DATA 586: Advanced Machine Learning

2023W2

Shan Du

Recurrent Neural Networks

- The conditional probability of token x_t at time step t only depends on the $n-1$ previous tokens.
- If we want to incorporate the possible effect of tokens earlier than time step t $-(n-1)$ on x_t, we need to increase n.
- However, the number of model parameters would also increase exponentially with it.

Recurrent Neural Networks

- Hence, rather than modeling $P(x_t|x_{t-1}, \dots, x_{t-n+1})$ it is preferable to use a latent variable model:

$$P(x_t|x_{t-1}, \dots, x_1) \approx P(x_t|h_{t-1})$$

where h_{t-1} is a hidden state that stores the sequence information up to time step $t-1$.

- In general, the hidden state at any time step t could be computed based on both the current input x_t and the previous hidden state h_{t-1}:

$$h_t = f(x_t, h_{t-1})$$

Recurrent Neural Networks with Hidden States

- Assume that we have a minibatch of inputs $X_t \in R^{n \times d}$ at time step t.
- At any time step t, the computation of the hidden state can be treated as:

  (i) concatenating the input X_t at the current time step t and the hidden state H_{t-1} at the previous time step $t-1$;

  (ii) feeding the concatenation result into a fully connected layer with the activation function ϕ.

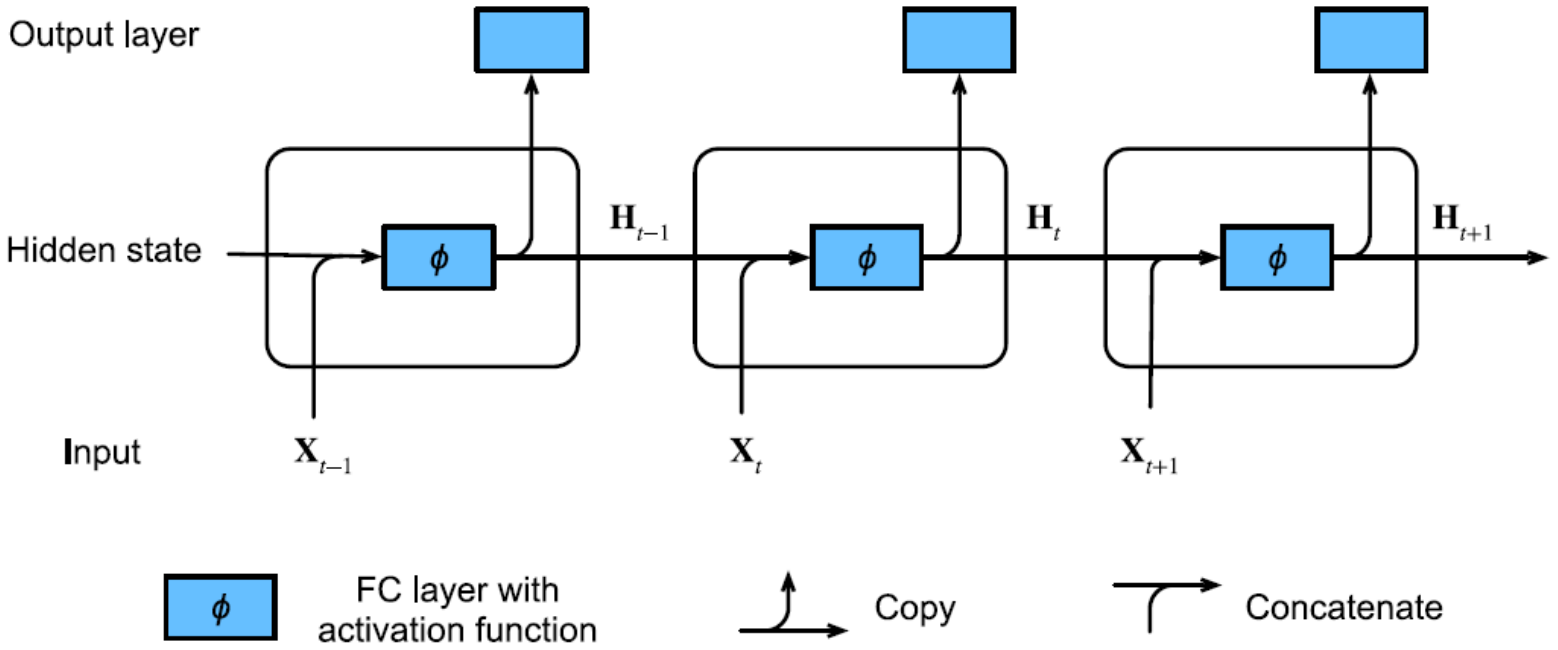
Recurrent Neural Networks with Hidden States

- The output of such a fully connected layer is the hidden state H_t of the current time step t.
- H_t will participate in computing the hidden state H_{t+1} of the next time step $t+1$. What is more, H_t will also be fed into the fully connected output layer to compute the output O_t of the current time step t.

$$H_t = \phi(X_t W_{xh} + H_{t-1} W_{hh} + b_h)$$

$$O_t = H_t W_{hq} + b_q$$

Recurrent Neural Networks with Hidden States



An RNN with a hidden state.

Recurrent Neural Networks with Hidden States

- The calculation of $X_t W_{xh} + H_t W_{hh}$ for the hidden state is equivalent to matrix multiplication of concatenation of X_t and H_t and concatenation of W_{xh} and W_{hh}.
- This can be proven in mathematics. In the following we just use a simple code snippet to show this.

Recurrent Neural Networks with Hidden States

- To begin with, we define matrices X, W_xh, H, and W_hh, whose shapes are (3, 1), (1, 4), (3, 4), and (4, 4), respectively.
- Multiplying X by W_xh, and H by W_hh, respectively, and then adding these two multiplications, we obtain a matrix of shape (3, 4).
- Then we concatenate the matrices X and H along columns (axis 1), and the matrices W_xh and W_hh along rows (axis 0). These two concatenations result in matrices of shape (3, 5) and of shape (5, 4), respectively. Multiplying these two concatenated matrices, we obtain the same output matrix of shape (3, 4) as above.

Recurrent Neural Networks with Hidden States

In [1]:
```
import torch
X, W_xh = torch.randn(3, 1), torch.randn(1, 4)
H, W_hh = torch.randn(3, 4), torch.randn(4, 4)
torch.matmul(X, W_xh) + torch.matmul(H, W_hh)
```

Out[1]:
```
tensor([[-3.2895,  0.0395,  0.4285,  2.6560],
        [ 2.6640,  2.3495,  0.8087, -0.3156],
        [ 4.1727,  8.1450,  6.6076,  5.8123]])
```

In [2]:
```
torch.matmul(torch.cat((X, H), 1), torch.cat((W_xh, W_hh), 0))
```

Out[2]:
```
tensor([[-3.2895,  0.0395,  0.4285,  2.6560],
        [ 2.6640,  2.3495,  0.8087, -0.3156],
        [ 4.1727,  8.1450,  6.6076,  5.8123]])
```

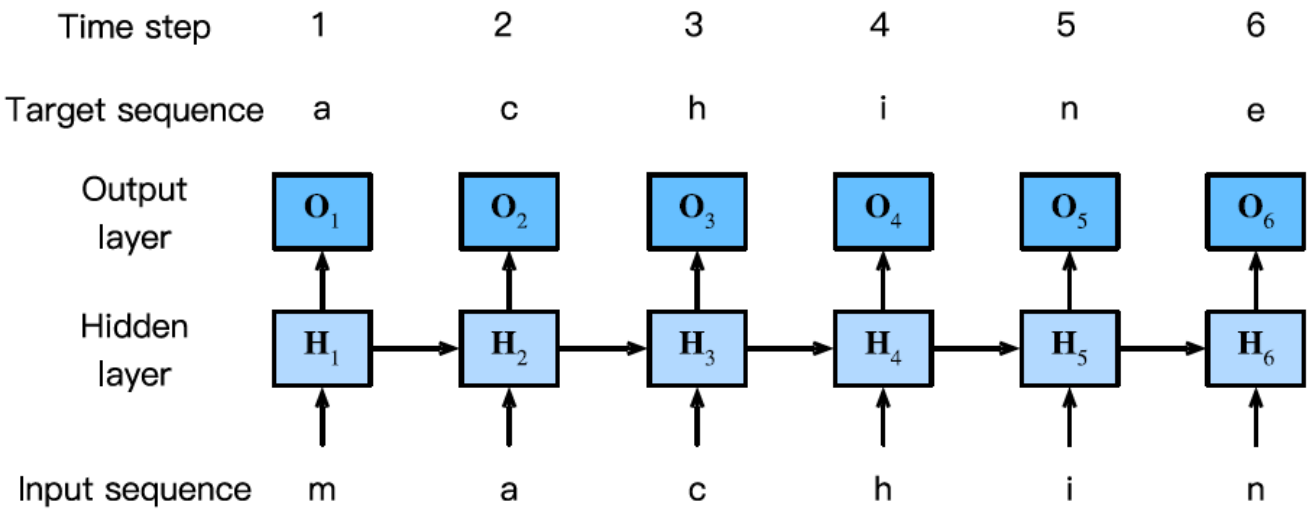
RNN-based Character-Level Language Models

- We aim to predict the next token based on the current and past tokens, thus we shift the original sequence by one token as the targets (labels).
- Let the minibatch size be one, and the sequence of the text be “machine”. To simplify training, we tokenize text into characters rather than words and consider a character-level language model.
- During the training process, we run a softmax operation on the output from the output layer for each time step, and then use the cross-entropy loss to compute the error between the model output and the target.

RNN-based Character-Level Language Models

- Due to the recurrent computation of the hidden state in the hidden layer, the output of time step 3 is determined by the text sequence “m”, “a”, and “c”.
- Since the next character of the sequence in the training data is “h”, the loss of time step 3 will depend on the probability distribution of the next character generated based on the feature sequence “m”, “a”, “c” and the target “h” of this time step.
- In practice, each token is represented by a d-dimensional vector, and we use a batch size $n > 1$. Therefore, the input X_t at time step t will be a $n \times d$ matrix.

RNN-based Character-Level Language Models



A character-level language model based on the RNN. The input and target sequences are machin and achine, respectively.

Backpropagation Through Time

- This procedure requires us to expand (or unroll) the computational graph of an RNN one time step at a time.
- The unrolled RNN is essentially a feedforward neural network with the special property that the same parameters are repeated throughout the unrolled network, appearing at each time step.
- Then, just as in any feedforward neural network, we can apply the chain rule, backpropagating gradients through the unrolled net.
- The gradient with respect to each parameter must be summed across all places that the parameter occurs in the unrolled net.

Backpropagation Through Time

- Complications arise because sequences can be rather long. It's not unusual to work with text sequences consisting of over a thousand tokens.
- Note that this poses problems both from a computational (too much memory) and optimization (numerical instability) standpoint.
- Input from the first step passes through over 1000 matrix products before arriving at the output, and another 1000 matrix products are required to compute the gradient.

Analysis of Gradients in RNNs

- We use a simplified model, where we denote h_t as the hidden state, x_t as input, and o_t as output at time step t.
- The input and the hidden state can be concatenated before being multiplied by one weight variable in the hidden layer. Thus, we use w_h and w_o to indicate the weights of the hidden layer and the output layer, respectively.

Analysis of Gradients in RNNs

- As a result, the hidden states and outputs at each time steps are

$$h_t = f(x_t, h_{t-1}, w_h)$$
$$o_t = g(h_t, w_o)$$

where f and g are transformations of the hidden layer and the output layer, respectively.

- Hence, we have a chain of values $\{\dots (x_{t-1}, h_{t-1}, o_{t-1}), (x_t, h_t, o_t) \dots\}$ that depend on each other via recurrent computation.

Analysis of Gradients in RNNs

- The forward propagation is fairly straightforward. All we need is to loop through the (x_t, h_t, o_t) triples one time step at a time.
- The discrepancy between output o_t and the desired target y_t is then evaluated by an objective function across all the T time steps as

$$L(x_1, \ldots, x_T, y_1, \ldots, y_T, w_h, w_o) = \frac{1}{T}\sum_{t=1}^{T} l(y_t, o_t)$$

Analysis of Gradients in RNNs

- For backpropagation, matters are a bit trickier, especially when we compute the gradients with regard to the parameters w_h of the objective function L. To be specific, by the chain rule,

$$\frac{\partial L}{\partial w_h} = \frac{1}{T}\sum_{t=1}^{T}\frac{\partial l(y_t, o_t)}{\partial w_h}$$

$$= \frac{1}{T}\sum_{t=1}^{T}\frac{\partial l(y_t,o_t)}{\partial o_t}\frac{\partial g(h_t,w_o)}{\partial h_t}\frac{\partial h_t}{\partial w_h}$$

The third factor $\frac{\partial h_t}{\partial w_h}$ is tricky, since we need to recurrently compute the effect of the parameter w_h on h_t.

Analysis of Gradients in RNNs

- h_t depends on both h_{t-1} and w_h, where computation of h_{t-1} also depends on w_h. Thus, evaluating the total derivate of h_t with respect to w_h using the chain rule yields

$$\frac{\partial h_t}{\partial w_h} = \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h} + \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial w_h}$$

- Remove the recurrent computation,

$$\frac{\partial h_t}{\partial w_h} = \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h} + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^{t} \frac{\partial f(x_j, h_{j-1}, w_h)}{\partial h_{j-1}} \right) \frac{\partial f(x_i, h_{i-1}, w_h)}{\partial w_h}$$

Backpropagation Through Time in Detail

- To keep things simple, we consider an RNN without bias parameters, whose activation function in the hidden layer uses the identity mapping ($\phi(x) = x$).
- For time step t, let the single example input and the target be $x_t \in R^d$ and y_t, respectively. The hidden state $h_t \in R^h$ and the output $o_t \in R^q$ are computed as

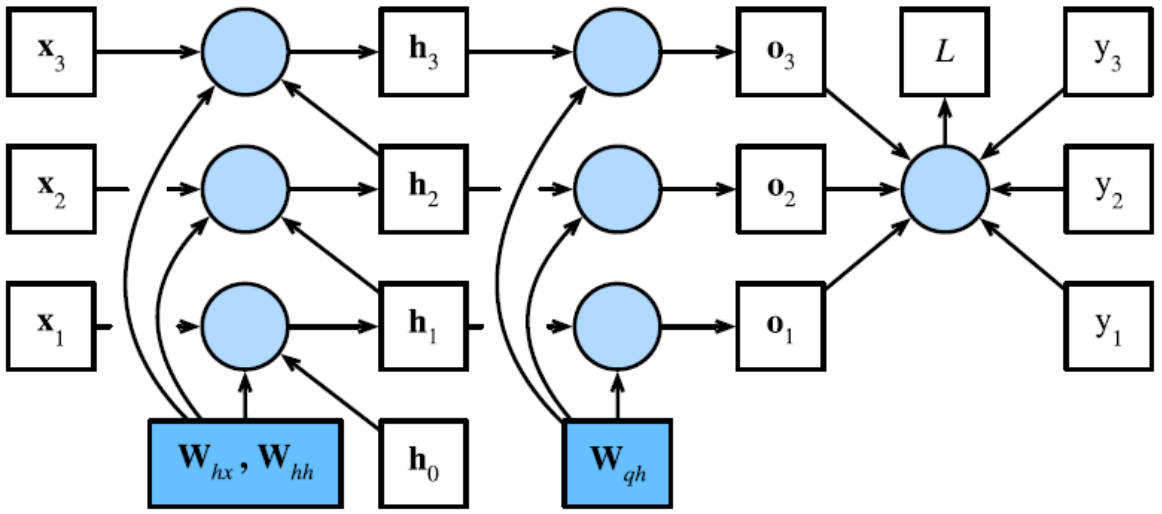
$$h_t = W_{hx}x_t + W_{hh}h_{t-1}$$
$$O_t = W_{qh}h_t$$

Backpropagation Through Time in Detail

where $W_{hx} \in R^{h \times d}$, $W_{hh} \in R^{h \times h}$, $W_{qh} \in R^{h \times q}$ are the weight parameters. Denote by $l(o_t, y_t)$ the loss at time step t. Our objective function, the loss over T time steps from the beginning of the sequence is thus

$$L = \frac{1}{T} \sum_{t=1}^{T} l(o_t, y_t)$$

Backpropagation Through Time in Detail



Computational graph showing dependencies for an RNN model with three time steps. Boxes represent variables (not shaded) or parameters (shaded) and circles represent operators.

The computation of the hidden states of time step 3, $\mathbf{h}_3$, depends on the model parameters $\mathbf{W}_{hx}$ and $\mathbf{W}_{hh}$, the hidden state of the last time step $\mathbf{h}_2$, and the input of the current time step $\mathbf{x}_3$.

Backpropagation Through Time in Detail

- Differentiating the objective function with respect to the model output at any time step t is fairly straightforward:

$$\frac{\partial L}{\partial o_t} = \frac{\partial l(o_t, y_t)}{T \cdot \partial o_t}$$

- $\frac{\partial L}{\partial W_{qh}} = \sum_{t=1}^{T} prod(\frac{\partial L}{\partial o_t}, \frac{\partial o_t}{\partial W_{qh}}) = \sum_{t=1}^{T} \frac{\partial L}{\partial o_t} h_t^T$
- $\frac{\partial L}{\partial W_{hx}} = \sum_{t=1}^{T} prod(\frac{\partial L}{\partial h_t}, \frac{\partial h_t}{\partial W_{hx}}) = \sum_{t=1}^{T} \frac{\partial L}{\partial h_t} x_t^T$
- $\frac{\partial L}{\partial W_{hh}} = \sum_{t=1}^{T} prod(\frac{\partial L}{\partial h_t}, \frac{\partial h_t}{\partial W_{hh}}) = \sum_{t=1}^{T} \frac{\partial L}{\partial h_t} h_{t-1}^T$

Modern Recurrent Neural Networks

- Long Short-Term Memory (LSTM)
- Gated Recurrent Units (GRU)
- Deep Recurrent Neural Networks
- Bidirectional Recurrent Neural Networks
- Encoder-Decoder Architecture

Long Short-Term Memory (LSTM)

- One of the first and most successful techniques for addressing vanishing gradients came in the form of the long short-term memory (LSTM) model due to Hochreiter and Schmidhuber (1997).
- LSTMs resemble standard recurrent neural networks but here each ordinary recurrent node is replaced by a *memory cell*. Each memory cell contains an *internal state*, i.e., a node with a self-connected recurrent edge of fixed weight 1, ensuring that the gradient can pass across many time steps without vanishing or exploding.

Long Short-Term Memory (LSTM)

- The term “long short-term memory” comes from the following intuition. Simple recurrent neural networks have *long-term memory* in the form of weights. The weights change slowly during training, encoding general knowledge about the data. They also have *short-term memory* in the form of ephemeral activations, which pass from each node to successive nodes.
- The LSTM model introduces an intermediate type of storage via the memory cell. A memory cell is a composite unit, built from simpler nodes in a specific connectivity pattern, with the novel inclusion of multiplicative nodes.

Gated Memory Cell

- Each memory cell is equipped with an *internal state* and a number of multiplicative gates that determine whether (i) a given input should impact the internal state (*the input gate*), (ii) the internal state should be flushed to 0 (*the forget gate*), and (iii) the internal state of a given neuron should be allowed to impact the cell's output (*the output gate*).

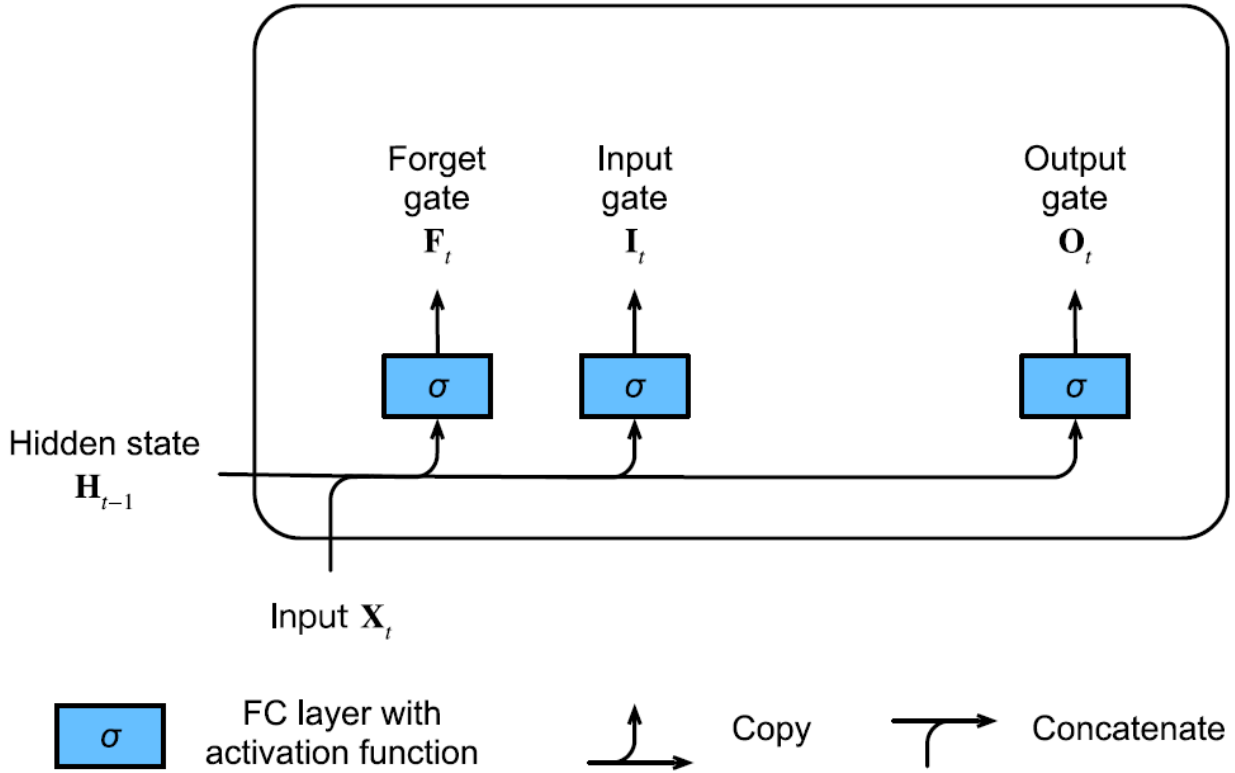
Gated Hidden State

- The key distinction between vanilla RNNs and LSTMs is that the latter support gating of the hidden state. This means that we have dedicated mechanisms for when a hidden state should be *updated* and also when it should be *reset*.
- These mechanisms are learned and they address the concerns listed above. For instance, if the first token is of great importance we will learn not to update the hidden state after the first observation. Likewise, we will learn to skip irrelevant temporary observations. Last, we will learn to reset the latent state whenever needed.

Input Gate, Forget Gate, and Output Gate

- The data feeding into the LSTM gates are the input at the current time step and the hidden state of the previous time step.
- Three fully connected layers with sigmoid activation functions compute the values of the input, forget, and output gates. As a result of the sigmoid activation, all values of the three gates are in the range of (0, 1).

Input Gate, Forget Gate, and Output Gate



Computing the input gate, the forget gate, and the output gate in an LSTM model.

Input Gate, Forget Gate, and Output Gate

- Additionally, we require an *input node*, typically computed with a *tanh* activation function. Intuitively, the *input gate* determines how much of the input node's value should be added to the current memory cell internal state. The *forget gate* determines whether to keep the current value of the memory or flush it. And the *output gate* determines whether the memory cell should influence the output at the current time step.

Input Gate, Forget Gate, and Output Gate

- Mathematically, suppose that there are h hidden units, the batch size is n, and the number of inputs is d. Thus, the input is $X_t \in R^{n\times d}$ and the hidden state of the previous time step is $H_{t-1} \in R^{n\times h}$. Correspondingly, the gates at time step t are defined as follows: the input gate is $I_t \in R^{n\times h}$, the forget gate is $F_t \in R^{n\times h}$, and the output gate is $O_t \in R^{n\times h}$. They are calculated as follows:

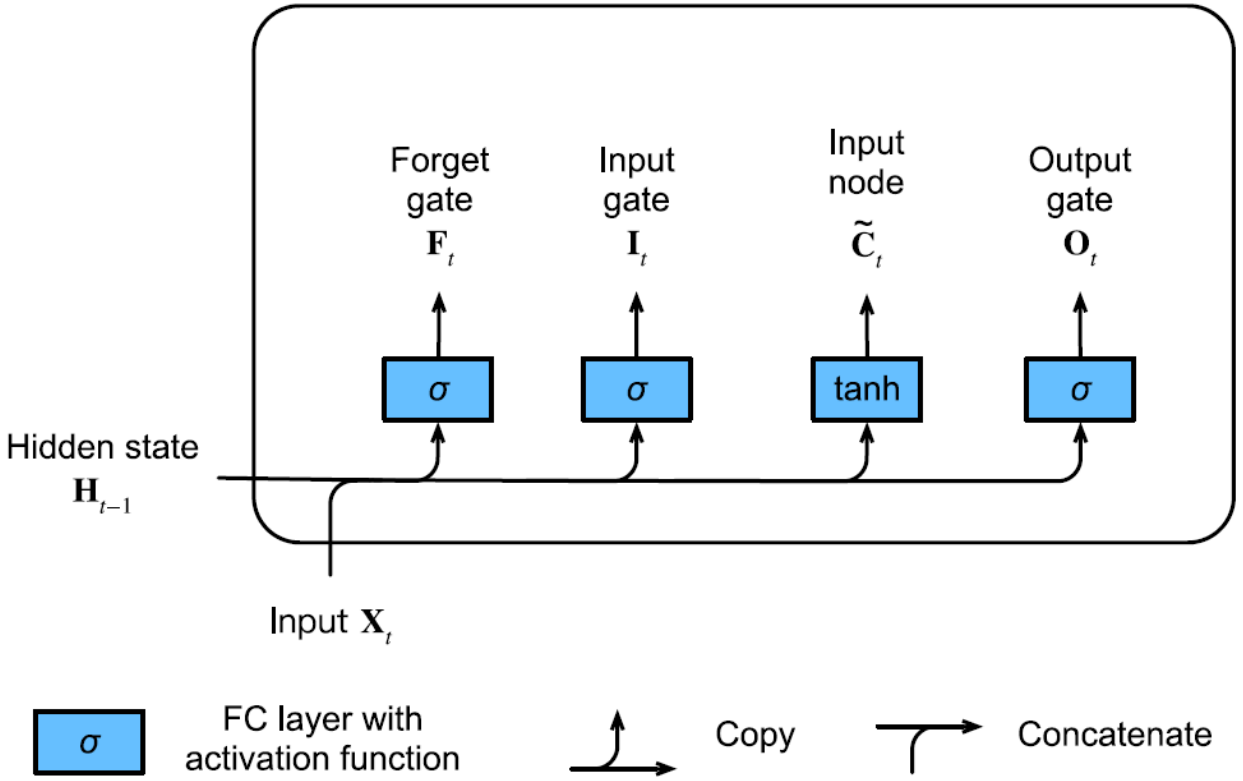
$$I_t = \sigma(X_t W_{xi} + H_{t-1} W_{hi} + b_i)$$
$$F_t = \sigma(X_t W_{xf} + H_{t-1} W_{hf} + b_f)$$
$$O_t = \sigma(X_t W_{xo} + H_{t-1} W_{ho} + b_o)$$

Input Node

- Next we design the memory cell. Since we have not specified the action of the various gates yet, we first introduce the *input node* $\tilde{C}_t \in R^{n \times h}$. Its computation is similar to that of the three gates described above, but using a tanh function with a value range for $(-1, 1)$ as the activation function. This leads to the following equation at time step t:

$$\tilde{C}_t = tanh(X_t W_{xc} + H_{t-1} W_{hc} + b_c)$$

Input Node



Computing the input node in an LSTM model.

Memory Cell Internal State

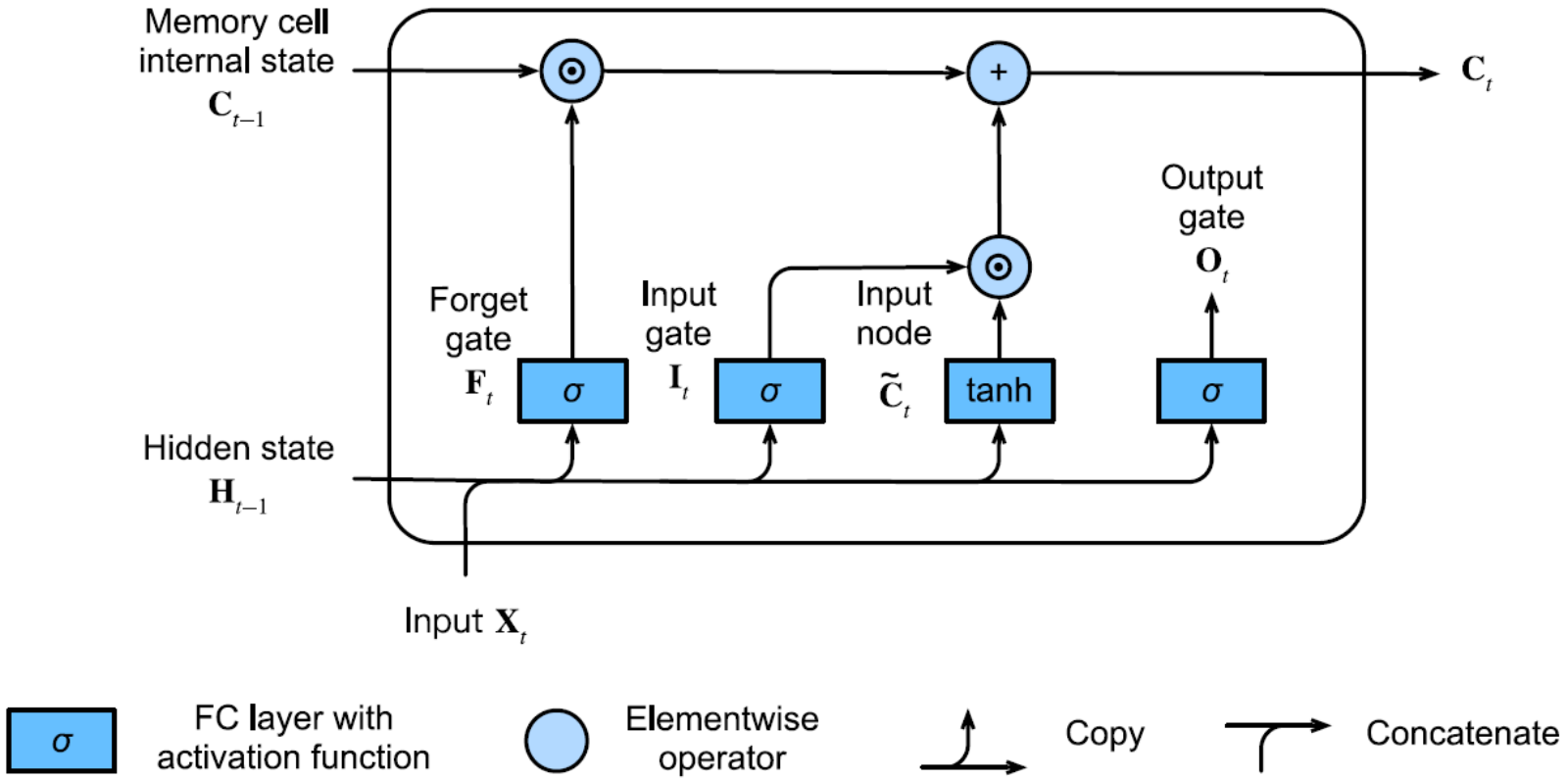
- In LSTMs, the input gate I_t governs how much we take new data into account via $\tilde{C}_t$ and the forget gate F_t addresses how much of the old cell internal state C_{t-1} we retain.
- Using the Hadamard (elementwise) product operator $\odot$ we arrive at the following update equation:

$$C_t = F_t \odot C_{t-1} + I_t \odot \tilde{C}_t$$

Memory Cell Internal State

- If the forget gate is always 1 and the input gate is always 0, the memory cell internal state C_{t-1} will remain constant forever, passing unchanged to each subsequent time step.
- However, input gates and forget gates give the model the flexibility to learn when to keep this value unchanged and when to perturb it in response to subsequent inputs. In practice, this design alleviates the vanishing gradient problem, resulting in models that are much easier to train, especially when facing datasets with long sequence lengths.

Memory Cell Internal State



Computing the memory cell internal state in an LSTM model.

Hidden State

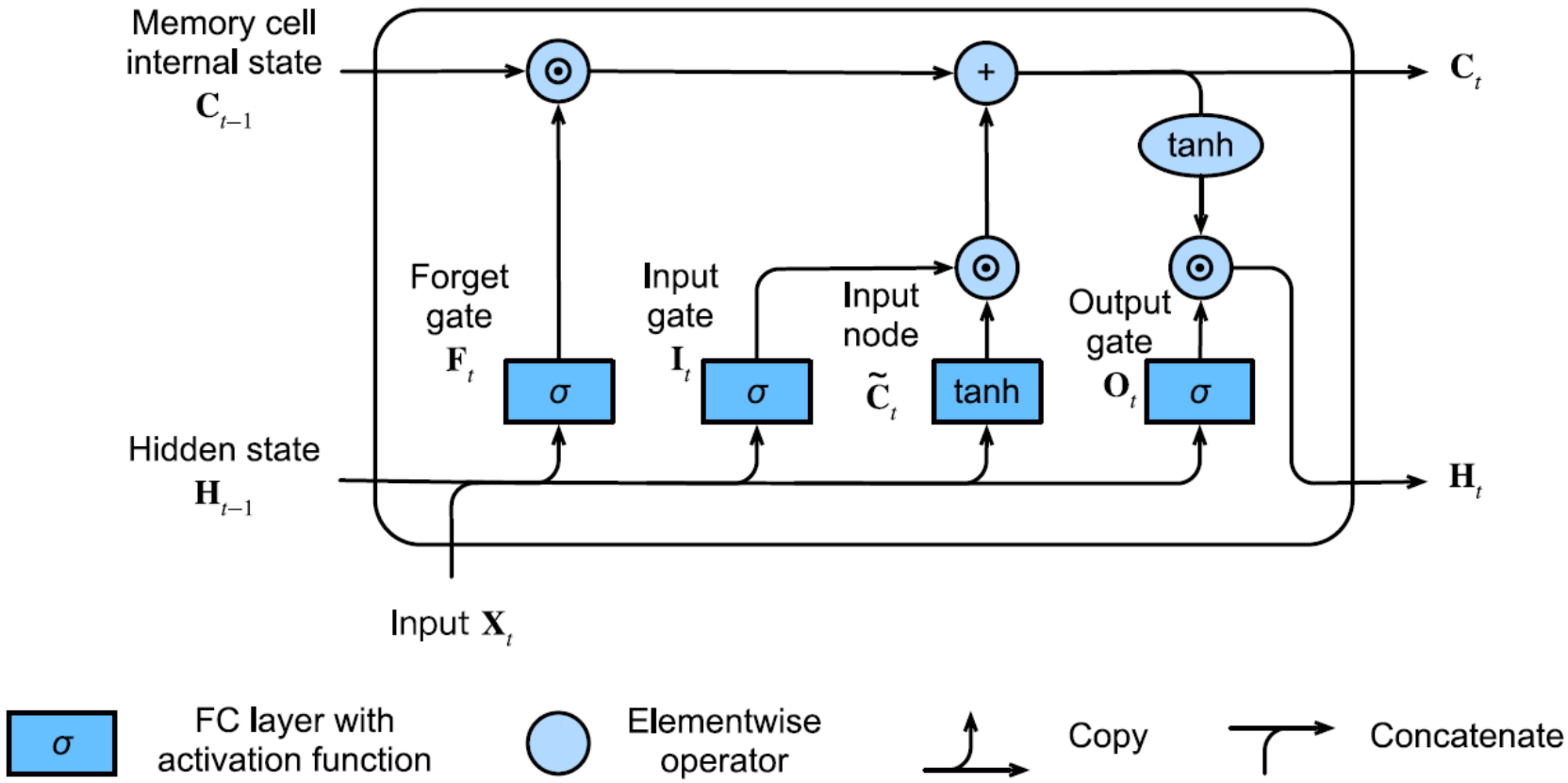
- Last, we need to define how to compute the output of the memory cell, i.e., the hidden state $H_t \in R^{n \times h}$, as seen by other layers. This is where the output gate comes into play.
- In LSTMs, we first apply *tanh* to the memory cell internal state and then apply another pointwise multiplication, this time with the output gate. This ensures that the values of H_t are always in the interval $(-1, 1)$:

$$H_t = O_t \odot \tanh(C_t)$$

Hidden State

- Whenever the output gate is close to 1, we allow the memory cell internal state to impact the subsequent layers uninhibited, whereas for output gate values close to 0, we prevent the current memory from impacting other layers of the network at the current time step.
- Note that a memory cell can accrue information across many time steps without impacting the rest of the network (so long as the output gate takes values close to 0), and then suddenly impact the network at a subsequent time step as soon as the output gate flips from values close to 0 to values close to 1.

Hidden State



Computing the hidden state in an LSTM model.