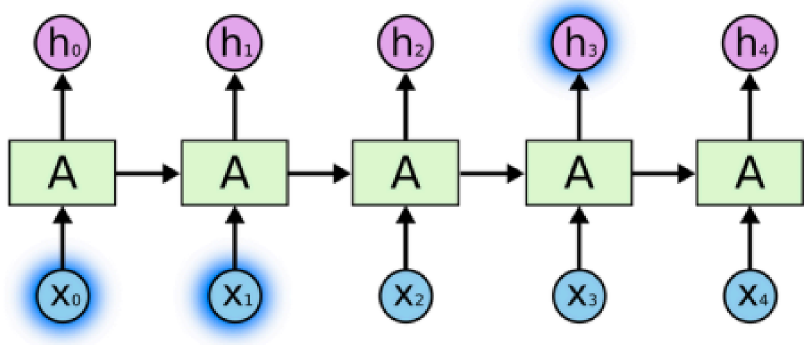
L11 LSTM

LSTM vs Standard Neural networks

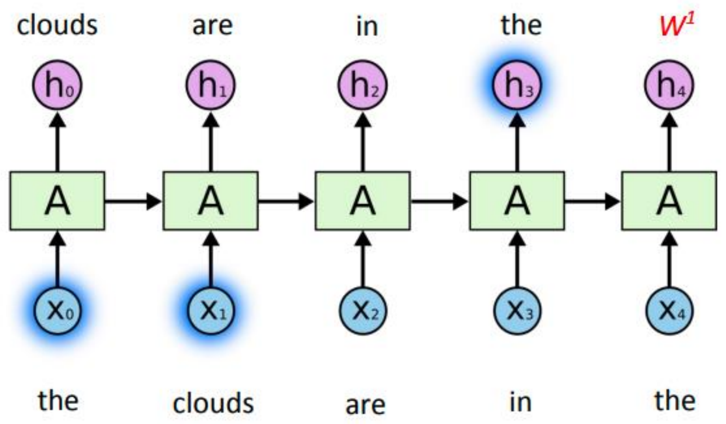
- Goal is to predict the current output, by incorporating the past behavior and the current input.
- A standard neural network cannot be used for prediction of sequential data
  - It predicts output at each time step independent of the past time steps. Though during training all the time steps can be considered, but it does not make use of the ordering in the data
  - A standard neural network can assume to have a memory, in the sense that it stores relevant past information through the training parameters. But it ignores ordering in the data.

LSTM Origin – Short-term dependency

- Sometimes, we only need to look at recent information to perform the present task.
- Ex: Consider a language model trying to predict the next word based on the previous ones.
- If we are trying to predict the last word in Ex1: “the clouds are in the sky” we do not need any further context – it is obvious the next word is going to be sky. Ex2: He opened the door and saw a bright light
- In such cases, where the gap between the relevant information and the place that it is needed is small, RNNs can learn to use the past information.
- “the clouds are in the sky” . sky → clouds [3 units apart]



h_3 depends on x_0 and x_1

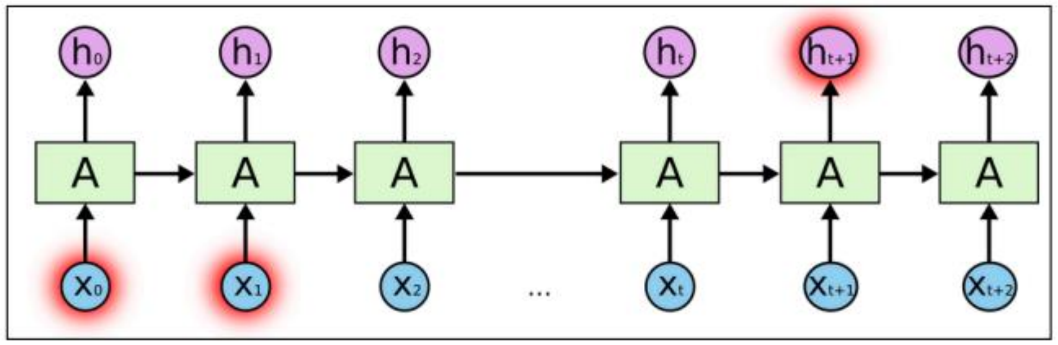


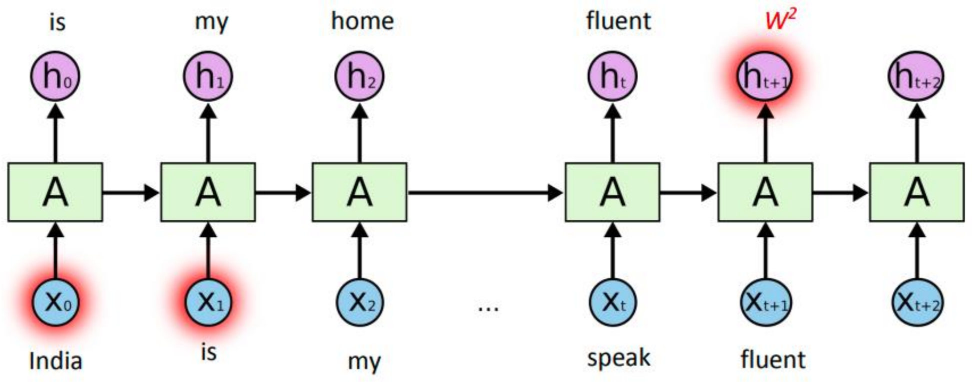
LSTM Origin - Long-term dependency

- But there are also cases where we need more context.
- Consider trying to predict the last word in the text Ex1: “I grew up in France… I speak fluent French.” Ex2: “India is my home country. I can speak fluent Hindi.”
- Recent information suggests that the next word is probably the name of a language
- But if we want to narrow down which language, we need the context of France, from further back.
- It is entirely possible for the gap between the relevant information and the point where it is needed to become very large.
- As that gap grows, RNNs become unable to learn to connect the information.
- LSTMs proposed in 1997 remain the most popular solution for overcoming this short coming of the RNNs.

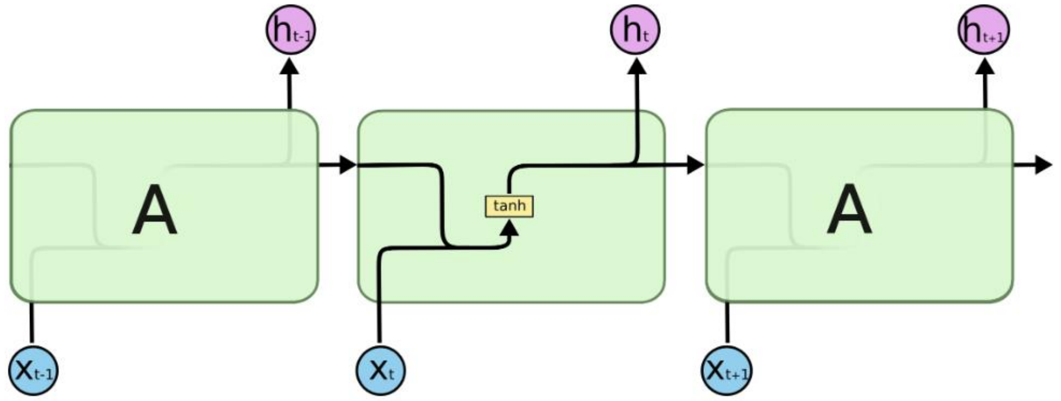
LSTM Origin - Long-term dependency

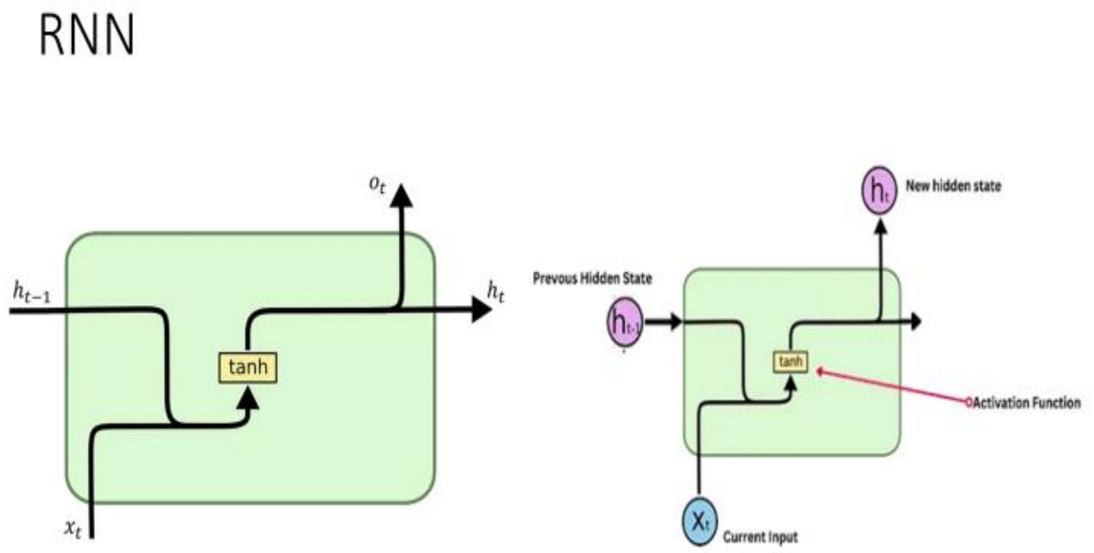
- RNNs work upon the fact that the result of an information is dependent on its previous state or previous n time steps.
- They have difficulty in learning long range dependencies.
- Ex2: The man who ate my pizza has purple hair.
- Note: purple hair is for the man and not the pizza. So this is a long dependency
- Example 1: "India is my home country. I can speak fluent Hindi." Hindi → India [9 units apart]





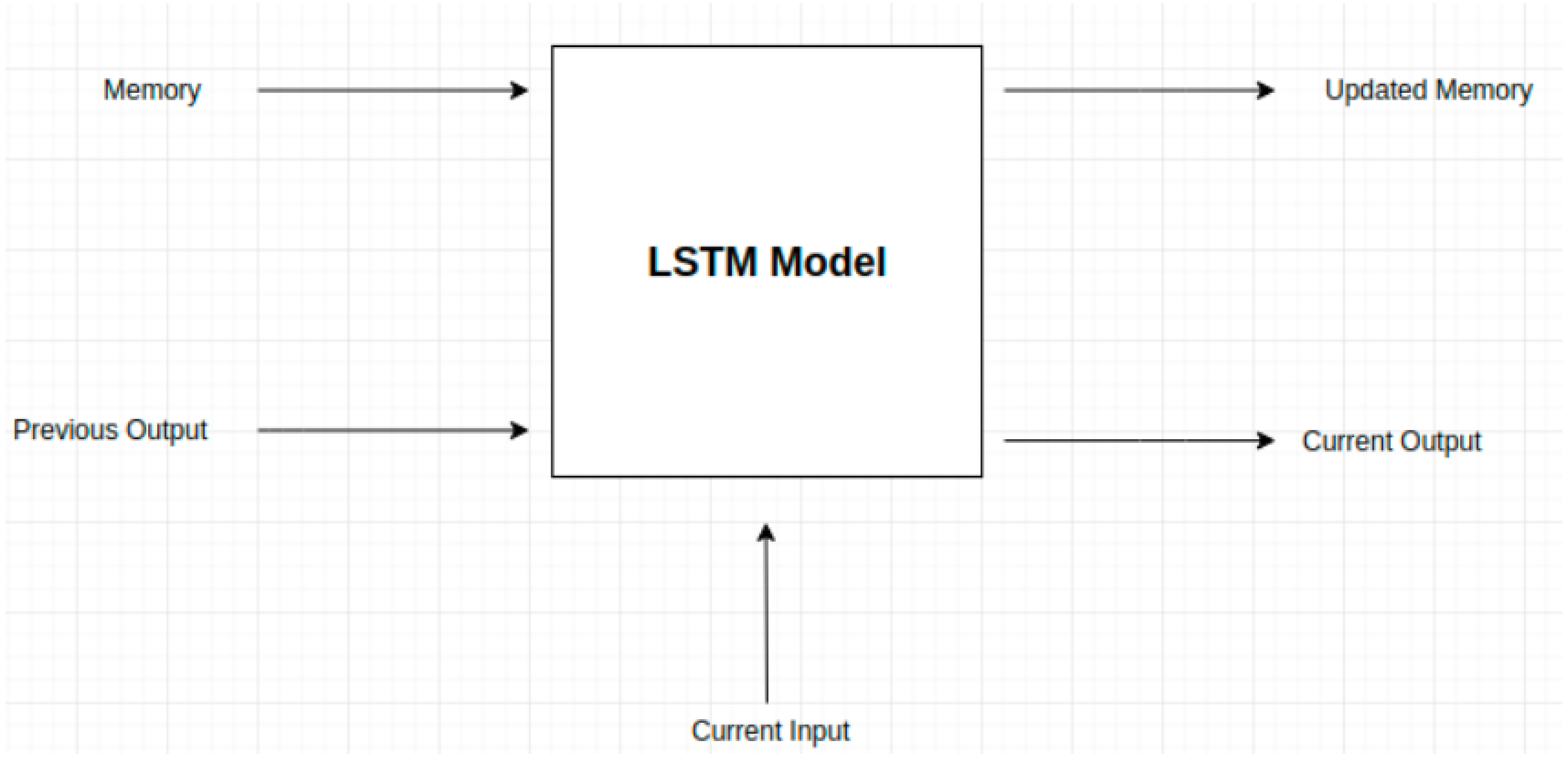
RNN



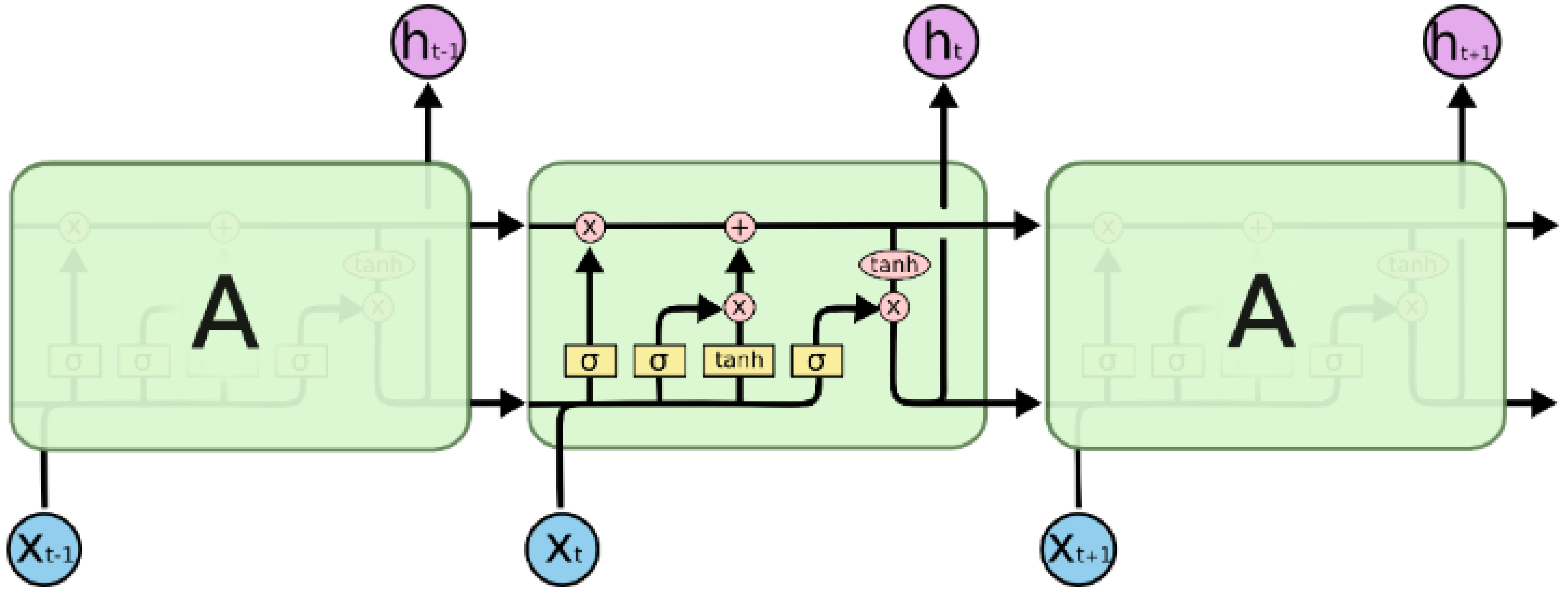


All recurrent neural networks have the form of a chain of repeating modules of neural network.
In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer.

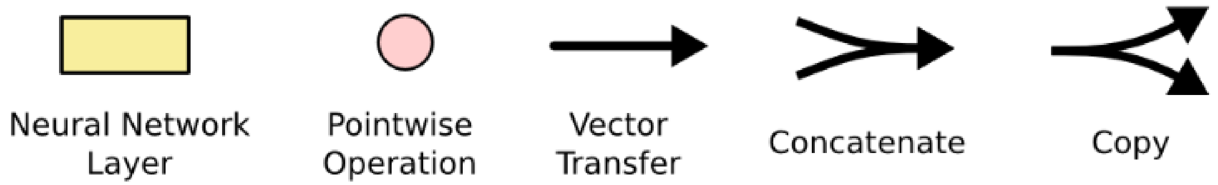
Building blocks of the LSTM model



Building blocks of the LSTM model



LSTMs also have this chain like structure
But the repeating module has a different structure.
Instead of having a single neural network layer, there are four layers



Building blocks of the LSTM model

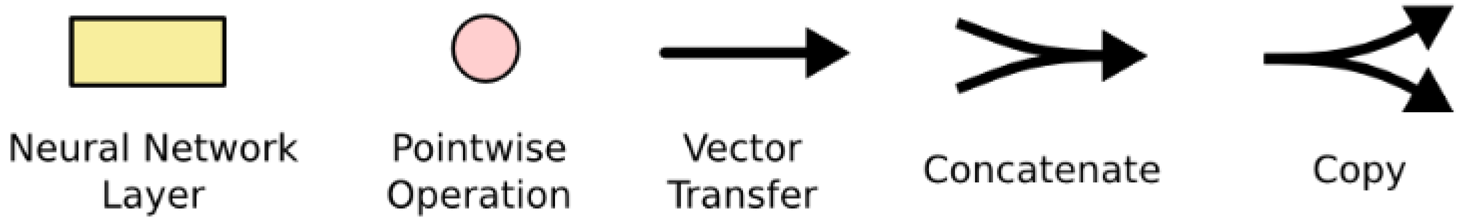
- LSTM has
- Three inputs(Ct-1, ht-1, Xt) - cell state(Ct-1), previous hidden state(ht-1), and current input(Xt) and
- Three outputs(Ct, ht, Ot), cell state(Ct), hidden state(ht), and current output(Ot).

- Note: The hidden state branches out to current output using a softmax layer.

Building blocks of the LSTM model

- Cell State: This line basically is the memory layer which contains the context across the inputs.
- Hidden State: This layer is similar to what we had in RNN for feeding previous input's output to the newer input.
- Forget Gate: Decides whether because of new X, should we remove previous memory of the network. For example — "Ram is a boy. Rama is a girl". When Rama will be fed as an input, previous context of Ram should be forgotten.
- Input Gate: Decides whether because of new X, should we update the previous memory of the network. For example — "Ram is a boy. Rama is a girl". When Rama will be fed as an input, previous context of person should be updated by Rama.
- Output Gate: Each input should give output based on current input, previous layer's output and persisted memory cell. This gate is responsible for the same.

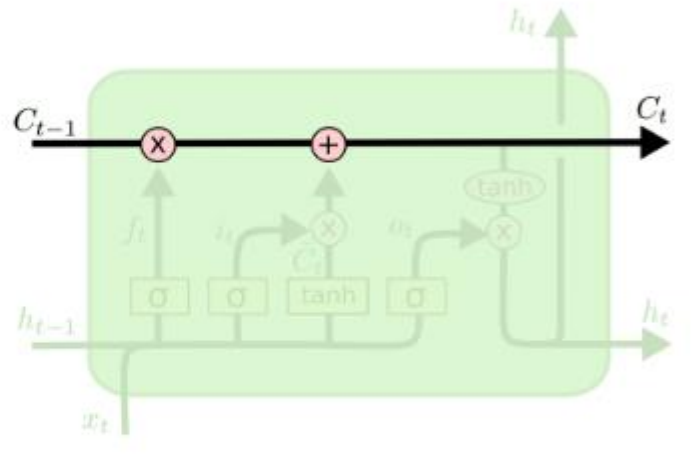
Convention



- Each line carries an entire vector, from the output of one node to the inputs of others.
- The pink circles represent pointwise operations, like vector addition
- yellow boxes are learned neural network layers.
- Lines merging denote concatenation,
- A line forking denote its content being copied and the copies going to different locations.

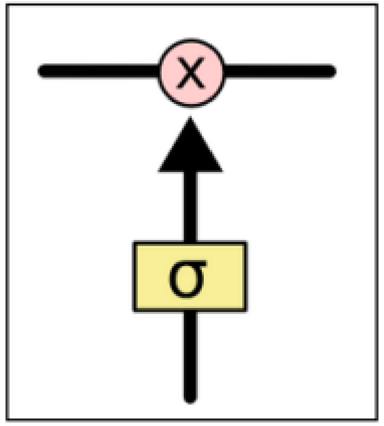
Building blocks of the LSTM model

- Cell state is represented in our diagram by the long horizontal line that runs through the top of the diagram.
- The cell state purpose is to decide what information to carry forward from the different observations that the network is trained on.
- information flows along it unchanged. The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.



Building blocks of the LSTM model

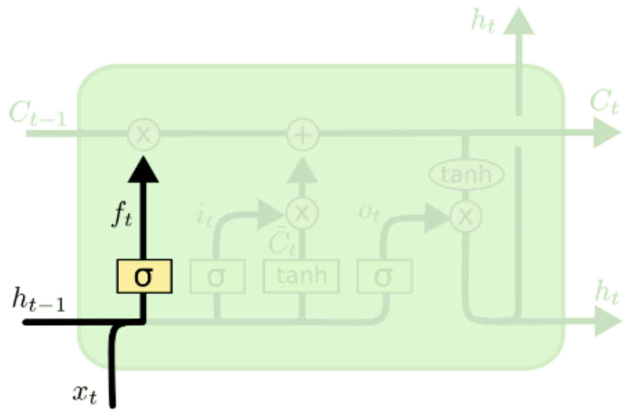
- Information can be added to or removed from the cell state in LSTM and is regulated by gates.
- These gates optionally let the information flow in and out of the cell.
- It contains a pointwise multiplication operation and a sigmoid neural net layer that assist the mechanism.
- The sigmoid layer gives out numbers between zero and one, where
    - zero means ‘nothing should be let through’, and
    - one means ‘everything should be let through’.
- LSTM has three of these Sigmoid gates, to protect and control cell state



- Note: In LSTM equations, * represents the element wise multiplication of the vectors

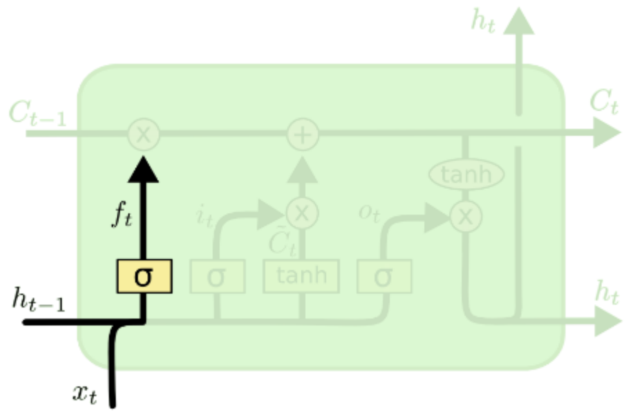
Step-by-Step LSTM Walk Through

- Forget Gate: Amount of memory it should forget
- The first step in LSTM is to decide what information we are going to throw away from the cell state.
- This decision is made by a sigmoid layer called the “forget gate layer.” It looks at ht−1 and xt , and outputs a number between 0 and 1 for each number in the cell state Ct−1 .
- 1 represents “completely keep this” while a 0 represents “completely get rid of this.”
- Ex: Assume a character level language model, to predict the next word based on all the previous ones. In such a problem, the cell state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject.



Step-by-Step LSTM Walk Through

- Forget Gate: Amount of memory it should forget
- The first step in LSTM is to decide what information we are going to throw away from the cell state.
- This decision is made by a sigmoid layer called the “forget gate layer.” It looks at ht−1 and xt , and outputs a number between 0 and 1 for each number in the cell state Ct−1 .
- 1 represents “completely keep this” while a 0 represents “completely get rid of this.”
- Ex: Assume a character level language model, to predict the next word based on all the previous ones. In such a problem, the cell state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject.



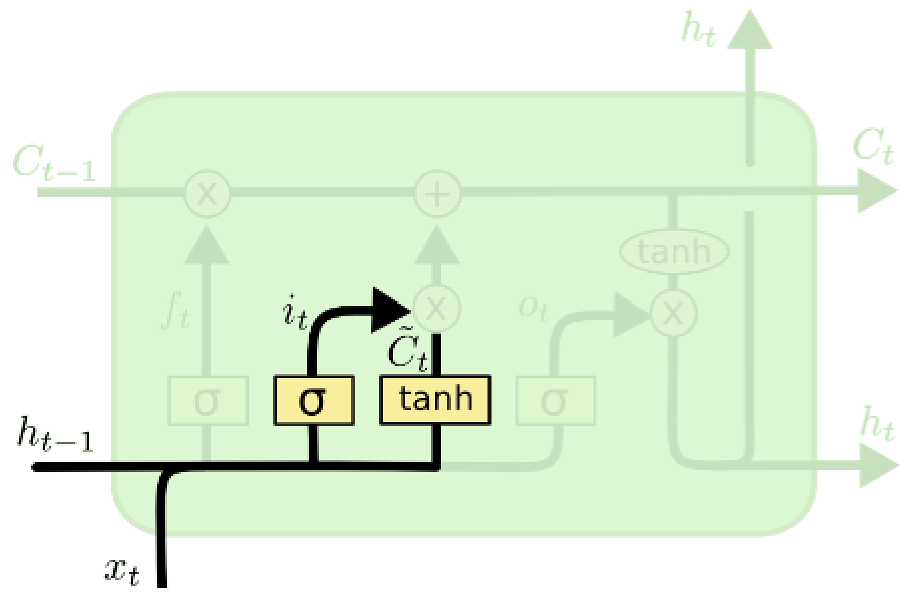
$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf})$$

Step-by-Step LSTM Walk Through

- Input Gate: Amount of new information it should memorize
- The next step is to decide what new information we are going to store in the cell state.
- This has two parts. First, a sigmoid layer called the “input gate layer” decides which values we will update. Next, a tanh layer creates a vector of new candidate values, $C^{\sim}_{t}$ that could be added to the state.
- In the next step, combine these two to create an update to the state.
- Ex: In the example of our language model, we would want to add the gender of the new subject to the cell state, to replace the old one we are forgetting.
- Note: To get the memory vector for the current timestamp the candidate $C^{\sim}_{t}$ is calculated. It is called cell gate and denoted as g_{t} in PyTorch

Step-by-Step LSTM Walk Through

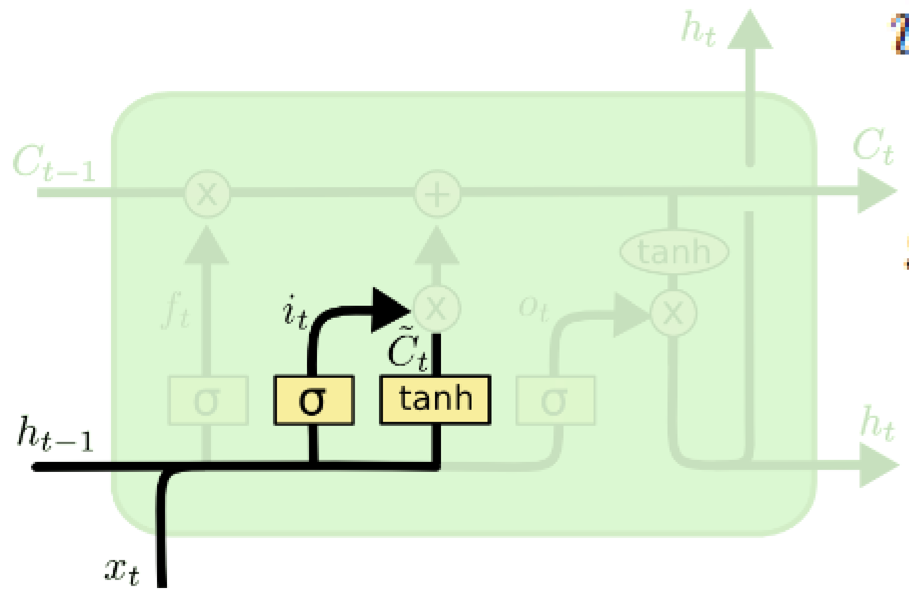
Input Gate: Amount of new information it should memorize



Note: To get the memory vector for the current timestamp the candidate $C^{\sim}_{t}$ is calculated. It is called cell gate and denoted as g_{t} in PyTorch

Step-by-Step LSTM Walk Through

Input Gate: Amount of new information it should memorize



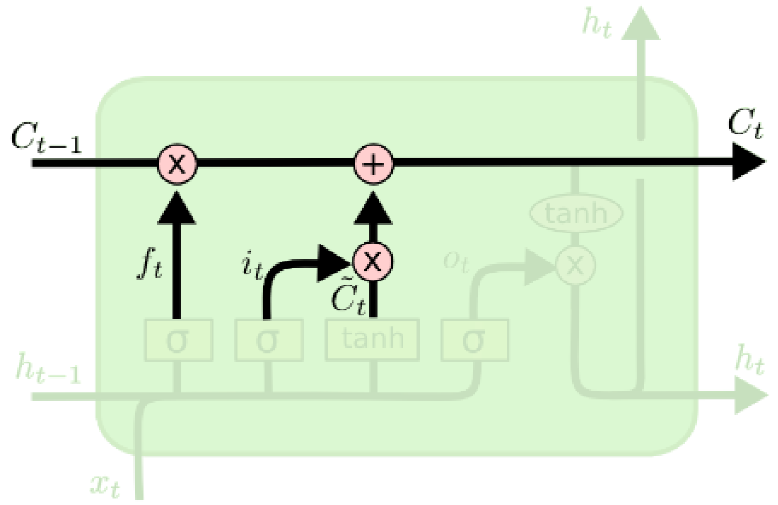
$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi})$$

$$g_t = \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg})$$

Note: To get the memory vector for the current timestamp the candidate $C^{\sim}_{t}$ is calculated. It is called cell gate and denoted as g_{t} in PyTorch

Step-by-Step LSTM Walk Through

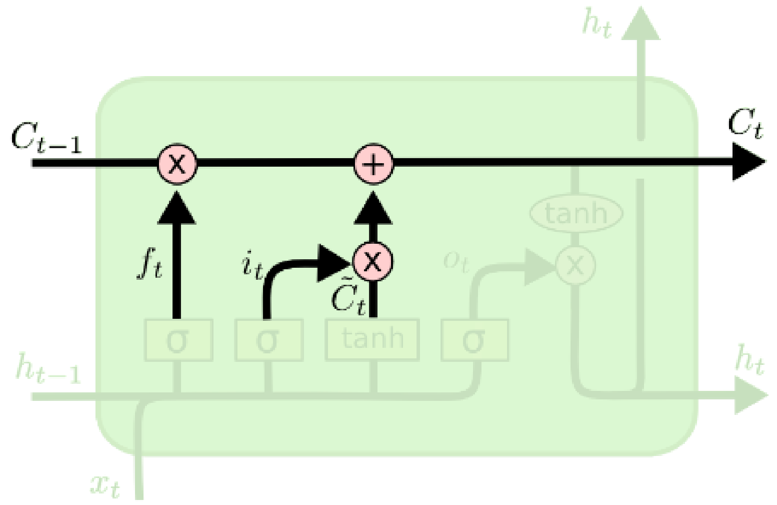
- Update the old cell state, Ct−1 , into the new cell state Ct
- Multiply the old state by ft, forgetting the things we decided to forget earlier. Then we add $i_t * \tilde{C}_t$
- This is the new candidate values, scaled by how much we decided to update each state value.
- Ex: In the case of the language model, this is where we would actually drop the information about the old subject’s gender and add the new information, as we decided in the previous steps.



Note: From the above equation, see that at any timestamp, cell state knows that what it needs to forget from the previous state(i.e $f_t \odot c_{t-1}$ and what it needs to consider from the current timestamp (i.e $i_t \odot g_t$)

Step-by-Step LSTM Walk Through

- Update the old cell state, Ct−1 , into the new cell state Ct
- Multiply the old state by ft, forgetting the things we decided to forget earlier. Then we add $i_t * C^{\sim}_t$
- This is the new candidate values, scaled by how much we decided to update each state value.
- Ex: In the case of the language model, this is where we would actually drop the information about the old subject’s gender and add the new information, as we decided in the previous steps.



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

Note: From the above equation, see that at any timestamp, cell state knows that what it needs to forget from the previous state(i.e $f_t \odot c_{t-1}$ and what it needs to consider from the current timestamp (i.e $i_t \odot g_t$)

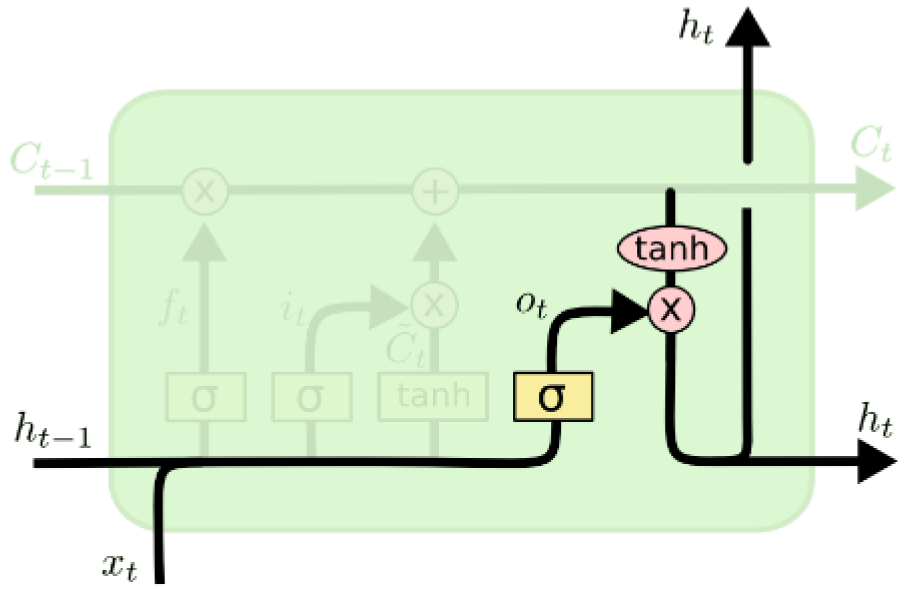
Step-by-Step LSTM Walk Through

- Output Gate: Amount of information it should pass to next unit
- Finally, we need to decide what we are going to output.
- This output will be based on our cell state, but will be a filtered version.
- First, we run a sigmoid layer which decides what parts of the cell state we are going to output.
- Then, we put the cell state through tanh (to push the values to be between -1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

- Ex: For the language model example, since it just saw a subject, it might want to output information relevant to a verb, in case that's what is coming next.
- It might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that is what follows next.

Step-by-Step LSTM Walk Through

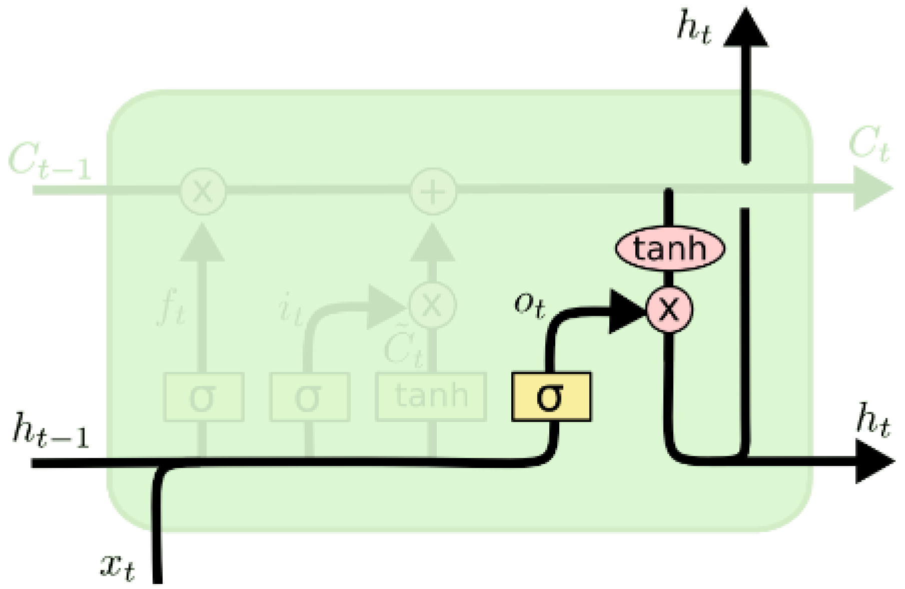
- Output Gate: Amount of information it should pass to next unit



Note: We can pass h_t the output from current LSTM block through the Softmax layer to get the predicted output y_t from the current block.

Step-by-Step LSTM Walk Through

- Output Gate: Amount of information it should pass to next unit

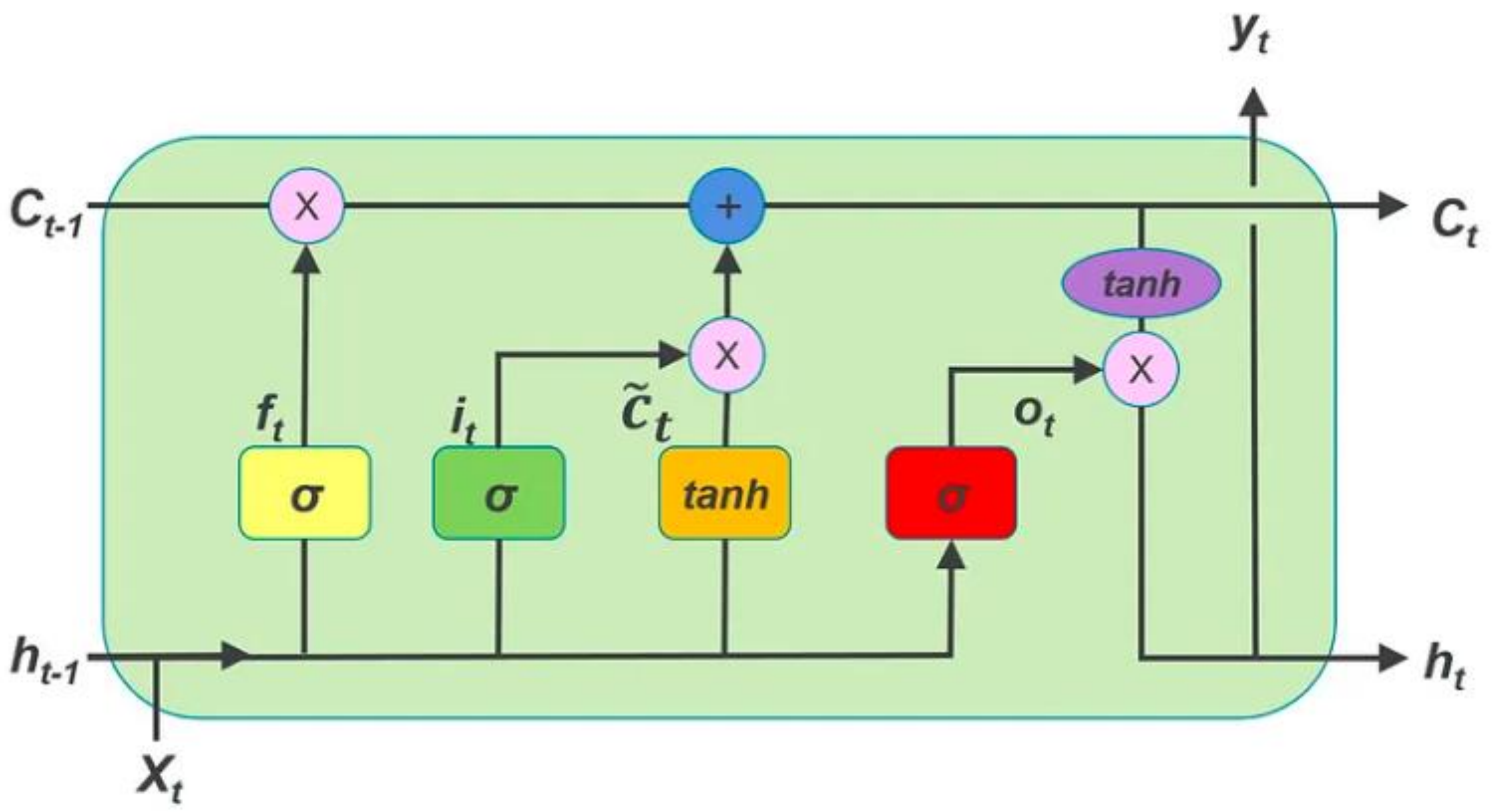


$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho})$$

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

Note: We can pass h_t the output from current LSTM block through the Softmax layer to get the predicted output y_t from the current block.

A block of LSTM at any timestamp {t}



We can pass h_t the output from current LSTM block through the Softmax layer to get the predicted output y_t from the current block.

LSTM Equations – PyTorch nn. LSTM()

torch.nn.LSTM(*input_size, hidden_size, num_layers=1, bias=True, batch_first=False, dropout=0.0, bidirectional=False*)

torch.nn.RNN(input_size, hidden_size, num_layers=1, nonlinearity='tanh', bias=True, batch_first=False, dropout=0.0, bidirectional=False)

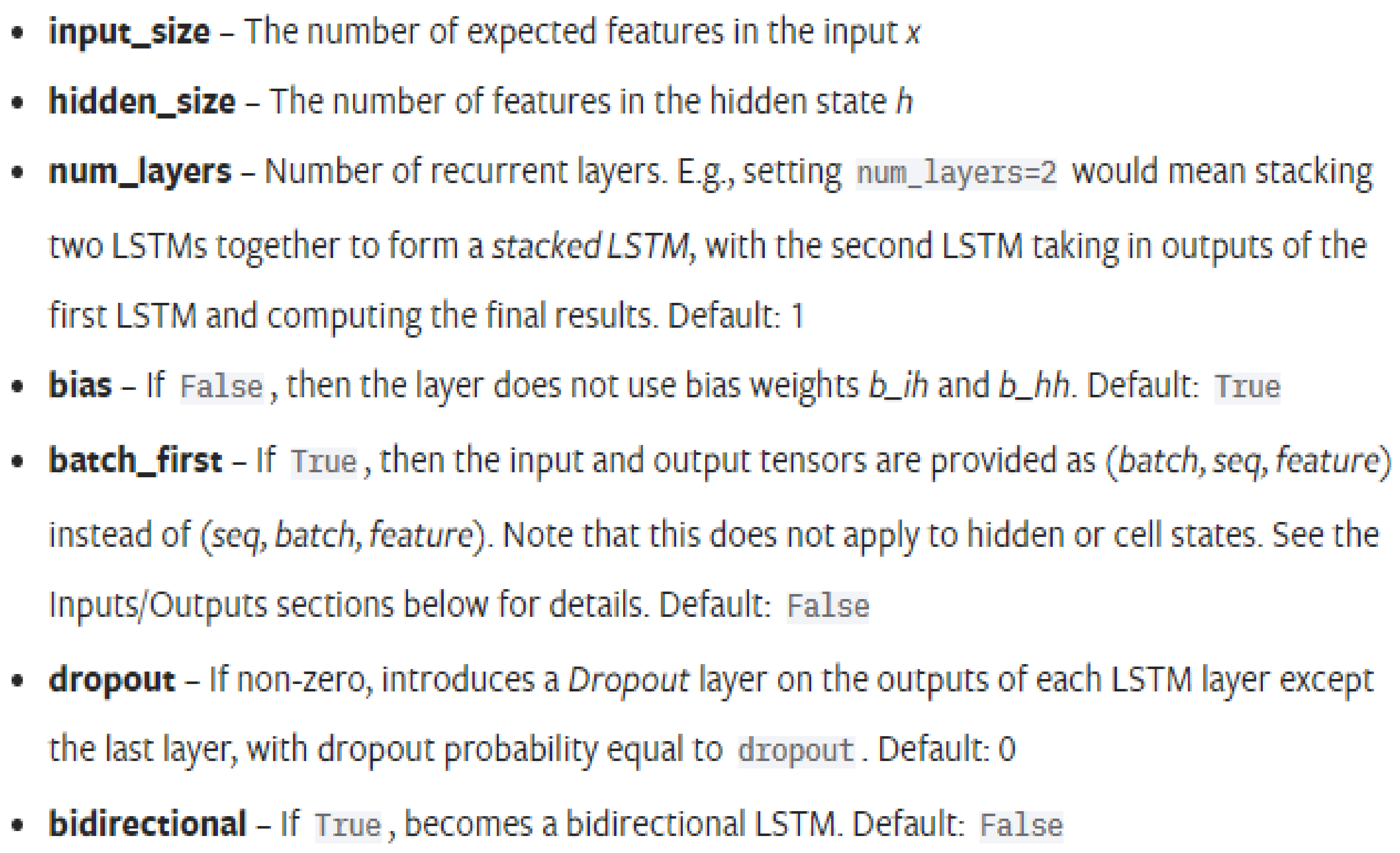
Apply a multi-layer long short-term memory (LSTM) to an input sequence. For each element in the input sequence, each layer computes the following function:

$$
\begin{aligned}
i_t &= \sigma(W_{ii} x_t + b_{ii} + W_{hi} h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if} x_t + b_{if} + W_{hf} h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig} x_t + b_{ig} + W_{hg} h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io} x_t + b_{io} + W_{ho} h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

where h_t is the hidden state at time t, c_t is the cell state at time t, x_t is the input at time t, h_{t-1} is the hidden state of the layer at time *t-1* or the initial hidden state at time *0*, and i_t, f_t, g_t, o_t are the input, forget, cell, and output gates, respectively. σ is the sigmoid function, and $\odot$ is the Hadamard product.

Hadmard product $\odot$ is element-wise multiplication, and . denotes matrix multiplication

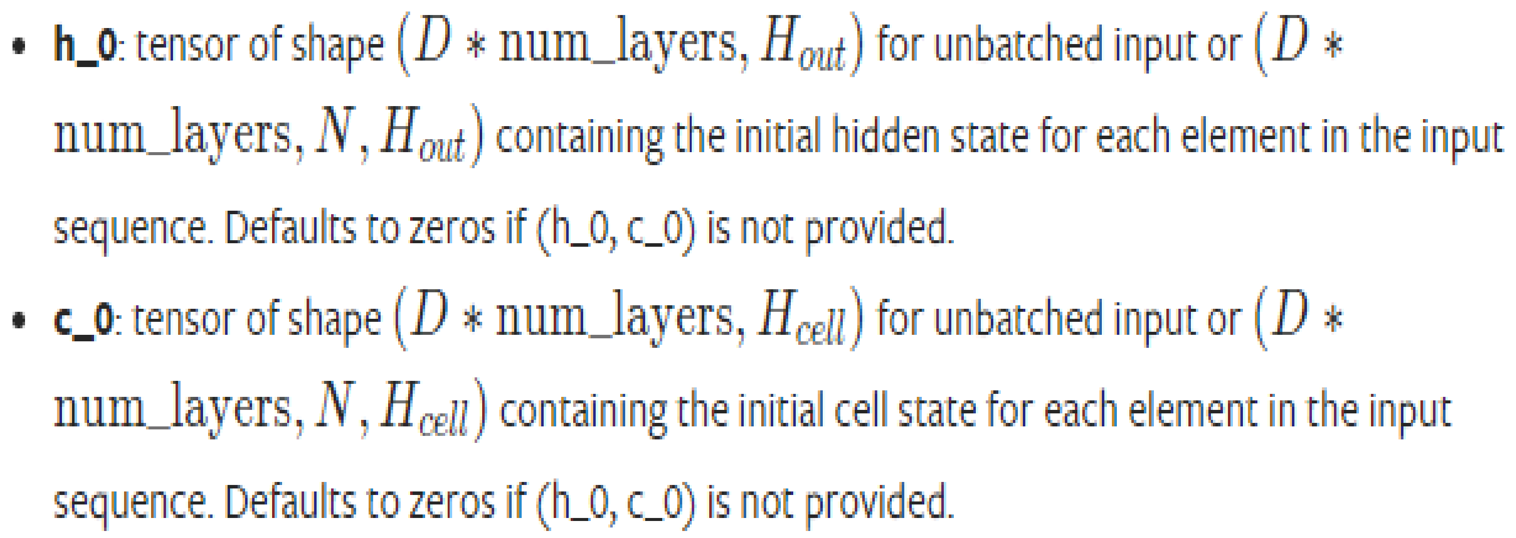
LSTM Equations – PyTorch nn. LSTM()



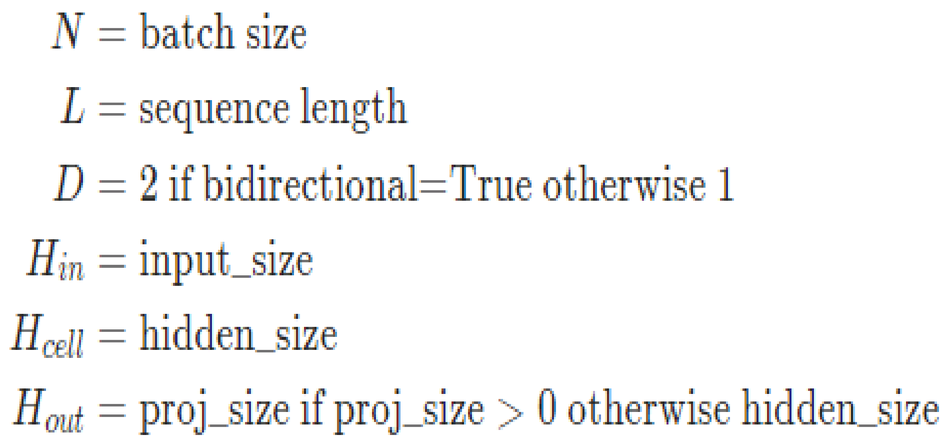
- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two LSTMs together to form a *stacked LSTM*, with the second LSTM taking in outputs of the first LSTM and computing the final results. Default: 1
- **bias** – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **batch_first** – If `True`, then the input and output tensors are provided as (*batch, seq, feature*) instead of (*seq, batch, feature*). Note that this does not apply to hidden or cell states. See the Inputs/Outputs sections below for details. Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional LSTM. Default: `False`

LSTM Equations – PyTorch nn. LSTM()

- Inputs: input, (h_0, c_0)
- input: tensor of shape (L, Hin) for unbatched input
- (L,N,Hin) when batch_first=False
- or (N,L, Hin) when batch_first=True containing the features of the input sequence.



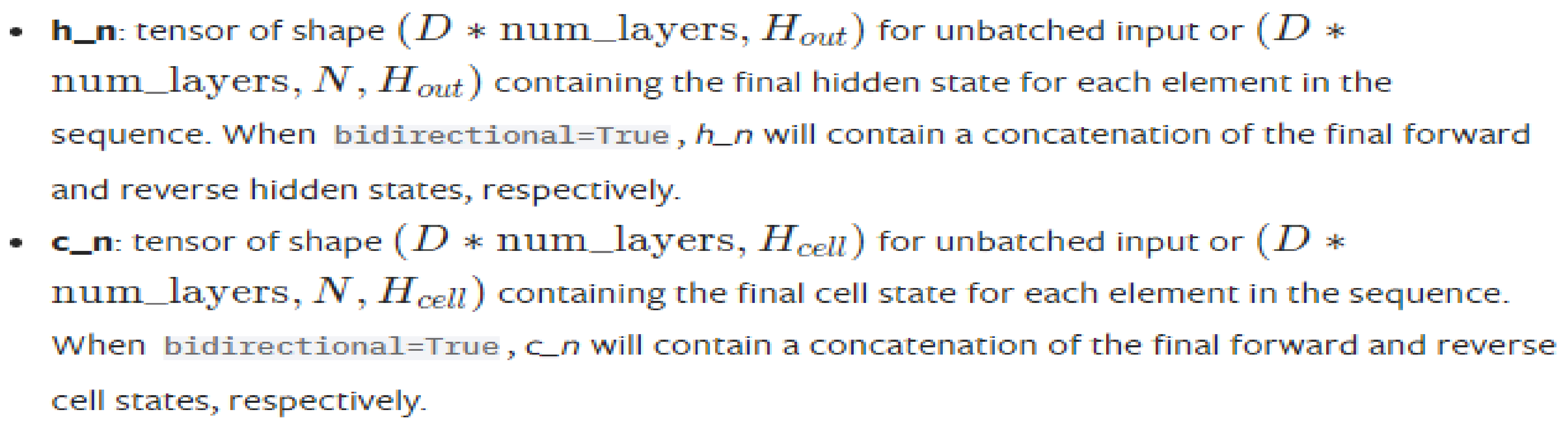
- **h_0**: tensor of shape $(D * \text{num\_layers}, H_{out})$ for unbatched input or $(D * \text{num\_layers}, N, H_{out})$ containing the initial hidden state for each element in the input sequence. Defaults to zeros if (h_0, c_0) is not provided.
- **c_0**: tensor of shape $(D * \text{num\_layers}, H_{cell})$ for unbatched input or $(D * \text{num\_layers}, N, H_{cell})$ containing the initial cell state for each element in the input sequence. Defaults to zeros if (h_0, c_0) is not provided.



$$
\begin{aligned}
N &= \text{batch size} \\
L &= \text{sequence length} \\
D &= 2 \text{ if bidirectional=True otherwise } 1 \\
H_{in} &= \text{input\_size} \\
H_{cell} &= \text{hidden\_size} \\
H_{out} &= \text{proj\_size if proj\_size} > 0 \text{ otherwise hidden\_size}
\end{aligned}
$$

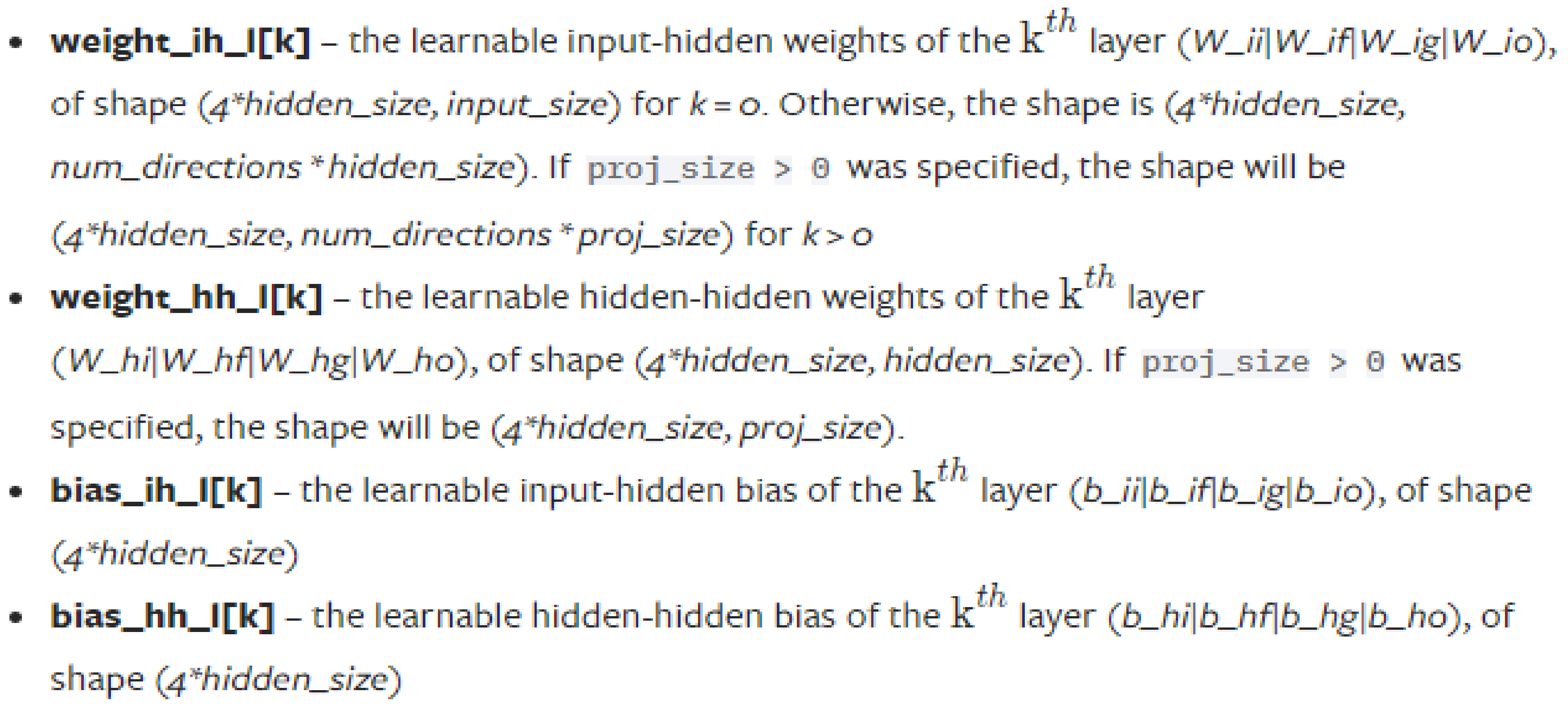
LSTM Equations – PyTorch nn. LSTM()

- Outputs: output, h_n
- output: tensor of shape
- (L, D∗H_{out}) for unbatched input,
- (L, N, D∗ H_{out}) when batch_first=False or
- (N, L, D∗H_{out}) when batch_first=True containing the output features (h_t) from the last layer of the RNN, for each t



- **h_n**: tensor of shape $(D * \text{num\_layers}, H_{out})$ for unbatched input or $(D * \text{num\_layers}, N, H_{out})$ containing the final hidden state for each element in the sequence. When `bidirectional=True`, *h_n* will contain a concatenation of the final forward and reverse hidden states, respectively.
- **c_n**: tensor of shape $(D * \text{num\_layers}, H_{cell})$ for unbatched input or $(D * \text{num\_layers}, N, H_{cell})$ containing the final cell state for each element in the sequence. When `bidirectional=True`, *c_n* will contain a concatenation of the final forward and reverse cell states, respectively.

LSTM Equations – PyTorch nn. LSTM()



- **weight_ih_l[k]** – the learnable input-hidden weights of the k^{th} layer (*W_ii|W_if|W_ig|W_io*), of shape (*4*hidden_size, input_size*) for $k = 0$. Otherwise, the shape is (*4*hidden_size, num_directions *hidden_size*). If `proj_size > 0` was specified, the shape will be (*4*hidden_size, num_directions *proj_size*) for $k > 0$
- **weight_hh_l[k]** – the learnable hidden-hidden weights of the k^{th} layer (*W_hi|W_hf|W_hg|W_ho*), of shape (*4*hidden_size, hidden_size*). If `proj_size > 0` was specified, the shape will be (*4*hidden_size, proj_size*).
- **bias_ih_l[k]** – the learnable input-hidden bias of the k^{th} layer (*b_ii|b_if|b_ig|b_io*), of shape (*4*hidden_size*)
- **bias_hh_l[k]** – the learnable hidden-hidden bias of the k^{th} layer (*b_hi|b_hf|b_hg|b_ho*), of shape (*4*hidden_size*)

Character Level Prediction model – LSTM/RNN

- The main task of the character-level language model is to predict the next character given all previous characters in a sequence of data, i.e. generate text character by character.
- Character-Level Language Model are similar to word-language models.

Steps:

- Encode a character into a one-hot encoding.
- Data preparation
- Convert a sequence of characters into a tensor representation.
- Define the LSTM model by extending the nn.Module class
- predict(s) : to make predictions

Character Level Prediction model – LSTM/RNN

ltt(ch): to encode a character into a one-hot tensor.

```
import string
import torch
import torch.nn as nn

def ltt(ch):
        ans = torch.zeros(n_letters)
        ans[letters.find(ch)]=1
        return ans
```

Character Level Prediction model – LSTM/RNN

- Data Preparation: The given dataset is "i love neural networks".
- seq-len: length of the sequence
- letters: All possible characters (lowercase letters + “ #")
- n_letters: length of vocabulary: 28 (26 lower case alphabets + ‘space’ + EOF is #

```
data = "i love neural networks"
EOF = "#"
data = data.lower()
seq_len = len(data)
letters = string.ascii_lowercase+’  #'
n_letters = len(letters)
print('Letter set = ', letters, "len=", n_letters)
print("Encoding of 'a' ",ltt('a'))
print("Encoding of 'b' ",ltt('b'))
print("Encoding of '#' ",ltt('#'))
```

```
Letter set =
abcdefghijklmnopqrstuvwxyz
#
len= 28
Encoding of 'a'  tensor([1., 0.,
0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0.,
0., 0.])

Encoding of 'b'  tensor([0., 1.,
0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0.,
0., 0.])

Encoding of '#'  tensor([0., 0.,
0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0.,
0., 1.])
```

Character Level Prediction model – LSTM/RNN

- getLine(s) to convert a sequence of characters into a tensor representation.

```
def getLine(s):
    ans = []
    for c in s:
        ans.append(ltt(c))
    return torch.cat(ans,dim=0).view(len(s),1,n_letters)
```

Character Level Prediction model – LSTM/RNN

```
# Output dimensions is (seq_len, batch , hidden_dim)
# Input dimensions = n_letters
# output dimensions = hidden_dimensions = 28
hidden_dim = n_letters
model = MyLSTM(n_letters,hidden_dim)
optimizer = torch.optim.Adam(params = model.parameters(),lr=0.01)
LOSS = torch.nn.CrossEntropyLoss()
#List to store targets
targets = []
#Iterate through all chars in the sequence, starting from second letter. Since output for 1st letter is the 2nd letter
for x in data[1:]+'#':
    #Find the target index. For a, it is 0, For 'b' it is 1 etc..
    targets.append(letters.find(x))
#Convert into tensor
targets = torch.tensor(targets)
print("targets=", targets)
```

Character Level Prediction model – LSTM/RNN

- Neural Network Class (MyLSTM): to define the LSTM model by extending the nn.Module class (base class for all neural network modules in PyTorch)
- __init__ method: initialize the LSTM layer with input and hidden dimensions.
- forward method: perform the forward pass through the LSTM layer.

Character Level Prediction model – LSTM/RNN

Neural Network Class (MyLSTM):

```
class MyLSTM(nn.Module):
  def __init__(self,input_dim,hidden_dim):
    super(MyLSTM,self).__init__()
    self.input_dim = input_dim
    self.hidden_dim = hidden_dim
    #LSTM takes, input dimensions, hidden dimensions and num layers in case of stacked LSTMs (Default is 1)
    self.LSTM = nn.LSTM(input_dim,hidden_dim)

  # Input is 3 dimensional (Sequence len, batch, input dimensions)
  # hc is a tuple which contains the vectors h (hidden) and c (cell state vector)
  def forward(self,inp,hc):
    #this gives output for each input and also (hidden and cell state vector)
    output,_= self.LSTM(inp,hc)
    return output
```

Character Level Prediction model – LSTM/RNN

```
#preparing input tensor
inpl = []
#Iterate through all inputs in the sequence
for c in data:
    #Convert into tensor
    inpl.append(ltt(c))
#Convert list to tensor
inp = torch.cat(inpl,dim=0)
#Reshape tensor into 3 dimensions (sequence length, batches = 1, dimensions = n_letters (28))
inp = inp.view(seq_len,1,n_letters)
#Number of iterations
n_iters = 150
```

Character Level Prediction model – LSTM/RNN

```
#Training loop
for itr in range(n_iters):
    #Zero the previous gradients
    model.zero_grad()
    #Initialize h and c vectors
    h = torch.rand(hidden_dim).view(1,1,hidden_dim)
    c = torch.rand(hidden_dim).view(1,1,hidden_dim)
    output = model(inp,(h,c))
    #Reshape the output to 2 dimensions. This is done, so that we can compare with target and get loss
    output = output.view(seq_len,n_letters)
    #Find loss
    loss = LOSS(output,targets)
    #Print loss for every 10th iteration
    if itr%10==0:
        print(itr,' ',(loss) )
    #Back propagate the loss
    loss.backward()
    #Perform weight updation
    optimizer.step()
```

Character Level Prediction model – LSTM/RNN

```
# predict(s) : to make predictions
def predict(s):
    #Get the vector for input
    print("s= ", s)
    inp = getLine(s)
    print("\n inp=", inp)
    #Initialize h and c vectors
    h = torch.rand(1,1,hidden_dim)
    c = torch.rand(1,1,hidden_dim)
    #Get the output
    out = model(inp,(h,c))
    print(" letters[out[-1][0].topk(1)[1].detach().numpy().item()", letters[out[-1][0].topk(1)[1].detach().numpy().item()])
    #Find the corresponding letter from the output
    return letters[out[-1][0].topk(1)[1].detach().numpy().item()]
```

Character Level Prediction model – LSTM/RNN

```
# predict(s) : to make predictions

str="i love neu“
predict(str)
```