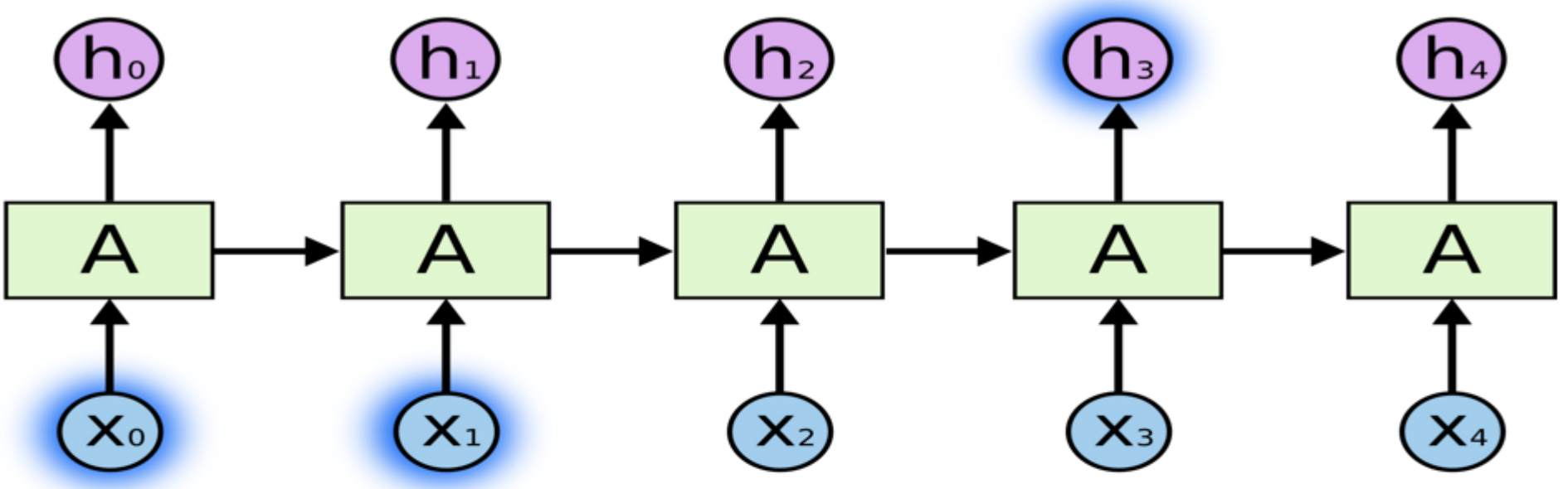
Sequence Modelling with Gated Units Intenals

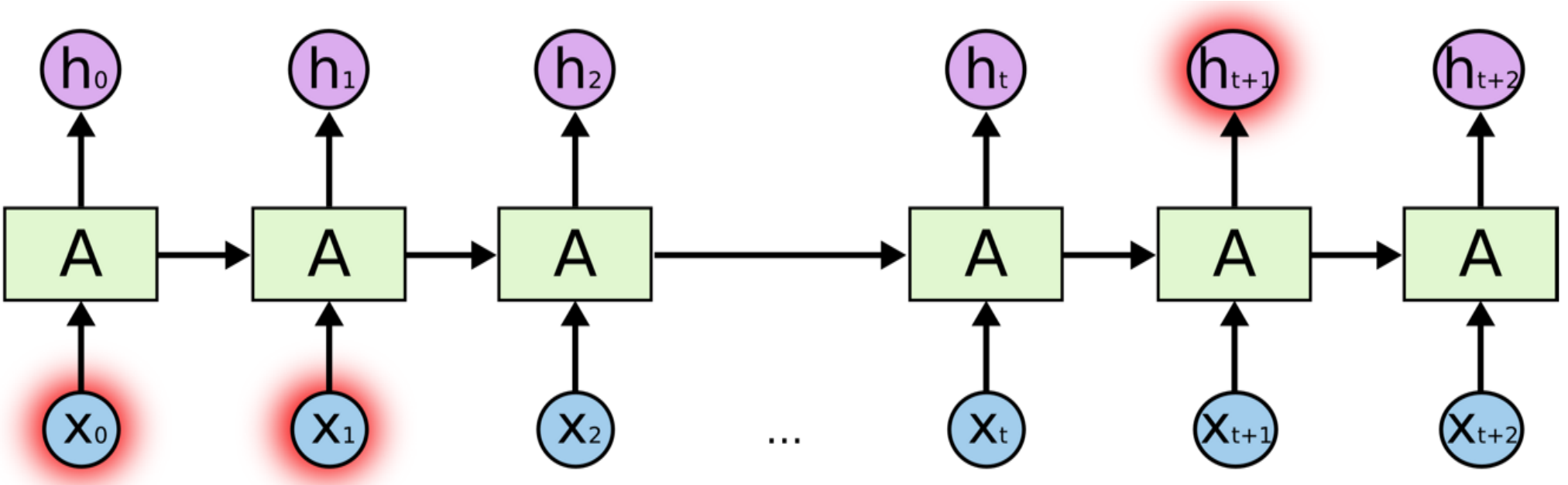
By Mohit Kumar

LSTM(Long Short Term Memory):Why



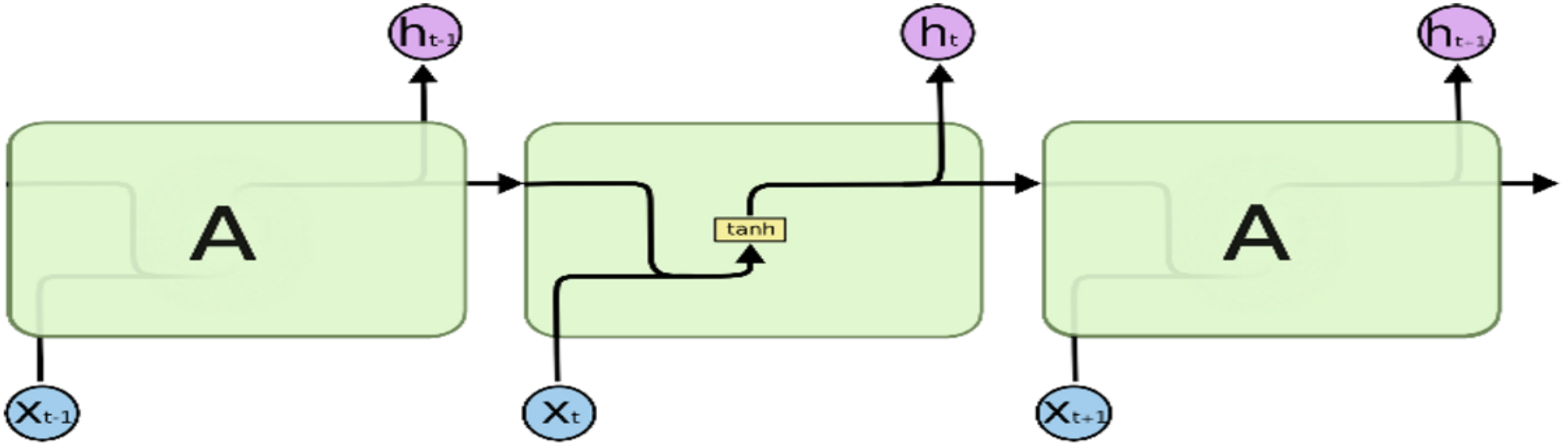
If the RNN is trying to predict the last word in "The clouds are in the sky". It does not need any further context. It is obviously "sky"!

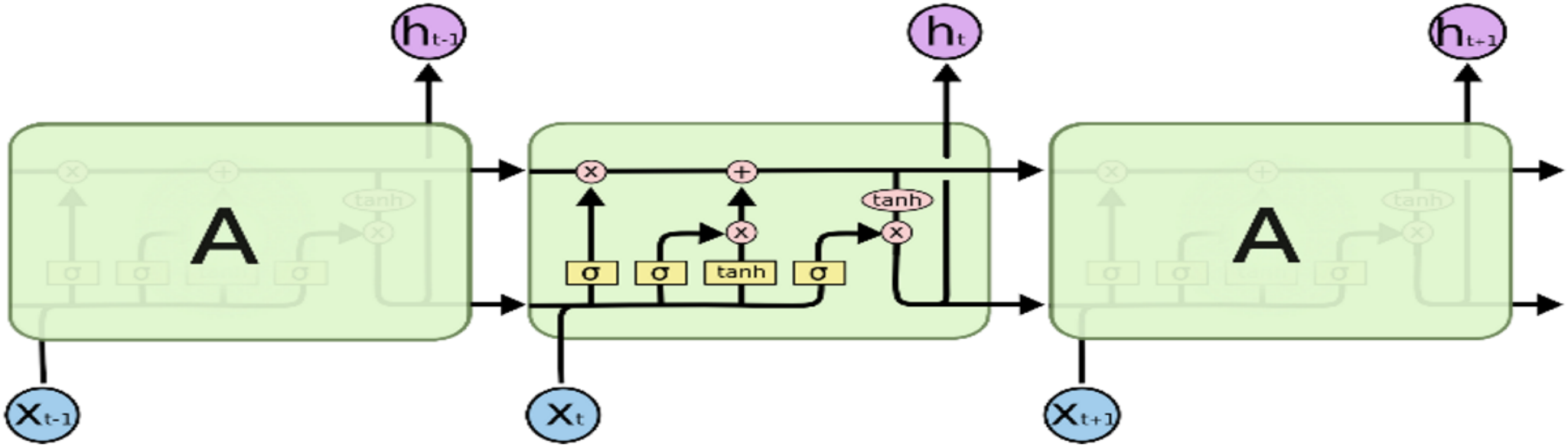
LSTM(Long Short Term Memory):Why



But there are cases when the RNN would require more context than available in the unrolled RNN. Consider "I grew up in France .....I speak fluent French." For this the RNN would require the context of "France" from further back.

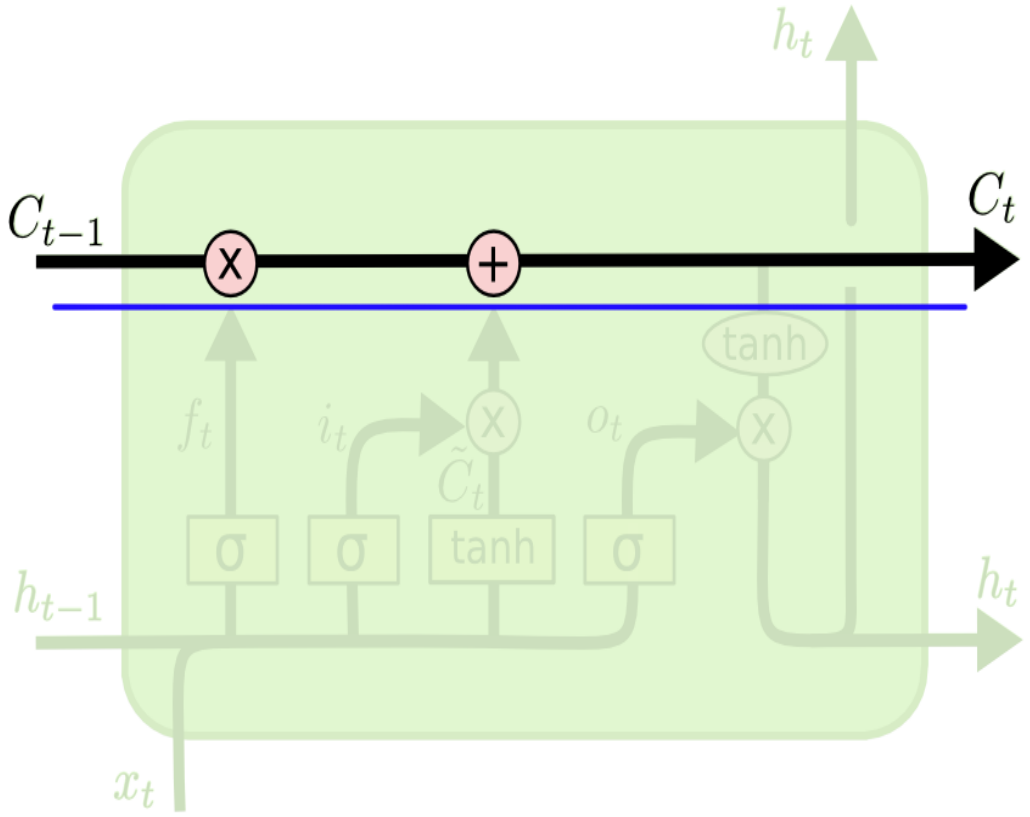
LSTM(Long Short Term Memory):Why

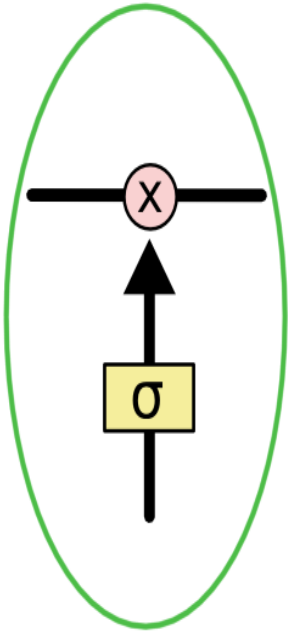




Thinking of RNN as an interface

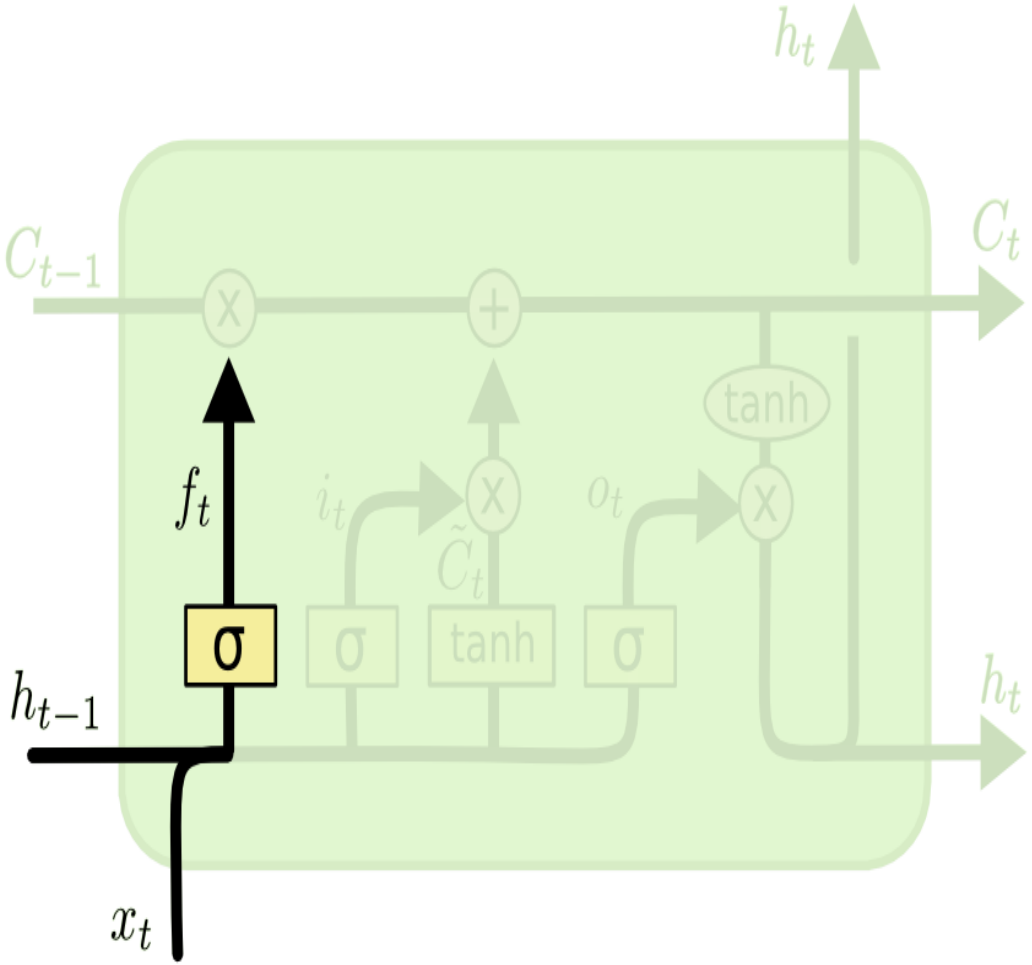
LSTM(Long Short Term Memory):Why





- The key idea in LSTM is the cell state
- The LSTM has the capability To remove or add information carefully controlled by structures called Gates
- They are composed out of a sigmoid neural net layer and a pointwise multiplication operation

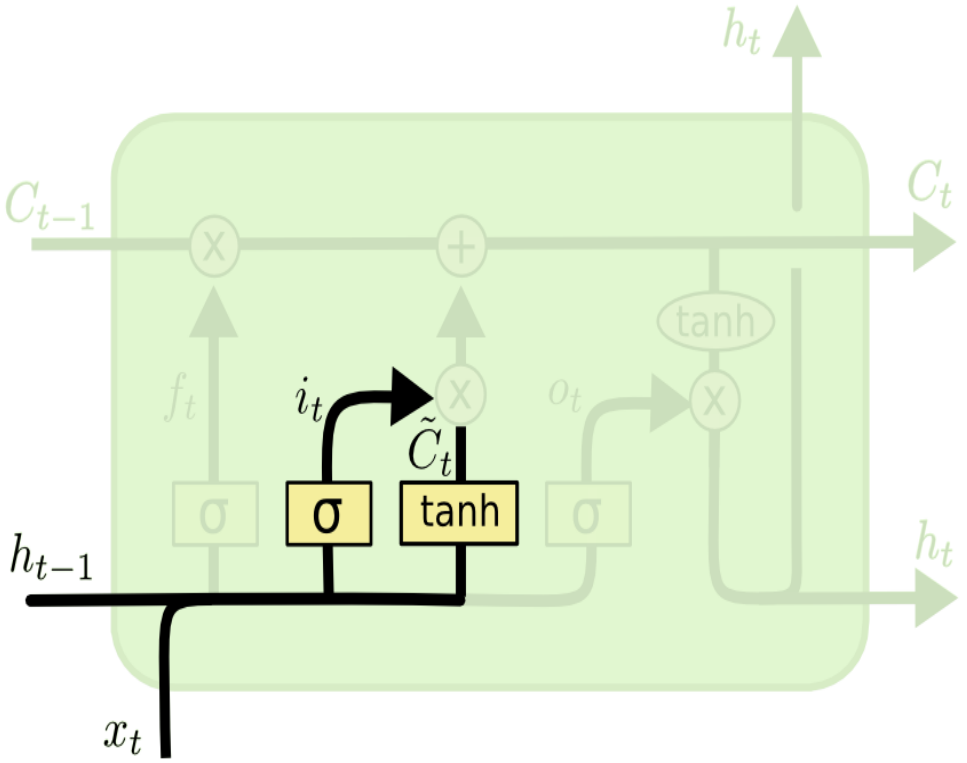
LSTM(Long Short Term Memory):Why



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

- Forget gate layer- it decides what information is irrelevant to current subject and decides to forget it.

LSTM(Long Short Term Memory):Why

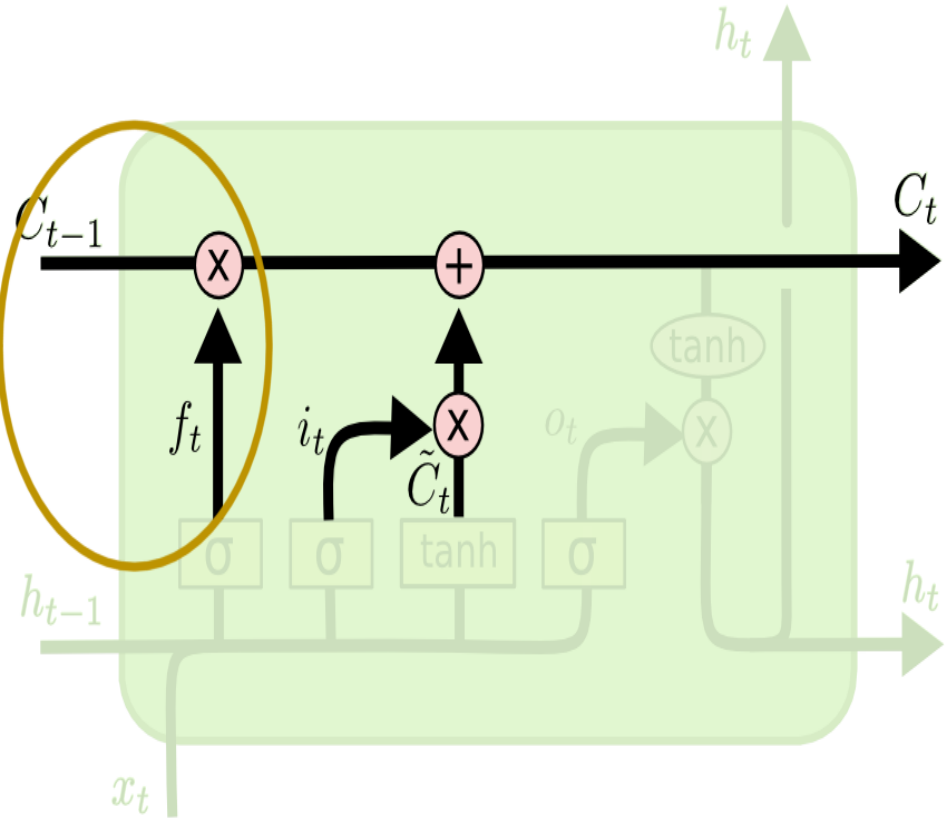


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

- Next step is to decide what information to store
- Input Gate layer- which values it is going to update
- Tanh layer create the vector of new candidate value(C_t) That could be added to state.(probably the details of the subject)

LSTM(Long Short Term Memory):Why



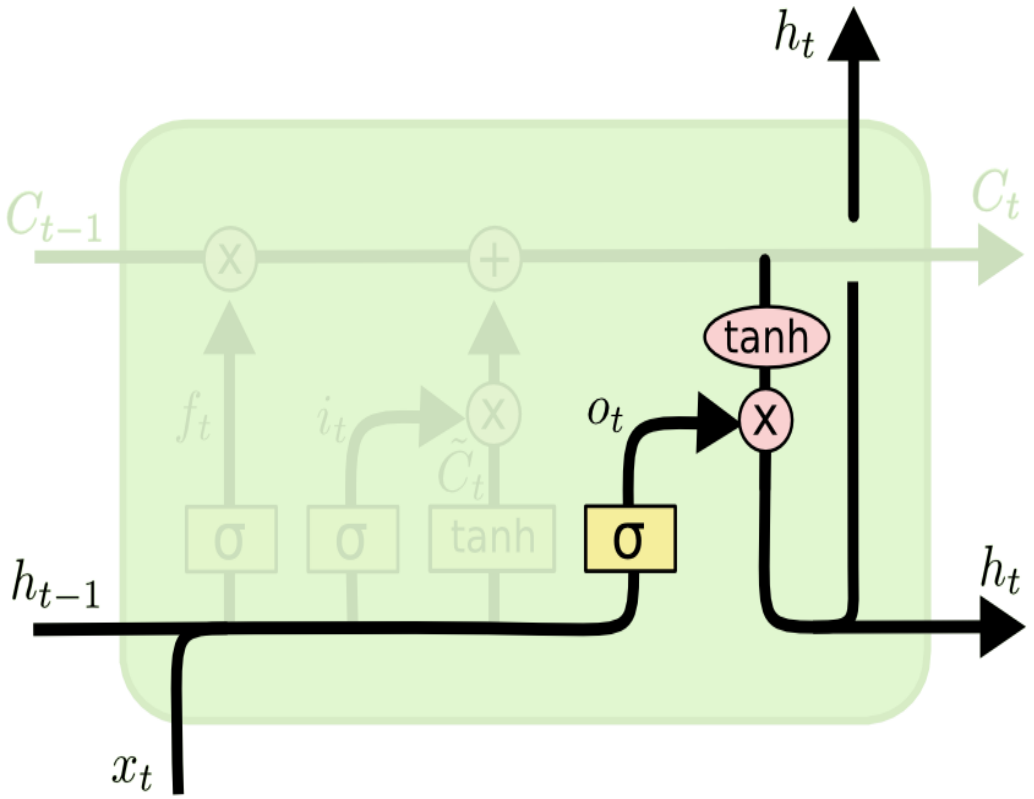
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Scale of how much

Candidate value.

- Now time to update the old cell state(C_{t-1}) to new cell state(C_t).
- Forget what we decided to forget earlier.

LSTM(Long Short Term Memory):Why



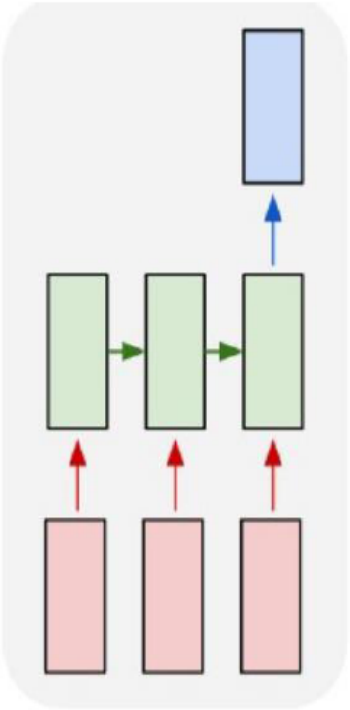
$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

- Finally decide what to output
- it is the cell state but a filtered version
- for a language model, it might use a pronoun relevant to the gender of the subject

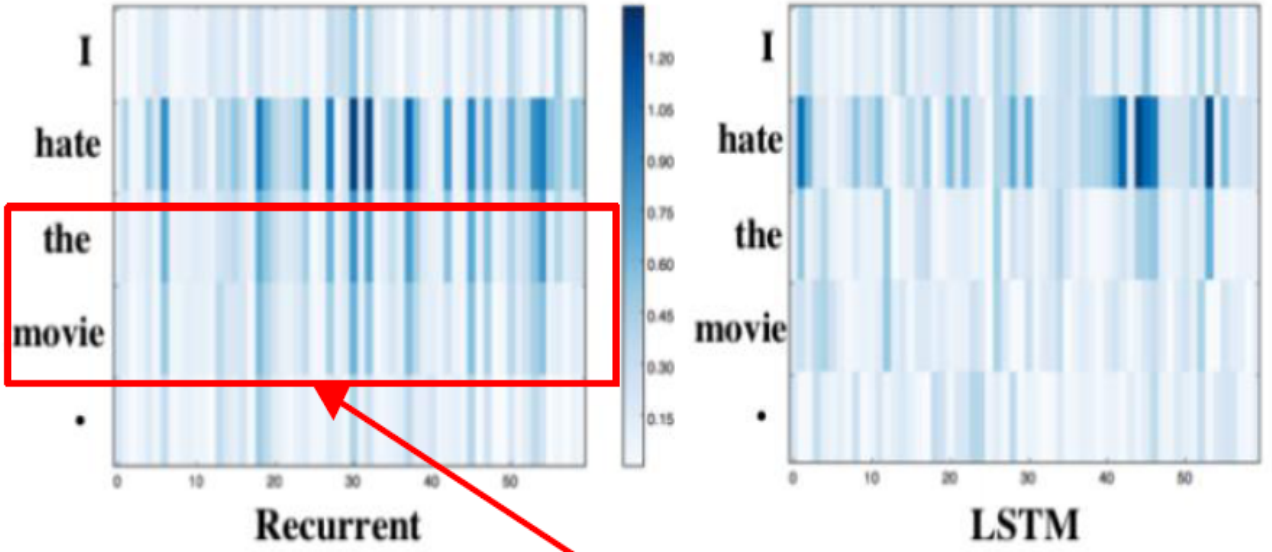
LSTM(Long Short Term Memory):Why

Sentiment Analysis



Many-one network

Saliency Heatmap

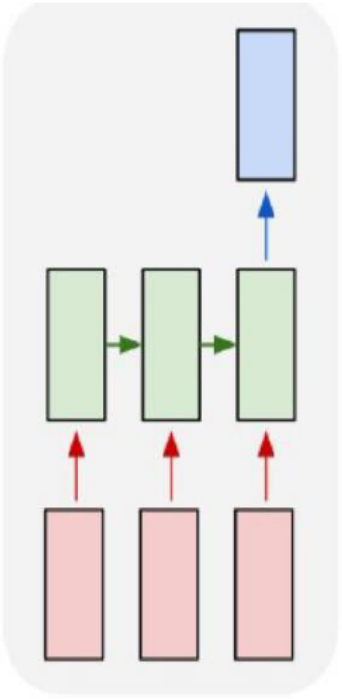


Recent words more salient

LSTM captures long term dependencies

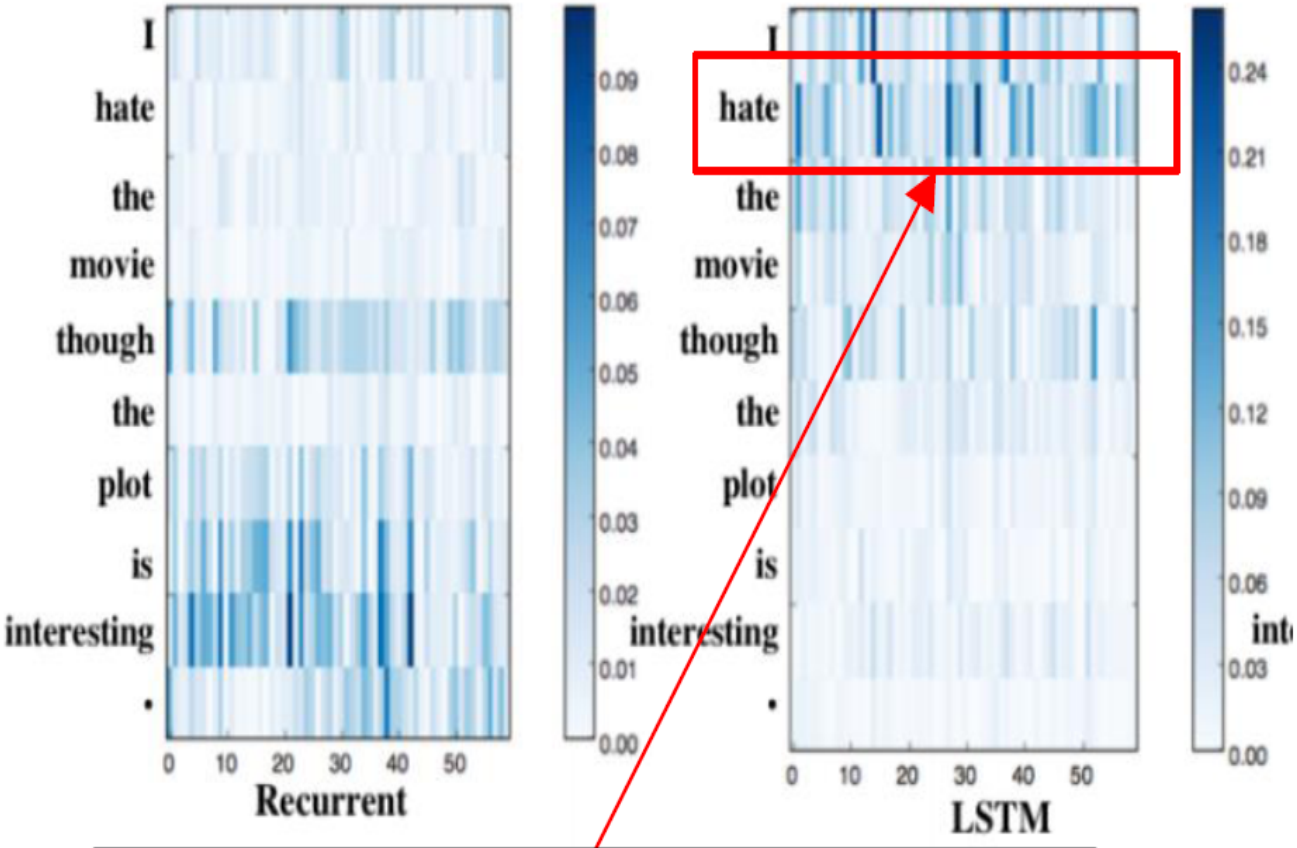
LSTM(Long Short Term Memory):Why

Sentiment Analysis



Many-one network

Saliency Heatmap

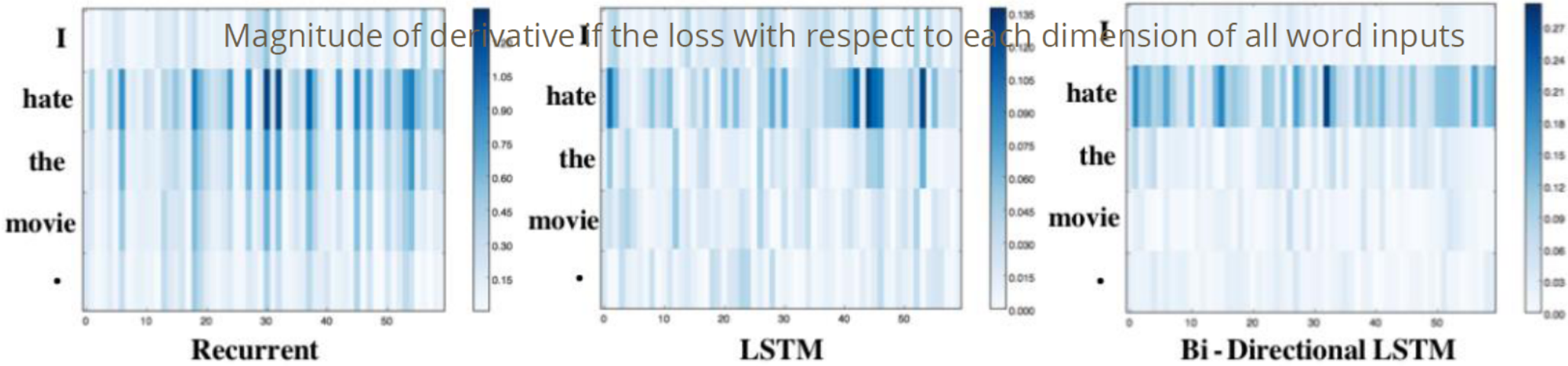


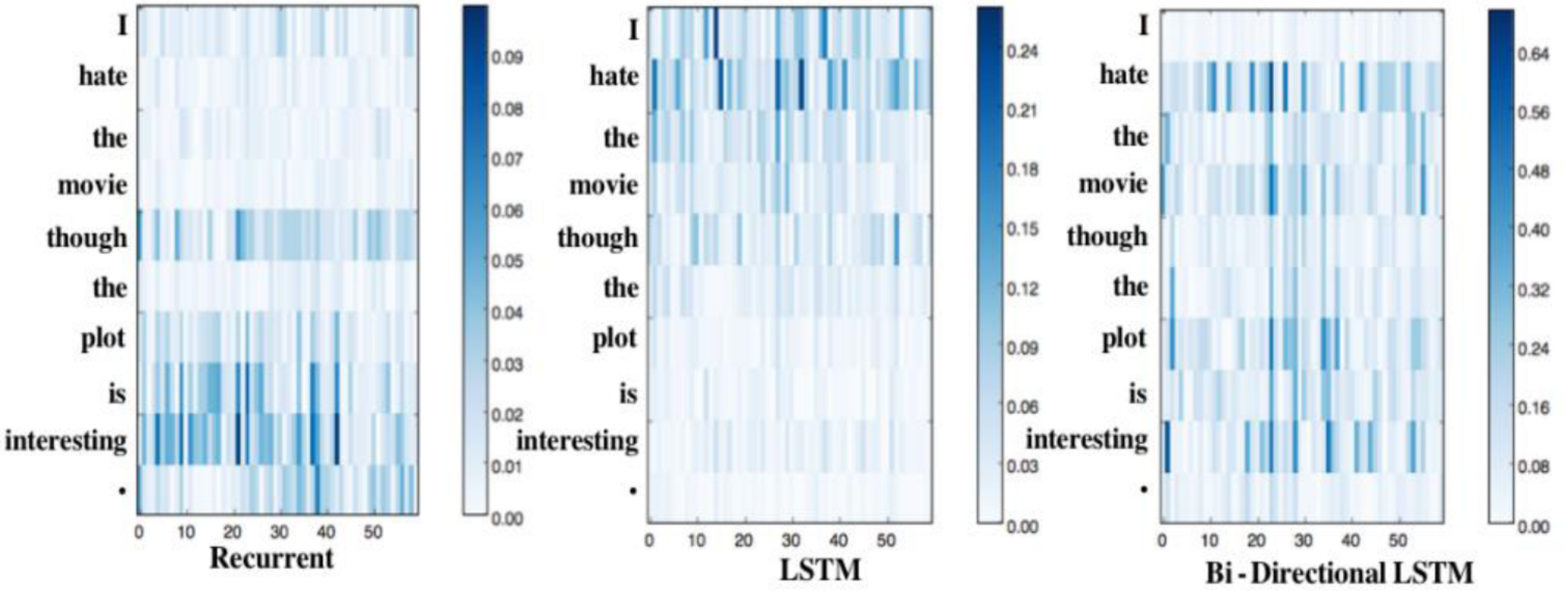
LSTM captures long term dependencies

LSTM(Long Short Term Memory):Why

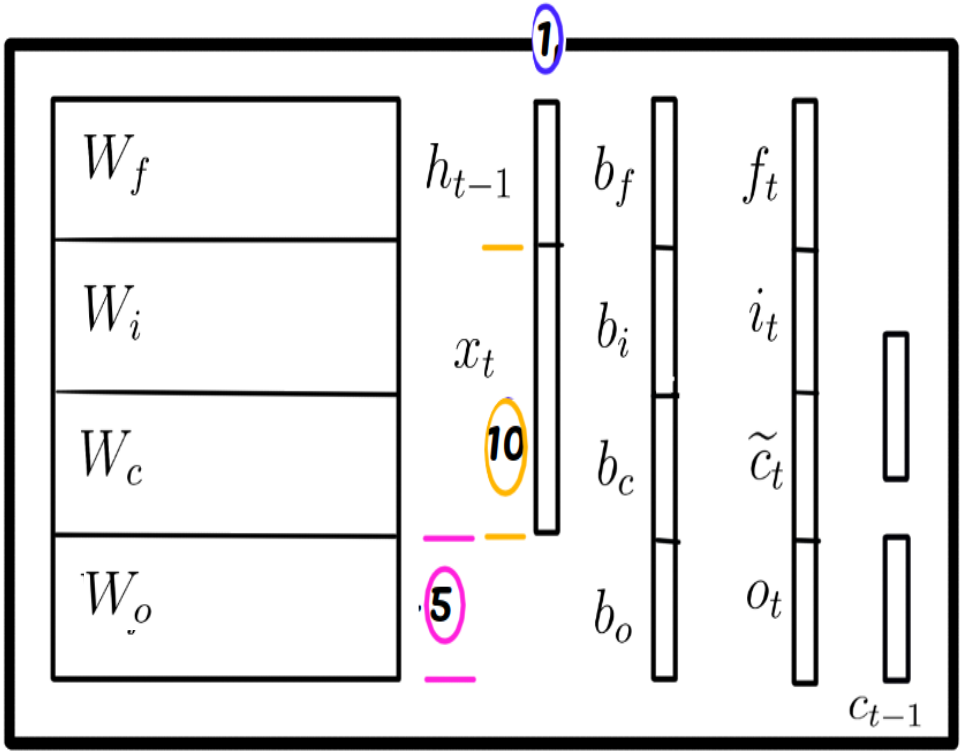
How much each unit contributes to the decision ?

Magnitude of derivative of the loss with respect to each dimension of all word inputs





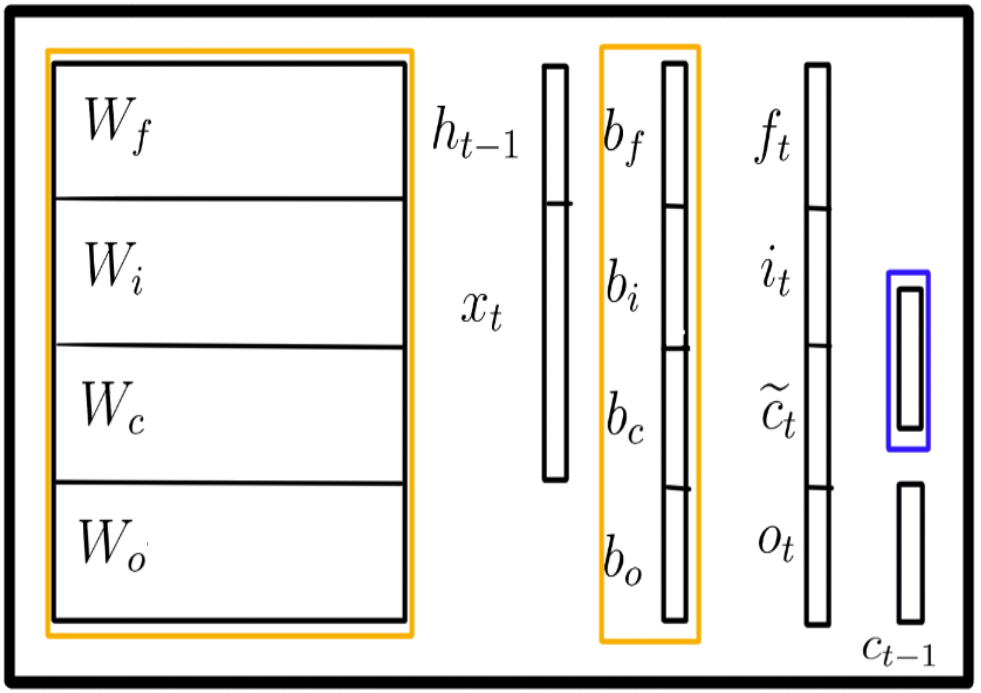
LSTM(Long Short Term Memory):Internals



$$f_t = \sigma\left(W_f.[h_{t-1}, x_t] + b_f\right)$$
$$i_t = \sigma\left(W_i.[h_{t-1}, x_t] + b_i\right)$$
$$\widetilde{c}_t = \tanh\left(W_c.[h_{t-1}, x_t] + b_c\right)$$
$$o_t = \sigma\left(W_o.[h_{t-1}, x_t] + b_o\right)$$
$$c_t = \left(f_t * c_{t-1}\right) + \left(i_t * \widetilde{c}_t\right)$$
$$h_t = o_t * tanh(c_t)$$
$$pred_t = \left(h_t.W_y\right) + b_y$$
$$\hat{y}_t = softmax(pred_t)$$
$$E_t = crossEntropyLoss(\hat{y}_t, y_t)$$

- Assume a batch size of 1, state size of 5 and, vocabulary size of 10 for one hot representation.

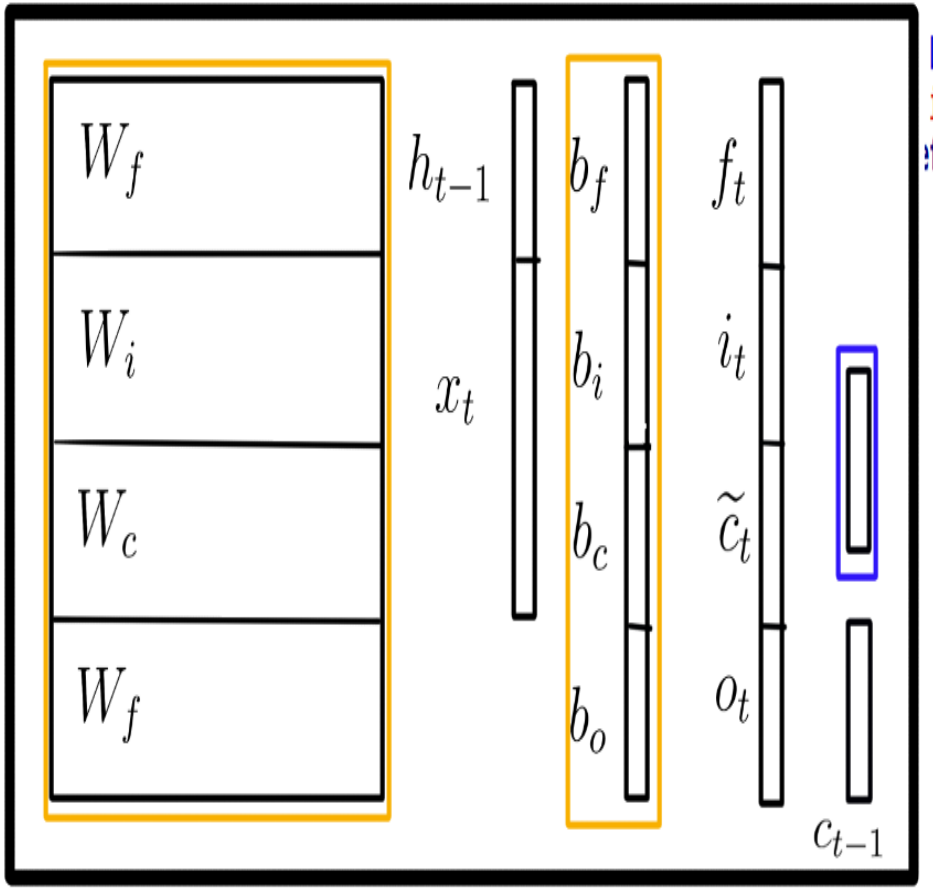
LSTM(Long Short Term Memory):Internals



$$f_t = \sigma\left(W_f.[h_{t-1}, x_t] + b_f\right)$$
$$i_t = \sigma\left(W_i.[h_{t-1}, x_t] + b_i\right)$$
$$\widetilde{c}_t = \tanh\left(W_c.[h_{t-1}, x_t] + b_c\right)$$
$$o_t = \sigma\left(W_o.[h_{t-1}, x_t] + b_o\right)$$
$$c_t = (f_t * c_{t-1}) + (i_t * \widetilde{c}_t)$$
$$h_t = o_t * tanh(c_t)$$
$$pred_t = (h_t.W_y) + b_y$$
$$\hat{y}_t = softmax(pred_t)$$
$$E_t = crossEntropyLoss(\hat{y}_t, y_t)$$

- Randomly initialised weights and biases to begin with.
- and the all important cell state that makes LSTM remember long term dependencies

LSTM(Long Short Term Memory):Internals



- Randomly initialised weights and biases to begin with.
- and the all important cell state that makes LSTM remember long term dependencies

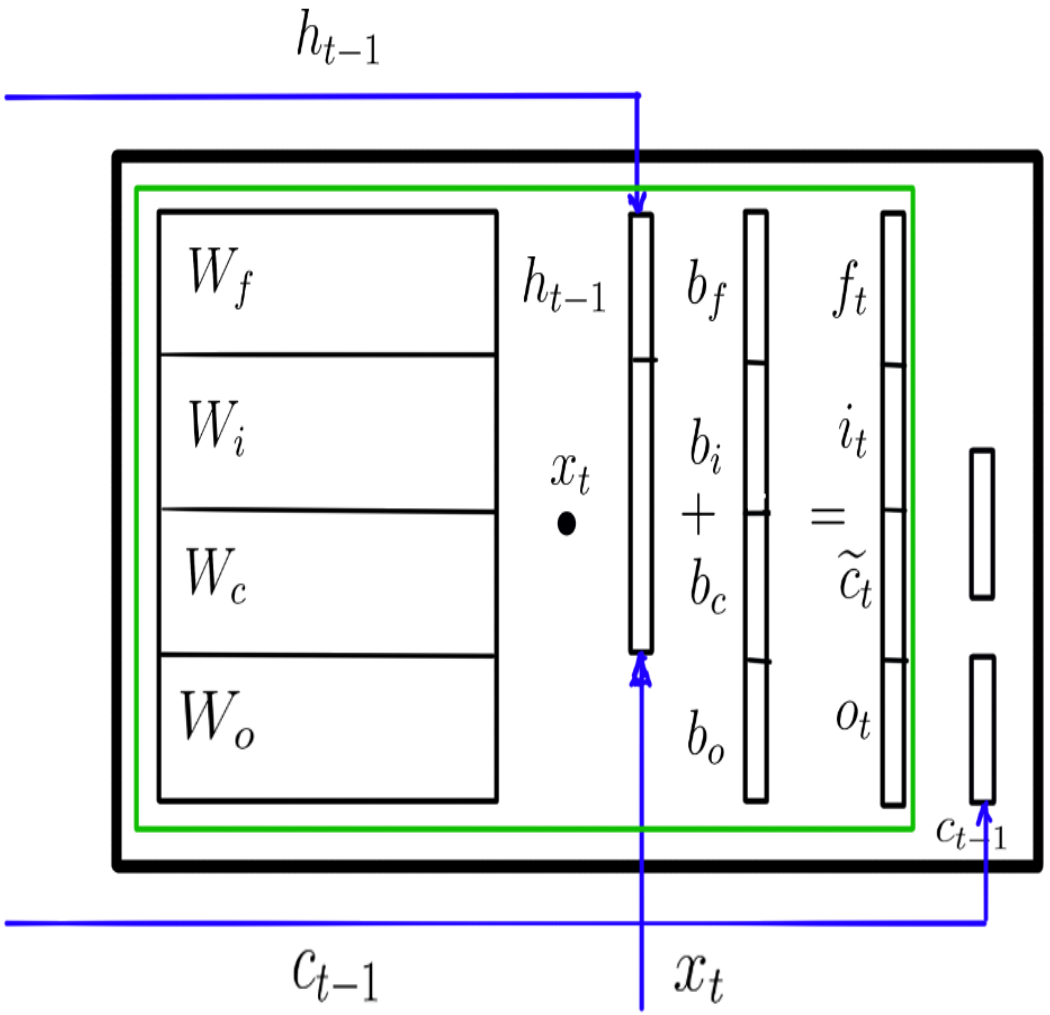
```
LSTMCell:
initialise the Recurrent Neural Network
f __init__(self, input_size, hidden_size,forget_bias=1, w=None,wout=None,biasout=None ,debug=False):
  self.input_size = input_size
  self.hidden_size = hidden_size
  self.forget_bias = forget_bias
  self.debug = debug

  # first column is for biases
  if w is None:
      self.WLSTM = np.random.randn(4 * hidden_size,input_size + hidden_size + 1) / np.sqrt(input_size + hidden_size)
  else:
      self.WLSTM=w

  if wout is None:
      self.WOUT = np.random.randn(self.hidden_size,self.input_size) / np.sqrt(self.input_size)
  else:
      self.WOUT=wout

  if biasout is None:
      self.BIASOUT = np.random.randn(1,self.input_size) / np.sqrt(self.input_size)
  else:
      self.BIASOUT=biasout
```

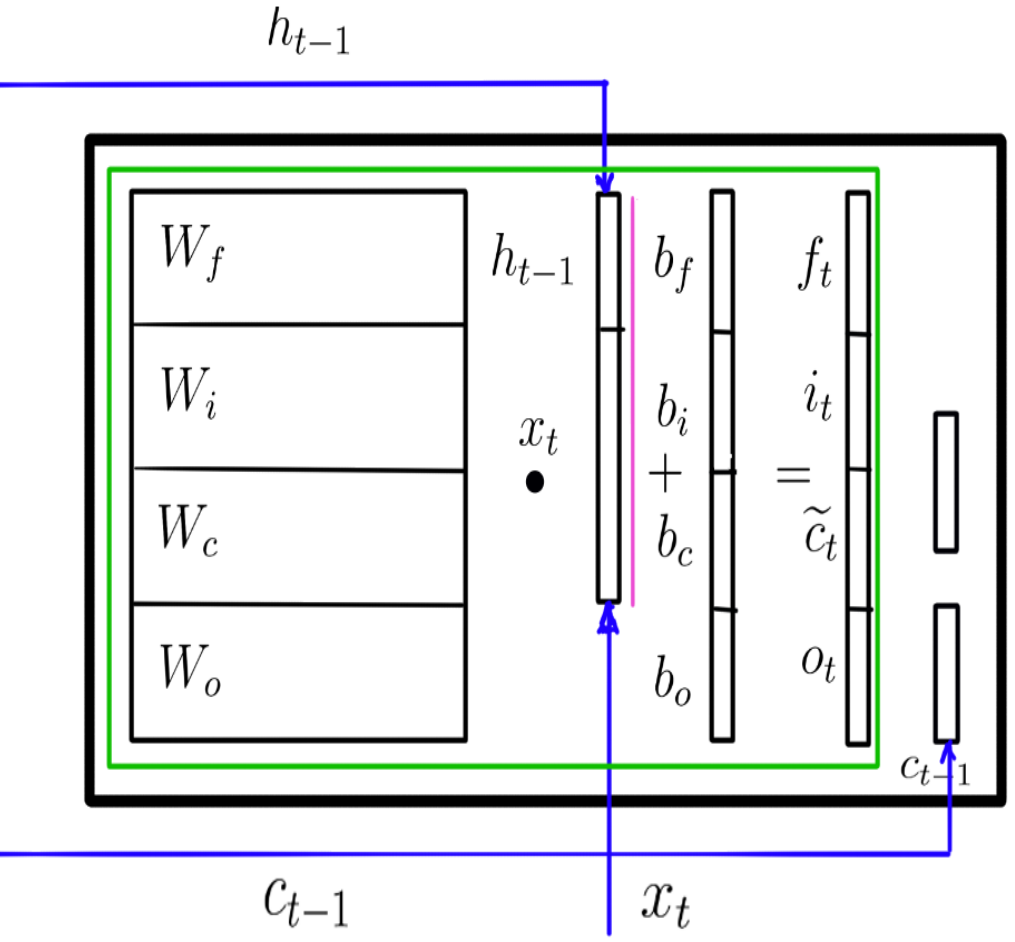
LSTM(Long Short Term Memory):Internals



- Recurrent state from previous time step
- All the gated calculations

$$f_t = \sigma\left(W_f.[h_{t-1}, x_t] + b_f\right)$$
$$i_t = \sigma\left(W_i.[h_{t-1}, x_t] + b_i\right)$$
$$\widetilde{c}_t = \tanh\left(W_c.[h_{t-1}, x_t] + b_c\right)$$
$$o_t = \sigma\left(W_o.[h_{t-1}, x_t] + b_o\right)$$
$$c_t = (f_t * c_{t-1}) + (i_t * \widetilde{c}_t)$$
$$h_t = o_t * tanh(c_t)$$
$$pred_t = (h_t.W_y) + b_y$$
$$\hat{y}_t = softmax(pred_t)$$
$$E_t = crossEntropyLoss(\hat{y}_t, y_t)$$

LSTM(Long Short Term Memory):Internals

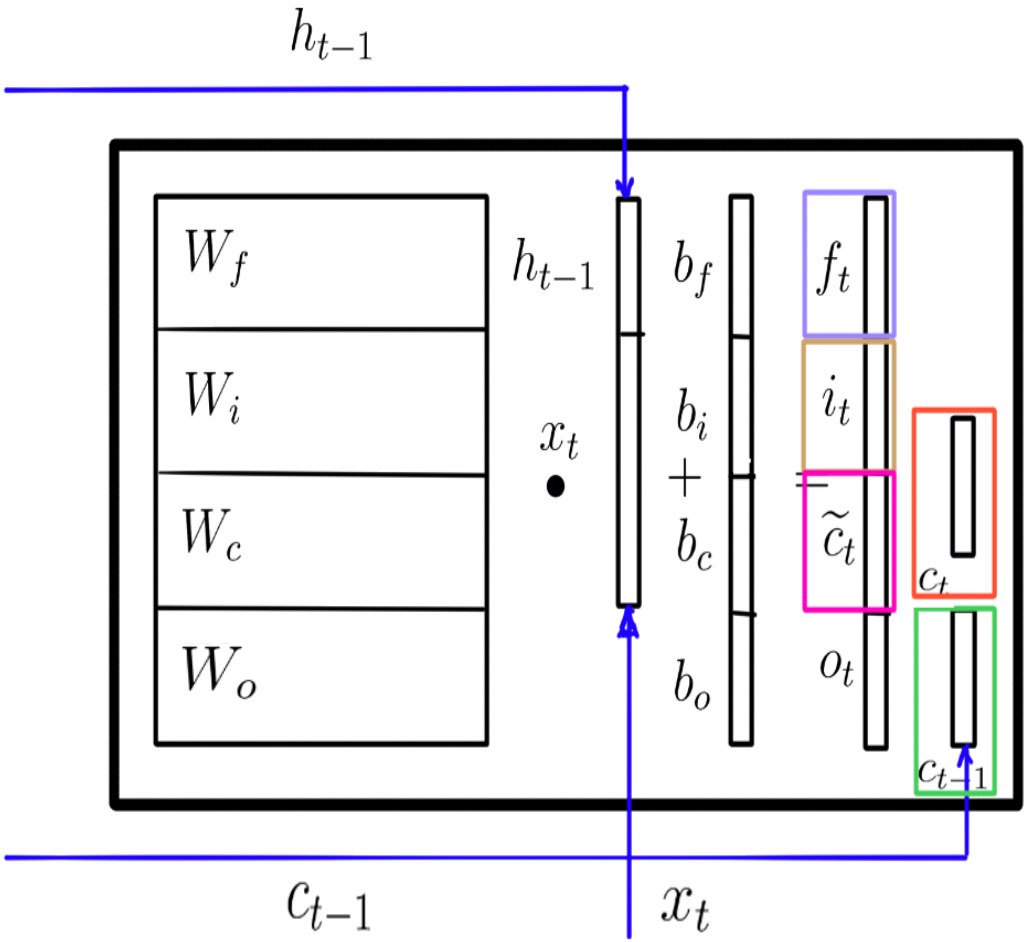


$$f_t = \sigma\left(W_f.[h_{t-1}, x_t] + b_f\right)$$
$$i_t = \sigma\left(W_i.[h_{t-1}, x_t] + b_i\right)$$
$$\widetilde{c}_t = \tanh\left(W_c.[h_{t-1}, x_t] + b_c\right)$$
$$o_t = \sigma\left(W_o.[h_{t-1}, x_t] + b_o\right)$$

```
for seqnum in range(seq):
    (c,h)=prevstate
    if self.debug:
        print("c:",c," h:",h)
        print("initstate:",prevstate)
    x=np.reshape(X[0][seqnum],[input_size,-1])
    row, col=x.shape
    if self.input_size != row:
        print("self.input_size:",self.input_size," c:",row)
        raise ValueError('Input size must match. This is the typically the vocab si
    z=np.concatenate((x,h),0)
    fico=np.dot(self.WLSTM[:,1:],z)+self.WLSTM[:,0].reshape(self.hidden_size*4,col)
    f=self.sigmoid_array(fico[0:self.hidden_size,:]+self.forget_bias)
    i=self.sigmoid_array(fico[self.hidden_size*1:self.hidden_size*2,:])
    cproj=self.tanh_array(fico[self.hidden_size*2:self.hidden_size*3,:])
    o=self.sigmoid_array(fico[self.hidden_size*3:self.hidden_size*4,:])
    cnew=(c * f) + (cproj * i)
    hnew=o*self.tanh_array(cnew)
```

- Recurrent state from previous time step
- All the gated calculations
- Concatenate (h, x) and then matrix multiply the whole lot.
- Individually apply sigmoid or tanh.

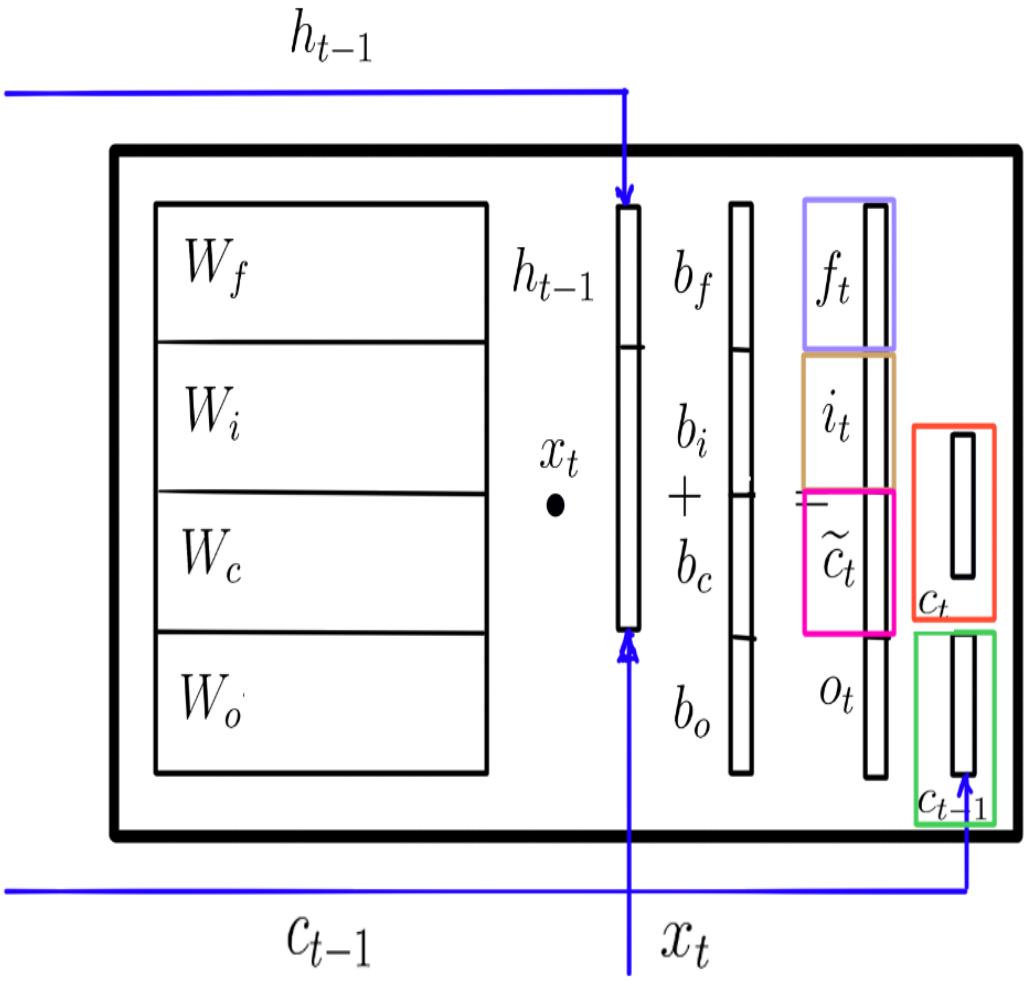
LSTM(Long Short Term Memory):Internals



$$f_t = \sigma\left(W_f.[h_{t-1}, x_t] + b_f\right)$$
$$i_t = \sigma\left(W_i.[h_{t-1}, x_t] + b_i\right)$$
$$\tilde{c}_t = \tanh\left(W_c.[h_{t-1}, x_t] + b_c\right)$$
$$o_t = \sigma\left(W_o.[h_{t-1}, x_t] + b_o\right)$$
$$c_t = (f_t * c_{t-1}) + (i_t * \tilde{c}_t)$$
$$h_t = o_t * tanh(c_t)$$
$$pred_t = (h_t.W_y) + b_y$$
$$\hat{y}_t = softmax(pred_t)$$
$$E_t = crossEntropyLoss(\hat{y}_t, y_t)$$

- New cells state calculation 4 time step t.
- Forget old state
- Remember new state

LSTM(Long Short Term Memory):Internals

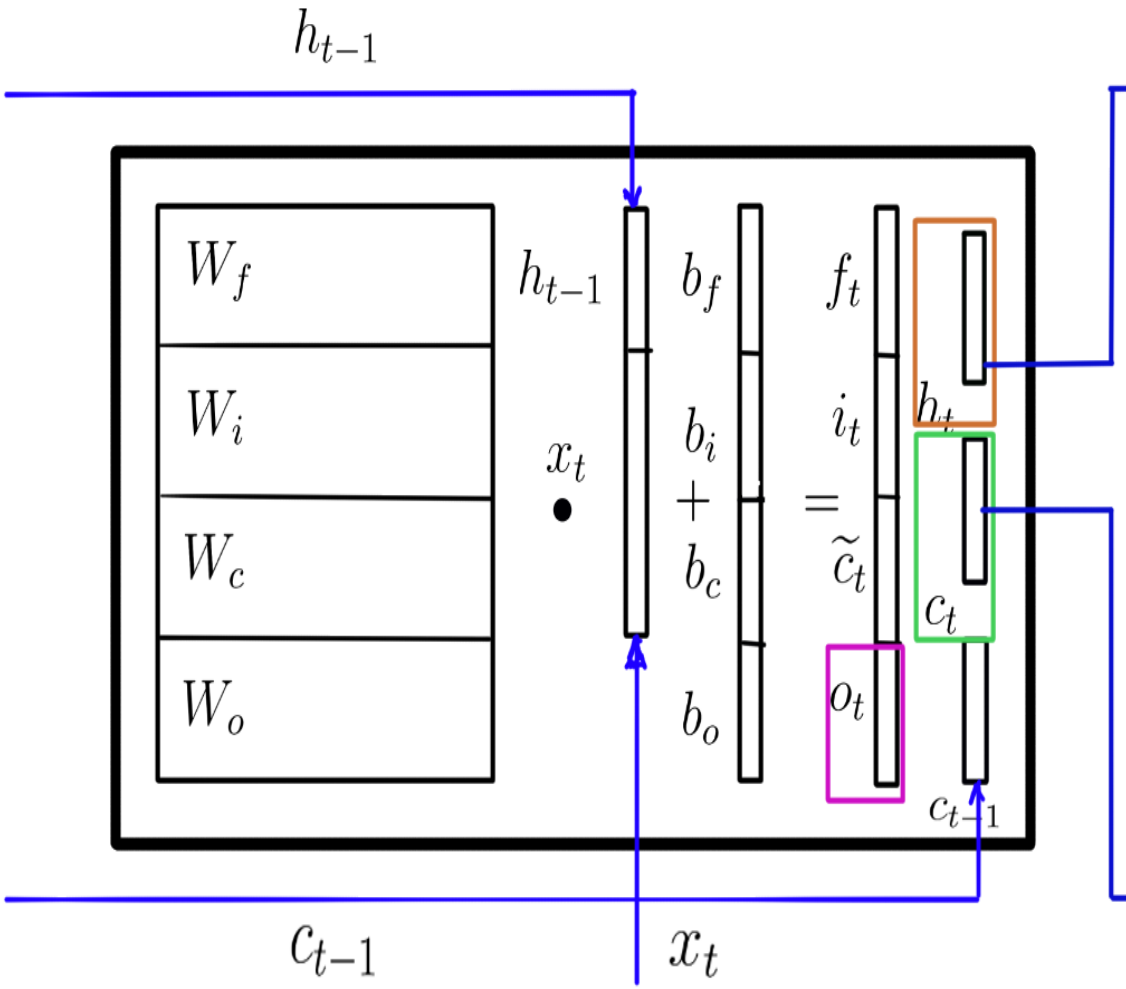


$$f_t = \sigma\left(W_f.[h_{t-1}, x_t] + b_f\right)$$
$$i_t = \sigma\left(W_i.[h_{t-1}, x_t] + b_i\right)$$
$$\tilde{c}_t = \tanh\left(W_c.[h_{t-1}, x_t] + b_c\right)$$
$$o_t = \sigma\left(W_o.[h_{t-1}, x_t] + b_o\right)$$
$$c_t = (f_t * c_{t-1}) + (i_t * \tilde{c}_t)$$

- New cells state calculation 4 time step t.
- Forget old state
- Remember new state

```
for seqnum in range(seq):
    (c,h)=prevstate
    if self.debug:
        print("c:",c," h:",h)
        print("initstate:",prevstate)
    x=np.reshape(X[0][seqnum],[input_size,-1])
    row, col=x.shape
    if self.input_size != row:
        print("self.input_size:",self.input_size," c:",row)
        raise ValueError('Input size must match. This is the typically the vocab si
    z=np.concatenate((x,h),0)
    fico=np.dot(self.WLSTM[:,1:],z)+self.WLSTM[:,0].reshape(self.hidden_size*4,col)
    f=self.sigmoid_array(fico[0:self.hidden_size,:]+self.forget_bias)
    i=self.sigmoid_array(fico[self.hidden_size*1:self.hidden_size*2,:])
    cproj=self.tanh_array(fico[self.hidden_size*2:self.hidden_size*3,:])
    o=self.sigmoid_array(fico[self.hidden_size*3:self.hidden_size*4,:])
    cnew=(c * f) + (cproj * i)
    hnew=o*self.tanh_array(cnew)
```

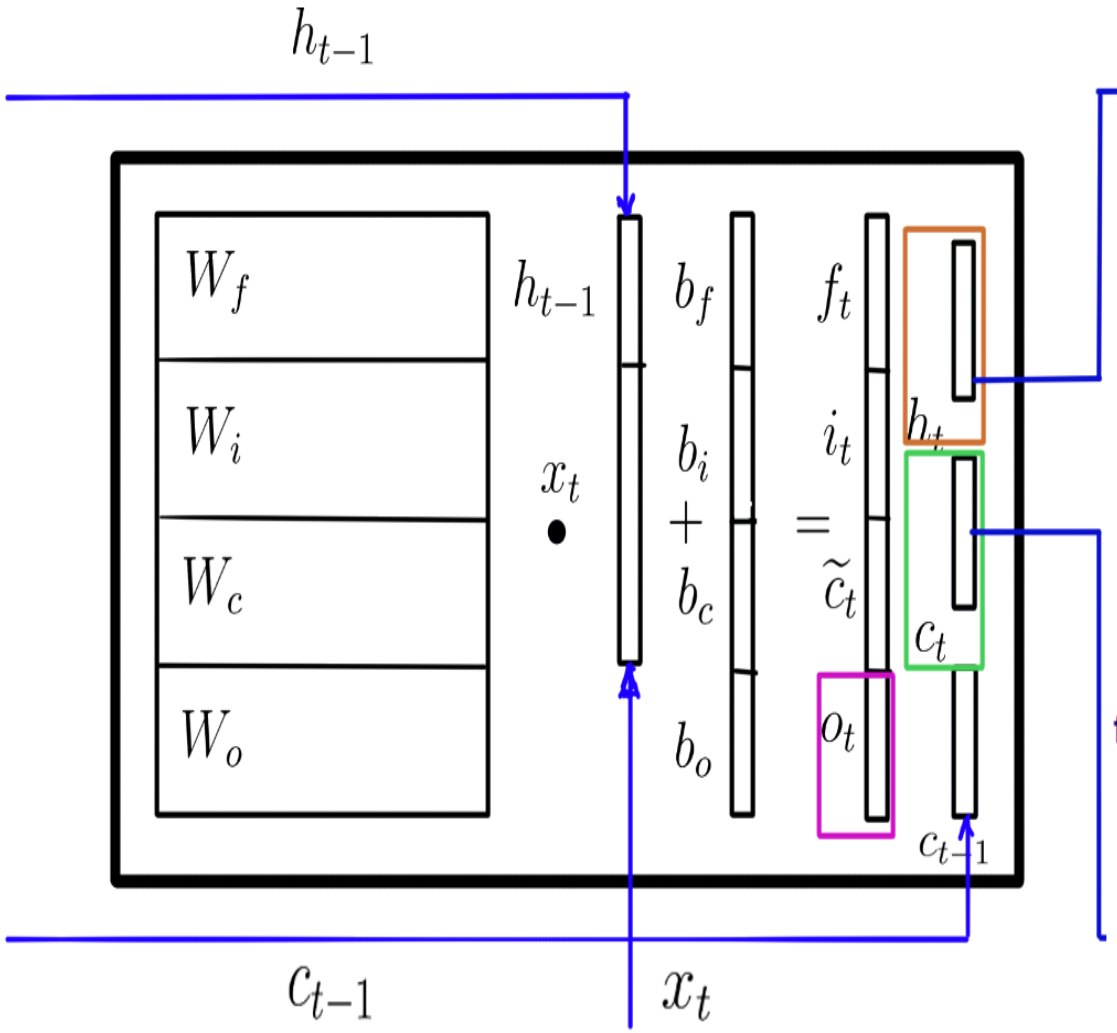
LSTM(Long Short Term Memory):Internals



$$f_t = \sigma\left(W_f.[h_{t-1}, x_t] + b_f\right)$$
$$i_t = \sigma\left(W_i.[h_{t-1}, x_t] + b_i\right)$$
$$\widetilde{c}_t = \tanh\left(W_c.[h_{t-1}, x_t] + b_c\right)$$
$$o_t = \sigma\left(W_o.[h_{t-1}, x_t] + b_o\right)$$
$$c_t = \left(f_t * c_{t-1}\right) + \left(i_t * \widetilde{c}_t\right)$$
$$h_t = o_t * tanh(c_t)$$
$$pred_t = \left(h_t.W_y\right) + b_y$$
$$\hat{y}_t = softmax(pred_t)$$
$$E_t = crossEntropyLoss(\hat{y}_t, y_t)$$

- The new recurrent state h_t calculated based on the new cell state c_t.
- h_t, c_t passed on to the next time step

LSTM(Long Short Term Memory):Internals



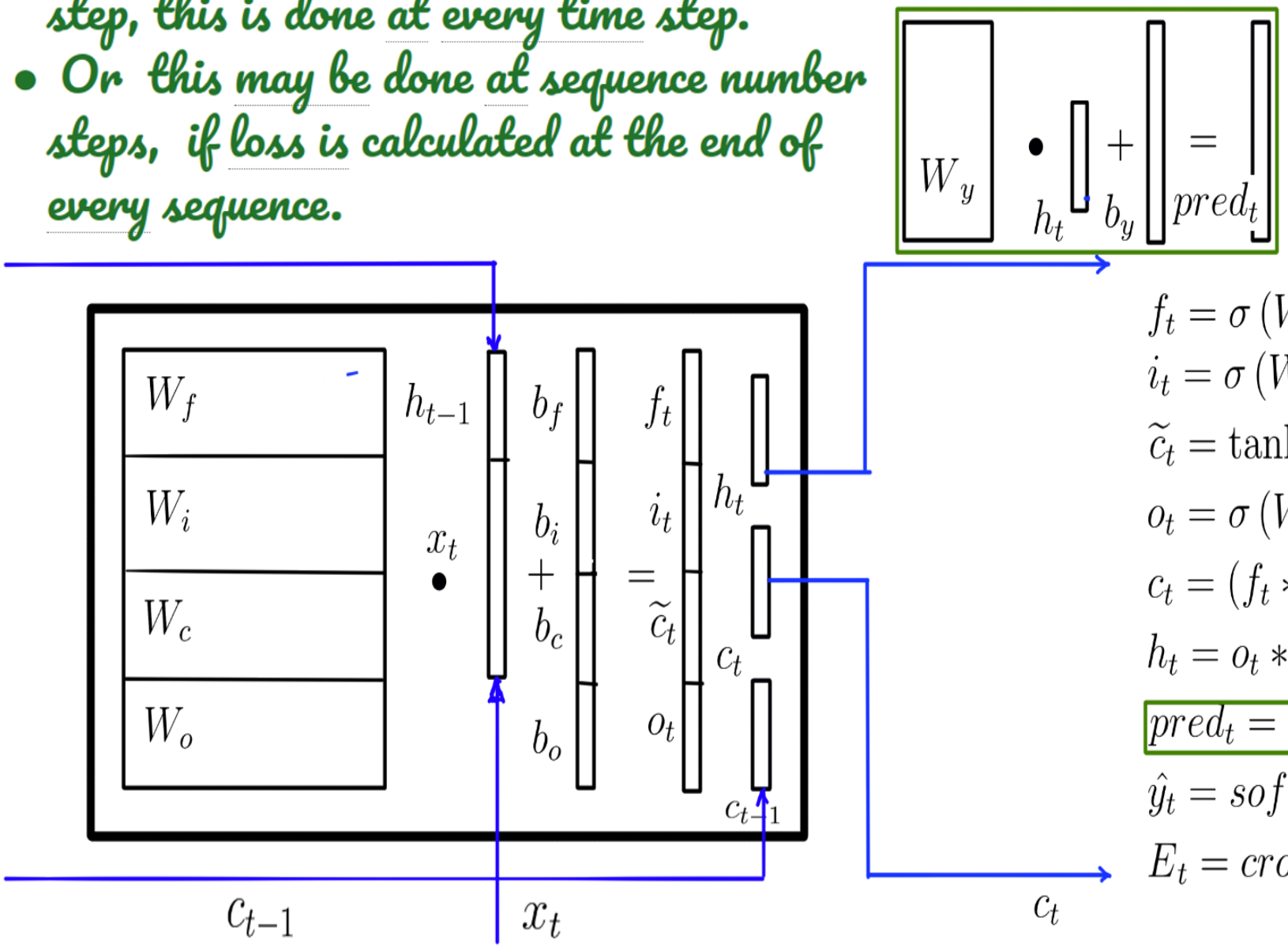
$$f_t = \sigma\left(W_f.[h_{t-1}, x_t] + b_f\right)$$
$$i_t = \sigma\left(W_i.[h_{t-1}, x_t] + b_i\right)$$
$$\tilde{c}_t = \tanh\left(W_c.[h_{t-1}, x_t] + b_c\right)$$
$$o_t = \sigma\left(W_o.[h_{t-1}, x_t] + b_o\right)$$
$$c_t = (f_t * c_{t-1}) + (i_t * \tilde{c}_t)$$
$$h_t = o_t * tanh(c_t)$$

```
for seqnum in range(seq):
    (c,h)=prevstate
    if self.debug:
        print("c:",c," h:",h)
        print("initstate:",prevstate)
    x=np.reshape(X[0][seqnum],[input_size,-1])
    row, col=x.shape
    if self.input_size != row:
        print("self.input_size:",self.input_size," c:",row)
        raise ValueError('Input size must match. This is the typically the vocab si
    z=np.concatenate((x,h),0)
    fico=np.dot(self.WLSTM[:,1:],z)+self.WLSTM[:,0].reshape(self.hidden_size*4,col)
    f=self.sigmoid_array(fico[0:self.hidden_size,:]+self.forget_bias)
    i=self.sigmoid_array(fico[self.hidden_size*1:self.hidden_size*2,:])
    cproj=self.tanh_array(fico[self.hidden_size*2:self.hidden_size*3,:])
    o=self.sigmoid_array(fico[self.hidden_size*3:self.hidden_size*4,:])
    cnew=(c * f) + (cproj * i)
    hnew=o*self.tanh_array(cnew)
```

- The new recurrent state h_t calculated based on the new cell state c_t.
- h_t, c_t passed on to the next time step

LSTM(Long Short Term Memory):Internals

- This is usually done outside the LSTM cell for most frameworks.
- If loss has to be calculated and every time step, this is done at every time step.
- Or this may be done at sequence number steps, if loss is calculated at the end of every sequence.

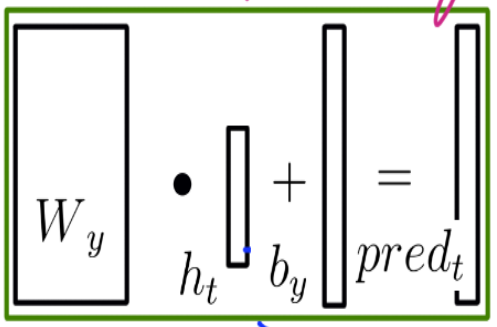


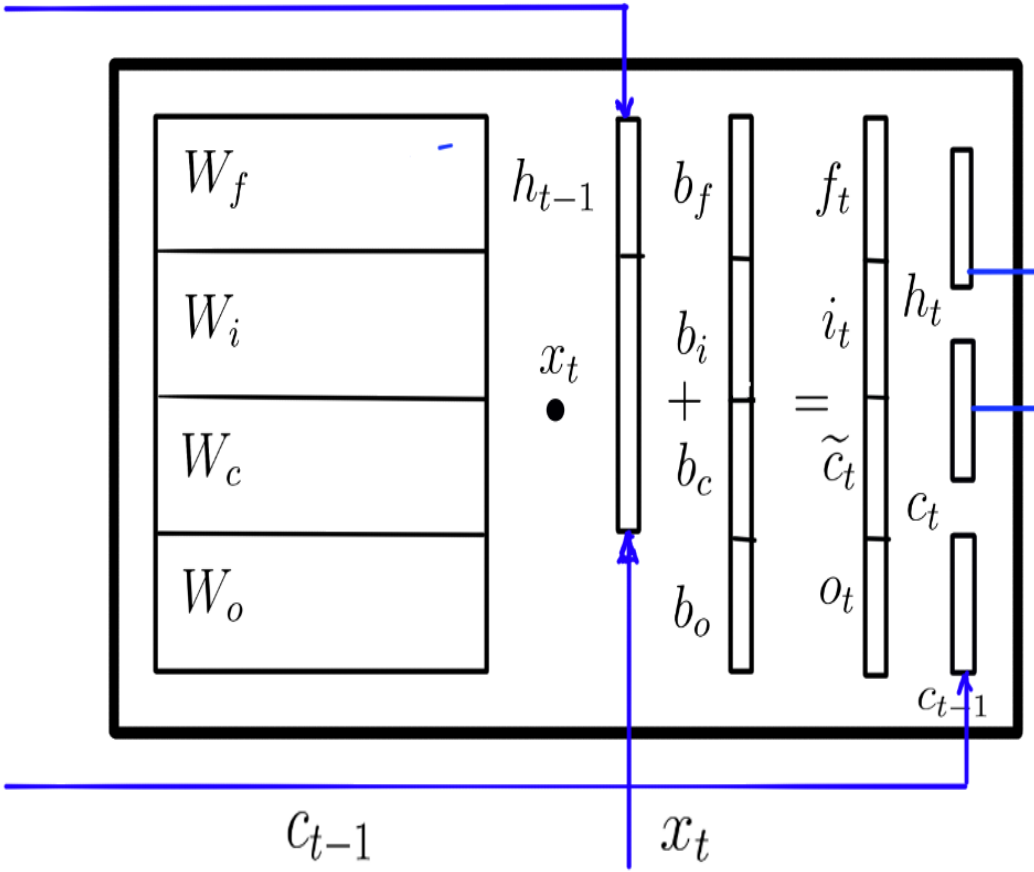
$$f_t = \sigma\left(W_f.[h_{t-1}, x_t] + b_f\right)$$
$$i_t = \sigma\left(W_i.[h_{t-1}, x_t] + b_i\right)$$
$$\widetilde{c}_t = \tanh\left(W_c.[h_{t-1}, x_t] + b_c\right)$$
$$o_t = \sigma\left(W_o.[h_{t-1}, x_t] + b_o\right)$$
$$c_t = (f_t * c_{t-1}) + (i_t * \widetilde{c}_t)$$
$$h_t = o_t * tanh(c_t)$$
$$\boxed{pred_t = (h_t.W_y) + b_y}$$
$$\hat{y}_t = softmax(pred_t)$$
$$E_t = crossEntropyLoss(\hat{y}_t, y_t)$$

LSTM(Long Short Term Memory):Internals

- This is usually done outside the LSTM cell for most frameworks.
- If loss has to be calculated and every time step, this is done at every time step.
- Or this may be done at sequence number steps, if loss is calculated at the end of every sequence.

- Notice the indentation, the last three steps are done after sequence number of steps.






```
        prevstates.append(prevstate)
    print("finalstate:",prevstate)
    (c,h)=prevstate
    pred=np.dot(np.reshape(h,[-1,self.hidden_size]),self.WOUT)+self.BIASOUT
    yhat = self.softmax(pred)
    lossesperoneseq = self.loss(np.reshape(yhat,[-1,self.input_size]),symbols_out_onehot)
```

$$pred_t = (h_t.W_y) + b_y$$

$$\hat{y}_t = softmax(pred_t)$$

$$E_t = crossEntropyLoss(\hat{y}_t, y_t)$$

c_t

LSTM(Long Short Term Memory):Internals

• sums up to one.

```
[[ 2.4]
 [ 2.0]
 [-0.0]
 [ 0.9]
 [ 0.2]
 [ 0.5]
 [-1.1]
 [-3.1]
 [ 2.3]
 [-0.7]]
```

$pred_t$

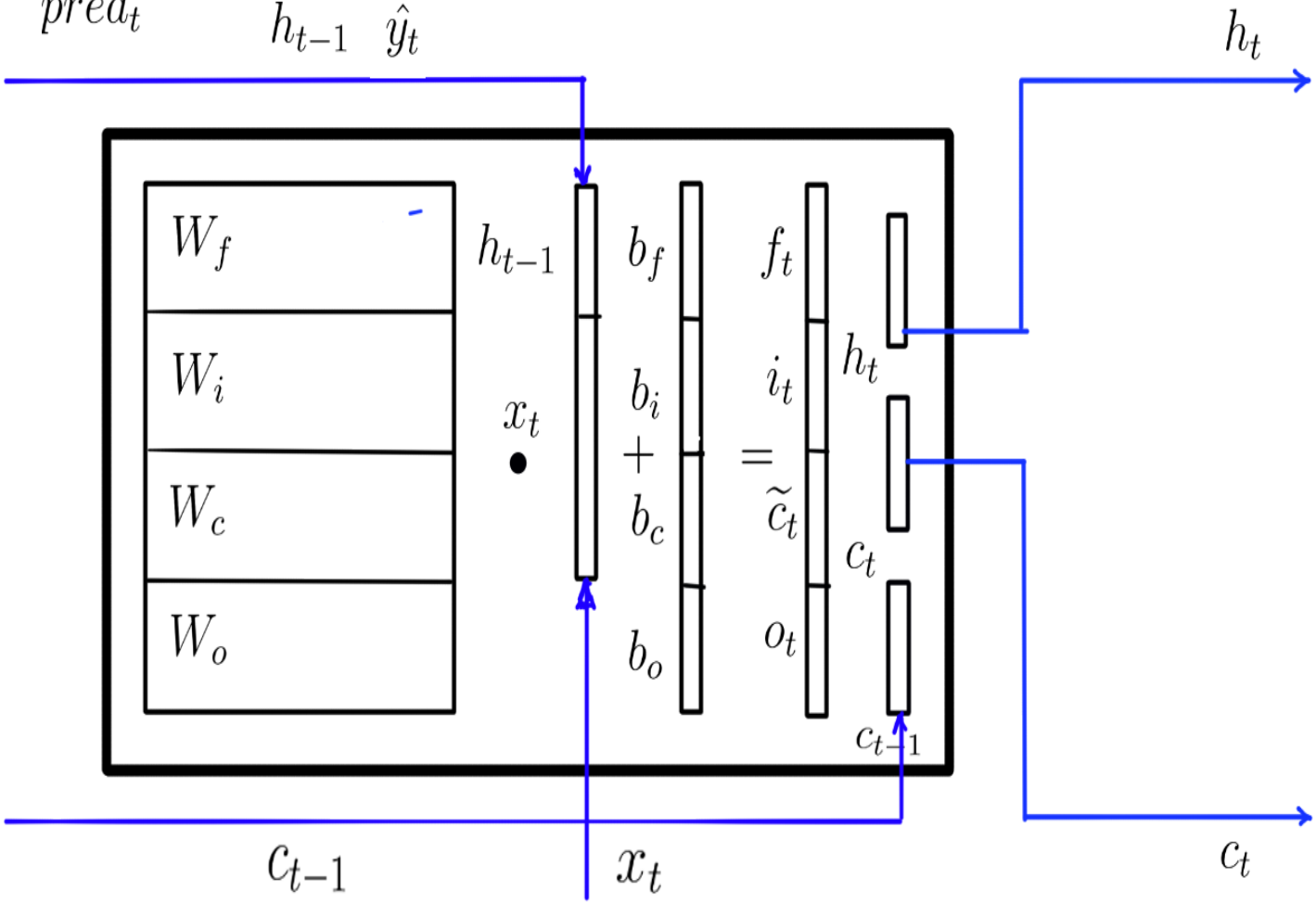
$softmax$

```
[[0.3]
 [0.2]
 [0.0]
 [0.0]
 [0.0]
 [0.0]
 [0.0]
 [0.0]
 [0.2]
 [0.0]]
```

$\hat{y}_t$

$$softmax(x_i) = \frac{exp(x_i))}{\sum_{i=0}^{k} exp(x_i)}$$

```
def softmax(self,logits):
    """row represents num classes but they may be real numbers
    That needs to be converted to probability"""
    r, c = logits.shape
    predsl = []
    for row in logits:
        inputs = np.squeeze(np.asarray(row))
        predsl.append(np.exp(inputs) / float(sum(np.exp(inputs))))
    return np.matrix(predsl).T
```



$$f_t = \sigma\left(W_f.[h_{t-1}, x_t] + b_f\right)$$
$$i_t = \sigma\left(W_i.[h_{t-1}, x_t] + b_i\right)$$
$$\widetilde{c}_t = \tanh\left(W_c.[h_{t-1}, x_t] + b_c\right)$$
$$o_t = \sigma\left(W_o.[h_{t-1}, x_t] + b_o\right)$$
$$c_t = \left(f_t * c_{t-1}\right) + \left(i_t * \widetilde{c}_t\right)$$
$$h_t = o_t * tanh(c_t)$$
$$pred_t = (h_t.W_y) + b_y$$
$$\hat{y}_t = softmax(pred_t)$$
$$E_t = crossEntropyLoss(\hat{y}_t, y_t)$$

LSTM(Long Short Term Memory):Internals

```
[[0.3]   [[1.]
 [0.2]    [0.]
 [0.0]    [0.]
 [0.0]    [0.]
 [0.0]    [0.]
 [0.0]    [0.]
 [0.0]    [0.]
 [0.0]    [0.]
 [0.2]    [0.]
 [0.0]]   [0.]]
```

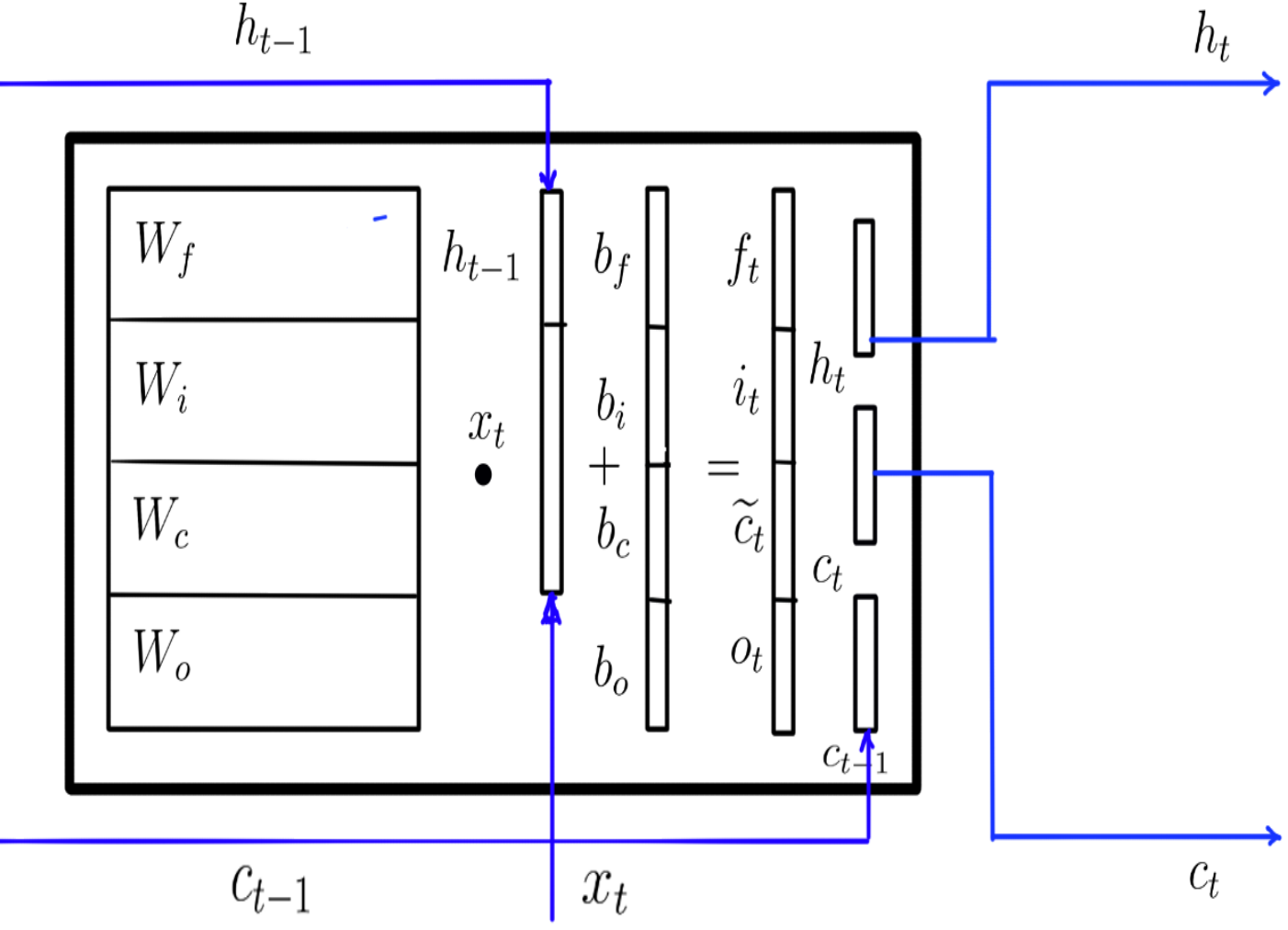
$\hat{y}_t$ y_t

$crossEntropyLoss$ [[1.171]] $crossEntropyLoss = -\sum_{i=0}^{t} y_i \log \hat{y}_i$

E_t

-log(.3) is 1.171, Everything else is zero, finally summed across the vertical axis

```
def loss(self,pred,labels):
    return np.multiply(labels, -np.log(pred)).sum(1)
```



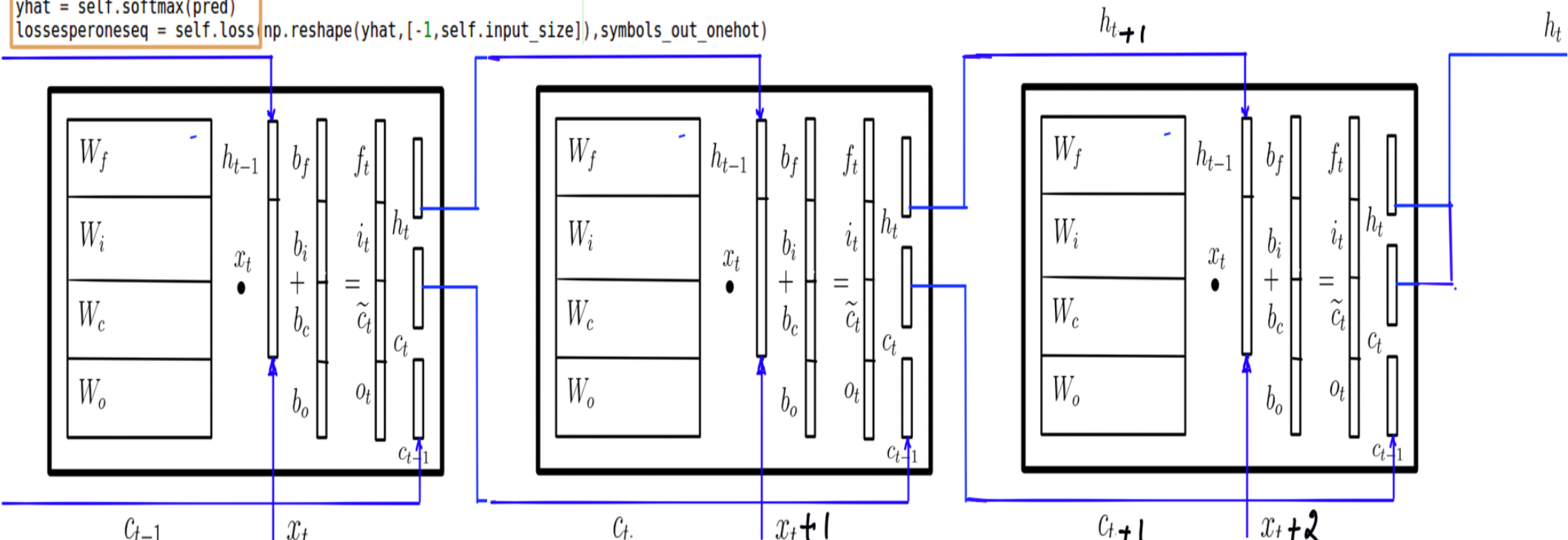
$$f_t = \sigma\left(W_f.[h_{t-1}, x_t] + b_f\right)$$
$$i_t = \sigma\left(W_i.[h_{t-1}, x_t] + b_i\right)$$
$$\widetilde{c}_t = \tanh\left(W_c.[h_{t-1}, x_t] + b_c\right)$$
$$o_t = \sigma\left(W_o.[h_{t-1}, x_t] + b_o\right)$$
$$c_t = (f_t * c_{t-1}) + (i_t * \widetilde{c}_t)$$
$$h_t = o_t * tanh(c_t)$$
$$pred_t = (h_t.W_y) + b_y$$
$$\hat{y}_t = softmax(pred_t)$$
$$E_t = crossEntropyLoss(\hat{y}_t, y_t)$$

LSTM(Long Short Term Memory):Internals

```
for seqnum in range(seq):
    (c,h)=prevstate
    z=np.concatenate((x,h),0)
    fico=np.dot(self.WLSTM[:,1:],z)+self.WLSTM[:,0].reshape(self.hidden_size*4,col)
    f=self.sigmoid_array(fico[0:self.hidden_size,:]+self.forget_bias)
    i=self.sigmoid_array(fico[self.hidden_size*1:self.hidden_size*2,:])
    cproj=self.tanh_array(fico[self.hidden_size*2:self.hidden_size*3,:])
    o=self.sigmoid_array(fico[self.hidden_size*3:self.hidden_size*4,:])
    cnew=(c * f) + (cproj * i)
    hnew=o*self.tanh_array(cnew)
    prevstate=cnew,hnew
    coldt[seqnum]=ct[seqnum-1]
    ct[seqnum]=cnew
    ht[seqnum]=hnew
    ft[seqnum]=f
    it[seqnum]=i
    ot[seqnum]=o
    cprojt[seqnum]=cproj
    zt[seqnum]=z
    prevstates.append(prevstate)
print("finalstate:",prevstate)
(c,h)=prevstate
pred=np.dot(np.reshape(h,[-1,self.hidden_size]),self.WOUT)+self.BIASOUT
yhat = self.softmax(pred)
lossesperoneseq = self.loss(np.reshape(yhat,[-1,self.input_size]),symbols_out_onehot)
```

$$pred_t = (h_t.W_y) + b_y$$
$$\hat{y}_t = softmax(pred_t)$$
$$E_t = crossEntropyLoss(\hat{y}_t, y_t)$$

- These steps done after sequence number of steps
- A sequence of 3 depicted below.



LSTM(Long Short Term Memory):Internals:Back Prop

```
#reverse
dh_next = np.zeros_like(h) #dh from the next character
dC_next = np.zeros_like(c)
dx=np.zeros_like(z).T
dy=yhat.copy()
dy=dy-np.reshape(symbols_out_onehot,[-1,1])
dWy=np.dot(dy,h.T)
dBy=dy
dht=np.dot(dy.T,self.WOUT.T).T
for t in reversed(range(seq)):
    z=  zt[t].T
    dht=dht+dh_next
    dot=np.multiply(dht,self.tanh_array(ct[t])*self.dsigmoid(ot[t]))
    dct=np.multiply(dht,ot[t]*self.dtanh(ct[t]))+dC_next
    dcproj=np.multiply(dct,it[t]*(1-cprojt[t]*cprojt[t]))
    dft=np.multiply(dct,coldt[t]*self.dsigmoid(ft[t]))
    dit=np.multiply(dct,cprojt[t]*self.dsigmoid(it[t]))
```

$$\frac{dE_t}{dBy_t} = \frac{\partial E_t}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial pred_t}$$

$$\frac{dE_t}{dBy_t} = \hat{y}_t - y_t$$

$$E_t = crossEntropyLoss(\hat{y}_t, y_t)$$

$$\hat{y}_t = softmax(pred_t)$$

$$pred_t = (h_t.W_y) + b_y$$

$$h_t = o_t * tanh(c_t)$$

$$c_t = (f_t * c_{t-1}) + (i_t * \widetilde{c}_t)$$

$$o_t = \sigma\left(W_o.[h_{t-1}, x_t] + b_o\right)$$

$$\widetilde{c}_t = \tanh\left(W_c.[h_{t-1}, x_t] + b_c\right)$$

$$i_t = \sigma\left(W_i.[h_{t-1}, x_t] + b_i\right)$$

$$f_t = \sigma\left(W_f.[h_{t-1}, x_t] + b_f\right)$$

LSTM(Long Short Term Memory):Internals:Back Prop

```
#reverse
dh_next = np.zeros_like(h) #dh from the next character
dC_next = np.zeros_like(c)
dx=np.zeros_like(z).T
dy=yhat.copy()
dy=dy-np.reshape(symbols_out_onehot,[-1,1])
dWy=np.dot(dy,h.T)
dBy=dy
dht=np.dot(dy.T,self.WOUT.T).T
for t in reversed(range(seq)):
    z=  zt[t].T
    dht=dht+dh_next
    dot=np.multiply(dht,self.tanh_array(ct[t])*self.dsigmoid(ot[t]))
    dct=np.multiply(dht,ot[t]*self.dtanh(ct[t]))+dC_next
    dcproj=np.multiply(dct,it[t]*(1-cprojt[t]*cprojt[t]))
    dft=np.multiply(dct,coldt[t]*self.dsigmoid(ft[t]))
    dit=np.multiply(dct,cprojt[t]*self.dsigmoid(it[t]))
```

$$\frac{\mathrm{d}E_t}{\mathrm{d}Wy_t} = \frac{\partial E_t}{\partial \hat{y}_t} . \frac{\partial \hat{y}_t}{\partial pred_t} . \frac{\partial pred_t}{\partial Wy_t}$$

$$\frac{\mathrm{d}E_t}{\mathrm{d}Wy_t} = (\hat{y}_t - y_t).h_t$$

$$E_t = crossEntropyLoss(\hat{y}_t, y_t)$$

$$\hat{y}_t = softmax(pred_t)$$

$$pred_t = (h_t.W_y) + b_y$$

$$h_t = o_t * tanh(c_t)$$

$$c_t = (f_t * c_{t-1}) + (i_t * \widetilde{c}_t)$$

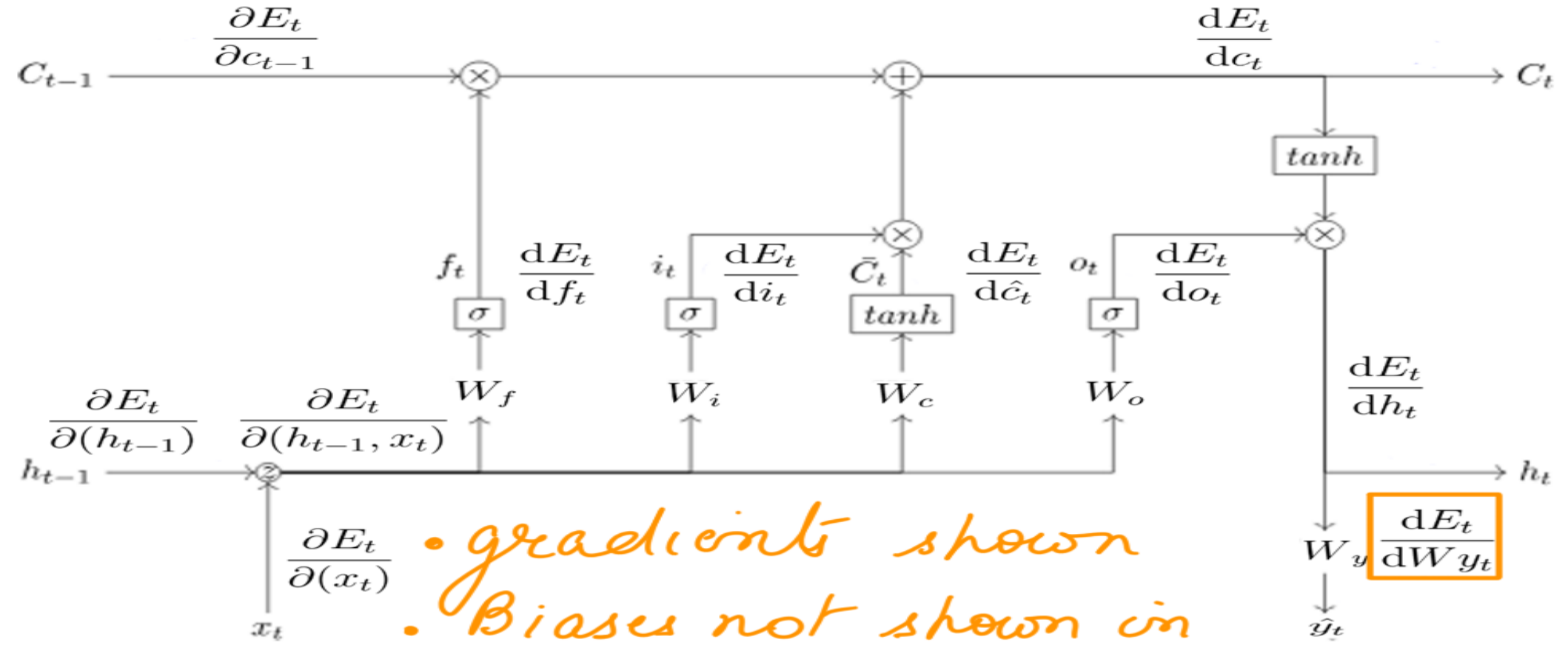
$$o_t = \sigma\left(W_o.[h_{t-1}, x_t] + b_o\right)$$

$$\widetilde{c}_t = \tanh\left(W_c.[h_{t-1}, x_t] + b_c\right)$$

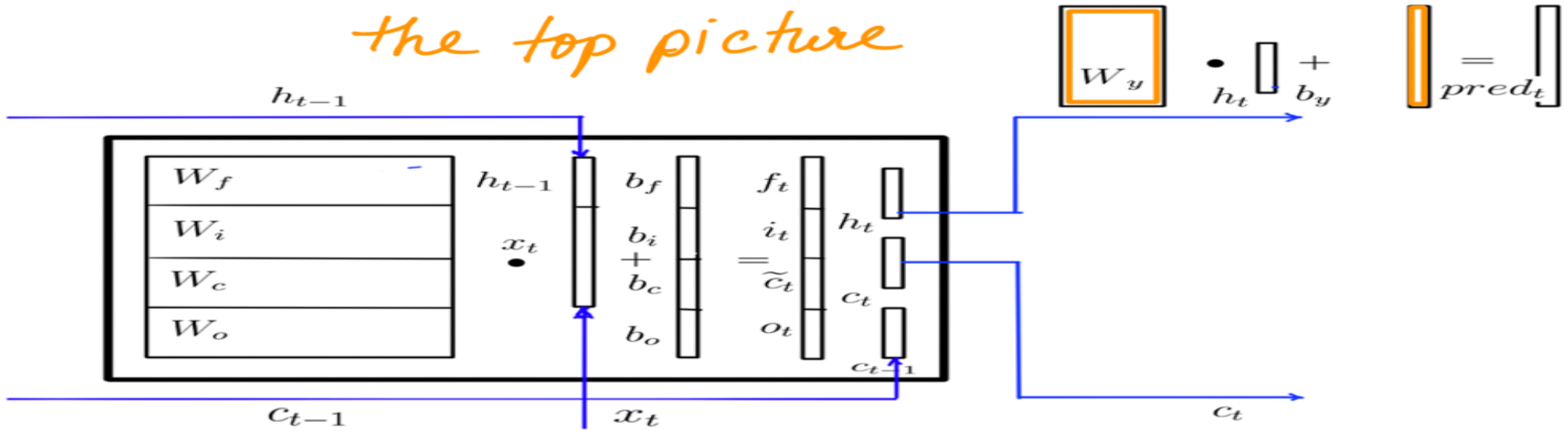
$$i_t = \sigma\left(W_i.[h_{t-1}, x_t] + b_i\right)$$

$$f_t = \sigma\left(W_f.[h_{t-1}, x_t] + b_f\right)$$

LSTM(Long Short Term Memory):Internals:Back Prop



- gradients shown
- Biases not shown in the top picture



LSTM(Long Short Term Memory):Internals:Back Prop

```
#reverse
dh_next = np.zeros_like(h) #dh from the next character
dC_next = np.zeros_like(c)
dx=np.zeros_like(z).T
dy=yhat.copy()
dy=dy-np.reshape(symbols_out_onehot,[-1,1])
dWy=np.dot(dy,h.T)
dBy=dy
dht=np.dot(dy.T,self.WOUT.T).T
for t in reversed(range(seq)):
    z=  zt[t].T
    dht=dht+dh_next
    dot=np.multiply(dht,self.tanh_array(ct[t])*self.dsigmoid(ot[t]))
    dct=np.multiply(dht,ot[t]*self.dtanh(ct[t]))+dC_next
    dcproj=np.multiply(dct,it[t]*(1-cprojt[t]*cprojt[t]))
    dft=np.multiply(dct,coldt[t]*self.dsigmoid(ft[t]))
    dit=np.multiply(dct,cprojt[t]*self.dsigmoid(it[t]))
```

$$\frac{dE_t}{dh_t} = \frac{\partial E_t}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial pred_t} \cdot \frac{\partial pred_t}{\partial h_t}$$

$$\frac{dE_t}{dh_t} = (\hat{y}_t - y_t.W_y) + dh\_next$$

- Wout is W_y.
- Recurrent element zeros to begin with
- Done sequence number of times.

$$E_t = crossEntropyLoss(\hat{y}_t, y_t)$$

$$\hat{y}_t = softmax(pred_t)$$

$$pred_t = (h_t.W_y) + b_y$$

$$h_t = o_t * tanh(c_t)$$

$$c_t = (f_t * c_{t-1}) + (i_t * \tilde{c}_t)$$

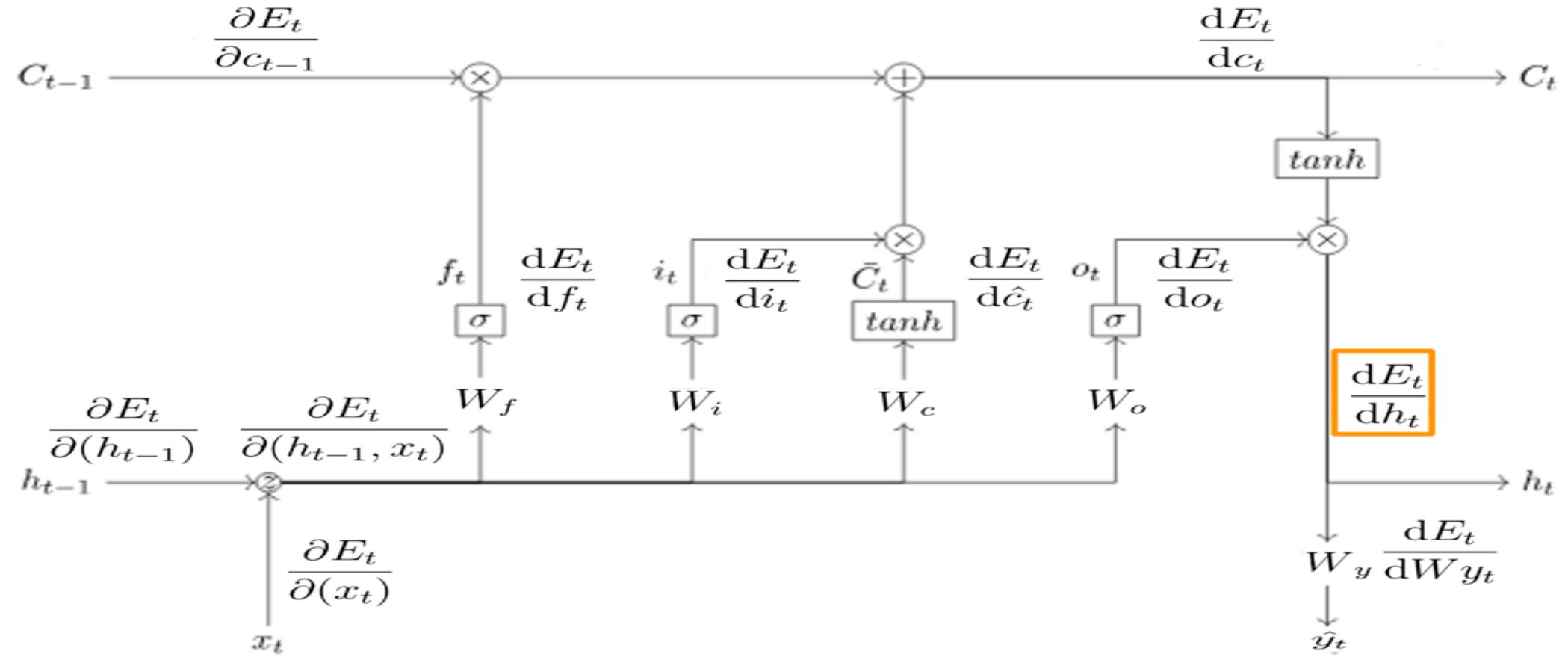
$$o_t = \sigma\left(W_o.[h_{t-1}, x_t] + b_o\right)$$

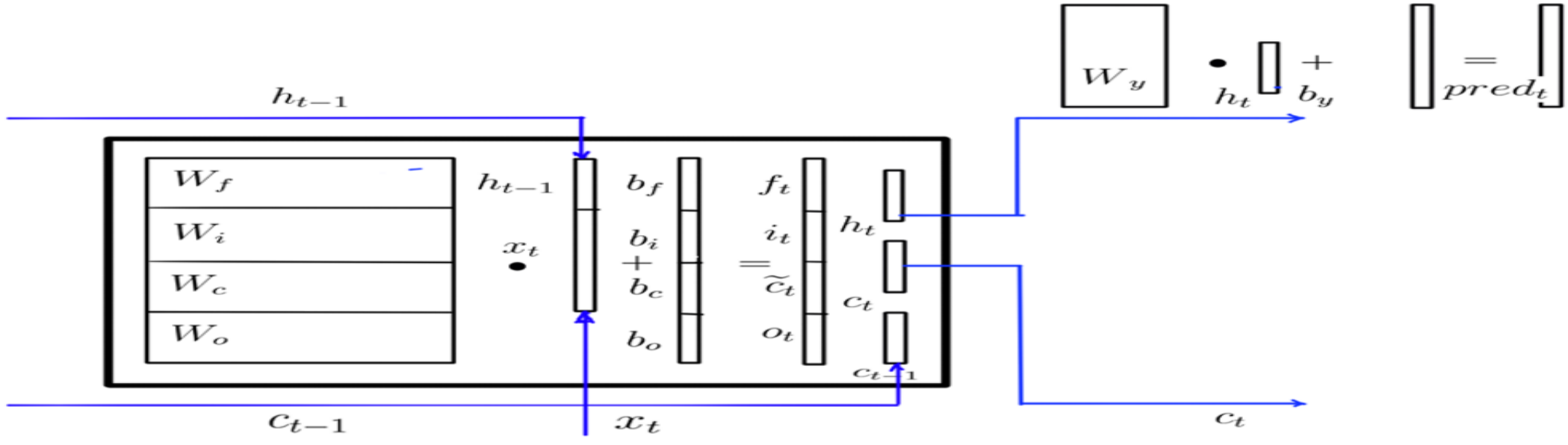
$$\tilde{c}_t = \tanh\left(W_c.[h_{t-1}, x_t] + b_c\right)$$

$$i_t = \sigma\left(W_i.[h_{t-1}, x_t] + b_i\right)$$

$$f_t = \sigma\left(W_f.[h_{t-1}, x_t] + b_f\right)$$

LSTM(Long Short Term Memory):Internals:Back Prop





LSTM(Long Short Term Memory):Internals:Back Prop

```
#reverse
dh_next = np.zeros_like(h) #dh from the next characte
dC_next = np.zeros_like(c)
dx=np.zeros_like(z).T
dy=yhat.copy()
dy=dy-np.reshape(symbols_out_onehot,[-1,1])
dWy=np.dot(dy,h.T)
dBy=dy
dht=np.dot(dy.T,self.WOUT.T).T
for t in reversed(range(seq)):
    z=  zt[t].T
    dht=dht+dh_next
    dot=np.multiply(dht,self.tanh_array(ct[t])*self.dsigmoid(ot[t]))
    dct=np.multiply(dht,ot[t]*self.dtanh(ct[t]))+dC_next
    dcproj=np.multiply(dct,it[t]*(1-cprojt[t]*cprojt[t]))
    dft=np.multiply(dct,coldt[t]*self.dsigmoid(ft[t]))
    dit=np.multiply(dct,cprojt[t]*self.dsigmoid(it[t]))
```

$$\frac{dE_t}{do_t} = \frac{\partial E_t}{\partial h_t}.\frac{\partial h_t}{\partial o_t}$$

$$\frac{dE_t}{do_t} = \frac{\partial E_t}{\partial h_t}.tanh(c_t).dsigmoid(o_t)$$

```
def dsigmoid(self,f):
    return f*(1-f)
```

$$E_t = crossEntropyLoss(\hat{y}_t, y_t)$$

$$\hat{y}_t = softmax(pred_t)$$

$$pred_t = (h_t.W_y) + b_y$$

$$h_t = o_t * tanh(c_t)$$

$$c_t = (f_t * c_{t-1}) + (i_t * \widetilde{c}_t)$$

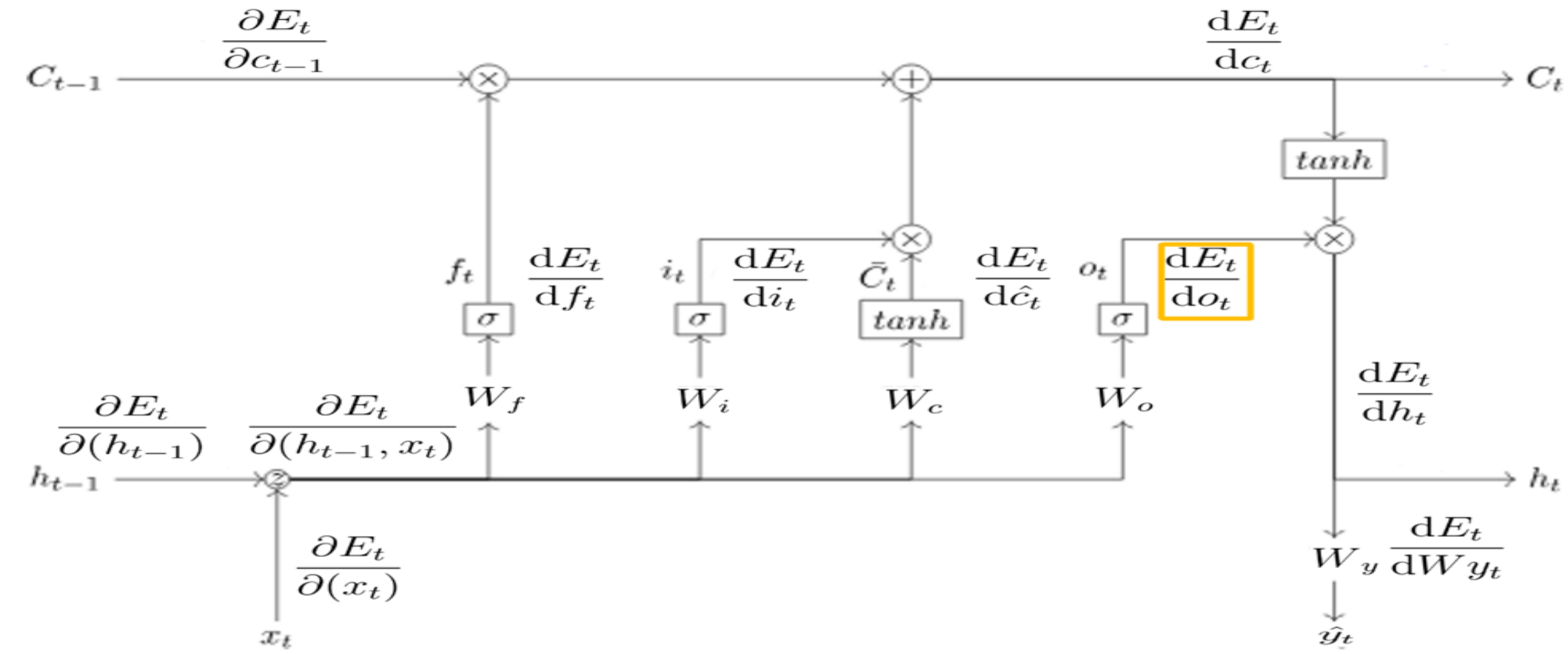
$$o_t = \sigma\left(W_o.[h_{t-1}, x_t] + b_o\right)$$

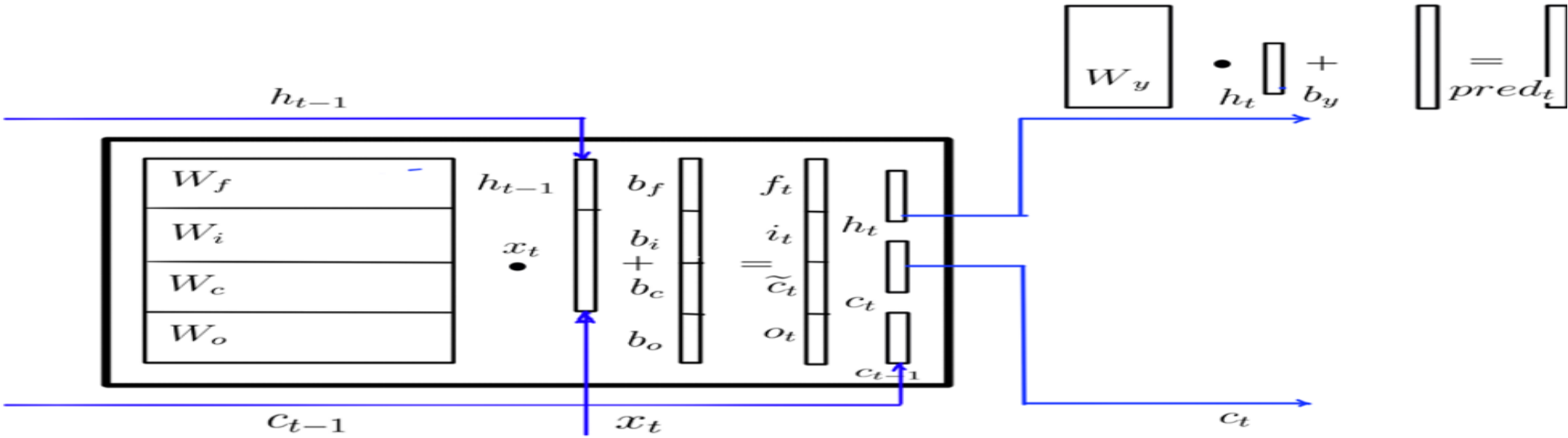
$$\widetilde{c}_t = \tanh\left(W_c.[h_{t-1}, x_t] + b_c\right)$$

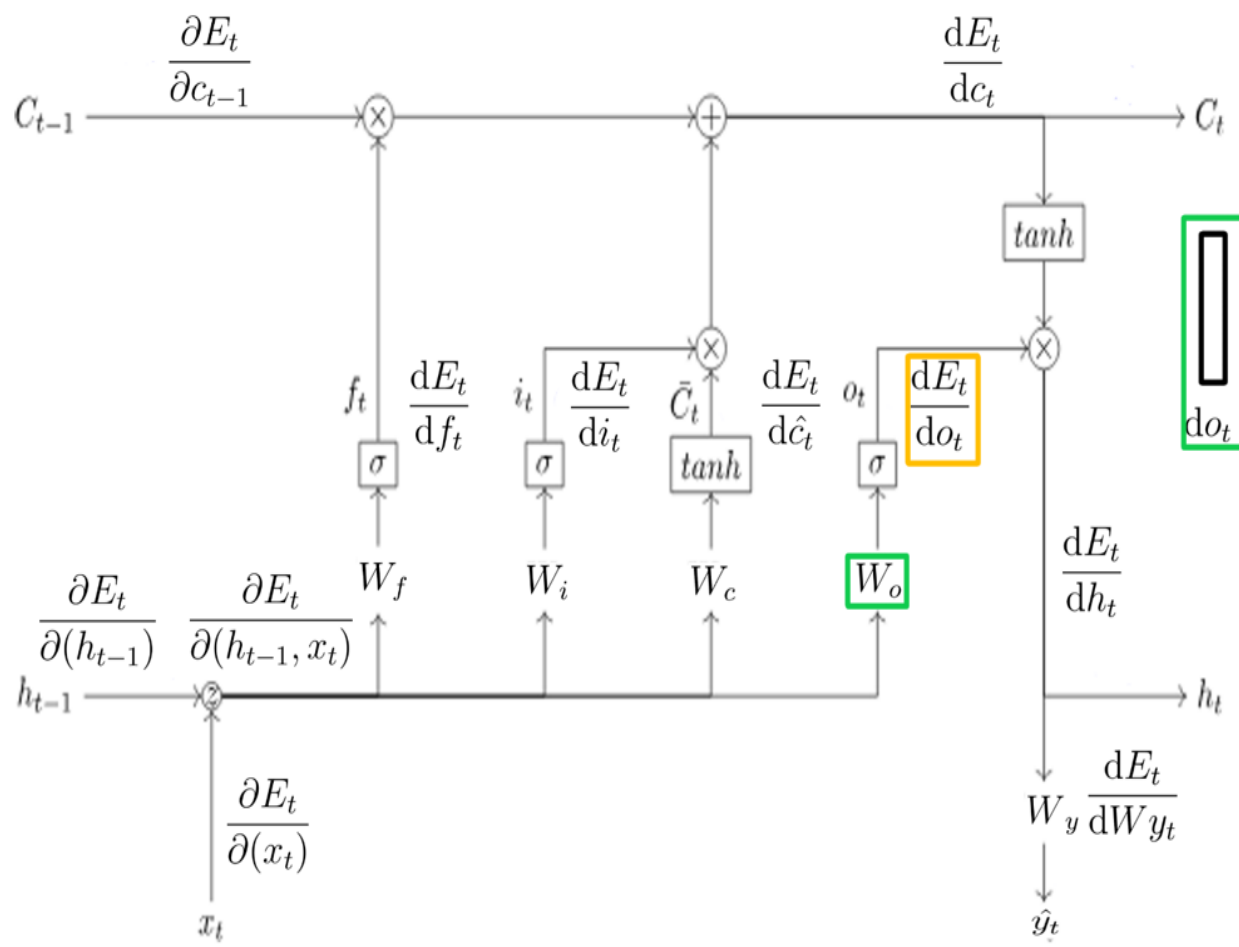
$$i_t = \sigma\left(W_i.[h_{t-1}, x_t] + b_i\right)$$

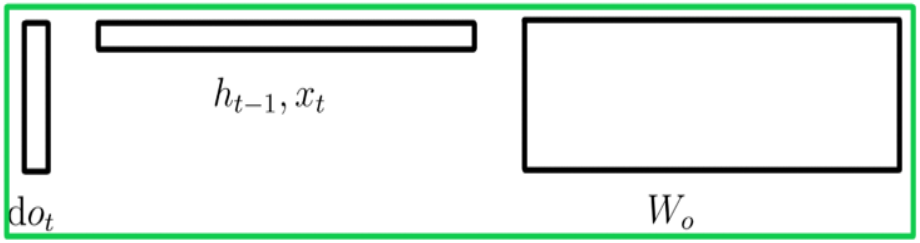
$$f_t = \sigma\left(W_f.[h_{t-1}, x_t] + b_f\right)$$

LSTM(Long Short Term Memory):Internals:Back Prop

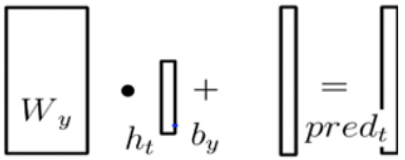


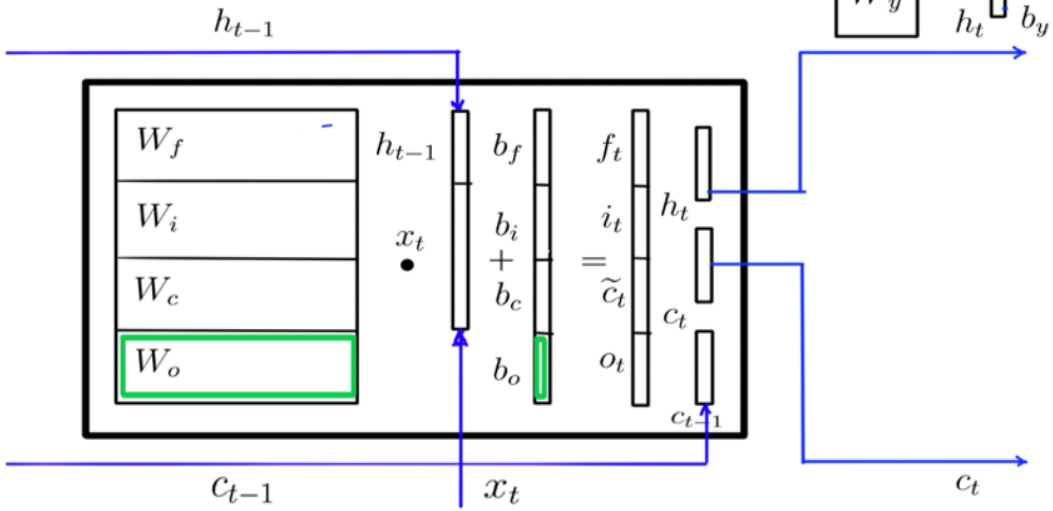







```
dwo+=np.dot(dot,z)
dbo+=dot
dxo=np.dot(dot.T,self.WLSTM[self.hidden size*3:self.hidden size*4,1:])
```





LSTM(Long Short Term Memory):Internals:Back Prop

```
#reverse
dh_next = np.zeros_like(h) #dh from the next char
dC_next = np.zeros_like(c)
dx=np.zeros_like(z).T
dy=yhat.copy()
dy=dy-np.reshape(symbols_out_onehot,[-1,1])
dWy=np.dot(dy,h.T)
dBy=dy
dht=np.dot(dy.T,self.WOUT.T).T
for t in reversed(range(seq)):
    z=  zt[t].T
    dht=dht+dh_next
    dot=np.multiply(dht,self.tanh_array(ct[t])*self.dsigmoid(ot[t]))
    dct=np.multiply(dht,ot[t]*self.dtanh(ct[t]))+dC_next
    dcproj=np.multiply(dct,it[t]*(1-cprojt[t]*cprojt[t]))
    dft=np.multiply(dct,coldt[t]*self.dsigmoid(ft[t]))
    dit=np.multiply(dct,cprojt[t]*self.dsigmoid(it[t]))
```

$$\frac{dE_t}{dc_t} = \frac{\partial E_t}{\partial h_t}.\frac{\partial h_t}{\partial c_t}$$

$$\frac{dE_t}{dc_t} = \frac{\partial E_t}{\partial h_t}.o_t.dtanh(c_t) + dc\_next$$

```
def dtanh(self,f):
    tanhf=np.tanh(f)
    return 1 - tanhf * tanhf
```

- Because of recurrent cell state $c_{(t-1)}$.

$$E_t = crossEntropyLoss(\hat{y}_t, y_t)$$

$$\hat{y}_t = softmax(pred_t)$$

$$pred_t = (h_t.W_y) + b_y$$

$$h_t = o_t * tanh(c_t)$$

$$c_t = (f_t * c_{t-1}) + (i_t * \widetilde{c}_t)$$

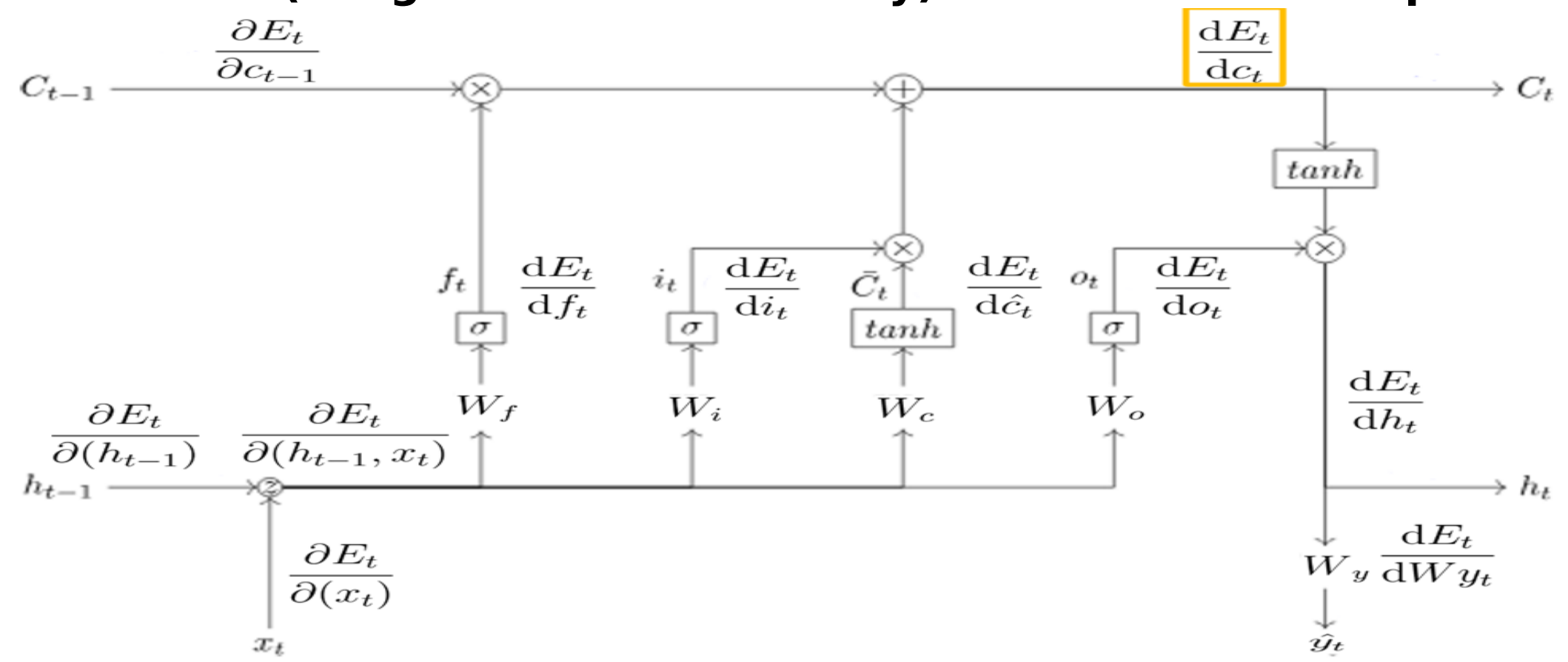
$$o_t = \sigma\left(W_o.[h_{t-1}, x_t] + b_o\right)$$

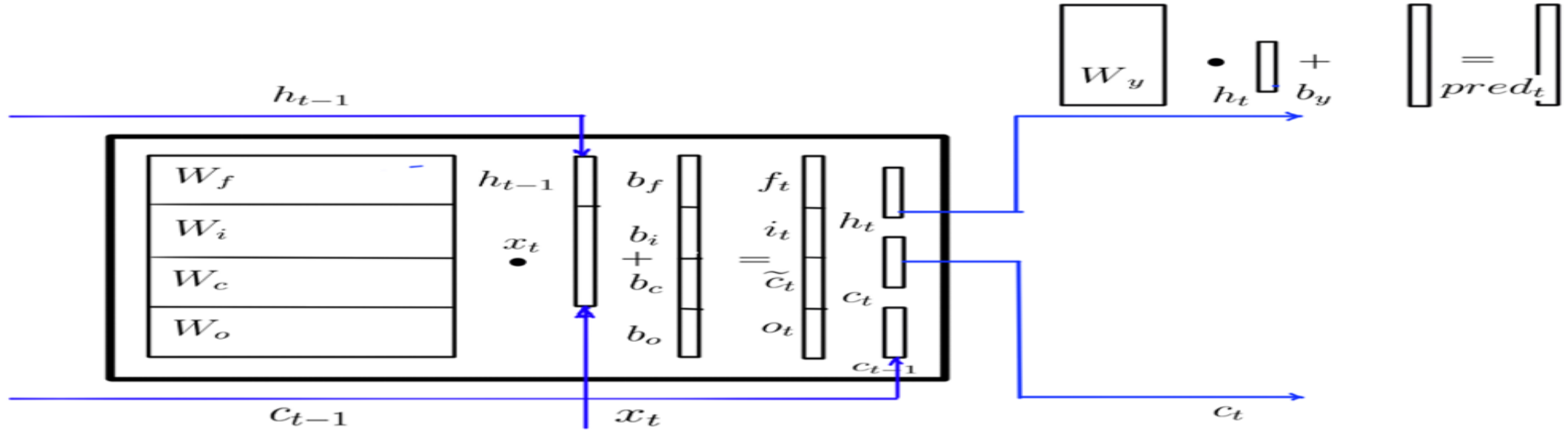
$$\widetilde{c}_t = \tanh\left(W_c.[h_{t-1}, x_t] + b_c\right)$$

$$i_t = \sigma\left(W_i.[h_{t-1}, x_t] + b_i\right)$$

$$f_t = \sigma\left(W_f.[h_{t-1}, x_t] + b_f\right)$$

LSTM(Long Short Term Memory):Internals:Back Prop





LSTM(Long Short Term Memory):Internals:Back Prop

```
#reverse
dh_next = np.zeros_like(h) #dh from the next character
dC_next = np.zeros_like(c)
dx=np.zeros_like(z).T
dy=yhat.copy()
dy=dy-np.reshape(symbols_out_onehot,[-1,1])
dWy=np.dot(dy,h.T)
dBy=dy
dht=np.dot(dy.T,self.WOUT.T).T
for t in reversed(range(seq)):
    z=  zt[t].T
    dht=dht+dh_next
    dot=np.multiply(dht,self.tanh_array(ct[t])*self.dsigmoid(ot[t]))
    dct=np.multiply(dht,ot[t]*self.dtanh(ct[t]))+dC_next
    dcproj=np.multiply(dct,it[t]*(1-cprojt[t]*cprojt[t]))
    dft=np.multiply(dct,coldt[t]*self.dsigmoid(ft[t]))
    dit=np.multiply(dct,cprojt[t]*self.dsigmoid(it[t]))
```

$$\frac{dE_t}{d\hat{c}_t} = \frac{\partial E_t}{\partial c_t}.\frac{\partial c_t}{\partial \hat{c}_t}$$

$$\frac{dE_t}{d\hat{c}_t} = \frac{\partial E_t}{\partial c_t}.i_t.dtanh(\hat{c}_t)$$

- Previously calculated dct
- inlined dtanh
- $\hat{c}$ is cproj.

$$E_t = crossEntropyLoss(\hat{y}_t, y_t)$$

$$\hat{y}_t = softmax(pred_t)$$

$$pred_t = (h_t.W_y) + b_y$$

$$h_t = o_t * tanh(c_t)$$

$$c_t = (f_t * c_{t-1}) + (i_t * \widetilde{c}_t)$$

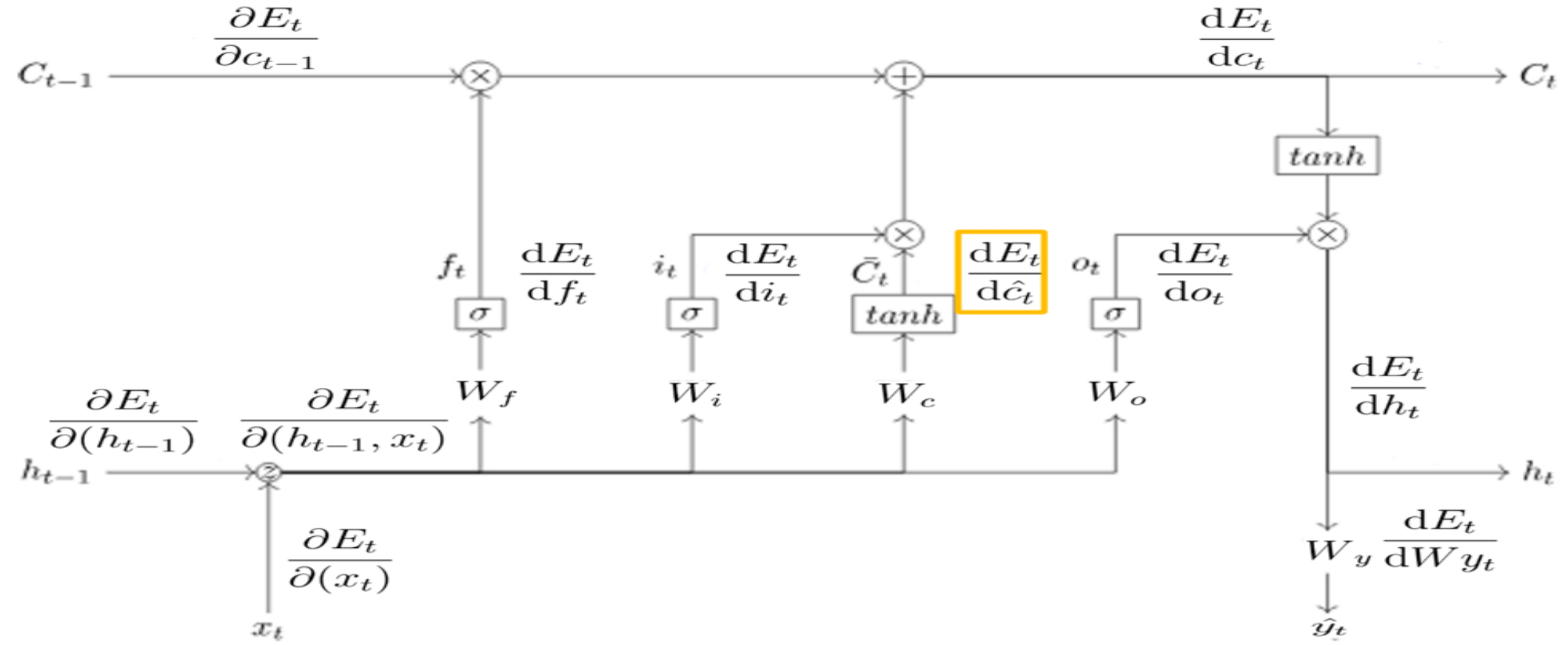
$$o_t = \sigma\left(W_o.[h_{t-1}, x_t] + b_o\right)$$

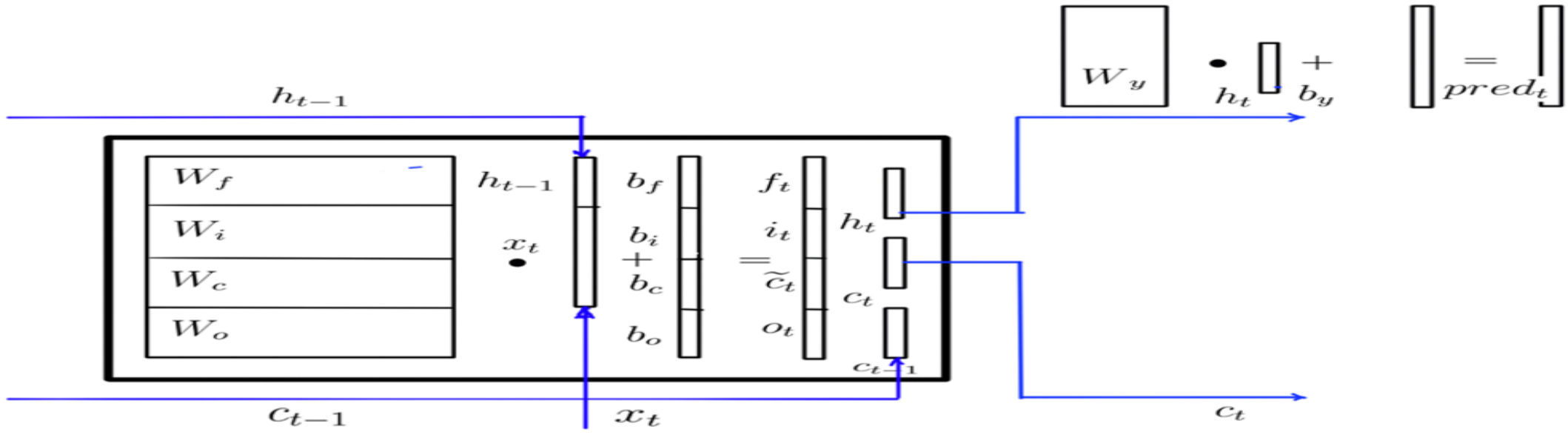
$$\widetilde{c}_t = \tanh\left(W_c.[h_{t-1}, x_t] + b_c\right)$$

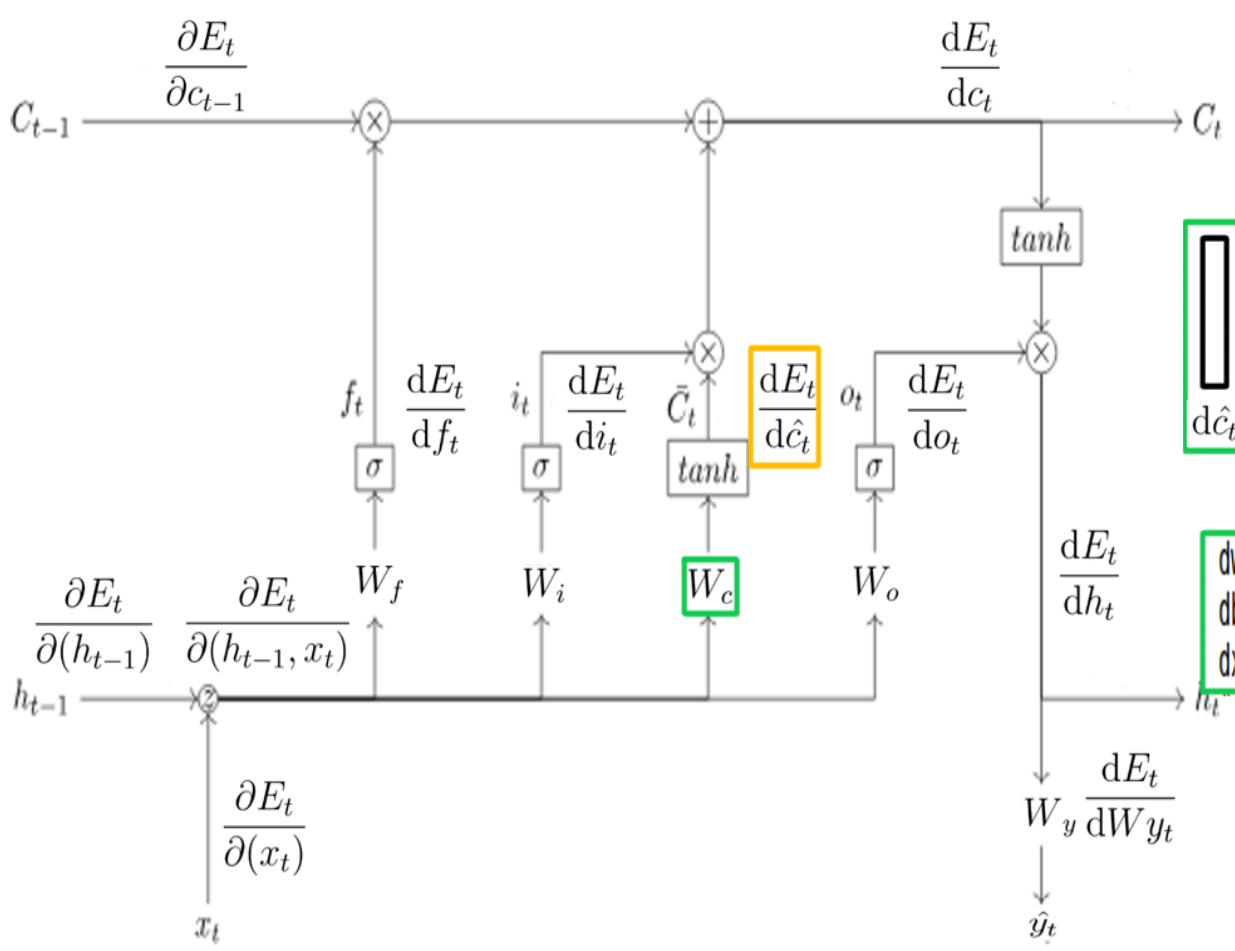
$$i_t = \sigma\left(W_i.[h_{t-1}, x_t] + b_i\right)$$

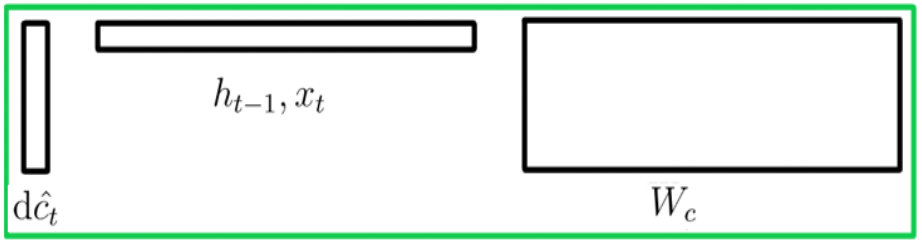
$$f_t = \sigma\left(W_f.[h_{t-1}, x_t] + b_f\right)$$

LSTM(Long Short Term Memory):Internals:Back Prop

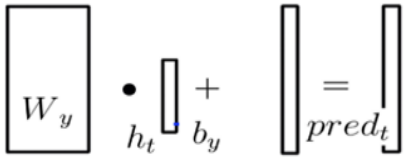


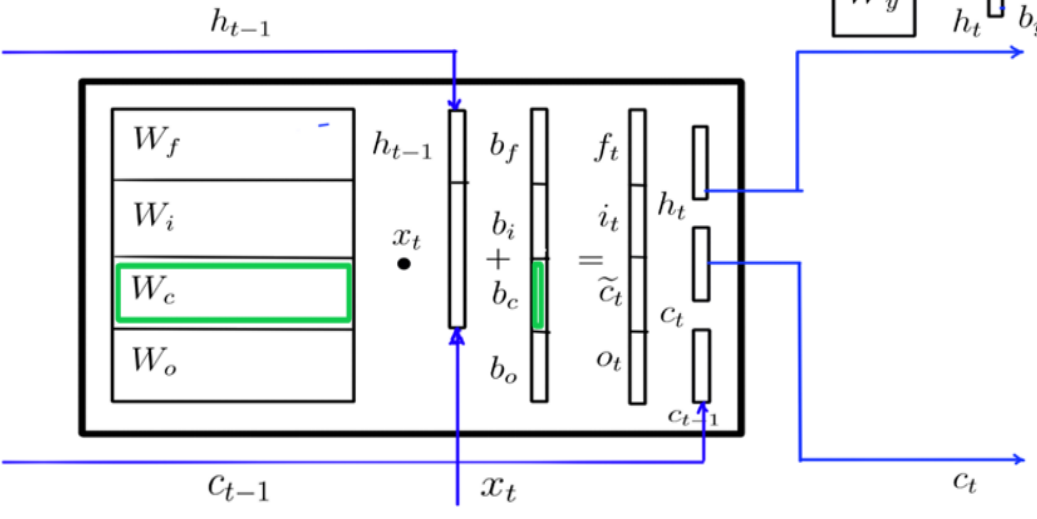







```
dwc+=np.dot(dcproj,z)
dbc+=dcproj
dxc=np.dot(dcproj.T,self.WLSTM[self.hidden_size*2:self.hidden_size*3,1:])
```





LSTM(Long Short Term Memory):Internals:Back Prop

```
#reverse
dh_next = np.zeros_like(h) #dh from the next charact
dC_next = np.zeros_like(c)
dx=np.zeros_like(z).T
dy=yhat.copy()
dy=dy-np.reshape(symbols_out_onehot,[-1,1])
dWy=np.dot(dy,h.T)
dBy=dy
dht=np.dot(dy.T,self.WOUT.T).T
for t in reversed(range(seq)):
    z=  zt[t].T
    dht=dht+dh_next
    dot=np.multiply(dht,self.tanh_array(ct[t])*self.dsigmoid(ot[t]))
    dct=np.multiply(dht,ot[t]*self.dtanh(ct[t]))+dC_next
    dcproj=np.multiply(dct,it[t]*(1-cprojt[t]*cprojt[t]))
    dft=np.multiply(dct,coldt[t]*self.dsigmoid(ft[t]))
    dit=np.multiply(dct,cprojt[t]*self.dsigmoid(it[t]))
```

$$\frac{dE_t}{df_t} = \frac{\partial E_t}{\partial c_t}.\frac{\partial c_t}{\partial f_t}$$

$$\frac{dE_t}{df_t} = \frac{\partial E_t}{\partial c_t}.c_{t-1}.dsigmoid(f_t)$$

$$E_t = crossEntropyLoss(\hat{y}_t, y_t)$$

$$\hat{y}_t = softmax(pred_t)$$

$$pred_t = (h_t.W_y) + b_y$$

$$h_t = o_t * tanh(c_t)$$

$$c_t = (f_t * c_{t-1}) + (i_t * \widetilde{c}_t)$$

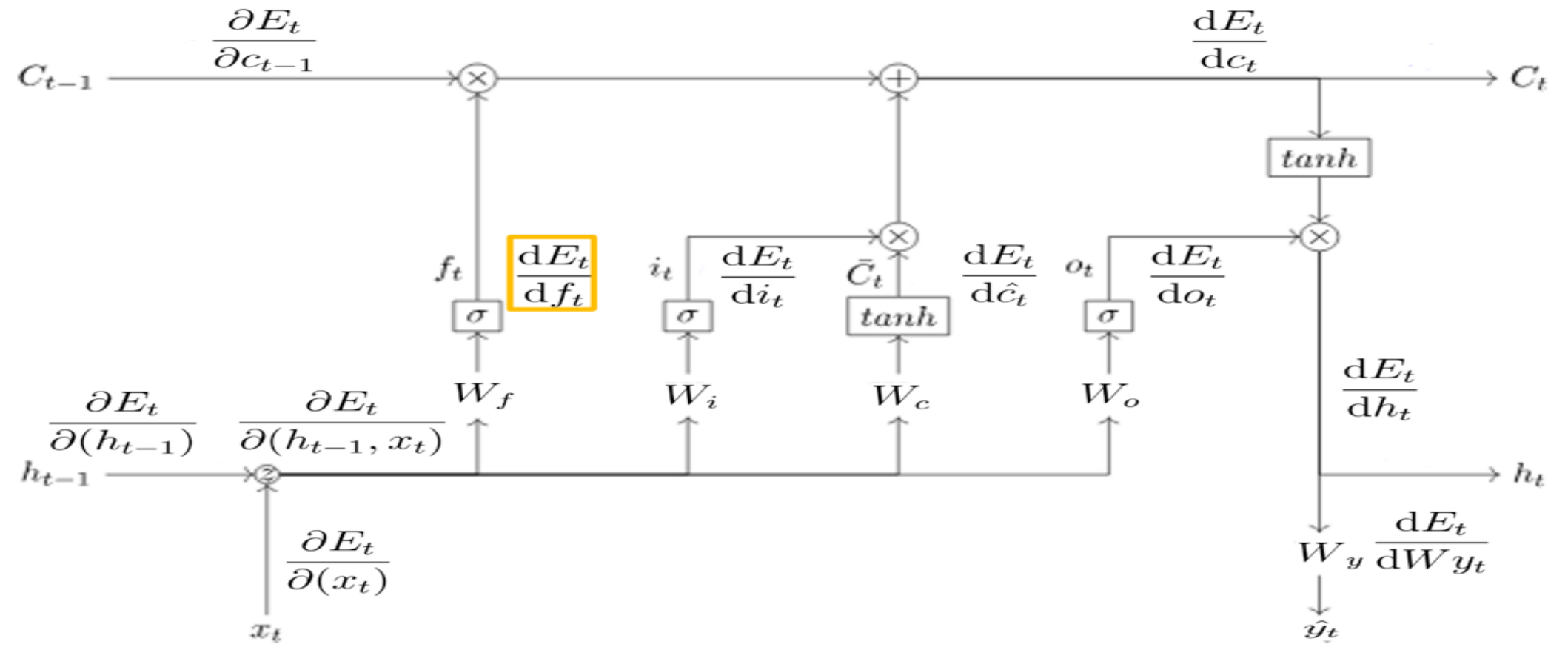
$$o_t = \sigma\left(W_o.[h_{t-1}, x_t] + b_o\right)$$

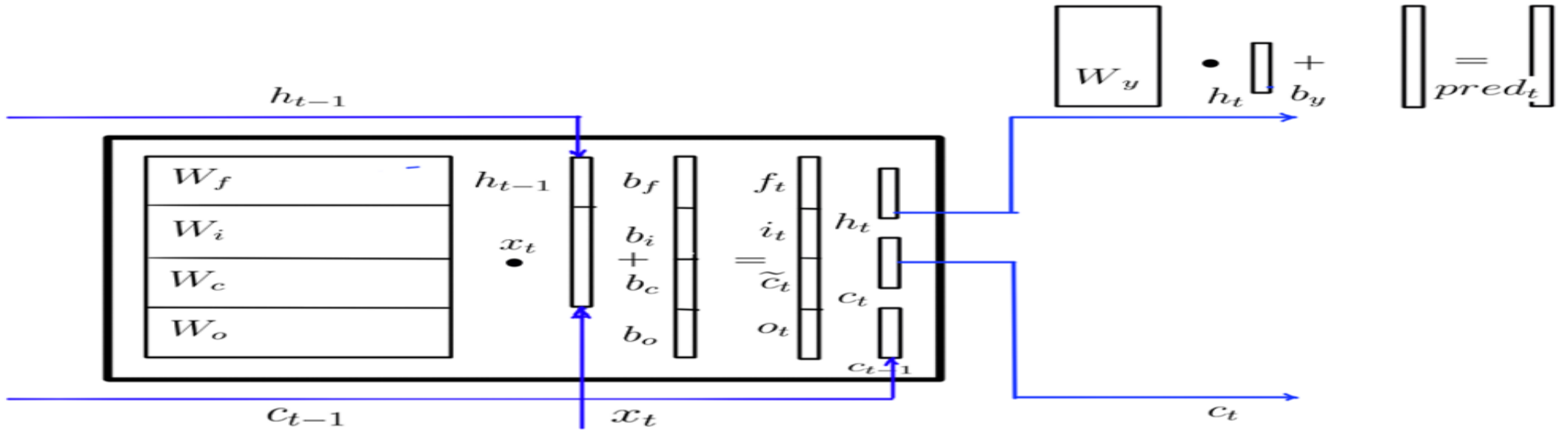
$$\widetilde{c}_t = \tanh\left(W_c.[h_{t-1}, x_t] + b_c\right)$$

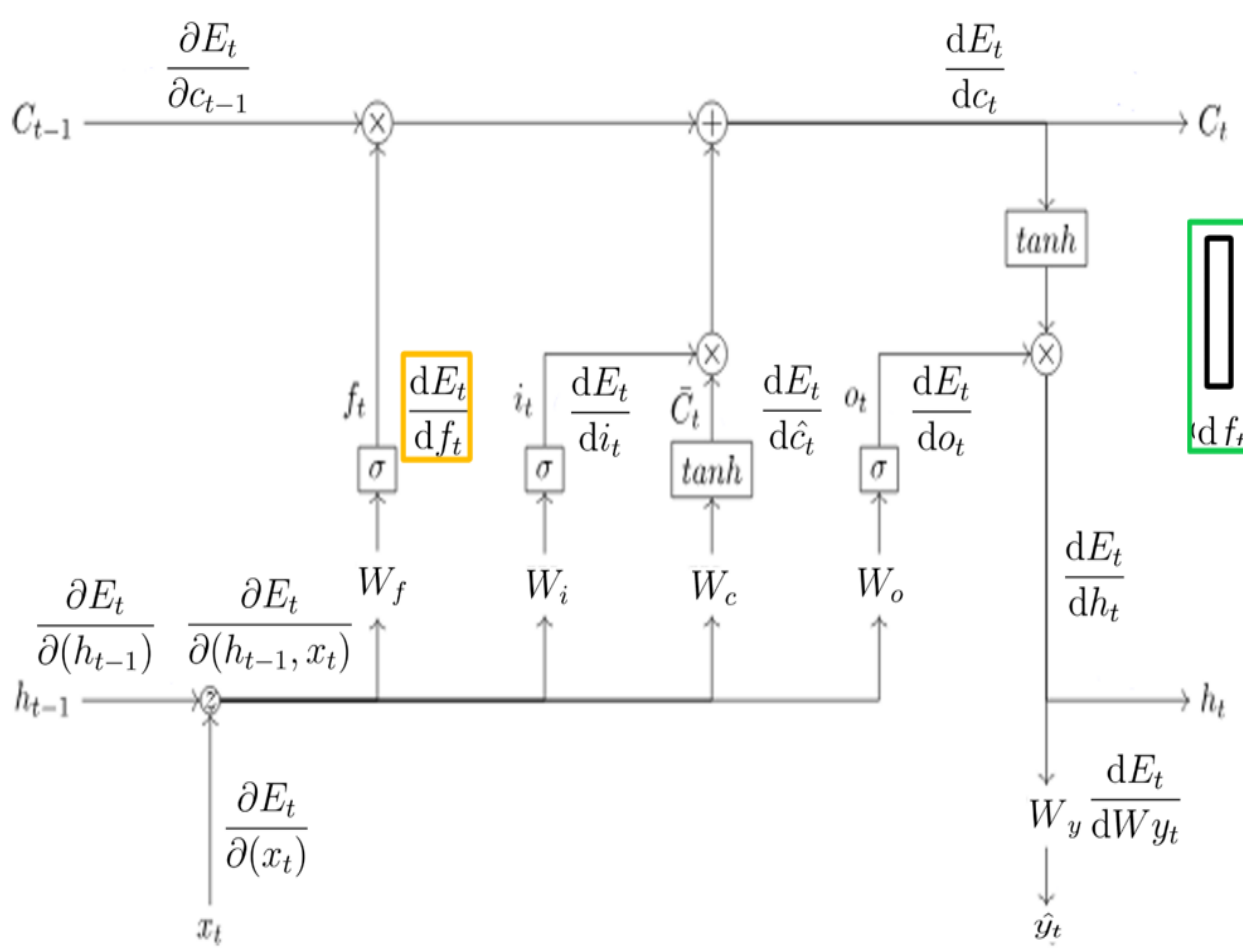
$$i_t = \sigma\left(W_i.[h_{t-1}, x_t] + b_i\right)$$

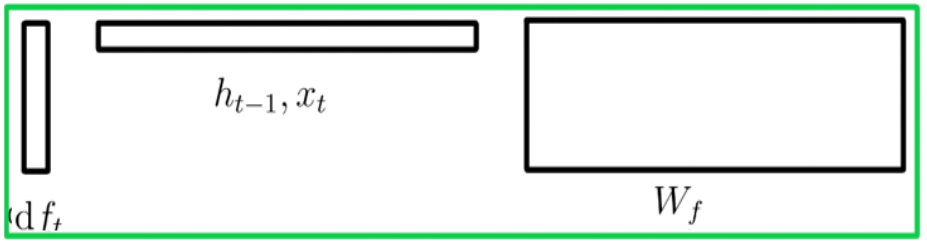
$$f_t = \sigma\left(W_f.[h_{t-1}, x_t] + b_f\right)$$

LSTM(Long Short Term Memory):Internals:Back Prop

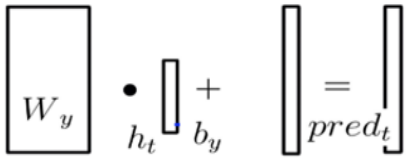


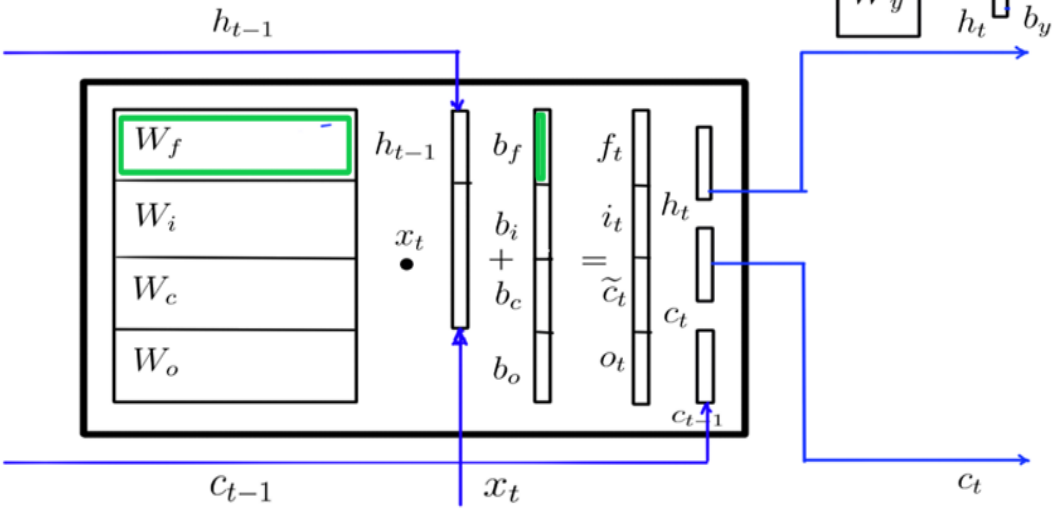







```
dwf+=np.dot(dft,z)
print("dxf:",dft.shape,":",z.shape,":",dwf.shape)
dbf+=dft
dxf=np.dot(dft.T,self.WLSTM[0:self.hidden_size,1:])
```





LSTM(Long Short Term Memory):Internals:Back Prop

```
#reverse
dh_next = np.zeros_like(h) #dh from the next character
dC_next = np.zeros_like(c)
dx=np.zeros_like(z).T
dy=yhat.copy()
dy=dy-np.reshape(symbols_out_onehot,[-1,1])
dWy=np.dot(dy,h.T)
dBy=dy
dht=np.dot(dy.T,self.WOUT.T).T
for t in reversed(range(seq)):
    z=  zt[t].T
    dht=dht+dh_next
    dot=np.multiply(dht,self.tanh_array(ct[t])*self.dsigmoid(ot[t]))
    dct=np.multiply(dht,ot[t]*self.dtanh(ct[t]))+dC_next
    dcproj=np.multiply(dct,it[t]*(1-cprojt[t]*cprojt[t]))
    dft=np.multiply(dct,coldt[t]*self.dsigmoid(ft[t]))
    dit=np.multiply(dct,cprojt[t]*self.dsigmoid(it[t]))
```

$$\frac{\mathrm{d}E_t}{\mathrm{d}i_t} = \frac{\partial E_t}{\partial c_t}.\frac{\partial c_t}{\partial i_t}$$

$$\frac{\mathrm{d}E_t}{\mathrm{d}i_t} = \frac{\partial E_t}{\partial c_t}.\hat{c}_t.dsigmoid(i_t)$$

$$E_t = crossEntropyLoss(\hat{y}_t, y_t)$$

$$\hat{y}_t = softmax(pred_t)$$

$$pred_t = (h_t.W_y) + b_y$$

$$h_t = o_t * tanh(c_t)$$

$$c_t = (f_t * c_{t-1}) + (i_t * \widetilde{c}_t)$$

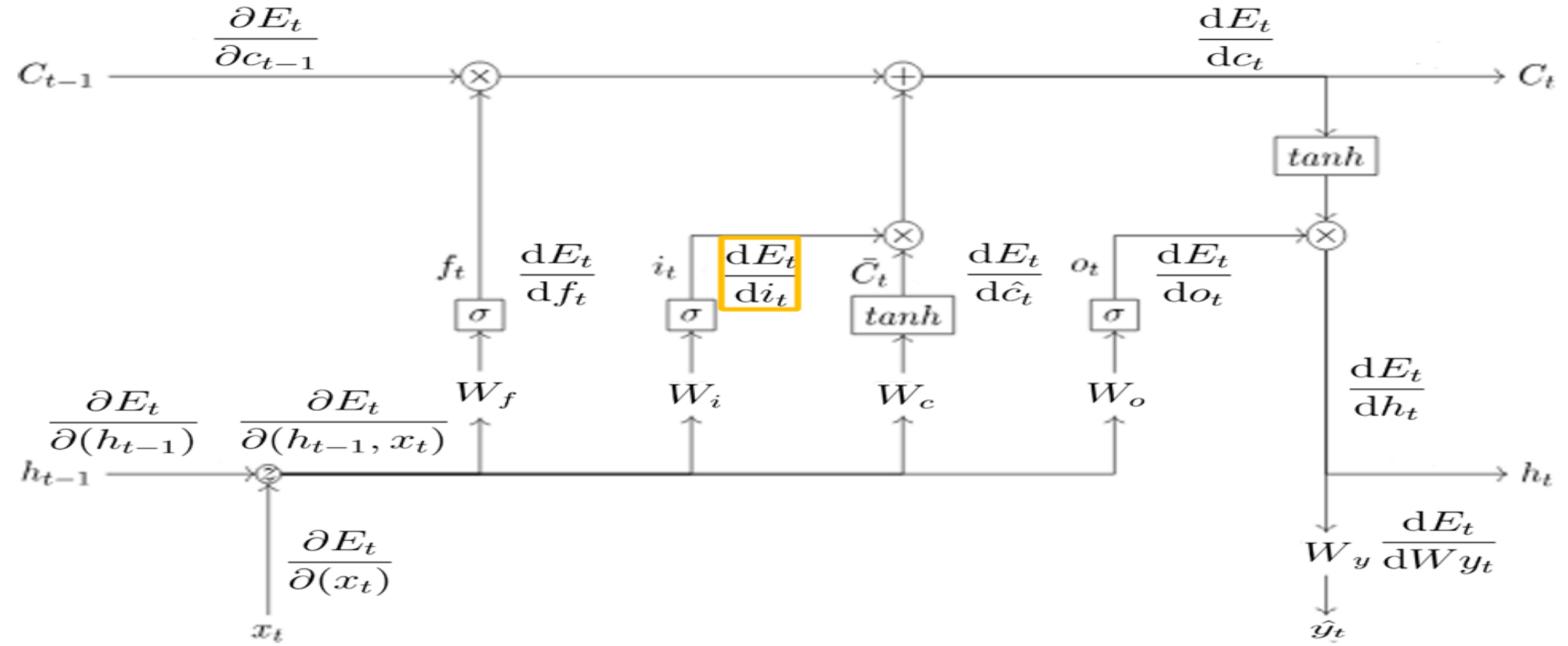
$$o_t = \sigma\left(W_o.[h_{t-1}, x_t] + b_o\right)$$

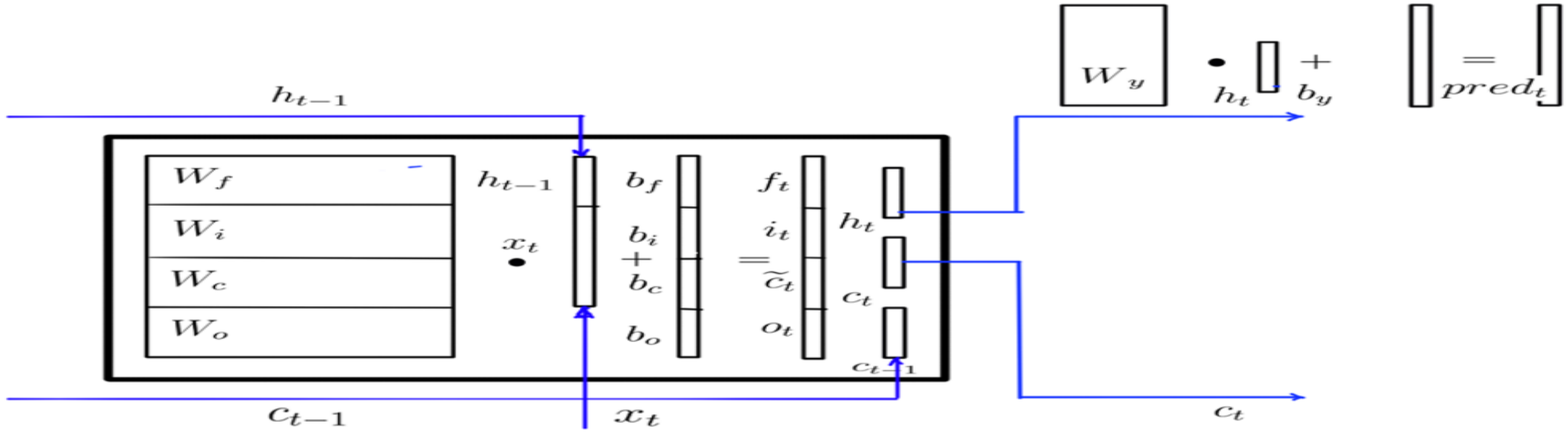
$$\widetilde{c}_t = \tanh\left(W_c.[h_{t-1}, x_t] + b_c\right)$$

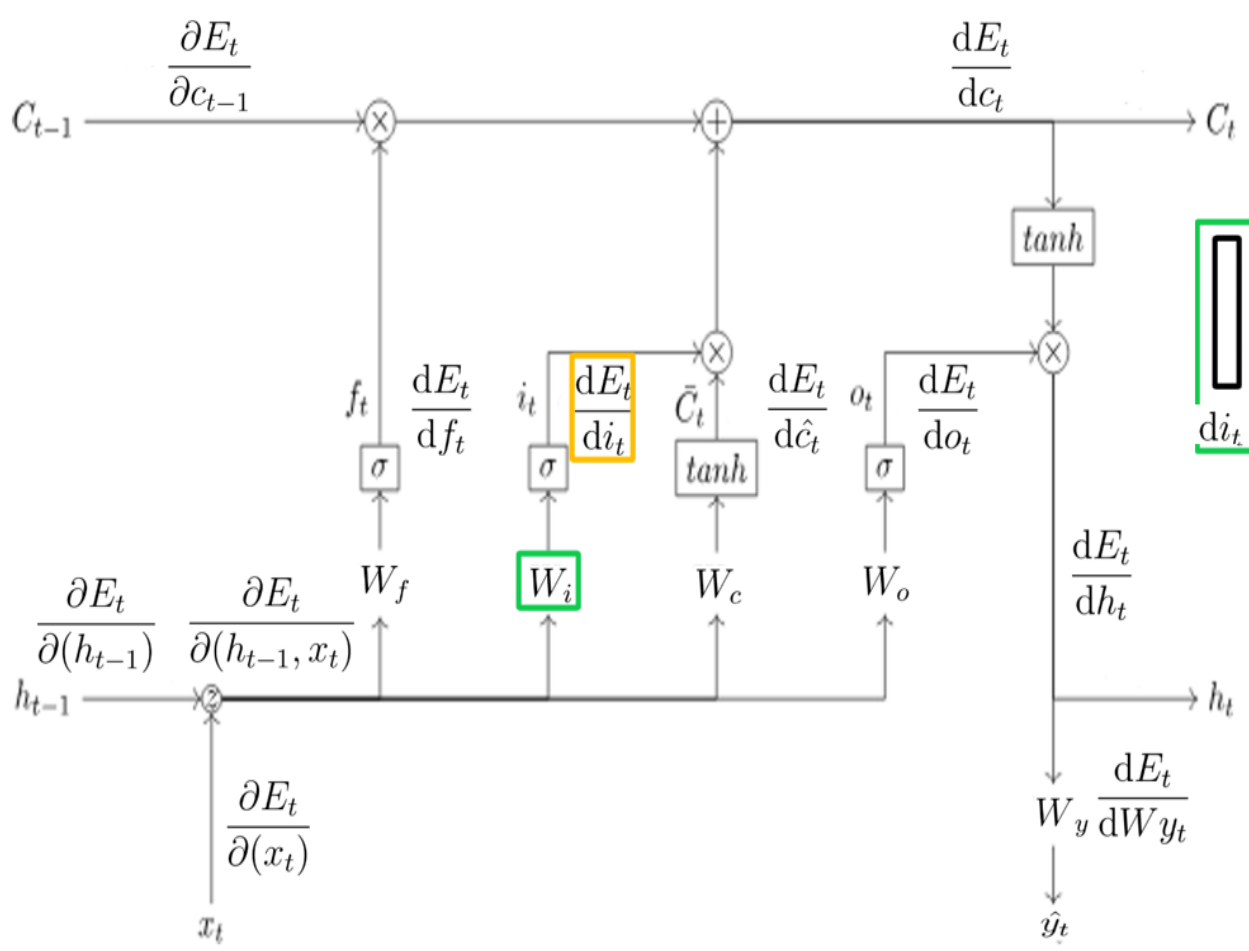
$$i_t = \sigma\left(W_i.[h_{t-1}, x_t] + b_i\right)$$

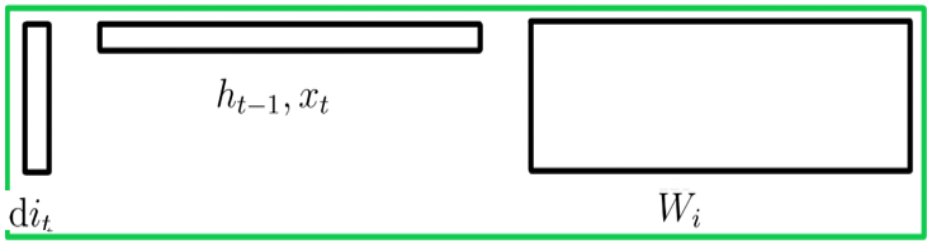
$$f_t = \sigma\left(W_f.[h_{t-1}, x_t] + b_f\right)$$

LSTM(Long Short Term Memory):Internals:Back Prop

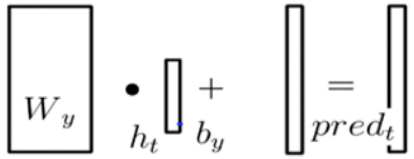


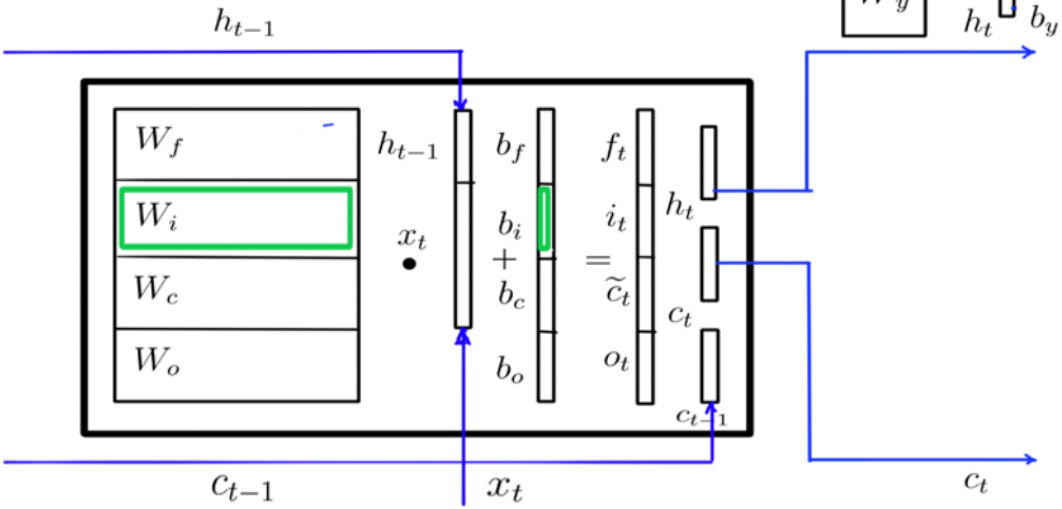







```
dwi+=np.dot(dit,z)
dbi+=dit
dxi=np.dot(dit.T,self.WLSTM[self.hidden_size*1:self.hidden_size*2,1:])
```





LSTM(Long Short Term Memory):Internals:Back Prop

```
dxf=np.dot(dft.T,self.WLSTM[0:self.hidden_size,1:])
dxi=np.dot(dit.T,self.WLSTM[self.hidden_size*1:self.hidden_size*2,1:])
dxc=np.dot(dcproj.T,self.WLSTM[self.hidden_size*2:self.hidden_size*3,1:])
dxo=np.dot(dot.T,self.WLSTM[self.hidden_size*3:self.hidden_size*4,1:])

dx=dxf+dxi+dxc+dxo
dh_next=dx[:,:self.hidden_size].T
```

- Previously calculated weights
- added and proportional part taken as dh_next.

$$E_t = crossEntropyLoss(\hat{y}_t, y_t)$$

$$\hat{y}_t = softmax(pred_t)$$

$$pred_t = (h_t.W_y) + b_y$$

$$h_t = o_t * tanh(c_t)$$

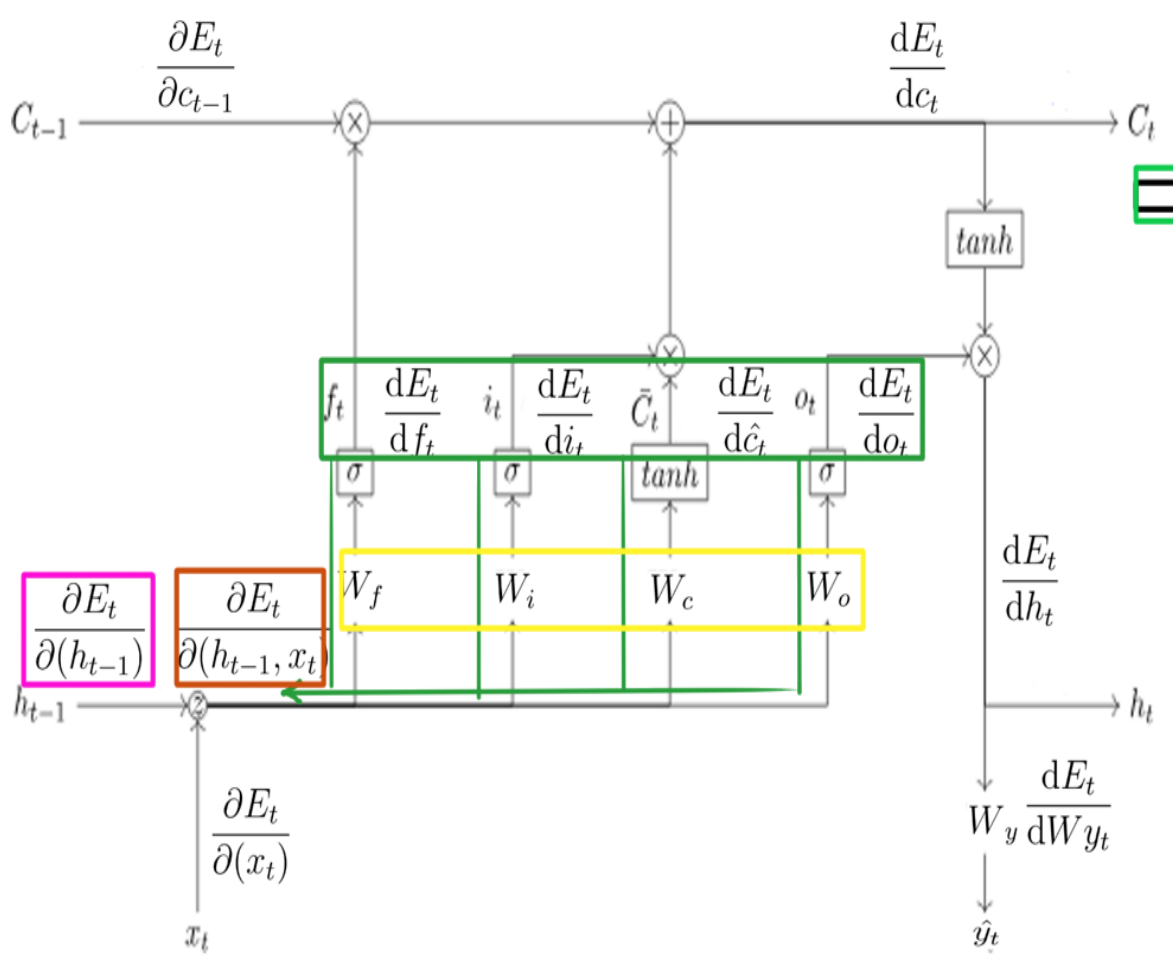
$$c_t = (f_t * c_{t-1}) + (i_t * \widetilde{c}_t)$$

$$o_t = \sigma\left(W_o.[h_{t-1}, x_t] + b_o\right)$$

$$\widetilde{c}_t = \tanh\left(W_c.[h_{t-1}, x_t] + b_c\right)$$

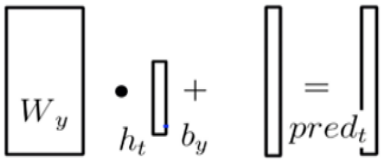
$$i_t = \sigma\left(W_i.[h_{t-1}, x_t] + b_i\right)$$

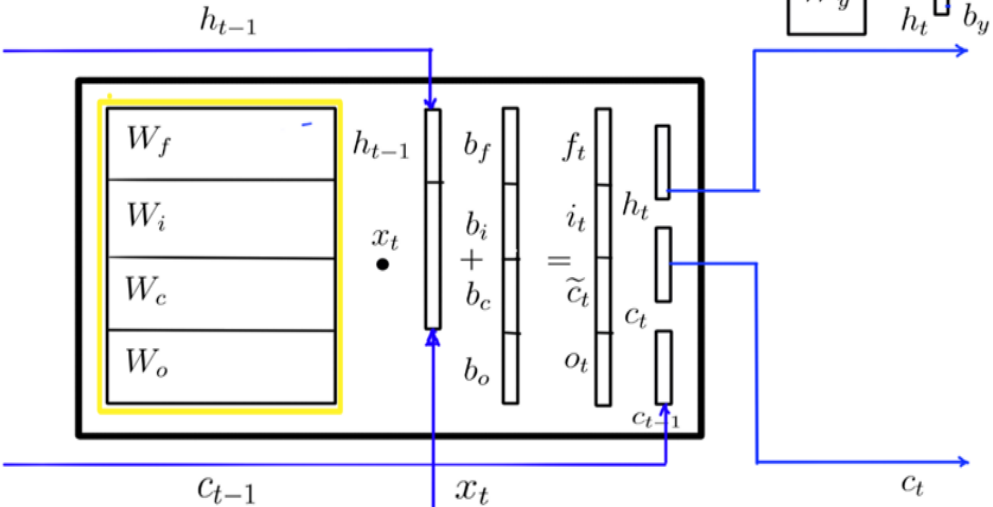
$$f_t = \sigma\left(W_f.[h_{t-1}, x_t] + b_f\right)$$



```
dxf=np.dot(dft.T,self.WLSTM[0:self.hidden_size,1:])
dxi=np.dot(dit.T,self.WLSTM[self.hidden_size*1:self.hidden_size*2,1:])
dxc=np.dot(dcproj.T,self.WLSTM[self.hidden_size*2:self.hidden_size*3,1:])
dxo=np.dot(dot.T,self.WLSTM[self.hidden_size*3:self.hidden_size*4,1:])

dx=dxf+dxi+dxc+dxo
dh_next=dx[:,:self.hidden_size].T
```





LSTM(Long Short Term Memory):Internals:Back Prop

```
dx=dxf+dxi+dxc+dxo
dh_next=dx[:,:self.hidden_size].T
dC_next=np.multiply(dct,ft[t])
db=np.concatenate((dit,dcproj,dft,dot),axis=0)
dht=np.zeros_like(dht)
```

- previously calculated dct multiplied by f_t.

$$E_t = crossEntropyLoss(\hat{y}_t, y_t)$$

$$\hat{y}_t = softmax(pred_t)$$

$$pred_t = (h_t.W_y) + b_y$$

$$h_t = o_t * tanh(c_t)$$

$$c_t = (f_t * c_{t-1}) + (i_t * \widetilde{c}_t)$$

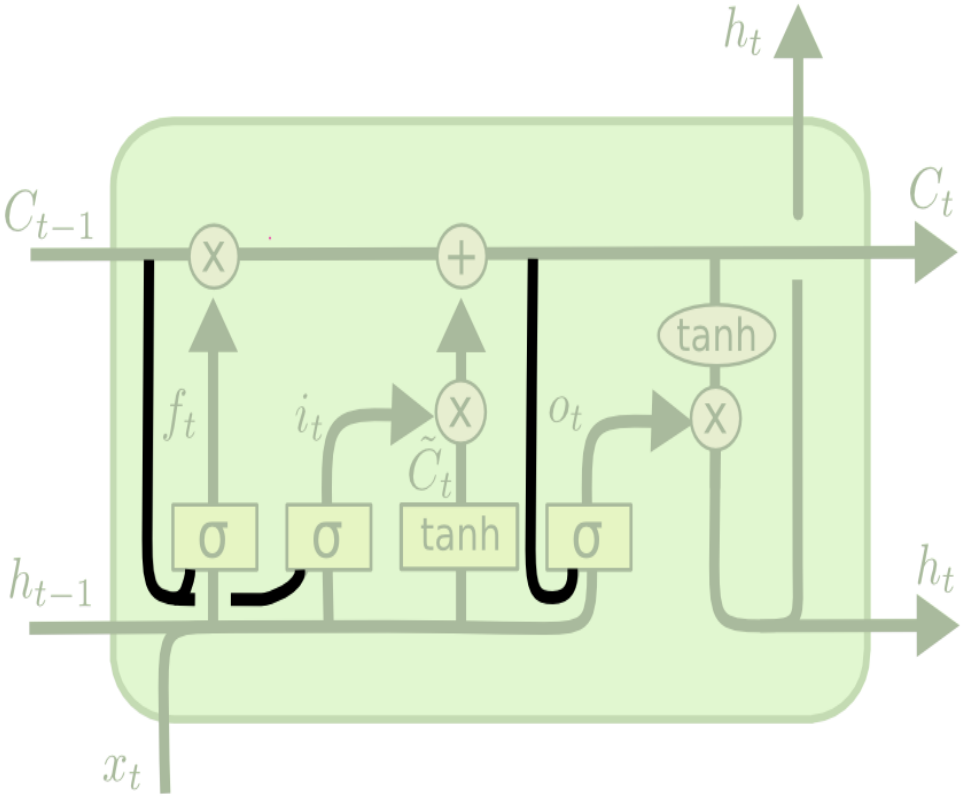
$$o_t = \sigma\left(W_o.[h_{t-1}, x_t] + b_o\right)$$

$$\widetilde{c}_t = \tanh\left(W_c.[h_{t-1}, x_t] + b_c\right)$$

$$i_t = \sigma\left(W_i.[h_{t-1}, x_t] + b_i\right)$$

$$f_t = \sigma\left(W_f.[h_{t-1}, x_t] + b_f\right)$$

LSTM(Long Short Term Memory):Variants:LSTM



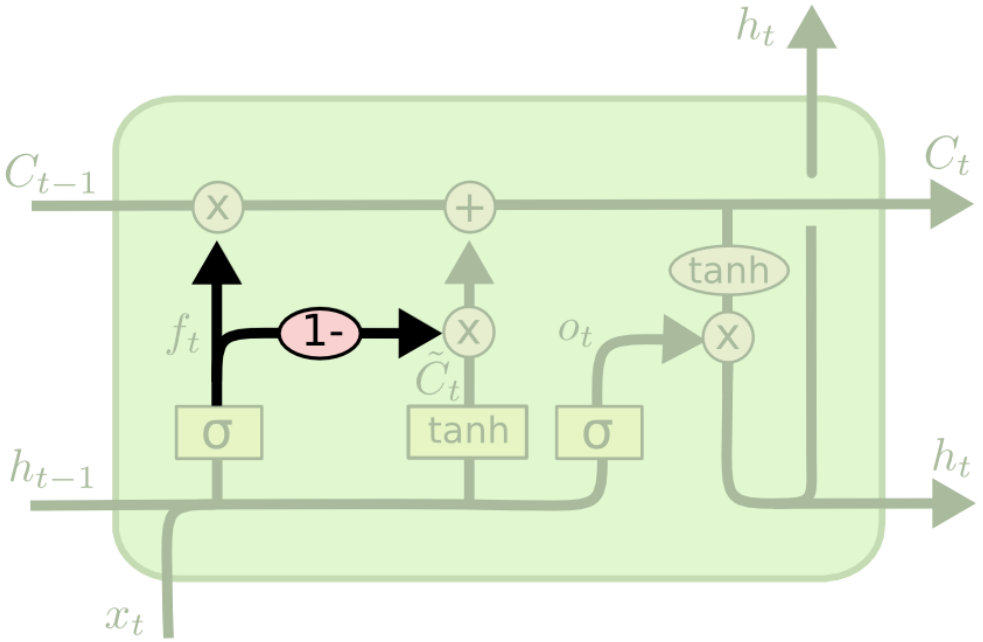
$$f_t = \sigma\left(W_f \cdot [C_{t-1}, h_{t-1}, x_t] + b_f\right)$$

$$i_t = \sigma\left(W_i \cdot [C_{t-1}, h_{t-1}, x_t] + b_i\right)$$

$$o_t = \sigma\left(W_o \cdot [C_t, h_{t-1}, x_t] + b_o\right)$$

- There are many variants.
  - Peep hole version allows gates to look at the cell state.

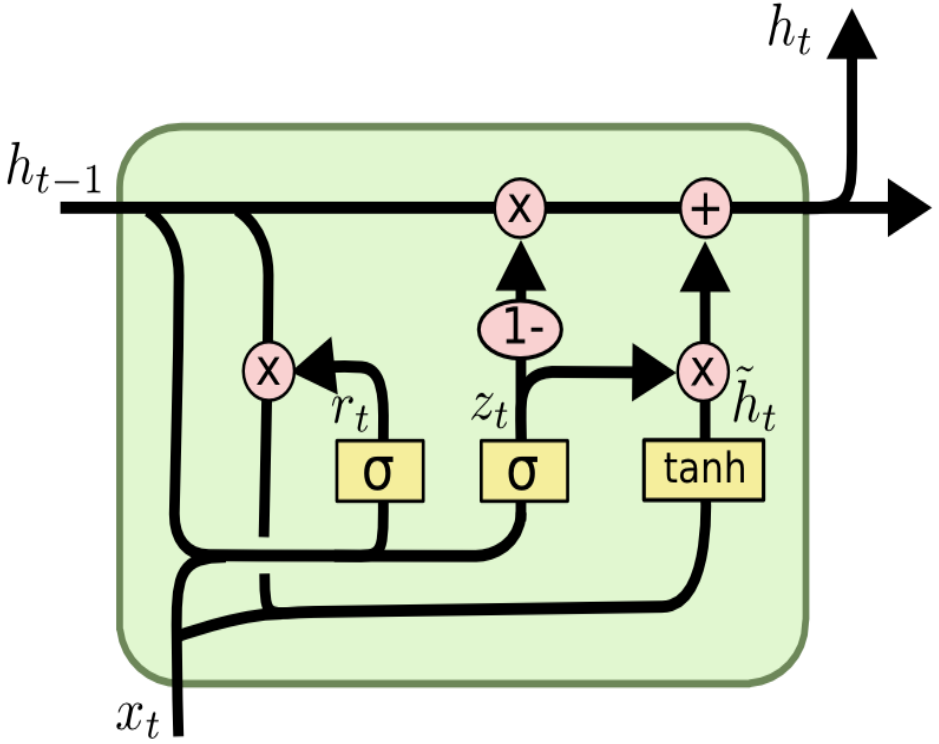
LSTM(Long Short Term Memory):Variants:LSTM



$$C_t = f_t * C_{t-1} + (1 - \boldsymbol{f_t}) * \tilde{C}_t$$

- Another variant couples the forget and input gates.

LSTM(Long Short Term Memory):Variants:GRU



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

- GRU - It combines forget and the input gate.
- It also combines the hidden state with the cell state.

RNN:Feed forward depth

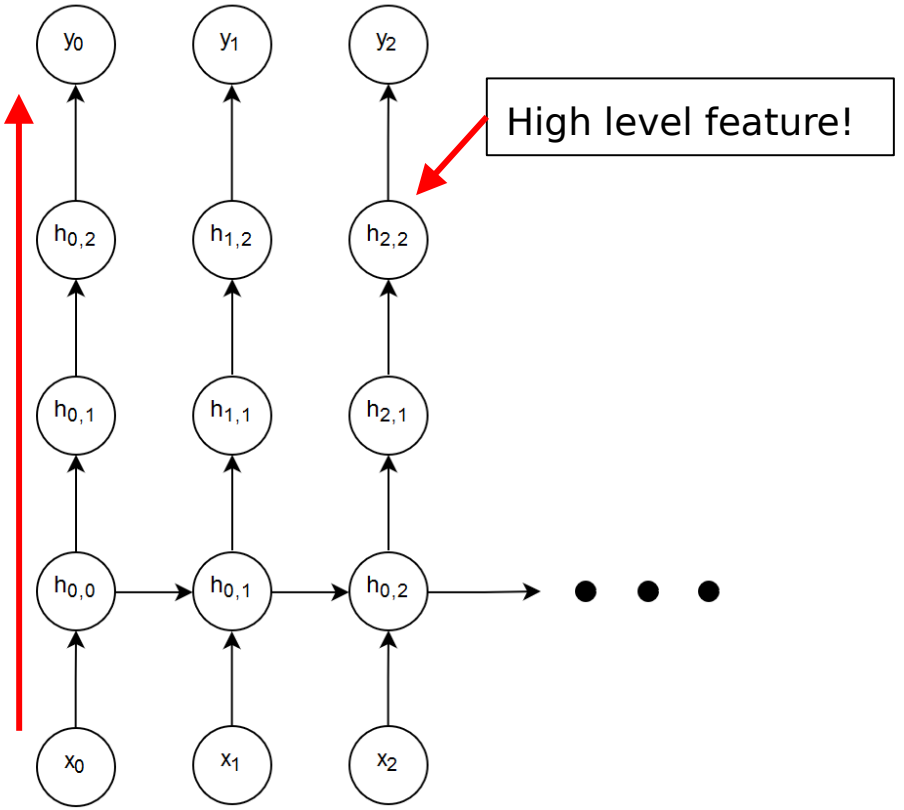
Notation: $h_{0,1} \Rightarrow$ time step 0, neuron #1

Feedforward depth: **longest** path

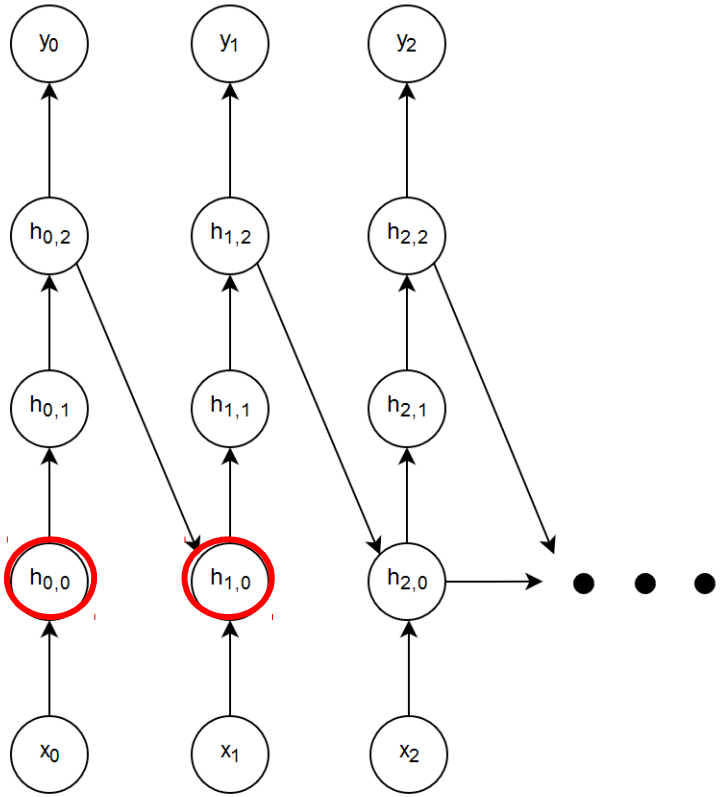
between an input and output at the

same timestep

Feedforward depth = 4



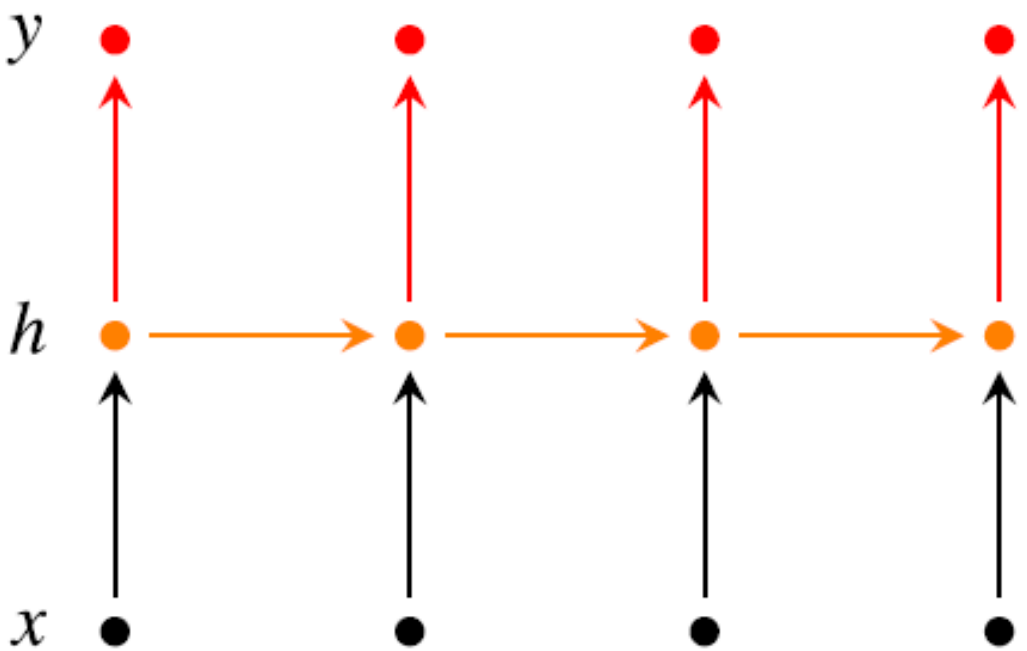
RNN:recurrent depth



Recurrent depth = 3

- **Recurrent depth**: Longest path between **same hidden state** in **successive timesteps**

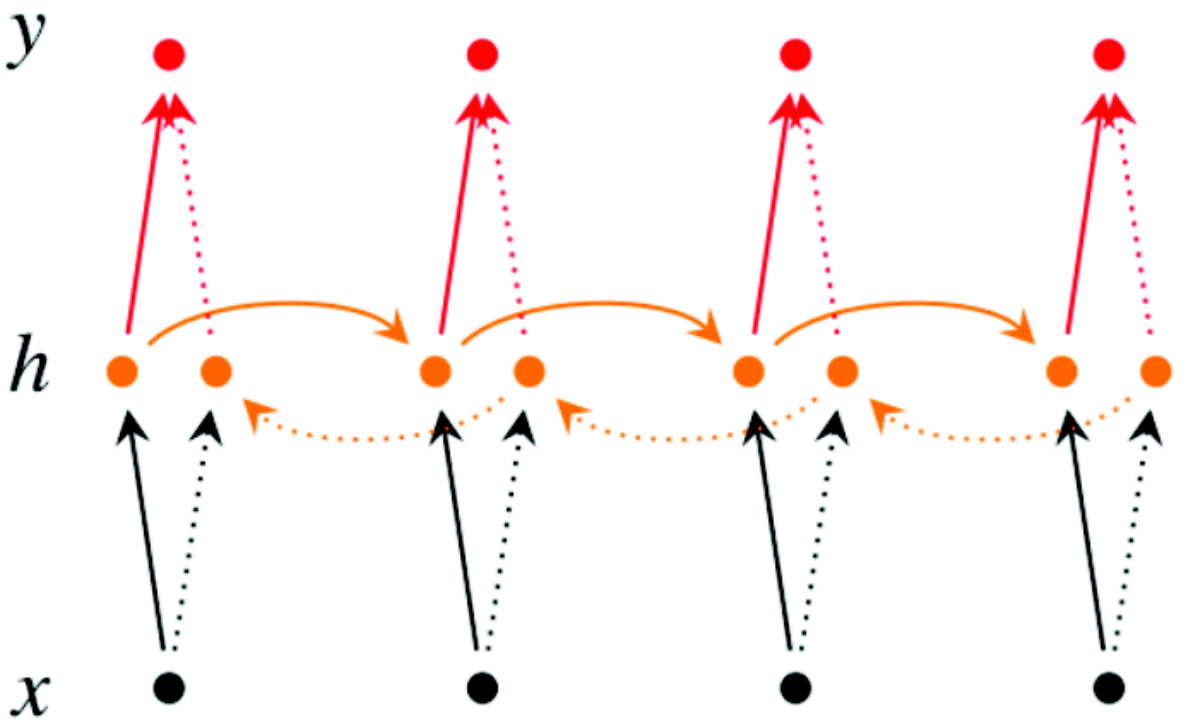
RNN:Uni



$$h_t = f(Wx_t + Vh_{t-1} + b)$$

$$y_t = g(Uh_t + c)$$

RNN:Bi

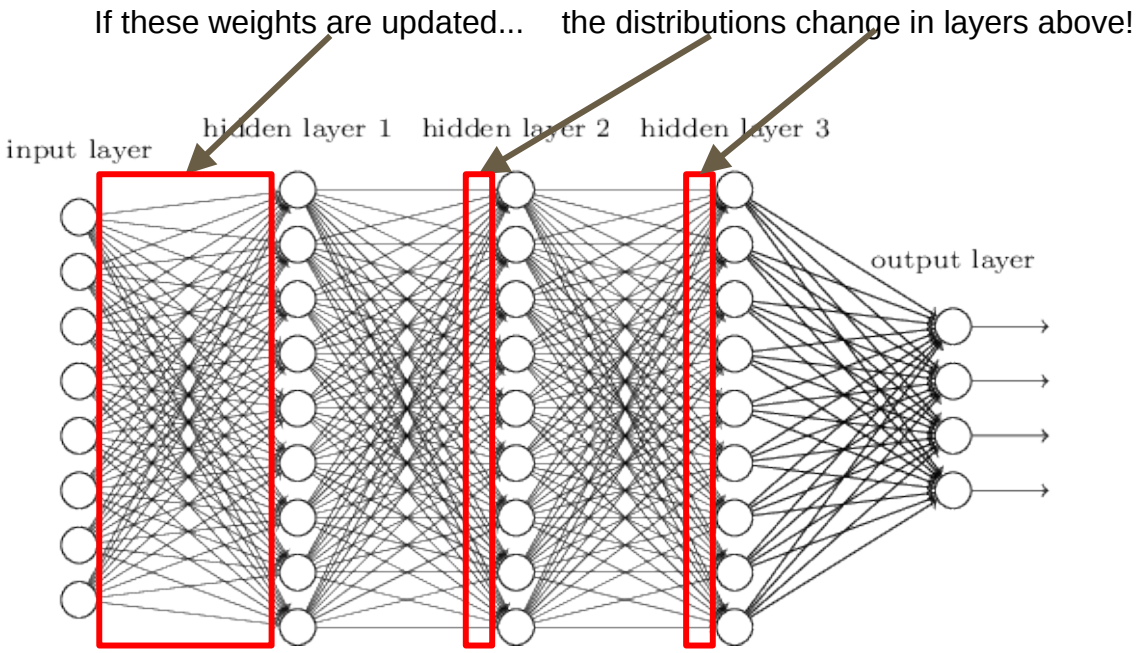


$$\overrightarrow{h}_t = f(\overrightarrow{W}x_t + \overrightarrow{V}\overrightarrow{h}_{t-1} + \overrightarrow{b})$$

$$\overleftarrow{h}_t = f(\overleftarrow{W}x_t + \overleftarrow{V}\overleftarrow{h}_{t+1} + \overleftarrow{b})$$

$$y_t = g(U[\overrightarrow{h}_t; \overleftarrow{h}_t] + c)$$

RNN:BN:Internal Covariate Shift



The model needs to learn parameters **while adapting to the changing input distribution**
⇒ slower model convergence!

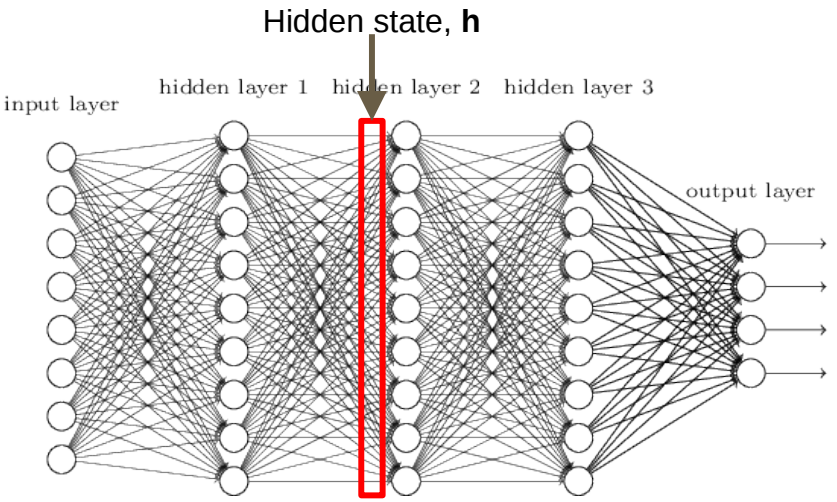
Source: https://i.stack.imgur.com/1bCQl.png

RNN:BN

Batch Normalization Equation:

$$\mathrm{BN}(\mathbf{h};\gamma,\beta) = \beta + \gamma \odot \frac{\mathbf{h} - \widehat{\mathbb{E}}[\mathbf{h}]}{\sqrt{\widehat{\mathrm{Var}}[\mathbf{h}] + \epsilon}}$$

Bias, Std Dev: **To be learned**



Cooijmans, Tim, et al. "Recurrent batch normalization."(2016).

RNN:BN

- RNNs **deepest along temporal dimension**
- Must be careful: repeated scaling could cause **exploding** gradients

RNN:BN

$$\begin{pmatrix} \tilde{\mathbf{f}}_t \\ \tilde{\mathbf{i}}_t \\ \tilde{\mathbf{o}}_t \\ \tilde{\mathbf{g}}_t \end{pmatrix} = \mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t + \mathbf{b}$$

$$\begin{aligned} \mathbf{c}_t &= \sigma(\tilde{\mathbf{f}}_t) \odot \mathbf{c}_{t-1} + \sigma(\tilde{\mathbf{i}}_t) \odot \tanh(\tilde{\mathbf{g}}_t) \\ \mathbf{h}_t &= \sigma(\tilde{\mathbf{o}}_t) \odot \tanh(\mathbf{c}_t), \end{aligned}$$

Original LSTM Equations

$$\begin{pmatrix} \tilde{\mathbf{f}}_t \\ \tilde{\mathbf{i}}_t \\ \tilde{\mathbf{o}}_t \\ \tilde{\mathbf{g}}_t \end{pmatrix} = \mathrm{BN}(\mathbf{W}_h \mathbf{h}_{t-1}; \gamma_h, \beta_h) + \mathrm{BN}(\mathbf{W}_x \mathbf{x}_t; \gamma_x, \beta_x) + \mathbf{b}$$

$$\begin{aligned} \mathbf{c}_t &= \sigma(\tilde{\mathbf{f}}_t) \odot \mathbf{c}_{t-1} + \sigma(\tilde{\mathbf{i}}_t) \odot \tanh(\tilde{\mathbf{g}}_t) \\ \mathbf{h}_t &= \sigma(\tilde{\mathbf{o}}_t) \odot \tanh(\mathrm{BN}(\mathbf{c}_t; \gamma_c, \beta_c)) \end{aligned}$$

Batch Normalized LSTM

Cooijmans, Tim, et al. "Recurrent batch normalization."(2016).

RNN:BN

- x, h_{t-1} normalized **separately**
- c_t **not normalized**

  (doing so may disrupt gradient flow) How?
- New state (h_t) normalized

$$\begin{pmatrix} \tilde{\mathbf{f}}_t \\ \tilde{\mathbf{i}}_t \\ \tilde{\mathbf{o}}_t \\ \tilde{\mathbf{g}}_t \end{pmatrix} = \mathrm{BN}(\mathbf{W}_h \mathbf{h}_{t-1}; \gamma_h, \beta_h) + \mathrm{BN}(\mathbf{W}_x \mathbf{x}_t; \gamma_x, \beta_x) + \mathbf{b}$$

$$\mathbf{c}_t = \sigma(\tilde{\mathbf{f}}_t) \odot \mathbf{c}_{t-1} + \sigma(\tilde{\mathbf{i}}_t) \odot \tanh(\tilde{\mathbf{g}}_t)$$

$$\mathbf{h}_t = \sigma(\tilde{\mathbf{o}}_t) \odot \tanh(\mathrm{BN}(\mathbf{c}_t; \gamma_c, \beta_c))$$

Cooijmans, Tim, et al. "Recurrent batch normalization."(2016).

RNN:BN

$$\mathrm{BN}(\mathbf{W}_h \mathbf{h}_{t-1}; \gamma_h, \beta_h)$$

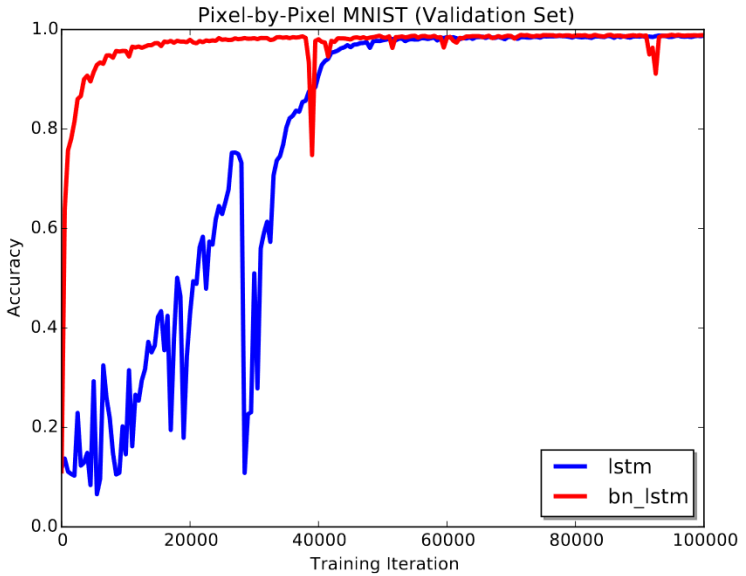
- Initialize **β to 0**, **γ** to a small value such as **~0.1**. Else vanishing gradients (think of the tanh plot!)
  - Learn statistics for each time step **independently** till some time step **T.** Beyond **T**, use statistics for **T**

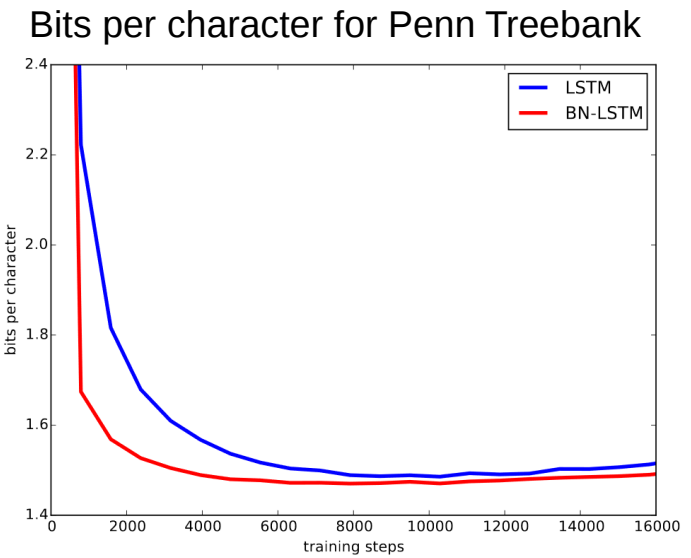
Cooijmans, Tim, et al. "Recurrent batch normalization."(2016).

RNN:BN

A: **Faster convergence** due to Batch Norm

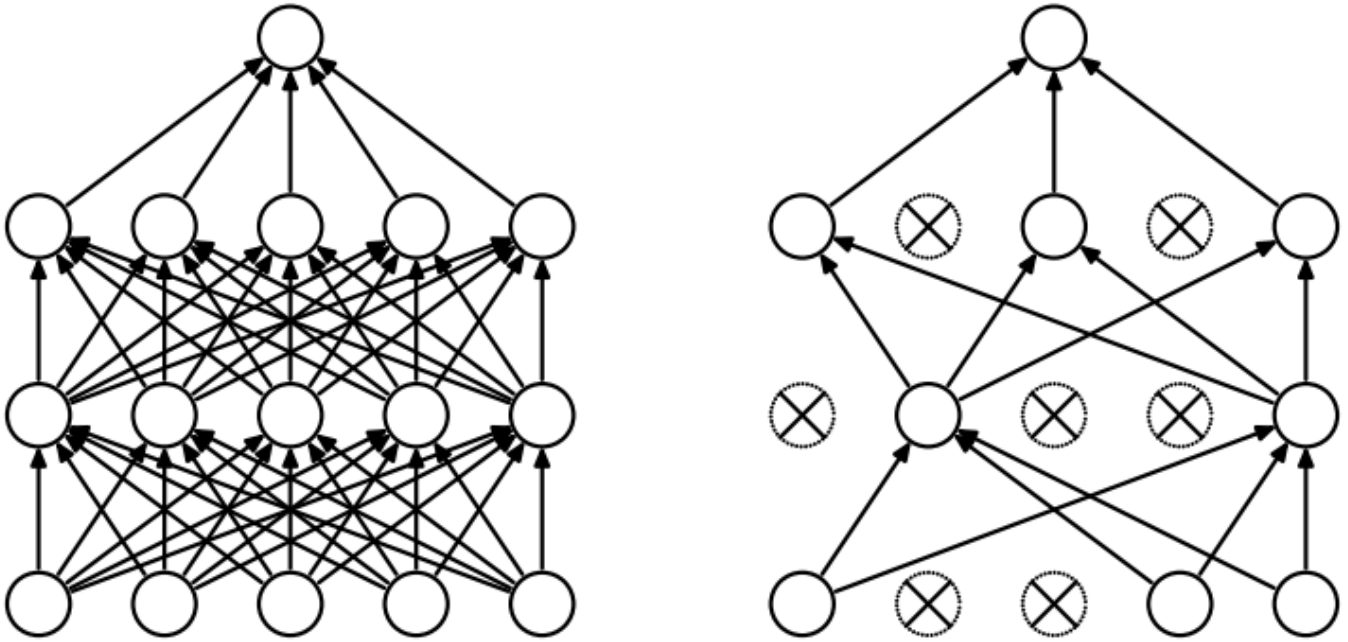
B: Performance **as good** as (if not better than) unnormalized LSTM





Cooijmans, Tim, et al. "Recurrent batch normalization."(2016).

RNN:Dropouts



(a) Standard Neural Net

(b) After applying dropout.

Srivastava et al. 2014. “Dropout: a simple way to prevent neural networks from overfitting”

RNN:Dropouts

To prevent over confident models

High Level Intuition: Ensemble of thinned networks sampled through dropout

Interested in a theoretical proof ?

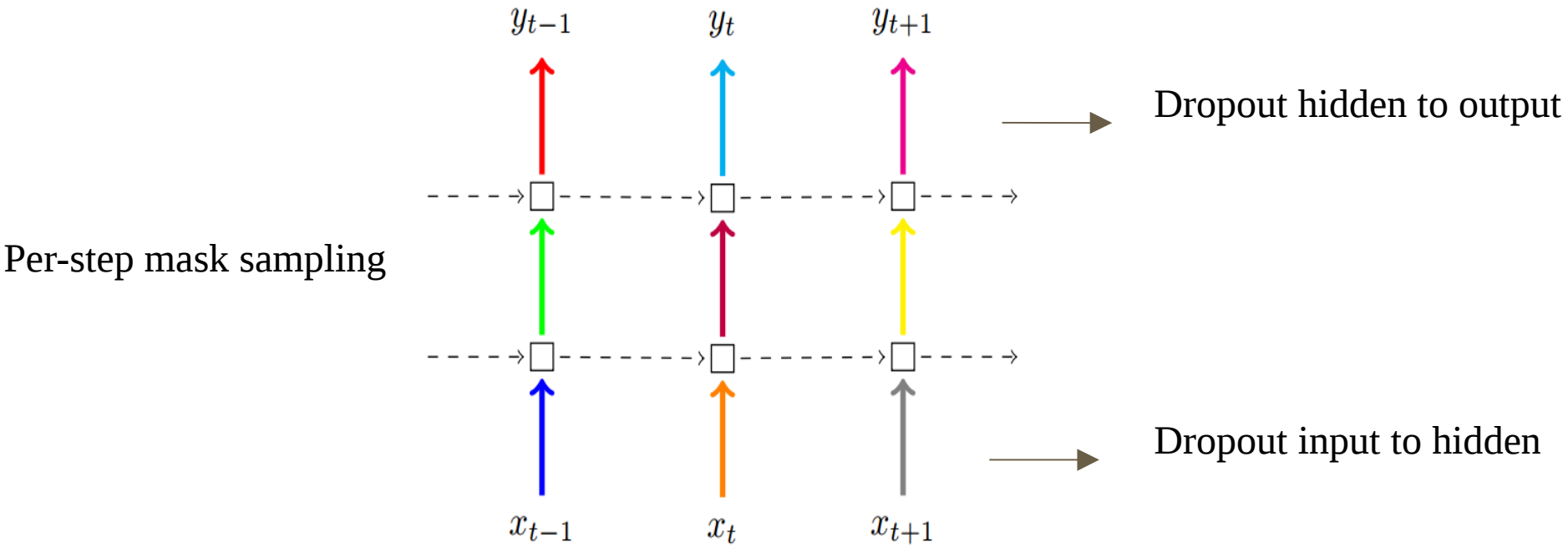
A Probabilistic Theory of Deep Learning, Ankit B. Patel, Tan Nguyen, Richard G. Baraniuk

Skip Proof Slides

RNN:Dropouts

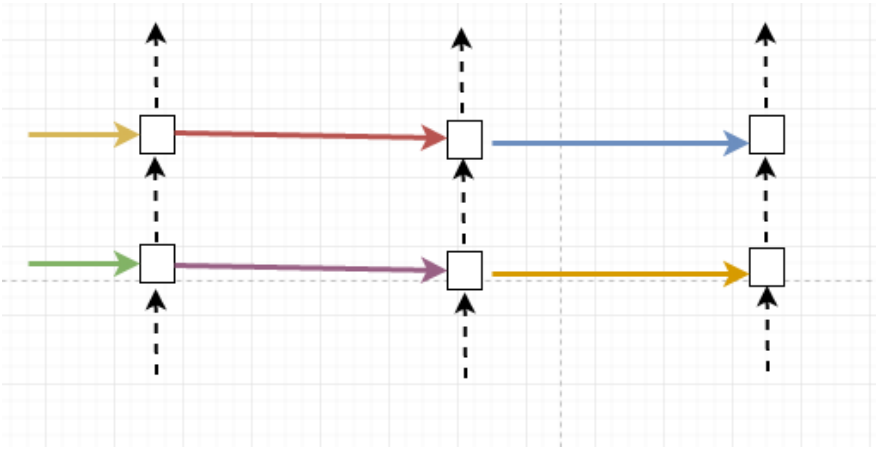
Beneficial to use it once in correct spot rather than put it everywhere

Each color represents a different mask



Zaremba et al. 2014. “Recurrent neural network regularization”

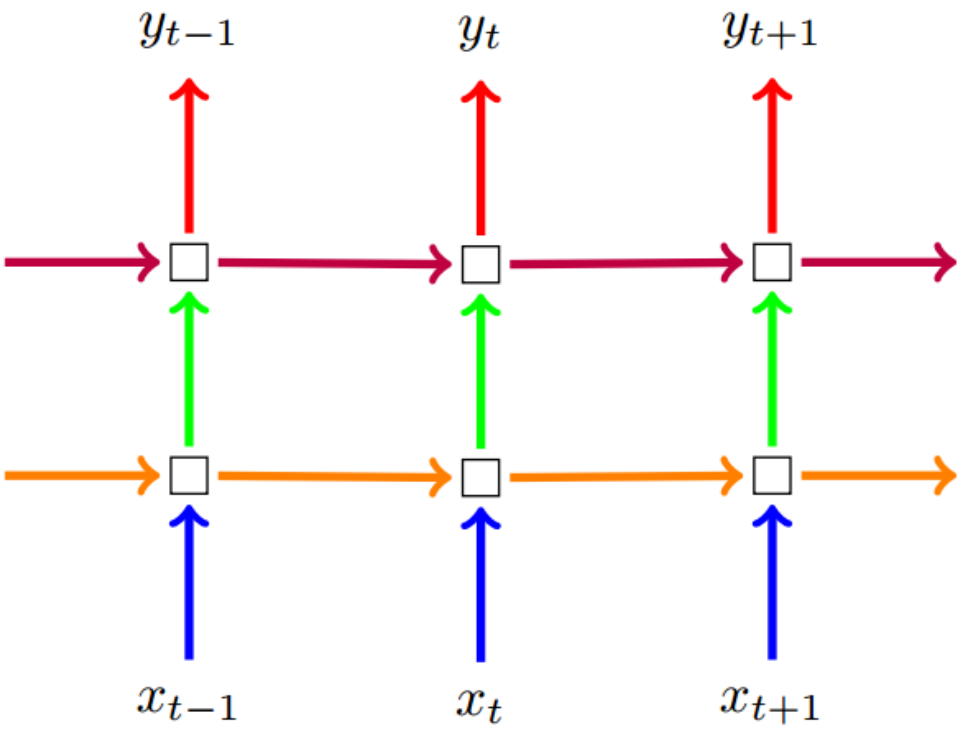
RNN:Dropouts



MEMORY LOSS !

Only tends to retain short term dependencies

RNN:Dropouts



- Per-sequence mask sampling
- Drop the time dependency of an entire feature

Gal 2015. “A theoretically grounded application of dropout in recurrent neural networks”

RNN:Dropouts

Dropout on cell state (c_t)

Inefficient

Dropout on cell state update ($\tanh(g)_t$) or (h_{t-1})

Optimal

$$\begin{pmatrix} \tilde{\mathbf{f}}_t \\ \tilde{\mathbf{i}}_t \\ \tilde{\mathbf{o}}_t \\ \tilde{\mathbf{g}}_t \end{pmatrix} = \mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t + \mathbf{b}$$

$$\begin{aligned} \mathbf{c}_t &= \sigma(\tilde{\mathbf{f}}_t) \odot \mathbf{c}_{t-1} + \sigma(\tilde{\mathbf{i}}_t) \odot \tanh(\tilde{\mathbf{g}}_t) \\ \mathbf{h}_t &= \sigma(\tilde{\mathbf{o}}_t) \odot \tanh(\mathbf{c}_t), \end{aligned}$$

Barth (2016) : “Semenuita et al. 2016. “Recurrent dropout without memory loss”

RNN:Dropouts

Model	Perplexity Scores
Original	125.2
Forward Dropout + Drop ($\tanh(g_t)$)	87 (-37)
Forward Dropout + Drop (h_{t-1})	88.4 (-36)
Forward Dropout	89.5 (-35)
Forward Dropout + Drop (c_t)	99.9 (-25)

Lower perplexity score is better !

Barth (2016) : “Semenuita et al. 2016. “Recurrent dropout without memory loss”

RNN:Implementation

```
cell_1 = tf.nn.rnn_cell.BasicRNNCell(num_units, input_size=None,
                                     activation=tanh)
```

- Basic RNNCell represents vanilla recurrent neuron layer.

```
cell_2 = tf.nn.rnn_cell.BasicLSTMCell(num_units,
                                      forget_bias=1.0,
                                      input_size=None,
                                      state_is_tuple=True,
                                      activation=tanh)
```

- Simple implementation of LSTM Unit

```
cell_3 = tf.nn.rnn_cell.LSTMCell(num_units, input_size=None,
                                 use_peepholes=False,
                                 cell_clip=None,
                                 initializer=None,
                                 num_proj=None,
                                 proj_clip=None,
                                 num_unit_shards=1,
                                 num_proj_shards=1,
                                 forget_bias=1.0,
                                 state_is_tuple=True,
                                 activation=tanh)
```

- More configuration options like peep hole structure, clipping of state values.

```
cell_4 = tf.nn.rnn_cell.GRUCell(num_units, input_size=None,
                                activation=tanh)
```

- GRUs.

RNN:Implementation

```
cell_1 = tf.nn.rnn_cell.BasicLSTMCell(10)
cell_2 = tf.nn.rnn_cell.BasicLSTMCell(10)
full_cell = tf.nn.rnn_cell.MultiRNNCell([cell_1, cell_2])
```

• Feed forward depth

```
cell_1 = tf.nn.rnn_cell.BasicLSTMCell(10)
tf.nn.rnn_cell.DropoutWrapper(cell_1, input_keep_prob=1.0,
                              output_keep_prob=1.0,
                              seed=None)
```

```
outputs, state = tf.nn.dynamic_rnn(cell, inputs,
                                   sequence_length=None,
                                   initial_state=None,
                                   dtype=None,
                                   parallel_iterations=None,
                                   swap_memory=False,
                                   time_major=False,
                                   scope=None)
```

• Simple implementation of dynamic RNN.

```
def dynamic_rnn(cell, X,state=None):
    batch,seq,input_size=X.shape

    if state is None:
        state=zero_state_initializer(cell.hidden_size,batch)
    result={}
    prevstate=state
    for seqnum in range(seq):
        newstate=cell(np.reshape(X[0][seqnum],[input_size,batch]),prevstate)
        result[seqnum]=newstate[1];
        prevstate=newstate

    return result,prevstate
```

RNN:Implementation

```
lstm_cell_fw = tf.contrib.rnn.BasicLSTMCell(lstm_units)
lstm_cell_bw = tf.contrib.rnn.BasicLSTMCell(lstm_units)


(output_fw, output_bw), state =
tf.nn.bidirectional_dynamic_rnn(lstm_cell_fw,lstm_cell_bw, data,
  dtype=tf.float32)
context_rep = tf.concat([output_fw, output_bw], axis=-1)
context_rep_flat = tf.reshape(context_rep, [-1, 2*lstm_units])
```