated, suggesting that simply increasing precision does not compensate for the reduction in parameter count, not even boosting it's accuracy in a significant manner. When the number of parameters was increased for the 16-bit float model, it failed to surpass the performance of the model with 32-bit parameters, which was an expected outcome. This further implies that the advantages of a larger parameter count are constrained by the limitations of precision in 16-bit floats, confirming the standard use of 32-bit floats due to their optimal balance of computation speed, precision, and parameter count.

5 Comparison

In this section, we draw a comparison between the current results and my prior work, where an LSTM model was employed alongside a single perceptron to process the LSTM states. The previous work was dedicated to predicting review ratings for an Amazon dataset using a custom word embedding implemented in NumPy with Python3. This embedding was uniquely a vector of length five, representing a probability distribution over five rating classes.

The embedding equation was formulated as follows:

$$\vec{e} = (p(r = 1|\mathbf{x}), p(r = 2|\mathbf{x}), p(r = 3|\mathbf{x}), p(r = 4|\mathbf{x}), p(r = 5|\mathbf{x}))$$

where r denotes the actual rating. Admittedly, this form of embedding might seem advantageous - bordering on the edge of 'data leakage' - since it directly incorporates the probability of each rating class. An accuracy of 61.91 % was achieved through simple statistical analysis on the dataset.

When training a LSTM model of 64 long hidden state size, trained on a 1750000 reviews with learning rate 0.00001, it was able to achieve 58.22%. Even though we employed much larger models with much more complex training routine and also adding a hidden layer into the MLP, we did not achieve even compareable results. This might be because of the inherent sneaking of the information about the review rating in the word embeddings, which lead to much more accurate predictions. When we actually try to use word embedding based on their similarity, the information about the rating fades away and the model is actually forced to learn to analyze the sentiment of such reviews.

I did not achieve at least the same accuracy of the model as with the model before, what was set as a goal during the assignement registration, later decreased to the goal of 50% accuracy. Though the example and training process was completely different and data leakage was forbidden in the newer model, given the model simplicity and fact it was forced to learn actual semantics analysis, it performed quite well.

6 Conclusion

This project successfully implemented a Long Short-Term Memory (LSTM) neural network from scratch, primarily focusing on the application of sentiment analysis on textual data from Amazon and TripAdvisor reviews. Utilizing Python3 and libraries such as PyTorch, NumPy, nltk, and gensim, the project demonstrated the process and challenges of developing a LSTM neural network without relying on high-level abstractions and automatic gradient computations.

The core of this work was an extensive series of insightful experiments designed to assess the performance of the LSTM model under various configurations. These experiments revealed several key insights. First, the choice of aggregation method significantly impacts model accuracy, with the mean aggregation method showing the best results. Second, the size of the LSTM and MLP layers influences the model's ability to capture nuances in the data, though larger models are more prone to overfitting, more so coupled with high-dimension token embeddings and low count of training data. Learning rate and the batch size used drastically affects the model learning speed and it's training stability. Also, the bidirectional model overperforms the unidirectional architecture. Also, the use of 16-bit floats as network parameters will not provide enough granularity for the parameters and the model will not be able to capture nuances well.

An intended goal was to explore the impact of reviewer attributes on the model's performance. However, this was not feasible due to data anonymization, highlighting a common challenge in machine learning projects where data availability can limit the scope of analysis.

Even though the goal of 50% accuracy was not met, the achieved accuracy of 43.33% is quite sufficent, given the model sheer simplicity, combined with rather compicated, complex task of sentiment analysis. Apart from this project, further experiments will be conducted, which will be published on my github page[4]. The project was very insightful and I enjoyed the implementation, as well as the experimenting and subsequent experiment analysis.

Since LSTM is very old architecture, there are many methods suitable for increasing it's accuracy. We could either use different architecture as GRU, transformer or other LSTM variant. The word embeddings could also capture more nuance with a different method. Making the LSTM or MLP model bigger, with more layers and more parameters could improve its accruracy as well as regularization techniques.

[4]GitHub account: github.com/michal-glos-brq/LSTM-from-scratch

References

[1] Matthias Braunhofer and Francesco Ricci. Tripadvisor dataset, 10 2016.

[2] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 11 1997.

[3] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[4] Jianmo Ni, Jiacheng Li, and Julian McAuley. Justifying recommendations using distantly-labeled reviews and fine-grained aspects. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan, editors, *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 188–197, Hong Kong, China, November 2019. Association for Computational Linguistics.

[5] Christopher Olah. Understanding lstm networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/, 2015. 2/1/2024.

[6] P.J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.