LSTM Word Prediction from Scratch

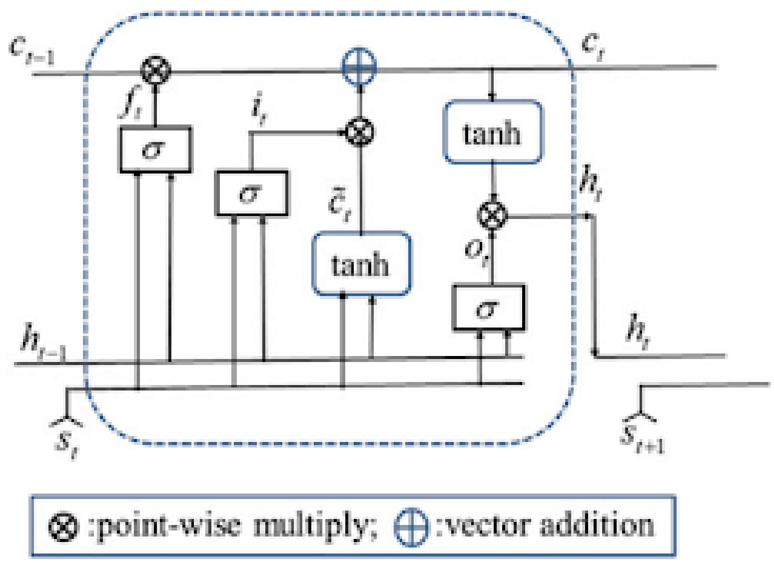
Thuong Dang, Phuoc Bui

Abstract

This is a documentation for our work on constructing an LSTM network that uses only Numpy for word prediction. In our work, we use previous two words to predict a next word. The formulas and structure of our program are described in detail in this documentation.

1 LSTM architecture

In short, LSTM is a variant of RNN, and it has *memory cells* and it *learns to forget*. That means, at each step, we will forget unimportant information and memorize important information. Let us look at the architecture of LSTM.



The picture is taken from [...]. For each block of an LSTM network, there will be four gates: a forget gate, an input gate, an output gate, and an intermediate gate, and their values are denoted $f_t, i_t, o_t, \tilde{c}_t$, respectively. We also denote c_t the internal memory state, and h_t the hidden state as in the architecture of RNN. Let us take a explain a bit further on this gated architecture. For the inputs, we denote

h_{t-1}: the previous hidden state.
$s_t = W_{\_}x_t + b_{\_}$: where $W_{\_}$ and $b_{\_}$ denote the weight and bias for gate $\_$.
c_{t-1}: the previous memory cell.
For gates, we have

Forget gate. As we can see, the output of the forget gate f_t has inputs h_{t-1} and s_t, where $s_t = W_f x_t + b_f$. Those are weight bias for the forget gate. And

$$f_t = \sigma(U_f h_{t-1} + W_f x_t + b_f),$$

where σ is the sigmoid function, U_f is the weight for the forget gate of h_{t-1}.

Input gate. The output of the input gate, denoted i_t has inputs h_{t-1} and s_t, where $s_t = W_i x_t + b_i$. Those are weight and bias for the input gate. And

$$i_t = \sigma(U_i h_{t-1} + W_i x_t + b_i),$$

where U_i is the weight of h_{t-1} for the input gate.

Output gate. Similarly, we have

$$o_t = \sigma(U_i h_{t-1} + W_o x_t + b_o)$$

Intermediate gate. We have

$$\tilde{c_t} = \tanh(U_c h_{t-1} + W_c x_t + b_c)$$

Through gates, we can define our memory cell

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c_t},$$

where $\odot$ denotes the Hadamard product, i.e. it is element-wise multiplication of matrices. Let us explain a bit about this. Our f_t is defined as a sigmoid function of previous state and current input, and its range is between 0 and 1. With Hadamard product, if a value f_{ij} of f_t is very close to 0, then $(c_{t-1})_{ij} f_{ij}$ is also very close to zero, and it means, the value of this position is not very important we can forget it. On the other hand, the product $i_t \odot \tilde{c_t}$ decides which information will be updated. Why? Because if a value i_{jk} of i_t is very close to 1, then $(i_t \odot \tilde{c_t})_{jk}$ is very close $(\tilde{c_t})_{jk}$. In some variants of LSTM, i_t is replaced by $(1 - f_t)$. In short, c_t helps us filter information: it determines which information we should forget or remember.

Now, we can update our current hidden state

$$h_t = \tanh(c_t) \odot o_t$$

And again, we see a filter o_t here: it decides which information to pass through. To produce an output, we can again use softmax

$$\widehat{y} = \text{softmax}(W_y h_n + b_y),$$

where h_n is the last hidden state.

2 Forward method

In the forward method, we initialized

- random matrices $U_f, W_f, U_i, W_i, U_c, W_c, U_o, W_o$ and
- random vectors b_f, b_i, b_c, b_o

for the four gates. Moreover, we will also initialize randomly the weight W_y and the bias b_y for the output. Their sizes depend on our input size, output size and a chosen hidden size. In our case, we set input size and output size to be the size of our dictionary, and hidden size 64.

Note that, because of the LSTM architecture, we will need to add the zero vector h_0 and c_0 for the initialized hidden state and memory cell. Because we will start from time $t = 1$, and the length of hidden states and gates are the same, it is easier if we also set those vectors $f_0, i_0, \tilde{c_0}, o_0$ to be zero (we never use them though). Now, we can easily code the foward method based on the formulas above. Here are the codes

```
# Compute the forget gate
f = sigmoid(self.Uf @ self.hs[t] + self.Wf @ x + self.bf)
# compute the intermediate gate
c_tilde = np.tanh(self.Uc @ self.hs[t] + self.Wc @ x + self.bc)
# Compute the input gate
i = sigmoid(self.Ui @ self.hs[t] + self.Wi @ x + self.bi)
# Compute the output gate
o = sigmoid(self.Uo @ self.hs[t] + self.Wo @ x + self.bo)
# Compute the memory cell
c = hadamard(c_tilde, i) + hadamard(self.cs[t], f)
# Compute the hidden state
h = hadamard(o, np.tanh(c))
```

3 Backward method

We now come to the most difficult part: compute backpropagation. But this can be easily done if we write everything explicitly. First, we can define the loss function L to be the categorical loss function

$$L = -y_{\text{true}} \log y_{\text{pred}},$$

where y_{true} is the true label and y_{pred} is the output of the softmax function. Let $W = W_y h_n + b_y$, by [...], we have

$$\frac{\partial L}{\partial W} = y_{\text{pred}} - y_{\text{true}}$$

Because

$$\frac{\partial W}{\partial W_y} = h_n, \frac{\partial W}{\partial h_n} = W_y, \frac{\partial W}{\partial b_y} = 1,$$

where 1 is the column vector with all entries are 1. By the chain rule, we can compute

$$dW_y = \frac{\partial L}{\partial W_y} = \frac{\partial L}{\partial W}\frac{\partial W}{\partial W_y}, dh_n = \frac{\partial L}{\partial h_n} = \frac{\partial L}{\partial W}\frac{\partial W}{\partial h_n}, db_y = \frac{\partial L}{\partial b_y} = \frac{\partial L}{\partial W}\frac{\partial W}{\partial b_y}$$

And we can use gradient descent to update

$$W_y - = (\text{learning rate}) \times dW_y, b_y - = (\text{learning rate}) \times db_y$$

We now move to gates to update $U_\_, W_\_$ and $b_\_$. We will first need some temporary values, namely

$$\text{temp}_\text{f} = f_t(1 - f_t), \text{temp}_\text{i} = i_t(1 - i_t), \text{temp}_\text{o} = o_t(1 - o_t),$$

$$\text{temp}_\text{c} = 1 - \tanh^2 c_t, \text{temp}_{\tilde{\text{c}}} = 1 - \tanh^2 \tilde{c}_t$$

Note that they are all derivatives of sigmoid and tanh functions. In codes, we have

```
# Derivatives of sigmoid function for forget, input, output gate
tmpf = self.fs[t] * (1 - self.fs[t])
tmpi = self.iss[t] * (1 - self.iss[t])
tmpo = self.os[t] * (1 - self.os[t])

# Derivatives of tanh function for memory cell and intermediate gate
tmpc = 1 - np.tanh(self.cs[t])**2
tmpc_tilde = 1 - np.tanh(self.css[t])**2
```

We next have

$$h_t = \tanh c_t \odot o_t,$$

and this yields

$$\begin{cases} \frac{\partial h_t}{\partial c_t} = o_t \, \text{temp}_\text{c} \\ \frac{\partial h_t}{\partial o_t} = \tanh c_t \end{cases}$$

For the memory cell, we have

$$c_t = c_{t-1} \odot f_t + i_t \odot \tilde{c}_t,$$

and this yields

$$\begin{cases} \frac{\partial c_t}{\partial c_{t-1}} = f_t \\ \frac{\partial c_t}{\partial f_t} = c_{t-1} \\ \frac{\partial c_t}{\partial i_t} = c_t \\ \frac{\partial c_t}{\partial \tilde{c}_t} = i_t \end{cases}$$

For the gates, we have

$$\text{gate}_t = \sigma(U_{\text{gate}} h_{t-1} + W_{\text{gate}} x_t + b_{\text{gate}}),$$

and this follows that

$$\begin{cases} \frac{\partial\, \text{gate}_t}{\partial h_{t-1}} = \text{temp}_{\text{gate}}\, U_{\text{gate}} \\ \frac{\partial\, \text{gate}_t}{\partial U_{\text{gate}}} = \text{temp}_{\text{gate}}\, h_{t-1} \\ \frac{\partial\, \text{gate}_t}{\partial W_{\text{gate}}} = \text{temp}_{\text{gate}}\, x_t \\ \frac{\partial\, \text{gate}_t}{\partial b_{\text{gate}}} = (\text{sum of rows of}\, \text{temp}_o), \end{cases}$$

where gate can be forget, input, output gate or intermediate gate, and $\text{temp}_{\text{gate}}$ is the temporary value for the corresponding gate. Here are sample codes for the output gate

```
dotdht_1 = self.Uo @ tmpo
dotdUo = tmpo @ self.hs[t-1].T
dotdWo = tmpo @ self.inputs[t].T
dotdbo = np.sum(tmpo, axis=1, keepdims=True)
```

To compute, for example $dU_o = \frac{\partial L}{\partial U_o}$, we note that each h_t is a function of U_o. Hence, by the chain rule, we have

$$dU_o = \sum_t \frac{\partial L}{\partial h_t} \frac{\partial h_t}{\partial o_t} \frac{\partial o_t}{\partial U_o}$$

Because $\frac{\partial h_t}{\partial o_t}$ is computed already, the problem is now reduced to compute $\frac{\partial L}{\partial h_t}$. Because we already know $\frac{\partial L}{\partial h_n}$, the value of $\frac{\partial L}{\partial h_t}$ can be computed inductively by the value of $\frac{\partial h_{t+1}}{\partial h_t}$. Because $f_t, i_t, o_t, \tilde{c}_t$ are functions of h_{t-1} and those are used to compute h_t, we have

$$\frac{\partial h_t}{\partial h_{t-1}} = \frac{\partial h_t}{\partial c_t} \frac{\partial c_t}{\partial h_{t-1}} + \frac{\partial h_t}{\partial o_t} \frac{\partial o_t}{\partial h_{t-1}}$$

$$= \frac{\partial h_t}{\partial c_t} \left(\frac{\partial c_t}{\partial \tilde{c}_t} \frac{\partial \tilde{c}_t}{\partial h_{t-1}} + \frac{\partial c_t}{\partial f_t} \frac{\partial f_t}{\partial h_{t-1}} + \frac{\partial c_t}{\partial i_t} \frac{\partial i_t}{\partial h_{t-1}} \right) + \frac{\partial h_t}{\partial o_t} \frac{\partial o_t}{\partial h_{t-1}}$$

And inductively, we can compute

$$\frac{\partial L}{\partial h_{t-1}} = \frac{\partial L}{\partial h_t} \frac{\partial h_t}{\partial h_{t-1}}$$

In codes, we have

```
# Update dh and clip its values
dhtdht_1 = dhtdct @ dctdct_tilde @ dc_tildedht_1.T + dhtdct @ dctdft @ dftdht_1.T + dhtdct @ dctdit @ ditdht_1.T + dhtdot @ dotdht_1.T
dh = dhtdht_1 @ dh
np.clip(dh, 1e-7, 1 - 1e-7, out = dh)
```

Using this and the value of $\frac{\partial h_t}{\partial o_t}$, we can compute

$$dU_o = \sum_t \frac{\partial L}{\partial h_t} \frac{\partial h_t}{\partial o_t} \frac{\partial o_t}{\partial U_o}$$

$$dW_o = \sum_t \frac{\partial L}{\partial h_t} \frac{\partial h_t}{\partial o_t} \frac{\partial o_t}{\partial W_o}$$

$$db_o = \sum_t \frac{\partial L}{\partial h_t} \frac{\partial h_t}{\partial o_t} \frac{\partial o_t}{\partial b_o}$$

Here are codes for the computations above

```
dUo += dh @ (dhtdot.T @ dotdUo)
dWo += dh @ (dhtdot.T @ dotdWo)
dbo += dh @ (dhtdot.T @ dotdbo)
```

For forget, input, intermediate gates, we need to do a little bit more, because those are used to compute c_t. For gate $\in$ {forget gate, input gate, intermediate gate}, we have

$$d\,\text{gate}_t = \frac{\partial L}{\partial\,\text{gate}_t} = \frac{\partial L}{\partial h_t} \frac{\partial h_t}{\partial c_t} \frac{\partial c_t}{\partial\,\text{gate}_t}$$

Everything is now complete and we can compute the values of partial derivatives for gates

$$dU_{\text{gate}} = \frac{\partial L}{\partial U_{\text{gate}}} = \sum_t \frac{\partial L}{\partial h_t} \frac{\partial h_t}{\partial\,\text{gate}_t} \frac{\partial\,\text{gate}_t}{\partial U_{\text{gate}}}$$

$$dW_{\text{gate}} = \frac{\partial L}{\partial W_{\text{gate}}} = \sum_t \frac{\partial L}{\partial h_t} \frac{\partial h_t}{\partial\,\text{gate}_t} \frac{\partial\,\text{gate}_t}{\partial W_{\text{gate}}},$$

$$db_{\text{gate}} = \frac{\partial L}{\partial b_{\text{gate}}} = \sum_t \frac{\partial L}{\partial h_t} \frac{\partial h_t}{\partial\,\text{gate}_t} \frac{\partial\,\text{gate}_t}{\partial b_{\text{gate}}},$$

Here are sample codes for the forget gate

```
dUf += dh @ (dhtdct @ dctdft).T @ dftdUf
dWf += dh @ (dhtdct @ dctdft).T @ dftdWf
dbf += dh @ (dhtdct @ dctdft).T @ dftdbf
```

Finally, we can update parameters by gradient descent

```python
# Update parameters
self.Uf -= learning_rate * dUf
self.Wf -= learning_rate * dWf
self.bf -= learning_rate * dbf

self.Wi -= learning_rate * dWi
self.Ui -= learning_rate * dUi
self.bi -= learning_rate * dbi

self.Wo -= learning_rate * dWo
self.Uo -= learning_rate * dUo
self.bo -= learning_rate * dbo

self.Wc -= learning_rate * dWc
self.Uc -= learning_rate * dUc
self.bc -= learning_rate * dbc

self.by -= learning_rate * dby
self.Wy -= learning_rate * dWy
```

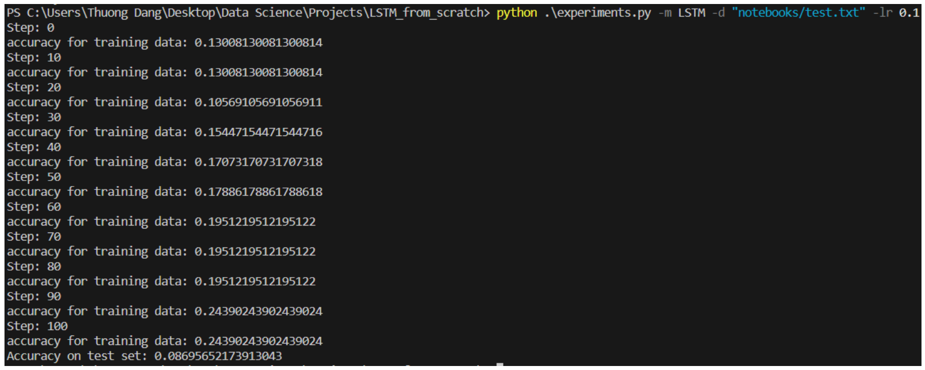
4 Experiment

Our dataset is very simple because of limited resources. It consists of 40 random sentences generated by chatGPT. Our task is to predict a word based on two previous words.

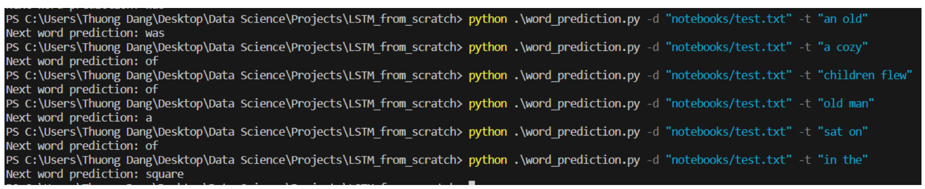
First, we created a dictionary consisting of all words in our dataset. With each sentence, we created pairs of inputs and outputs, where the inputs are two consecutive words, and outputs are the third consecutive words. For example, if we have a sentence "I am so happy", then there will be two pairs $X_0 =$ "i am", $y_0 =$ "so", and $X_1 =$ "am so", $y_1 =$ "happy".

We then transformed pairs into one-hot-encoded vectors, whose lengths are equal to the size of our dictionary. Of course there are other methods for word embeddings, such as word2vec or GloVe but it is not our main purposes. That's why we keep it simple.

Here are the results of our model



```
PS C:\Users\Thuong Dang\Desktop\Data Science\Projects\LSTM_from_scratch> python .\experiments.py -m LSTM -d "notebooks/test.txt" -lr 0.1
Step: 0
accuracy for training data: 0.13008130081300814
Step: 10
accuracy for training data: 0.13008130081300814
Step: 20
accuracy for training data: 0.10569105691056911
Step: 30
accuracy for training data: 0.15447154471544716
Step: 40
accuracy for training data: 0.17073170731707318
Step: 50
accuracy for training data: 0.17886178861788618
Step: 60
accuracy for training data: 0.1951219512195122
Step: 70
accuracy for training data: 0.1951219512195122
Step: 80
accuracy for training data: 0.1951219512195122
Step: 90
accuracy for training data: 0.24390243902439024
Step: 100
accuracy for training data: 0.24390243902439024
Accuracy on test set: 0.08695652173913043
```



```
PS C:\Users\Thuong Dang\Desktop\Data Science\Projects\LSTM_from_scratch> python .\word_prediction.py -d "notebooks/test.txt" -t "an old"
Next word prediction: was
PS C:\Users\Thuong Dang\Desktop\Data Science\Projects\LSTM_from_scratch> python .\word_prediction.py -d "notebooks/test.txt" -t "a cozy"
Next word prediction: of
PS C:\Users\Thuong Dang\Desktop\Data Science\Projects\LSTM_from_scratch> python .\word_prediction.py -d "notebooks/test.txt" -t "children flew"
Next word prediction: of
PS C:\Users\Thuong Dang\Desktop\Data Science\Projects\LSTM_from_scratch> python .\word_prediction.py -d "notebooks/test.txt" -t "old man"
Next word prediction: a
PS C:\Users\Thuong Dang\Desktop\Data Science\Projects\LSTM_from_scratch> python .\word_prediction.py -d "notebooks/test.txt" -t "sat on"
Next word prediction: of
PS C:\Users\Thuong Dang\Desktop\Data Science\Projects\LSTM_from_scratch> python .\word_prediction.py -d "notebooks/test.txt" -t "in the"
Next word prediction: square
```

As we can see, the results is quite low because of the following reasons:

- The dataset consists of random sentences and does not have a specific context.
- The size of the dataset is very small.
- The word prediction task is hard, especially with two or more words.
- The program is for educational purpose.

REFERENCES

[1] Fathi M. Salem, *Recurrent Neural Networks from Simple to Gated Architectures*, Springer 2022.

[2] Harrison Kinsley & Daniel Kukieła, *Neural Networks from Scratch*, NNFS `https://nnfs.io`