CogNETive **July 25, 2019**

Building a Long-Short Term Memory (LSTM) Unit

Vaishnavi Shrivastava

Github Link: https://github.com/vshrivas/CogNETive

Forward Propagation

Forward propagation is defined in the *forwardprop* method, which takes in h_{t-1} as h_{prev}, C_{t-1} as c_{prev}, and the current input x.
$z = [h_{t-1}, x_t]$, concatenation of h_{t-1} and x_t.

Forget Gate

The forget gate layer uses h_{t-1} and x_t to calculate a number between 0 and 1 for each value in C_{t-1}, where 0 means to completely forget and 1 means to completely remember. Sigmoid is used because it outputs a value between 0 and 1, which corresponds to an amount of information flow.

$f_t = \sigma(W_f \cdot z + b_f)$

Input Gate and Cell State

This is used to decide the values that need to be updated, and by how much they need to be updated. We also generate a set of candidate values $\tilde{C}_t$ that can be multiplied by the input gate and added to the new cell state.
$i_t = \sigma(W_i \cdot z + b_i)$

The cell state's values should be able to increase or decrease when we add the output of some function. Sigmoid output is always non-negative; meaning values in the state would only increase. The output from tanh can be positive or negative, allowing for increases and decreases in the state. That's why tanh is used to determine candidate values to get added to the internal state. On the other hand, to overcome the vanishing gradient problem, we need a function whose second derivative can sustain for a long range before going to zero. Tanh is a good function with the above property.
A tanh function ensures that the values stay between -1 and 1, thus regulating the output of the neural network.
$\tilde{C}_t = \tanh(W_C \cdot z + b_c)$

We multiply the old state by f_t, forgetting the things we decided to forget earlier. Then we add $i_t \cdot \tilde{C}_t$, the new candidate values, scaled by how much we decided to update each state value.
$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t$.

Sidenote:
In RNNs the weights are shared across the time steps. If you have a ReLU activation and the recurrence matrices enter the unstable regime (for Elman RNNs this occurs if the spectral norm of the rec. matrix is ¿ 1, for LSTMs the condition is more complicated, but still exists), the magnitude of the state will increase exponentially with the time steps during the forward pass. In turn this will increase the the magnitude of the gradients during the backward pass, the two effects can easily feed into each other, leading to numeric

overflow. Properly placed tanh functions can avoid this: at worst you will get saturation and vanishing gradients, which is better than overflow. There are some papers that used ReLUs with RNNs, but they require very careful weight initialization and low learning rates, and in general there is no benefit over LSTMs or GRUs.

Output Gate and Hidden State

The output from this timestep, h_t, will be a filtered version of the cell state, where not all of the information passes through. To decide the information that will pass through, we calculate a sigmoid layer o_t. The cell state is pushed through a tanh, to squeeze it between 1 and -1, and then multiplied by o_t to output only the parts we decided.
$o_t = \sigma(W_o \cdot z + b_o)$
$h_t = o_t \cdot tanh(C_t)$
These values are all returned in the dictionary *fresults*.

Backpropagation Through Time Derivation

`dv` $=$
$\frac{dL}{dv_t} =$
$\hat{y_t} - y_t$

`dh` $=$
$\frac{dL}{dh_t} =$
$\frac{dL}{dv_t} \cdot \frac{dv_t}{dh_t} + \frac{dL_{t+1}}{dh_t} =$
$\frac{dL}{\cancel{dv_t}} \cdot \frac{\cancel{dv_t}}{dh_t} + \frac{dL_{t+1}}{dh_t} =$
$\frac{dL}{dv_t} \cdot W_v + \frac{dL_{t+1}}{dh_t} =$
`dv` $\cdot \mathrm{W}_v$ + `dh`$_{\texttt{prev}}$
The second term is added because h_{t-1} is passed to the cell at the next timestep, and therefore has an effect on the cell in future timesteps that needs to be accounted for.

`dc` $=$
$\frac{dL}{dC_t} =$
$\frac{dL}{dh_t} \cdot \frac{dh_t}{dC_t} + \frac{dL}{dC_{t+1}} \cdot \frac{dC_{t+1}}{dC} =$
$\frac{dL}{\cancel{dh_t}} \cdot \frac{\cancel{dh_t}}{dC_t} + \frac{dL}{\cancel{dC_{t+1}}} \cdot \frac{\cancel{dC_{t+1}}}{dC} =$
$\frac{dL}{dh_t} \cdot [(o_t)(\tanh' C_t)] + \frac{dL}{dC_{t+1}} \cdot f_{t+1} =$
`dh` $\cdot [o_t(1 - \tanh^2 C_t)]$ + `dc`$_{\texttt{prev}}$
C_t impacts L by influencing h_t in the current timestep, and by influencing C_{t+1} in the next timestep, so we add both factors to find the derivative.

Forget Gate

`df` $=$
$\frac{dL}{df_t} =$
$\frac{dL}{dC_t} \cdot \frac{dC_t}{df_t} =$
$\frac{dL}{\cancel{dC_t}} \cdot \frac{\cancel{dC_t}}{df_t} =$
`dc` $\cdot C_{t-1}$

The `logit` function is the inverse of the `sigmoid` function, so applying it to f_t will yield $W_f Z + b_f$. Let

`dsigmoid` be the derivative of the sigmoid function.
$\texttt{dW}_\texttt{f} =$
$\frac{dL}{dW_f} =$
$\frac{dL}{df_t} \cdot \frac{df_t}{dW_f} =$
$\frac{dL}{\cancel{df_t}} \cdot \frac{\cancel{df_t}}{dW_f} =$
$\frac{dL}{df_t} \cdot [\sigma'(W_f Z + b_f)(Z)] =$
$\texttt{df} \cdot [\texttt{dsigmoid}(\texttt{logit}(f))] \cdot Z$

$\texttt{db}_\texttt{f} =$
$\frac{dL}{db_f} =$
$\frac{dL}{df_t} \cdot \frac{df_t}{db_f} =$
$\frac{dL}{\cancel{df_t}} \cdot \frac{\cancel{df_t}}{db_f} =$
$\frac{dL}{df_t} \cdot [\sigma'(W_f Z + b_f)(1)] =$
$\texttt{df} \cdot [\texttt{dsigmoid}(\texttt{logit}(f))]$

$\texttt{dZ}_\texttt{f} =$
$\frac{dL}{dZ} =$
$\frac{dL}{df_t} \cdot \frac{df_t}{dZ} =$
$\frac{dL}{\cancel{df_t}} \cdot \frac{\cancel{df_t}}{dZ} =$
$\frac{dL}{df_t} \cdot [\sigma'(W_f Z + b_f)(W_f)] =$
$\texttt{df} \cdot [\texttt{dsigmoid}(\texttt{logit}(f))] \cdot W_f$

In the code, the repeated multiplication of `df` and $[\texttt{dsigmoid}(\texttt{logit}(f))]$ is eliminated by setting
$\texttt{df = df} \cdot [\texttt{dsigmoid}(\texttt{logit}(f))]$.

Input Gate

The `logit` function is the inverse of the `sigmoid` function, so applying it to i_t will yield $W_i Z + b_i$. Let `dsigmoid` be the derivative of the sigmoid function.
$\texttt{di} =$
$\frac{dL}{di_t} =$
$\frac{dL}{dC_t} \cdot \frac{dC_t}{di_t} =$
$\frac{dL}{\cancel{dC_t}} \cdot \frac{\cancel{dC_t}}{di_t} =$
$\texttt{dc} \cdot \tilde{C_t}$

$\texttt{dW}_\texttt{i} =$
$\frac{dL}{dW_i} =$
$\frac{dL}{di_t} \cdot \frac{di_t}{dW_i} =$
$\frac{dL}{\cancel{di_t}} \cdot \frac{\cancel{di_t}}{dW_i} =$
$\frac{dL}{di_t} \cdot [\sigma'(W_i Z + b_i)(Z)] =$
$\texttt{di} \cdot [\texttt{dsigmoid}(\texttt{logit}(i_t))] \cdot Z$

$\texttt{db}_\texttt{i} =$
$\frac{dL}{db_i} =$
$\frac{dL}{di_t} \cdot \frac{di_t}{db_i} =$
$\frac{dL}{\cancel{di_t}} \cdot \frac{\cancel{di_t}}{db_f} =$

$\frac{dL}{di_t} \cdot [\sigma'(W_i Z + b_i)(1)] =$
`di` $\cdot$ [`dsigmoid`(`logit`(i_t))]

$\mathtt{dZ_i} =$
$\frac{dL}{dZ} =$
$\frac{dL}{di_t} \cdot \frac{di_t}{dZ} =$
$\frac{dL}{\cancel{di_t}} \cdot \frac{\cancel{di_t}}{dZ} =$
$\frac{dL}{di_t} \cdot [\sigma'(W_i Z + b_i)(W_i)] =$
`df` $\cdot$ [`dsigmoid`(`logit`(i_t))] $\cdot W_i$

In the code, the repeated multiplication of `di` and [`dsigmoid`(`logit`(i_t))] is eliminated by setting `di = di` $\cdot$ [`dsigmoid`(`logit`(i_t))].

Output Gate

The `logit` function is the inverse of the `sigmoid` function, so applying it to o_t will yield $W_o Z + b_o$. Let `dsigmoid` be the derivative of the sigmoid function.
`do` $=$
$\frac{dL}{do_t} =$
$\frac{dL}{dh_t} \cdot \frac{dh_t}{do_t} =$
$\frac{dL}{\cancel{dh_t}} \cdot \frac{\cancel{dh_t}}{do_t} =$
`dh` $\cdot \tanh C_t$

$\mathtt{dW_o} =$
$\frac{dL}{dW_o} =$
$\frac{dL}{do_t} \cdot \frac{do_t}{dW_o} =$
$\frac{dL}{\cancel{do_t}} \cdot \frac{\cancel{do_t}}{dW_o} =$
$\frac{dL}{do_t} \cdot [\sigma'(W_o Z + b_o)(Z)] =$
`do` $\cdot$ [`dsigmoid`(`logit`(o_t))] $\cdot Z$

$\mathtt{db_o} =$
$\frac{dL}{db_o} =$
$\frac{dL}{do_t} \cdot \frac{do_t}{db_o} =$
$\frac{dL}{\cancel{do_t}} \cdot \frac{\cancel{do_t}}{db_o} =$
$\frac{dL}{do_t} \cdot [\sigma'(W_o Z + b_o)(1)] =$
`do` $\cdot$ [`dsigmoid`(`logit`(o_t))]

$\mathtt{dZ_o} =$
$\frac{dL}{dZ} =$
$\frac{dL}{do_t} \cdot \frac{do_t}{dZ} =$
$\frac{dL}{\cancel{do_t}} \cdot \frac{\cancel{do_t}}{dZ} =$
$\frac{dL}{do_t} \cdot [\sigma'(W_o Z + b_o)(W_o)] =$
`do` $\cdot$ [`dsigmoid`(`logit`(o_t))] $\cdot W_o$

In the code, the repeated multiplication of `do` and [`dsigmoid`(`logit`(o_t))] is eliminated by setting `do = do` $\cdot$ [`dsigmoid`(`logit`(o_t))].

Candidate Gate

The `arctanh` function is the inverse of the `tanh` function, so applying it to c_{bar} will yield $W_{cbar}Z + b_{cbar}$. Let `dtanh` be the derivative of the tanh function.

$\mathtt{dc}_{\mathtt{bar}} =$
$\frac{dL}{dC_{bar_t}} =$
$\frac{dL}{dC_t} \cdot \frac{dC_t}{dC_{bar_t}} =$
$\frac{dL}{\cancel{dC_t}} \cdot \frac{\cancel{dC_t}}{dC_{bar_t}} =$
$\mathtt{dc} \cdot i_t$

$\mathtt{dW}_{\mathtt{cbar}} =$
$\frac{dL}{dW_{cbar}} =$
$\frac{dL}{dC_{bar_t}} \cdot \frac{dC_{bar_t}}{dW_{cbar}} =$
$\frac{dL}{\cancel{dC_{bar_t}}} \cdot \frac{\cancel{dC_{bar_t}}}{dW_{cbar}} =$
$\frac{dL}{dC_{bar_t}} \cdot [\tanh'(W_{cbar}Z + b_{cbar})(Z)] =$
$\mathtt{dc}_{\mathtt{bar}} \cdot [\mathtt{dtanh}(\mathtt{arctanh}(c_{bar}))] \cdot Z$

$\mathtt{db}_{\mathtt{cbar}} =$
$\frac{dL}{db_{cbar}} =$
$\frac{dL}{dC_{bar_t}} \cdot \frac{dC_{bar_t}}{db_{cbar}} =$
$\frac{dL}{\cancel{dC_{bar_t}}} \cdot \frac{\cancel{dC_{bar_t}}}{db_{cbar}} =$
$\frac{dL}{dC_{bar_t}} \cdot [\tanh'(W_{cbar}Z + b_{cbar})(1)] =$
$\mathtt{dc}_{\mathtt{bar}} \cdot [\mathtt{dtanh}(\mathtt{arctanh}(c_{bar}))]$

$\mathtt{dZ}_{\mathtt{cbar}} =$
$\frac{dL}{dZ} =$
$\frac{dL}{dC_{bar_t}} \cdot \frac{dC_{bar_t}}{dZ} =$
$\frac{dL}{\cancel{dC_{bar_t}}} \cdot \frac{\cancel{dC_{bar_t}}}{dZ} =$
$\frac{dL}{dC_{bar_t}} \cdot [\tanh'(W_{cbar}Z + b_{cbar})(W_{cbar})] =$
$\mathtt{dc}_{\mathtt{bar}} \cdot [\mathtt{dtanh}(\mathtt{arctanh}(c_{bar}))] \cdot W_{cbar}$

In the code, the repeated multiplication of $\mathtt{dc}_{\mathtt{bar}}$ and $[\mathtt{dtanh}(\mathtt{arctanh}(c_{bar}))]$ is eliminated by setting
$\mathtt{dc}_{\mathtt{bar}}$ `=` $\mathtt{dc}_{\mathtt{bar}} \cdot [\mathtt{dtanh}(\mathtt{arctanh}(c_{bar}))]$.

Training Process

The network is trained in steps, where in each step, first we run forward propagation through all of the training examples, and store the results of forward prop at each timestep in a dictionary and accumulate the overall loss by comparing the output of each timestep, with the expected result at each timestep. The we perform backpropagation through each timestep, starting from the last timestep and going backwards to the first, calculating and accumulating the gradients of the weight matrices and biases at each step, and sending back the calculated values of $\mathtt{dh}_{\mathtt{next}}$ and $\mathtt{dc}_{\mathtt{next}}$ to the previous timestep to be used in the gradient calculations there, since these are the values that link backprop across timesteps.