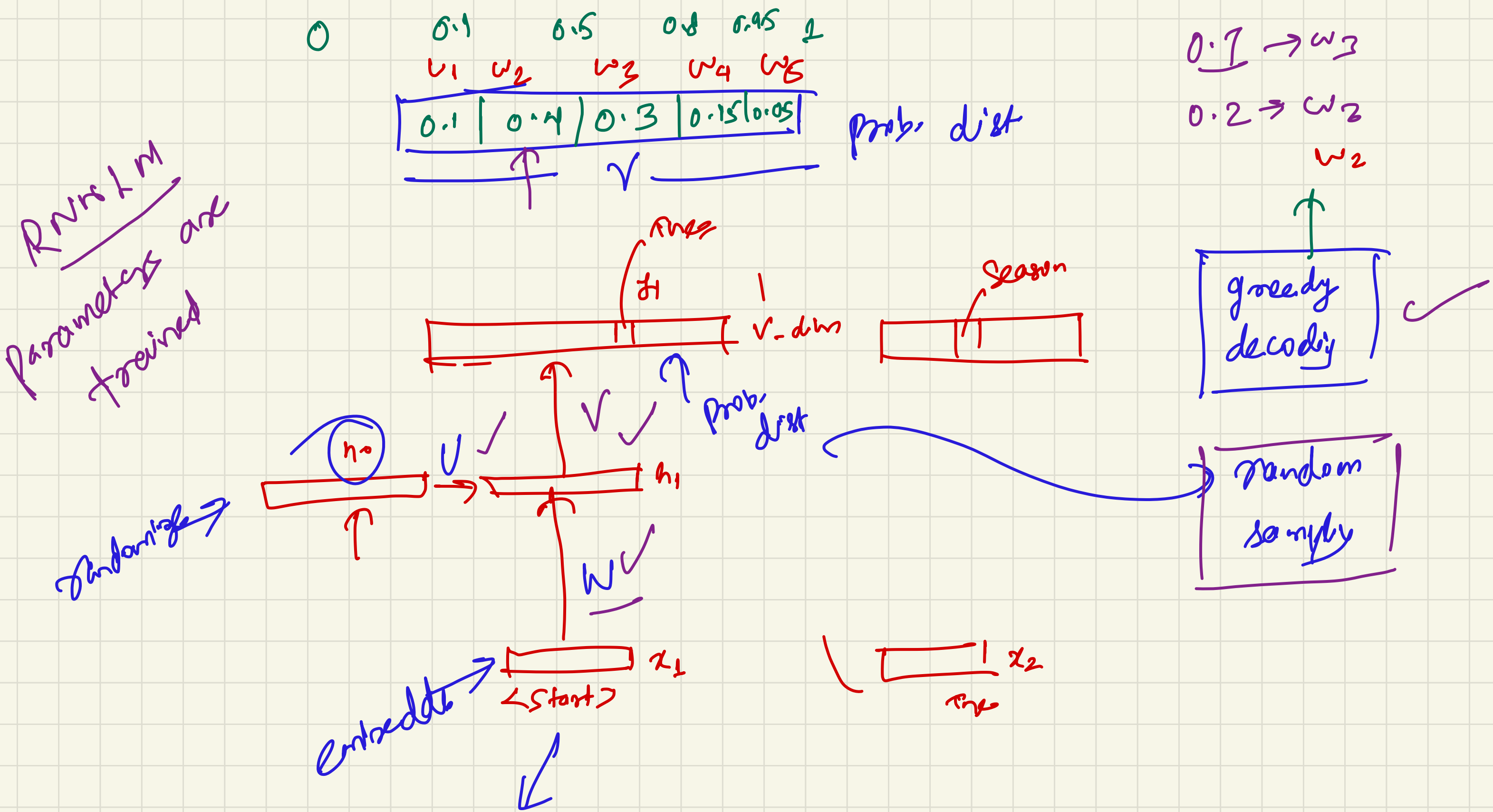

0 0.1 0.5 0.8 0.95 1
w1 w2 w3 w4 w5
0.1 | 0.4 | 0.3 | 0.15 | 0.05
prob. dist
V
0.1 → w3
0.2 → w2
w2
RNN LM
parameters are trained
season
V_dim
greedy decoding
random sampling
prob. dist
h0
U
V
h1
W
x1
<start>
x2
embedding

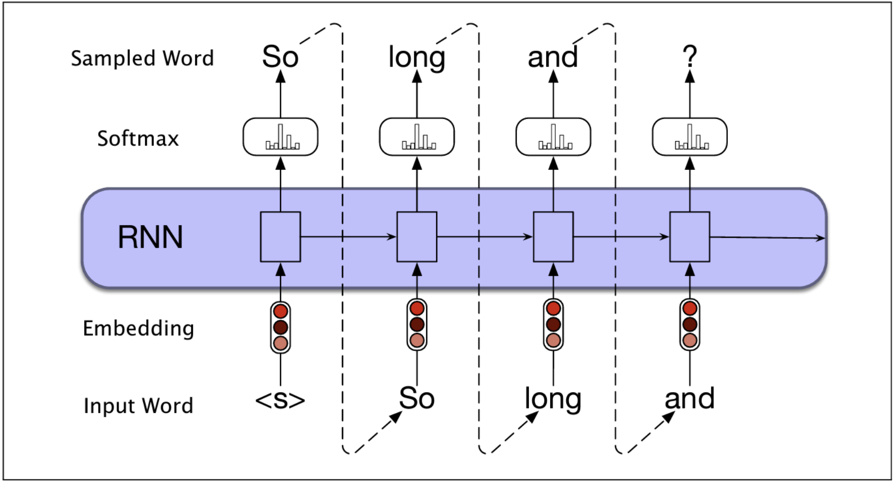
Generating text with an RNN Language Model

- RNN-based language models can be used for language generation (and hence, for machine translation, dialog, etc.)
- A language model can incrementally generate words by repeatedly sampling the words conditioned on the previous choices – also known as *autoregressive generation*.

Autoregressive Generation with RNNs

- All your parameters have already been trained.
- Start with a special begin of sentence token <s> as input
- Through forward propagation, obtain the probability distribution at the output, and sample a word
- Feed the word as input at the next time-step (its word vector)
- Continue generating until the end of sentence token is sampled, or a fixed length of the sentence has been reached.

Autoregressive Generation with RNNs



RNNs can be used for various other applications

o/p: o_1 - - o_n

I/p: sentence w_1 - - w_n

- Sequence labeling: Named Entity Recognition, Parts-of-Speech Tagging
- Text Classification: Sentiment Analysis, Spam Detection

I/p: sentence

o/p: label for the sentence

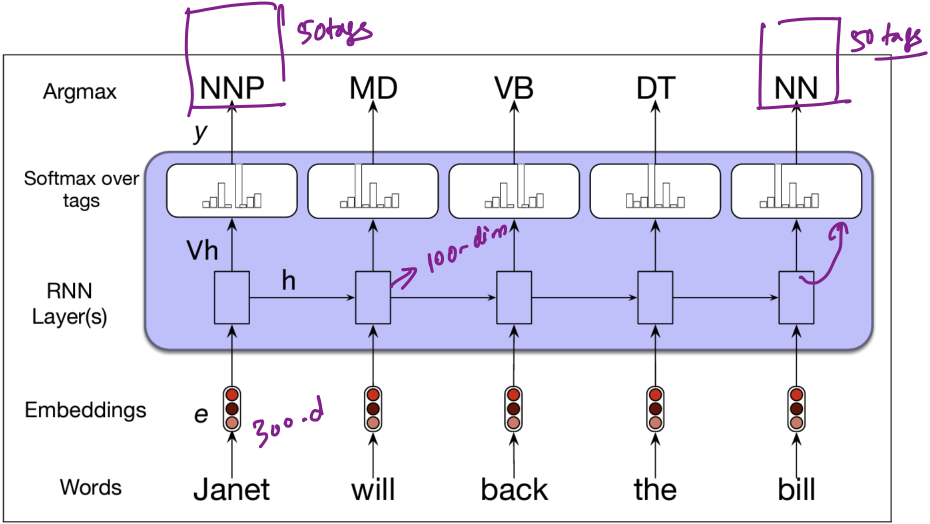
RNNs for Sequence Labeling

Task

Assign a label chosen from a small fixed set of labels to each element of the sequence

- Inputs: Word embeddings
- Outputs: Tag probabilities generated by the softmax layer

RNNs for Sequence Labeling



100×(300+100+50)

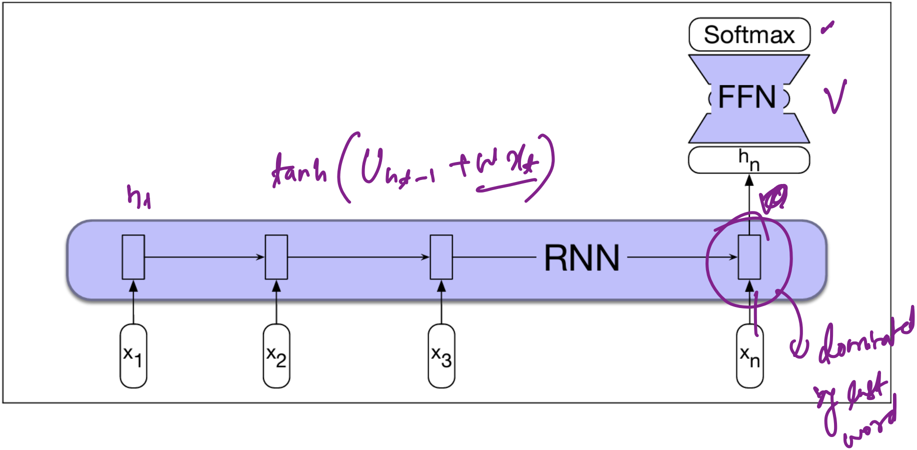
RNNs for Sequence Classification

Task

Classify the entire sequence rather than the token within them

- Pass the text to be classified a word at a time, generating new hidden states at each time step
- The hidden state of the last token can be thought of as a compressed representation of the entire sequence
- This last hidden state is passed through a feed-forward network that chooses a class via softmax
- There are other options of combining information from all the hidden states

RNNs for Sequence Classification



Other Variations: Stacked RNNs

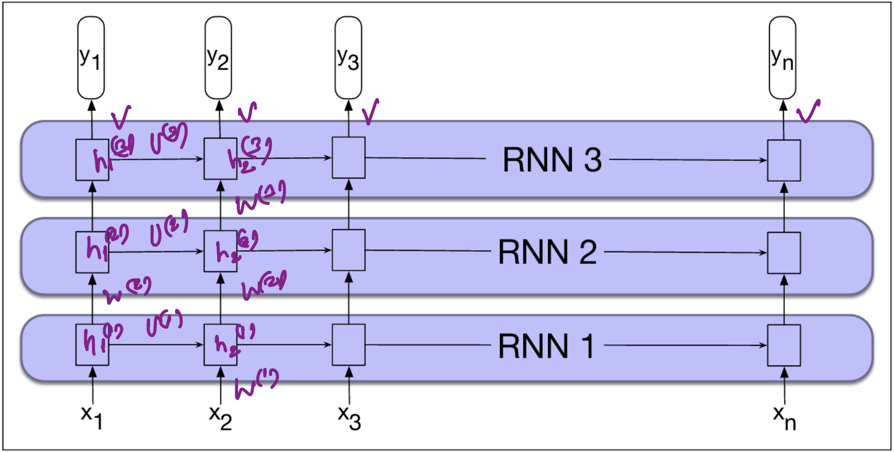
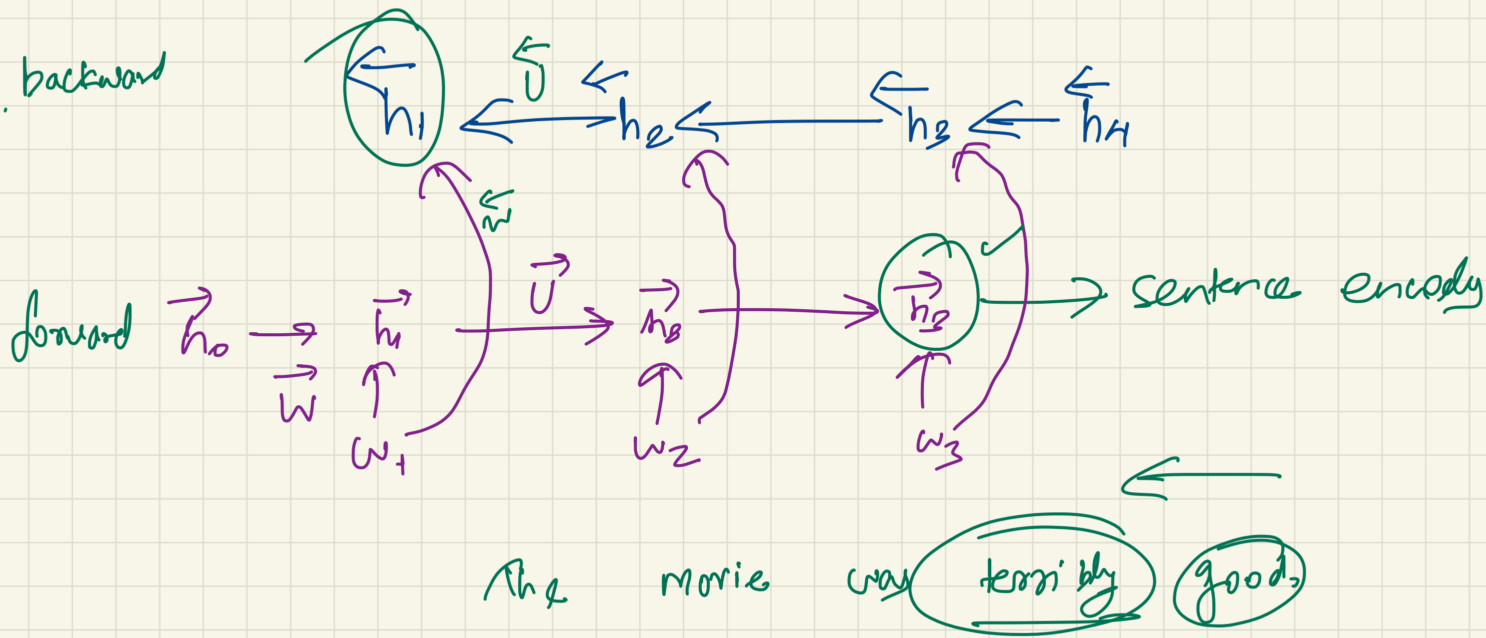


Figure 9.10 Stacked recurrent networks. The output of a lower level serves as the input to higher levels with the output of the last network serving as the final output.

bi-RNN for Text Classification

* Can't be used for LM!

$\boxed{\vec{h}_3 \,;\, \overleftarrow{h}_1} \rightarrow$ FFN



Other Variations: Bidirectional RNNs

- RNN makes use of information from left (prior) context to predict at time t
- In many applications, the entire sequence is available; so it makes sense to also make use of the right context to predict at time t
- Bidirectional RNNs combine two independent RNNs, one where the input is processed from left to right (forward RNN), and another from end to the start (backward RNN).

$$h_t^f = RNN_{forward}(x_1, \ldots, x_t)$$

$$h_t^b = RNN_{backward}(x_n, \ldots, x_t)$$

$$h_t = [h_t^f ; h_t^b]$$

Other Variations: Bidirectional RNNs

seq. labeling

slp3

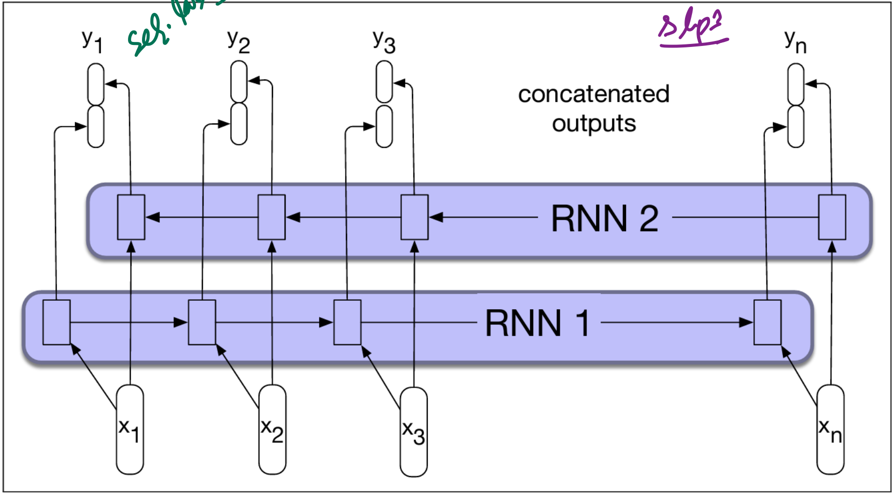


Figure 9.11 A bidirectional RNN. Separate models are trained in the forward and backward directions, with the output of each model at each time point concatenated to represent the bidirectional state at that time point.

Using Bidirectional RNNs for Sequence Classification

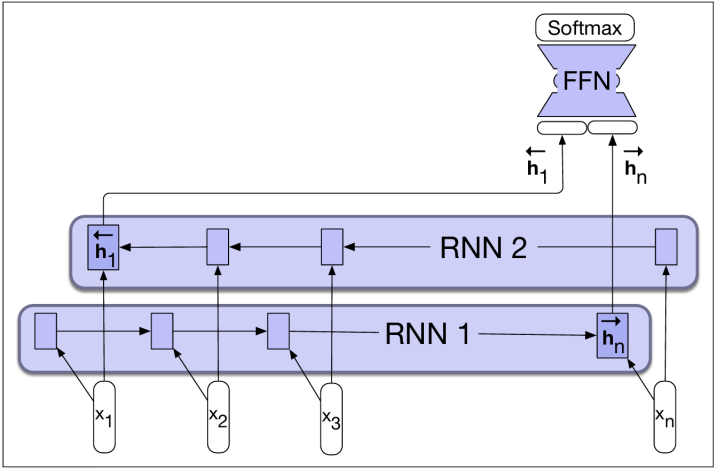


Figure 9.12 A bidirectional RNN for sequence classification. The final hidden units from the forward and backward passes are combined to represent the entire sequence. This combined representation serves as input to the subsequent classifier.

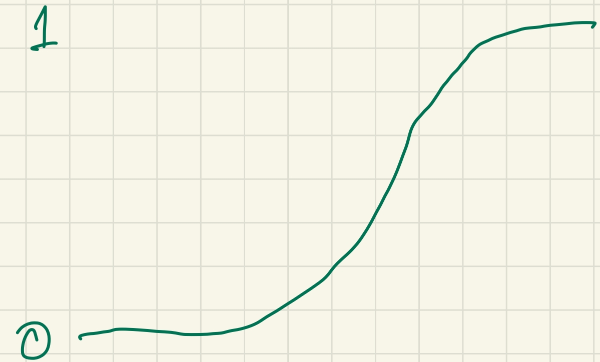
$$\begin{bmatrix} 1 & 0 & 0.5 \end{bmatrix}$$

✓ ✗

$$\begin{bmatrix} 1.2 & -0.3 & 0.5 \end{bmatrix}$$

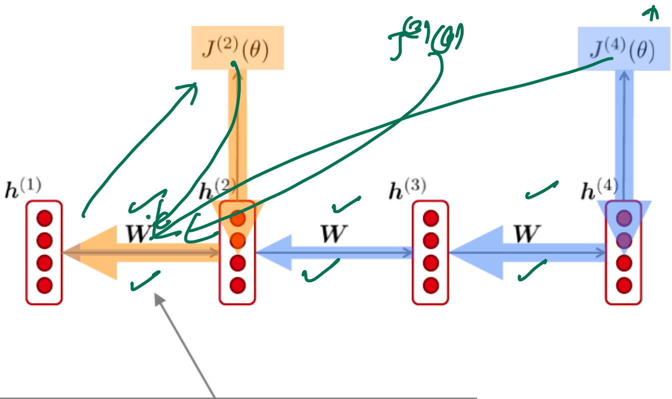
$$\tanh\left(U\vec{h}_1 + W\vec{x}_2\right)$$

$$\begin{tikzcd} \vec{h}_1 \arrow[r, "U"] & \vec{h}_2 \\ x_1 \arrow[u] & \arrow[u, "W"'] \end{tikzcd}$$



Sigmoid

Need for better units: Vanishing Gradient



Gradient signal from faraway is lost because it's much smaller than gradient signal from close-by.

So model weights are only updated only with respect to near effects, not long-term effects.

Effect of vanishing gradient on RNN LM

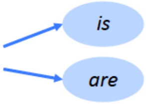
- **LM task:** *When she tried to print her tickets, she found that the printer was out of toner. She went to the stationery store to buy more toner. It was very overpriced. After installing the toner into the printer, she finally printed her ________*

- To learn from this training example, the RNN-LM needs to model the dependency between *"tickets"* on the 7th step and the target word *"tickets"* at the end.

- But if gradient is small, the model can't learn this dependency
  - So the model is unable to predict similar long-distance dependencies at test time

Effect of vanishing gradient on RNN LM

- **LM task:** *The writer of the books ___*



- **Correct answer**: *The writer of the books is planning a sequel*
- **Syntactic recency:** *The writer of the books is* (correct)
- **Sequential recency:** *The writer of the books are* (incorrect)
- Due to vanishing gradient, RNN-LMs are better at learning from sequential recency than syntactic recency, so they make this type of error more often than we'd like [Linzen et al 2016]

How to fix vanishing gradient problem?

- The main problem is that it is too difficult for the RNN to learn to preserve information over many timesteps.
- In a vanilla RNN, the hidden state is constantly being rewritten

$$h^{(t)} = tanh(Uh^{(t-1)} + Wx^{(t)})$$

- How about better RNN units?

Using Gates for better RNN units

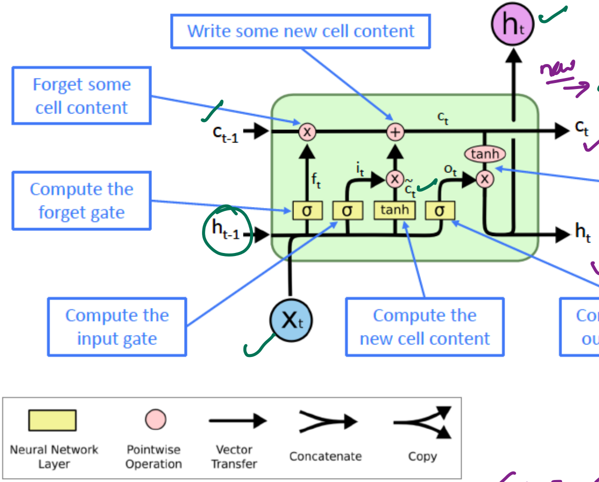
- The gates are also vectors
- On each timestep, each element of the gates can be open (1), close (0) or somewhere in-between.
- The gates are dynamic: their value is computed based on the current context.

Two famous architectures

GRUs, LSTMs

Long Short Term Memory (LSTM)

You can think of the LSTM equations visually like this:



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

new → $\tilde{C}_t = \tanh(W x_t + U h_{t-1})$

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

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

$h_t = o_t \odot f(\tilde{C}_t)$

$C_t = ?$ $\quad h_t = ?$

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

LSTM: More Details

- For context management, an explicit context layer is added to the architecture
- It makes use of specialized neural units (gates) to control the flow of information
- The gates share a common design feature, and choice of sigmoid pushes its output to 0 or 1, thus it works as a binary mask.

LSTM: In Equations

Forget Gate

Controls what is kept vs forgotten from the context

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

4 U, 4 W, 1 V

Input Gate

Controls what parts of new cell content are written to the context

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

Output Gate

Controls what part of context are output to hidden state

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

$\tilde{c}_t$

New Cell content: $g_t = tanh(U_g h_{t-1} + W_g x_t)$

∇h_t

New Context Vector: $c_t = \boxed{i_t \odot g_t} + \boxed{f_t \odot c_{t-1}}$

$h_t = c_t$

New Hidden State: $h_t = o_t \odot tanh(c_t)$