# Recurrent Neural Networks

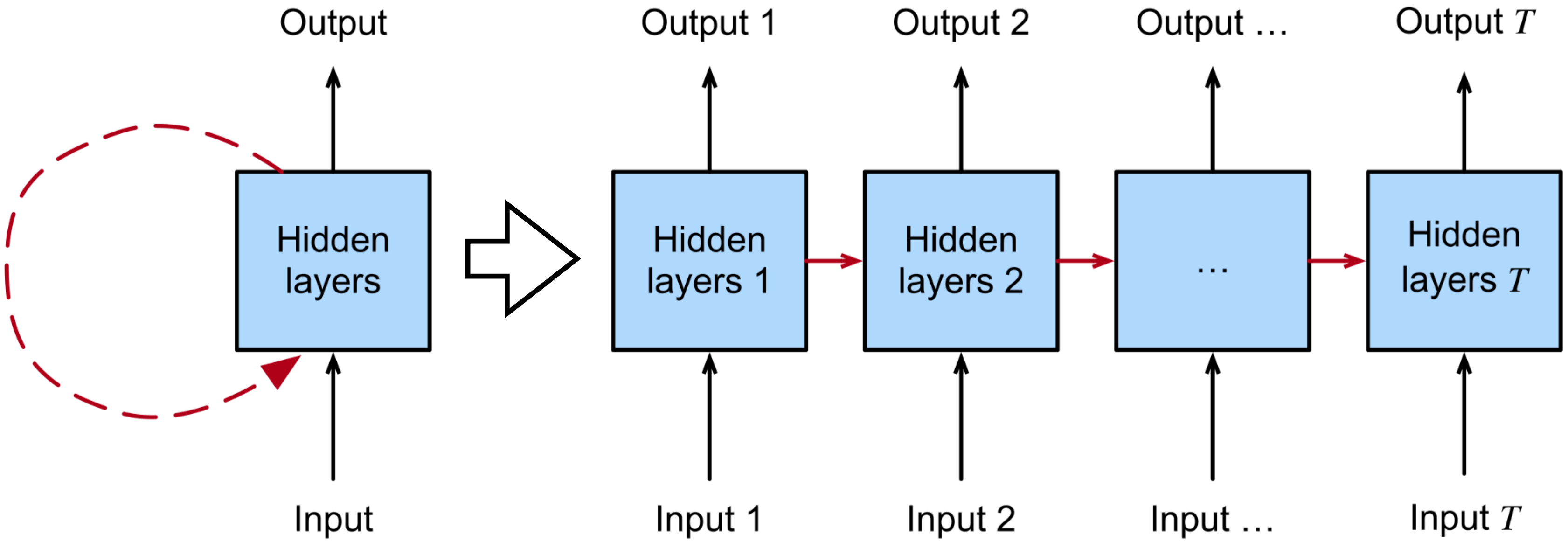
ITM528 Deep Learning

**Taemoon Jeong**

# Recurrent Neural Networks

- Until now, we have focused on fixed-length data.
  - These datasets are sometimes called tabular, because they can be arranged in tables, where each example gets its own row, and each attribute gets its own column.
- But what should we do when faced with a sequence of images, as in a video?
- Recurrent neural networks (RNNs) are deep learning models that capture the dynamics of sequences via recurrent connections, which can be thought of as cycles in the network of nodes.

# Recurrent Neural Networks



- We can unfold the RNN (on the left) over time steps.

# Sequence Modeling

ordered data

# Working with Sequences

- Up until now, we have focussed on models whose inputs consisted of a single feature vector $\mathbf{X} \in \mathbb{R}^d$.
- Now, we focus on inputs that consist of an ordered list of feature vectors $\mathbf{x}_1, \ldots, \mathbf{x}_T$ where each feature vector $\mathbf{x}_t$ indexed by a time step $t \in \mathbb{Z}^+$ lies in $\mathbb{R}^d$.
- Previously, when dealing with individual inputs, we assumed that they were sampled independently from the same underlying distribution $P(X)$.
    - While we still assume that entire sequences are sampled independently, we cannot assume that the data arriving at each time step are independent of each other.

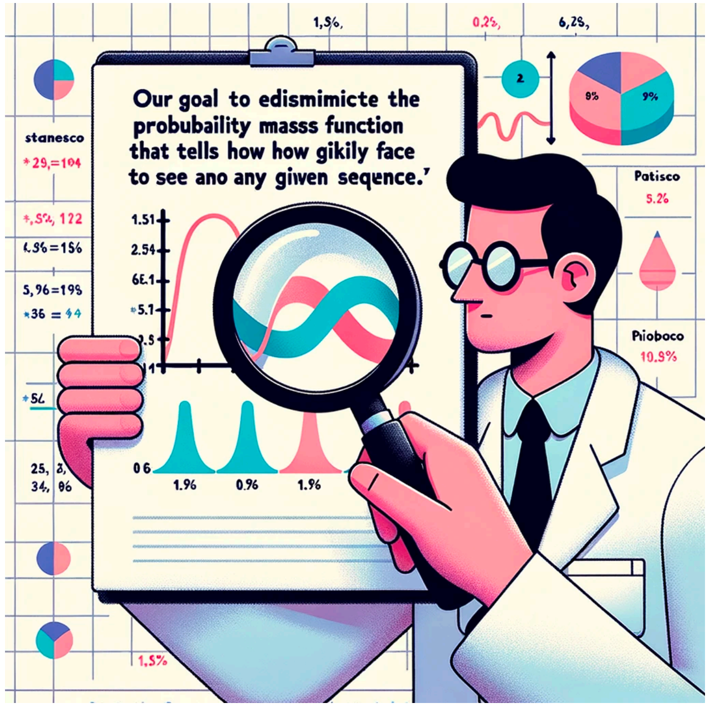
# Targets in Sequences

- We sometimes wish to predict a fixed target $y$ given sequentially structured input (e.g., sentiment classification based on a movie review).
- At other times, we wish to predict a sequentially structured target $(y_1, \ldots, y_T)$ given a fixed input (e.g., image captioning).
- Still, other times, our goal is to predict sequentially structured targets based on sequentially structured inputs (e.g., machine translation or video captioning).
- Sequence-to-sequence (seq2seq) tasks take two forms:

  (i) **aligned**: where the input at each time step aligns with a corresponding target (e.g., speech tagging).

  (ii) **unaligned**: where the input and target do not necessarily exhibit a step-for-step correspondence (e.g., machine translation).

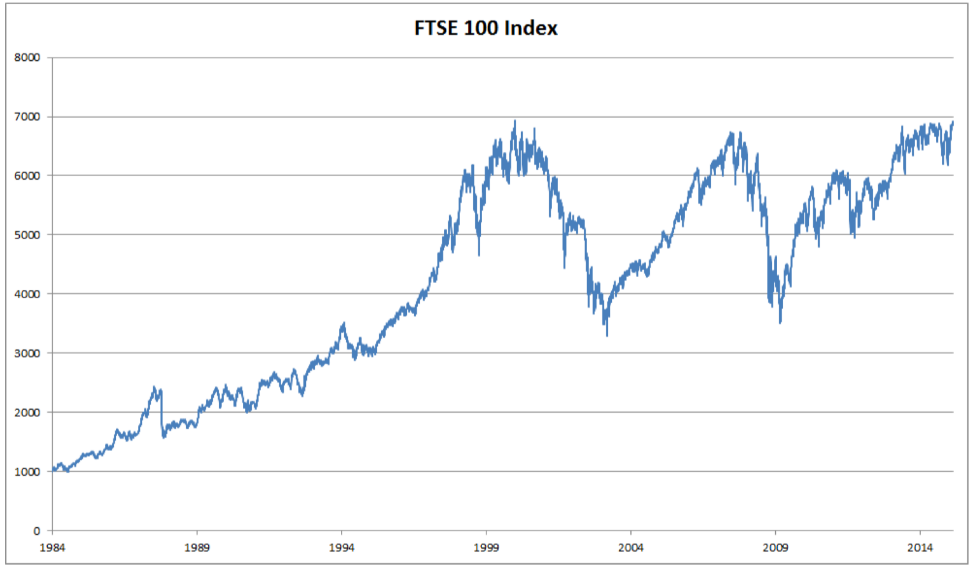
# Autoregressive Models


Our goal to edismimicte the probubaility masss function that tells how how gikily face to see ano any giwen seqence.'
stanesco
Patisco
5.2%
Pioboco
10.9%
1,5%

- We first tackle the most straightforward problem: unsupervised density modeling (i.e., sequence modeling) that does not require targets.
  - Our goal is to estimate the probability mass function that tells us how likely we are to see any given sequence, i.e., $p(\mathbf{x}_1, \ldots, \mathbf{x}_T)$.

https://chat.openai.com

# Financial Times Stock Exchange



- We will focus on stock price data from FTSE 100 index over about 30 years.
- The trader may interested in knowing the probability distribution

$$P(\mathbf{x}_t \,|\, \mathbf{x}_{t-1}, \ldots, \mathbf{x}_1)$$

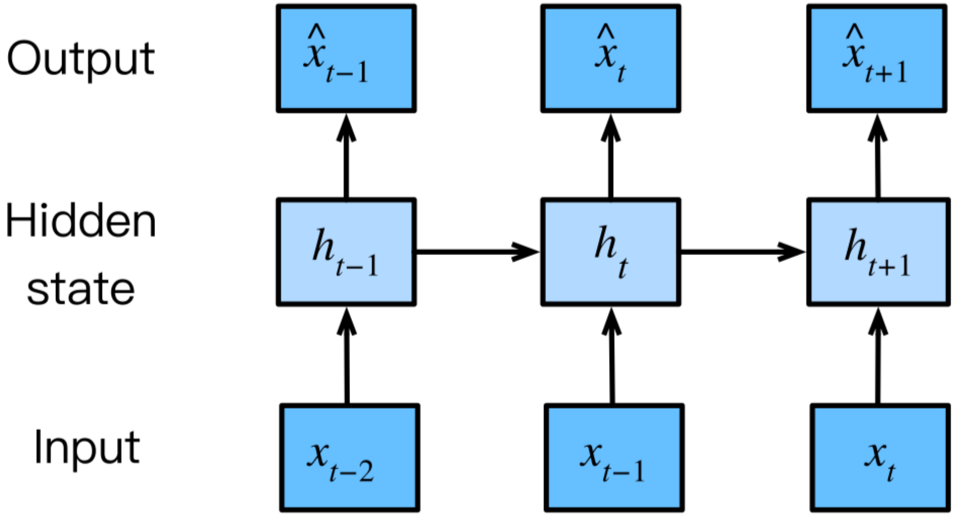
over prices that the index might take in the subsequent time step.

- While estimating the entire distribution can be difficult, one can simply estimate the conditional expectation

$$\mathbb{E}[\mathbf{x}_t \,|\, \mathbf{x}_{t-1}, \ldots, \mathbf{x}_1] \, .$$

전체로 접근하는건 힘들어서 -> 조건부로 접근

# Latent Autoregressive Model



- We might develop models that maintain some summary $\mathbf{h}_t$ of the past observations and at the same time update $\mathbf{h}_t$ in addition to the prediction $\hat{\mathbf{x}}_t$.
- This leads to models that estimate $\mathbf{x}_t$ with $\hat{\mathbf{x}}_t = P(\mathbf{x}_t \mid \mathbf{h}_t)$ and update of the form $\mathbf{h}_t = g(\mathbf{h}_{t-1}, \mathbf{x}_{t-1})$.
- Since $\mathbf{h}_t$ is never observed, these models are also called latent autoregressive models.

# Sequence Models

- Sometimes, especially when working with language, we wish to estimate the joint probability of an entire sequence.
    - This is a common task when working with sequences composed of discrete tokens, such as words.
- Generally, these estimated functions are called sequence models, and for natural language data, they are called language models.
    - Language models can be used to evaluate the naturalness of two candidates' output generated by a machine translation system.
    - They can by used to sample sequences.
- By applying the chain rule of probability:

$$P(\mathbf{x}_1, \ldots \mathbf{x}_T) = P(\mathbf{x}_1 \,|\, \mathbf{x}_{t-1}, \mathbf{x}_{t-2}, \ldots, \mathbf{x}_1)$$

# Markov Models

sp에서 한거랑 같은 개념 인듯

- We may condition only on the $\tau$ previous time steps, i.e., $x_{t-1}, \ldots, x_{t-\tau}$, rather than the entire sequence history $x_{t-1}, \ldots, x_1$.
- Whenever we can throw away the history beyond the previous $\tau$ steps without any loss in predictive power, we say that the sequence satisfies a Markov condition, i.e., that the future is conditionally independent of the past, given the recent history.
- When $\tau = 1$, we say that the data is characterized by a first-order Markov model, and when $\tau = k$, we say that the data is characterized by a $k$-th order Markov model.
- For when the first-order Markov condition holds ($\tau = 1$), our joint probability becomes a product of each word given the previous word:

$$P(x_1, \ldots, x_T) = P(x_1) \prod_{t=1}^{T} P(x_t | x_{t-1}).$$

# The Order of Decoding

- Why did we have to represent the factorization of a text sequence $P(x_1, \dots, x_T)$ as a left-to-right chain of conditional probabilities?
- Why not right-to-left or some other, seemingly random order?
- In principle, there is nothing wrong with unfolding $P(x_1, \dots, x_T)$ in reverse order. The result is valid factorization:

$$P(x_1, \dots, x_T) = \prod_{t=T}^{1} P(x_t \mid x_{t+1}, \dots, x_T)$$

- However, there are many reasons why factorizing text in the same directions as we read it (left-to-right for most languages, but right-to-left for Arabic and Hebrew) is preferred for the task of language modeling.

# Tokenization

*매우 중요 별표치심

- Tokens are the atomic (indivisible) units of text.
- For example, we could represent the sentence "Baby needs a new pair of shoes"
  - as a sequence of 7 words, where the set of all words comprises a large vocabulary (tens or hundreds of thousands of words), or
  - as a sequence of 30 characters, using a much smaller vocabulary (there are only 256 distinct ASCII characters).
- When machines encounter unseen words, we label them as UNK (Unknown Token) to signify words not in their vocabulary. This is referred to as the OOV (Out-Of-Vocabulary) issue.

# Byte Pair Encoding (1/3)

- Suppose we have 4 words in our dictionary ('low', 'lower', 'newest', 'widest').
  - If we use a word-based tokenizer, we cannot handle 'lowest' (OOV problem).
- When applying BPE,
  - We first construct a vocabulary using character-based tokenization ('l', 'l', 'w', 'e', 'r', 'n', 's', 't', 'i', 'd').
  - Then, we specify the number of iterations of BPE.

https://wikidocs.net/22592

# Byte Pair Encoding (2/3)

# dictionary update!
l o w : 5,
l o w e r : 2,
n e w es t : 6,
w i d es t : 3

# vocabulary update!
l, o, w, e, r, n, s, t, i, d, es

# dictionary update!
l o w : 5,
l o w e r : 2,
n e w est : 6,
w i d est : 3

# vocabulary update!
l, o, w, e, r, n, s, t, i, d, es, est

# dictionary update!
lo w : 5,
lo w e r : 2,
n e w est : 6,
w i d est : 3

# vocabulary update!
l, o, w, e, r, n, s, t, i, d, es, est, lo

• • •

# dictionary update!
low : 5,
low e r : 2,
newest : 6,
widest : 3

# vocabulary update!
l, o, w, e, r, n, s, t, i, d, es, est,
lo, low, ne, new, newest, wi,
wid, widest

# Byte Pair Encoding (3/3)

```
# dictionary update!
low : 5,
low e r : 2,
newest : 6,
widest : 3
```

```
# vocabulary update!
l, o, w, e, r, n, s, t, i, d, es, est,
lo, low, ne, new, newest, wi,
wid, widest
```

- In the inference phase, when 'lowest' is given, it would have been OOV for word-based tokenization.
- However, BPE encodes
  - 'lowest' into ('low','est')
  - 'lowing' into ('low', 'i', 'n','g')
  - 'highing' into ('h', 'i', 'g', 'h', 'i', 'n', 'g')

# Language Models

# Language Models

- Previously, we saw how to map text sequences into tokens.
- Assume that the tokens in a text sequence of length $T$ are in turn $x_1, x_2, \ldots, x_T$.
- The goal of language models is to estimate the joint probability of the whole sequence:

$$P(x_1, x_2, \ldots, x_T).$$

- Language models are incredibly useful.
  - an ideal language model would be able to generate natural text just on its own, simply by drawing one token at a time $x_t \sim P(x_t \mid x_{t-1}, \ldots, x_1)$.
  - all text emerging from such a model would pass as natural language.
  - it would be sufficient for generating a meaningful dialog, simply by conditioning the text on previous dialog fragments.

# Learning Language Models

- Let's start by applying the basic probability rules:

$$P(x_1, x_2, \ldots, x_T) = \sum_{t=1}^{T} P(x_t \,|\, x_1, \ldots x_{t-1}).$$

- For a sequence with 4 tokens and the Markov property of different orders leads to:
  - $P(x_1, x_2, x_3, x_4) = P(x_1)P(x_2)P(x_3)P(x_4)$
  - $P(x_1, x_2, x_3, x_4) = P(x_1)P(x_2 \,|\, x_1)P(x_3 \,|\, x_2)P(x_4 \,|\, x_3)$
  - $P(x_1, x_2, x_3, x_4) = P(x_1)P(x_2 \,|\, x_1)P(x_3 \,|\, x_1, x_2)P(x_4 \,|\, x_2, x_3)$
- The probability formulae that involve one, two, and three variables (as above) are typically referred to as unigram, bigram, and trigram, respectively.
- Note that such probabilities are language model parameters.

# Word Frequency

- Then, how can we estimate the word probabilities?
- The probability of words can be calculated from the relative word frequency of a given word in the training dataset.
  - For example, the estimate $\hat{P}$('deep') can be calculated as the probability of any sentence starting with the word 'deep'.
  - A slightly less accurate approach would be to count all occurrences of the word 'deep' and divide it by the total number of words in the corpus.
- Moving on, we could attempt to estimate

$$\hat{P}(\text{'learning'}\,|\,\text{'deep'}) = \frac{n(\text{deep', learning'})}{n(\text{deep'})}$$

where $n(x)$ and $n(x, x')$ are the number of occurrences of singletons and consecutive word pairs, respectively.

- Unfortunately, estimating the probability of a word pair (and $n$-word combinations) is somewhat more difficult, since the occurrences of 'deep learning' are a lot less frequent.
- If the dataset is small or if the words are very rare, we might not find even a single one of them.

# Laplace Smoothing

- A common strategy to handle the zero frequency problem is to perform some form of Laplace smoothing.
- The solution is to add a small constant to all counts.
- Denote by $n$ the total number of words in the training set and $m$ the number of unique words:

$$\hat{P}(x) = \frac{n(x) + \epsilon_1/m}{n + \epsilon_1}$$

$$\hat{P}(x'|x) = \frac{n(x, x') + \epsilon_2 \hat{P}(x')}{n(x) + \epsilon_2}$$

$$\hat{P}(x''|x, x') = \frac{n(x, x', x'') + \epsilon_3 \hat{P}(x'')}{n(x, x') + \epsilon_3}$$

- Here $\epsilon_1$, $\epsilon_2$, and $\epsilon_3$ are hyperparameters.
- Unfortunately, models like this get unwieldy quickly for the following reasons.
  1. Many $n$-grams occur very rarely, making Laplace smoothing rather unsuitable.
  2. We need to store all counts.
  3. This entirely ignores the meaning of the words.

# Perplexity

- Let's discuss about how to measure the language model quality, which will be used to evaluate our models.
  - One way is to check how surprising the text is.
  - A good language model is able to predict with high-accuracy tokens what we will see next.
- Consider the following continuations of the phrase "It is raining":
  1. "It is raining outside"
  2. "It is raining banana tree"
  3. "It is raining piouw;kcj pwepoiut"
- In terms of quality, example 1 is clearly the best.
- However, shorter sequences are much more likely to occur than the longer ones.
- In a nutshell, a quantity called perplexity is computed as follows:

$$\exp\left(-\frac{1}{n}\sum_{t=1}^{n}\log P(x_t \mid x_{t-1},\ldots,x_1)\right).$$

# Recurrent Neural Networks

# Recurrent Neural Networks

- Previously, we described Markov models and $n$-grams for language modeling, where the conditional probability of the token $x_t$ at time step $t$ only depends on the $n-1$ previous tokens.
- However, the number of model parameters would also increase exponentially with it, as we need to store $|\mathscr{V}|^n$ numbers for a vocabulary set $\mathscr{V}$.
- Hence, rather than modeling $P(x_t | x_{t-1}, \ldots, x_{t-n+1})$, it is preferable to use a latent variable model:

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

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 $h_{t-1}$:

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

# Networks without Hidden States

- Let's take a look at an MLP with a single hidden layer.
- Let the hidden layer's activation function be $\phi$, a minibatch of examples $\mathbf{X} \in \mathbb{R}^{n\times d}$ with batch size $n$ and $d$ inputs, the hidden layer output $\mathbf{H} \in \mathbb{R}^{n\times h}$ is calculated as

$$\mathbf{H} = \phi(\mathbf{X}\mathbf{W}_{wh} + \mathbf{b}_h)$$

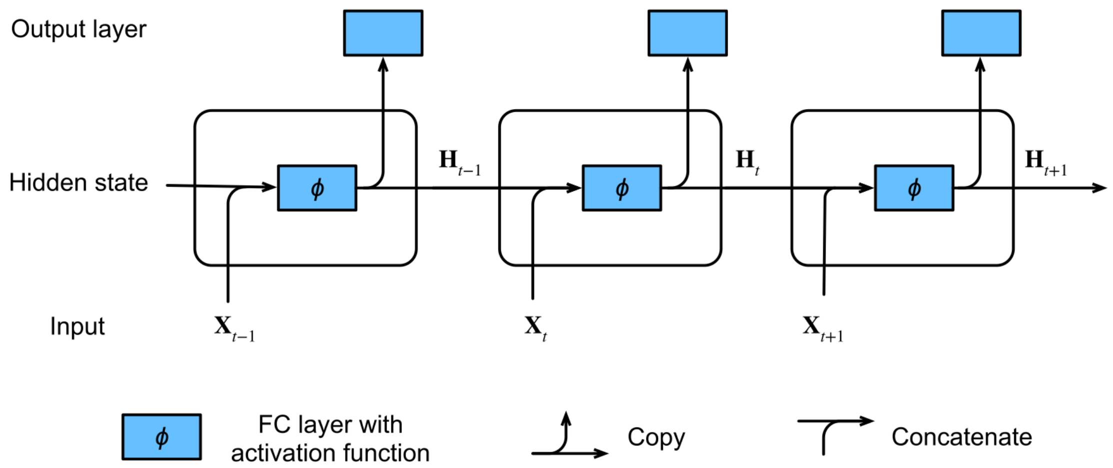
- We have the weight parameter $\mathbf{W}_{wh} \in \mathbb{R}^{d\times h}$, the bias parameter $\mathbf{b}_h \in \mathbb{R}^{1\times h}$, and the number of hidden units $h$, for the hidden layer.
- Next, the hidden layer output $\mathbf{H}$ is used as input of the output layer given by:

$$\mathbf{O} = \mathbf{H}\mathbf{W}_{hq} + \mathbf{b}_q,$$

where $\mathbf{O} \in \mathbb{R}^{n\times q}$ is the output variable, $\mathbf{W}_{hq} \in \mathbb{R}^{h\times q}$ is the weight parameter, and $\mathbf{b}_q \in \mathbb{R}^{1\times q}$ is the bias parameter of the output layer.

- If it is a classification problem, we can use softmax$(\mathbf{O})$ to compute the probability distribution of the output categories.

# Recurrent Networks with Hidden States



- Assume that we have a minibatch of inputs $\mathbf{X}_t \in \mathbb{R}^{n\times d}$ at time step $t$ and the hidden layer output $\mathbf{H}_t \in \mathbb{R}^{n\times h}$ at time step $t$.
- Unlike the MLP, we save the previous hidden layer output $\mathbf{H}_{t-1}$ and introduce a new weight parameter $\mathbf{W}_{hh} \in \mathbb{R}^{h\times h}$ to describe how to use the hidden layer output:

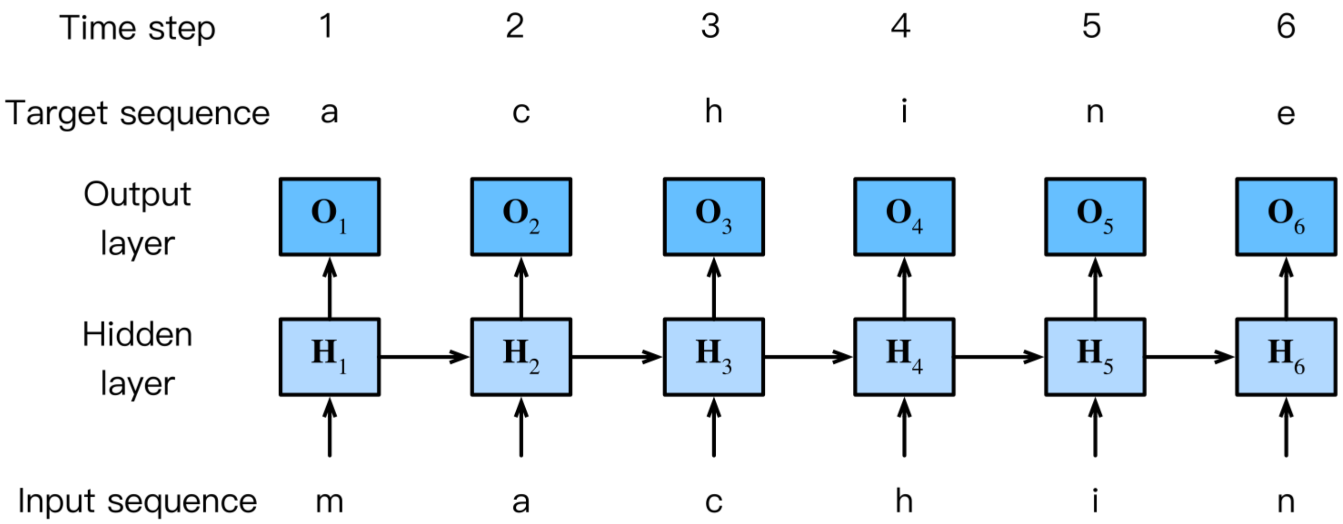
$$\mathbf{H}_t = \phi(\mathbf{X}_t\mathbf{W}_{xh} + \mathbf{H}_{t-1}\mathbf{W}_{hh} + \mathbf{b}_h).$$

- The hidden layer output captures and retains the sequence's historical information up to its time step. Therefore, such a hidden layer output is called a hidden state.
- The output of the output layer is computed as:

$$\mathbf{O}_t = \mathbf{H}_t\mathbf{W}_{hq} + \mathbf{b}_q.$$

# RNN-based Character-Level Language Models

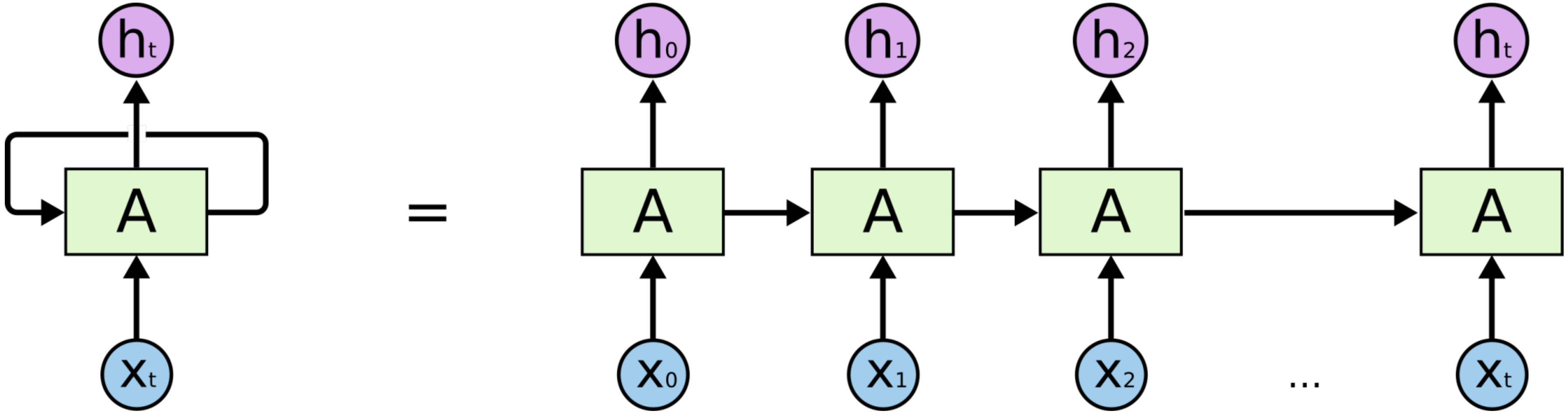
“machine”



- Recall that for language modeling, 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 target (labels).
- 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.

# Backpropagation Through Time

- Applying backpropagation in RNNs is called 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.



# Backpropagation Through Time

- We denote $x_t$, $h_t$, and $o_t$ as an input, hidden state, and output at a time step $t$. We use $w_h$ and $w_o$ to indicate the weights of the hidden layer and the output layer, respectively.

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

- Suppose we have a chain of values $\{\ldots, (x_{t-1}, h_{t-1}, o_{t-1}), (x_t, h_t, o_t), \ldots\}$ that depend on each other via recurrent computation.
- The objective function becomes

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

where $y_t$ is the output target at a time step $t$.

# Backpropagation Through Time

- From the chain rule,

여기 별표

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

이걸 ?

- While the first and second factors of the product are easy to compute, the third factor $\partial h_t / \partial w_h$ is tricky to compute since we need to recurrently compute the effect of the parameter $w_h$ on $h_t$ (where $t = 1, \ldots, T$).

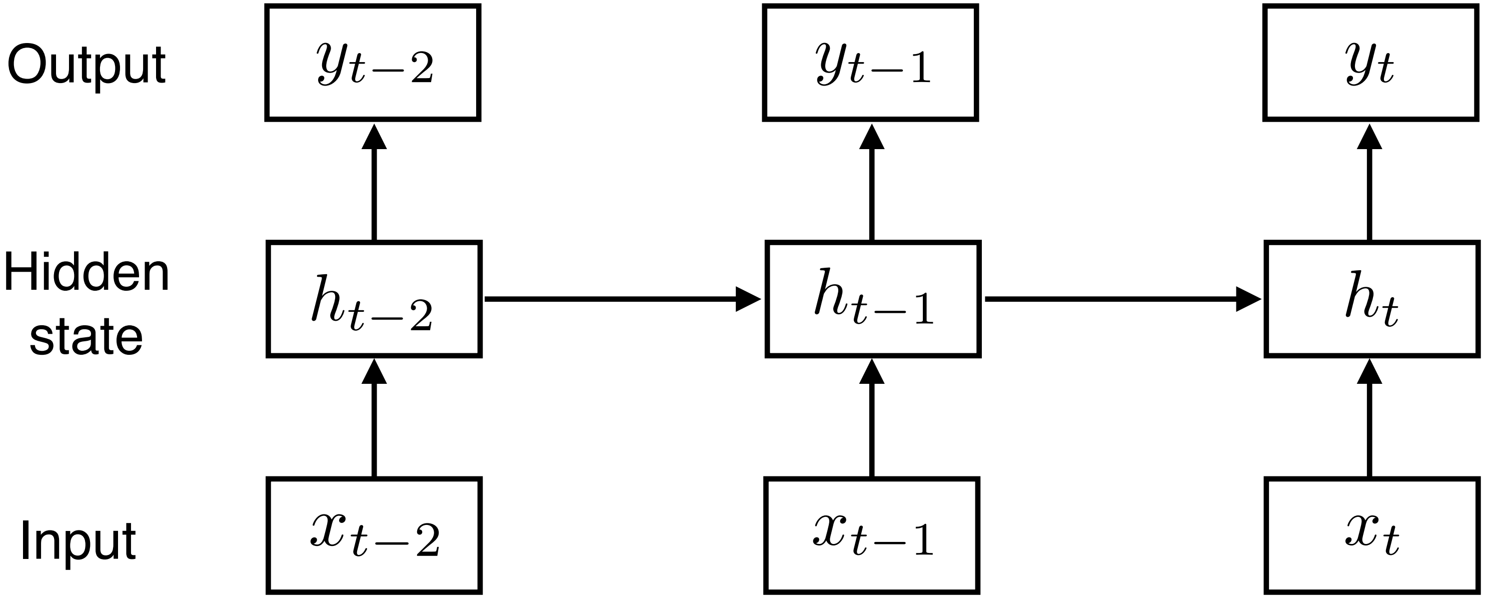
- To be specific, from the chain rule, we get

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

- While we can compute this, this chain can get very long whenever $t$ is large, especially due to the following term $\prod_{j=i+1}^{t}\frac{\partial f(x_j, h_{j-1}; w_h)}{\partial h_{j-1}}$.

# Long Short-Term Memory (LSTM)

# (Recall) Recurrent Neural Networks



$$h_1 = \phi(W^T h_0 + U^T x_1)$$

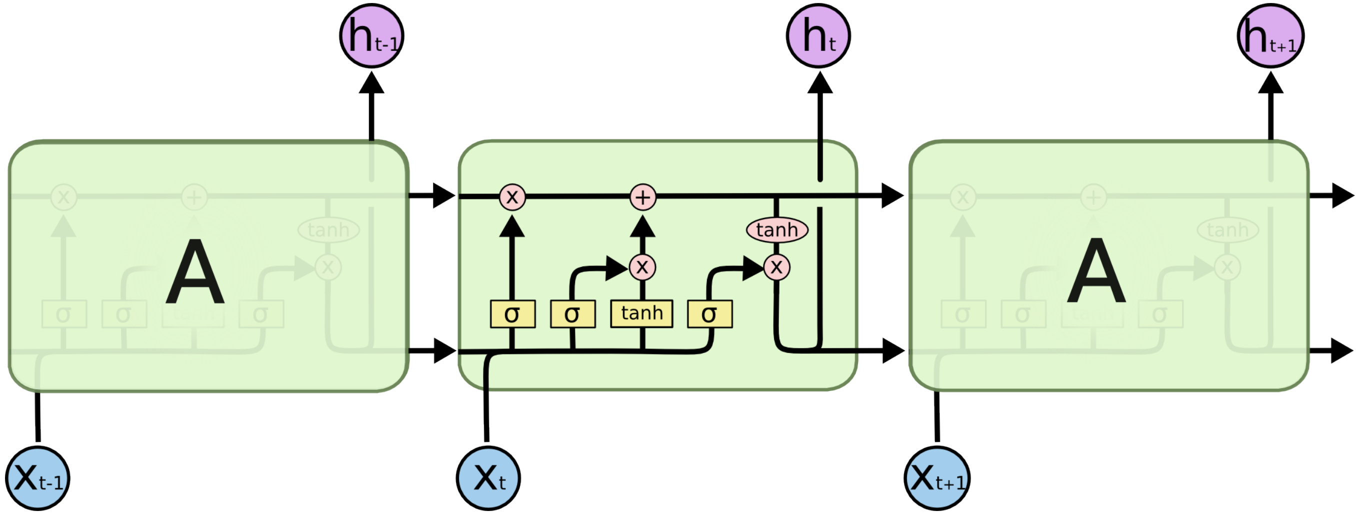
$$h_2 = \phi(W^T \phi(W^T h_0 + U^T x_1) + U^T x_2)$$

$$h_3 = \phi(W^T \phi(W^T \phi(W^T \phi(W^T h_0 + U^T x_1) + U^T x_2) + U^T x_1) + U^T x_3)$$

$$h_4 = \phi(W^T \phi(W^T \phi(W^T \phi(W^T \phi(W^T h_0 + U^T x_1) + U^T x_2) + U^T x_1) + U^T x_3) + U^T x_4)$$

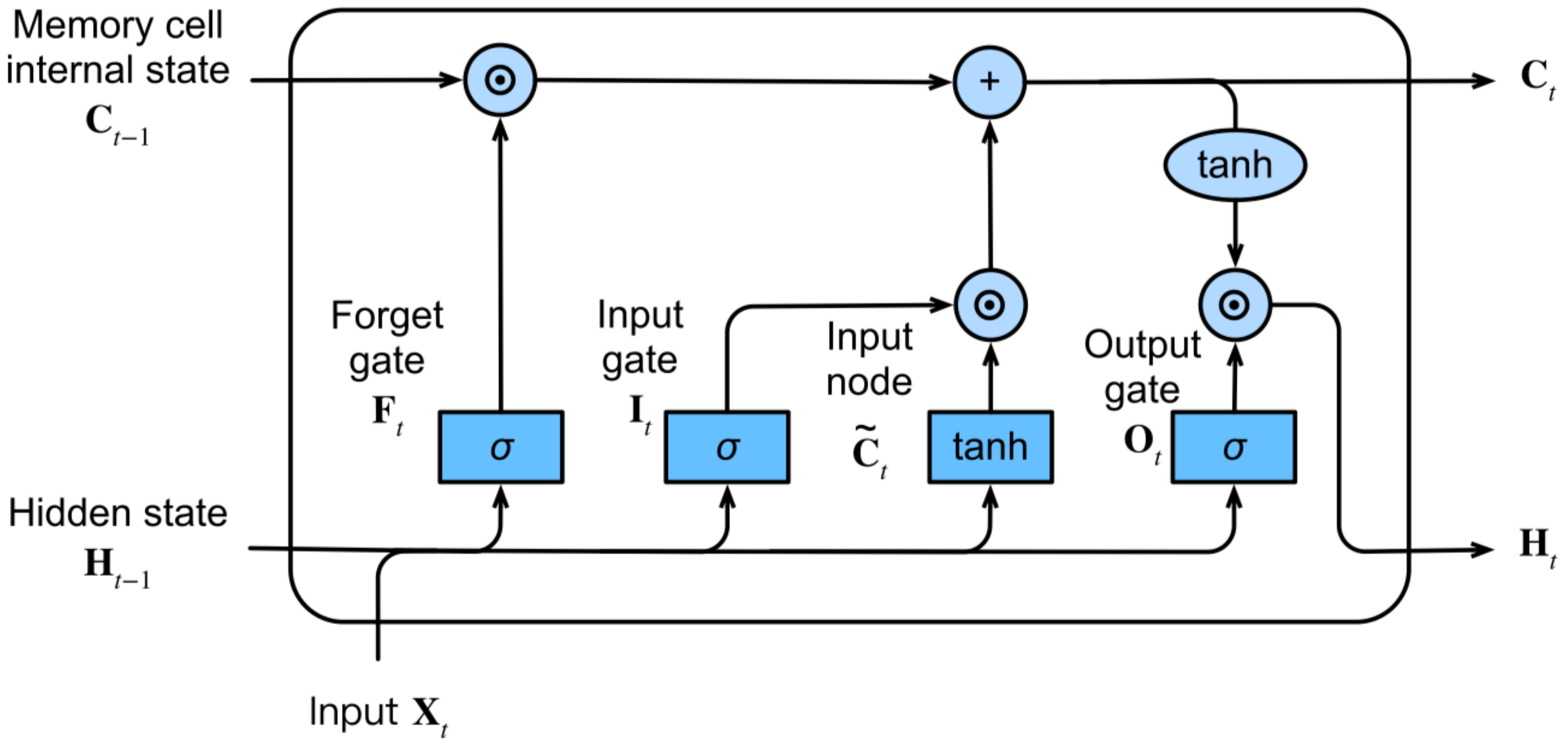
Vanishing / exploding gradient

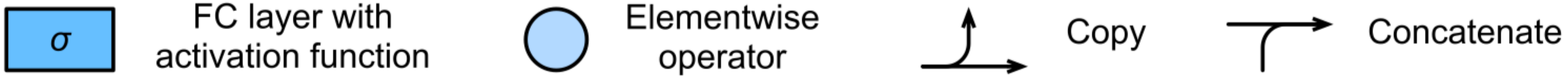
# Gated Memory Cell



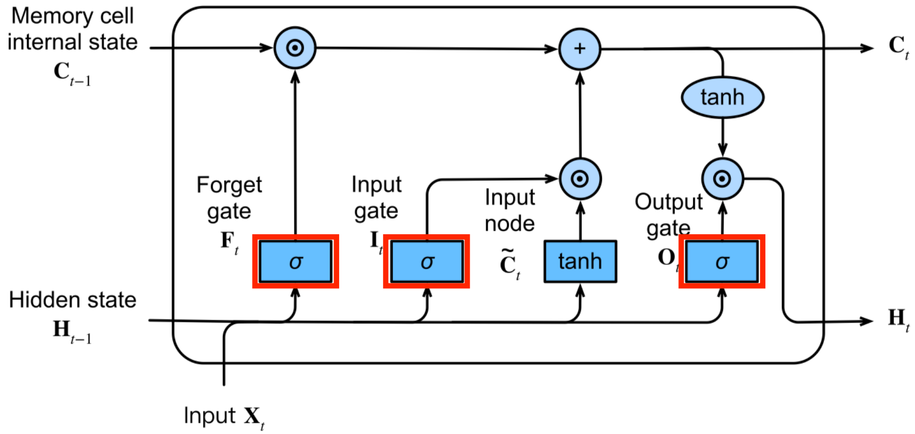
- A common problem with the vanilla RNNs is their inability to learn long-term dependencies (partially owing to vanishing and exploding gradients).
- To handle this, Hochreiter and Schmidhuber (1997) presented a long short-term memory (LSTM) model by replacing each ordinary recurrent node with a memory cell.
- The term "long short-term memory" comes from the following intuition.
    - Simple RNNS have <u>long-term memory</u> in the form of weight, which changes slowly during training.
    - They also have <u>short-term memory</u> in the form of ephemeral activations of nodes.
    - The LSTM model introduces an intermediate type of storage via the memory cell.

# LSTM Model


Memory cell
internal state
$\mathbf{C}_{t-1}$
Forget
gate
$\mathbf{F}_t$
$\sigma$
Input
gate
$\mathbf{I}_t$
$\sigma$
Input
node
$\tilde{\mathbf{C}}_t$
tanh
Output
gate
$\mathbf{O}_t$
$\sigma$
+
tanh
$\mathbf{C}_t$
Hidden state
$\mathbf{H}_{t-1}$
$\mathbf{H}_t$
Input $\mathbf{X}_t$


$\sigma$
FC layer with
activation function
Elementwise
operator
Copy
Concatenate

# Gated Hidden State



- The key distinction between vanilla RNNs and LSTMs is that the latter supports gaiting of the hidden state.
- Suppose that there are $h$ hidden units, the batch size is $n$, and the number of inputs is $d$. Thus, the input is $\mathbf{X}_t \in \mathbb{R}^{n\times d}$ and the hidden state of the previous time step is $\mathbf{H}_{t-1} \in \mathbb{R}^{n\times h}$.
- Then, we compute the input gate $\mathbf{I}_t \in \mathbb{R}^{n\times h}$, the forget gate $\mathbf{F}_t \in \mathbb{R}^{n\times h}$, and the output gate $\mathbf{O}_t \in \mathbb{R}^{n\times h}$ as follows:

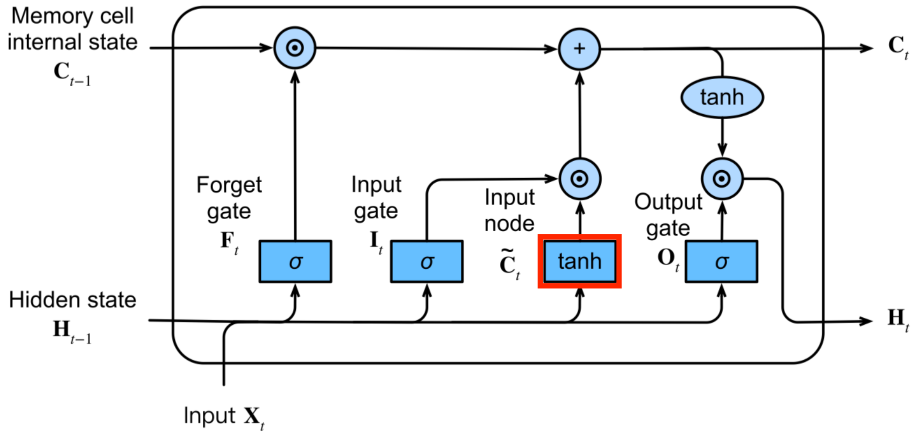
$$\mathbf{I}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xi} + \mathbf{H}_{t-1}\mathbf{W}_{hi} + \mathbf{b}_i)$$

$$\mathbf{F}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xf} + \mathbf{H}_{t-1}\mathbf{W}_{hf} + \mathbf{b}_f)$$

$$\mathbf{O}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xo} + \mathbf{H}_{t-1}\mathbf{W}_{ho} + \mathbf{b}_o)$$

where $\mathbf{W}$ and $\mathbf{b}$ are weights and biases and $\sigma(\cdot)$ is a sigmoid function to map the input value to $(0,1)$.

# Input Node

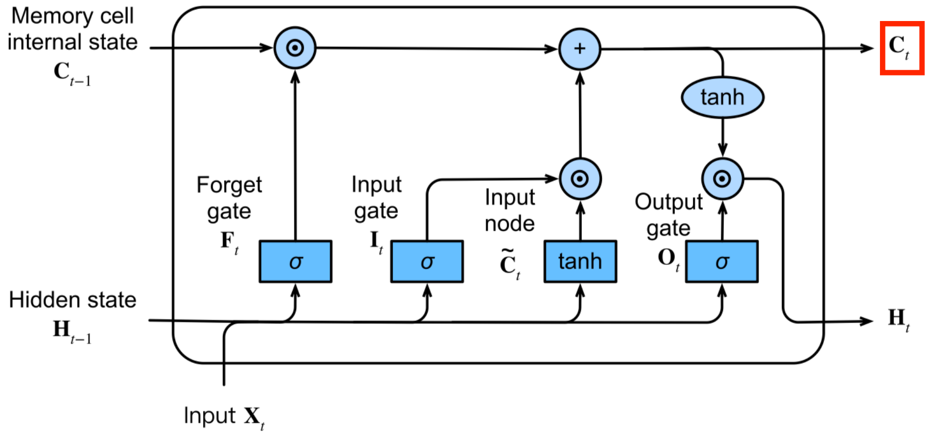


- Next, we design the memory cell.
- The input node is computed from the current input and previous cell state as follows:

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xc} + \mathbf{H}_{t-1}\mathbf{W}_{hc} + \mathbf{b}_c)$$

where $\mathbf{W}_{xc} \in \mathbb{R}^{d\times h}$ and $\mathbf{W}_{hc} \in \mathbb{R}^{h\times h}$ are weight parameters and $\mathbf{b}_c \in \mathbb{R}^{1\times h}$ is a bias parameter.

# Memory Cell



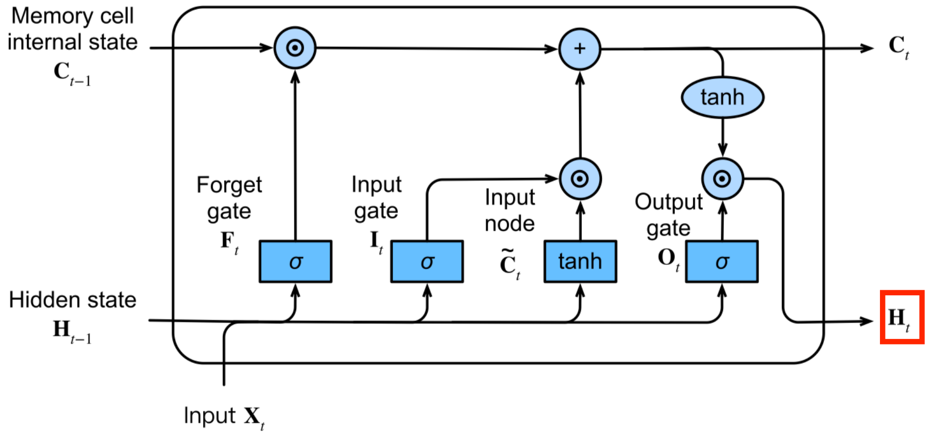
- The input gate $\mathbf{I}_t$ governs how much we take new data into account via $\tilde{\mathbf{C}}_t$ , and the forget gate $\mathbf{F}_t$ addresses how much of the old cell internal state $\mathbf{C}_{t-1} \in \mathbb{R}^{n\times h}$ we retain.
- Specifically, we arrive at the following update equation:

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t$$

where $\odot$ is the Hadamard (elementwise) product operator.

- If the forget gate is always $1$ and the input gate is always $0$, the memory cell internal state $\mathbf{C}_{t-1}$ will remain constant forever.

# Hidden State (output)



- Last, we need to define how to compute the output of the memory cell (i.e., the hidden state $\mathbf{H}_t \in \mathbb{R}^{n \times h}$)
- This is where the output gate comes into play:

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t)$$

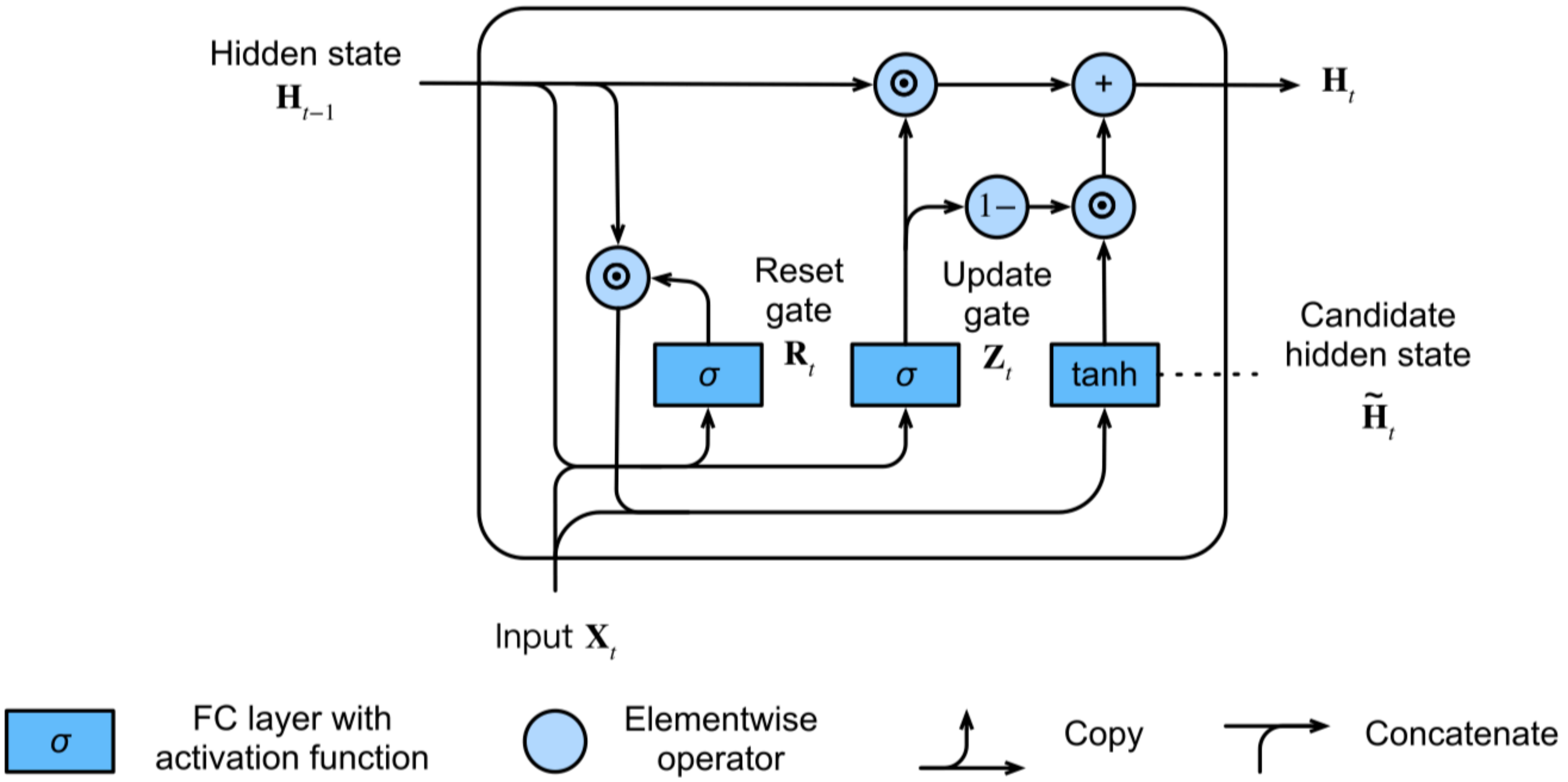
where the output gate values are always in the interval $(-1, +1)$.

# Gated Recurrent Units (GRU)

# Gated Recurrent Units

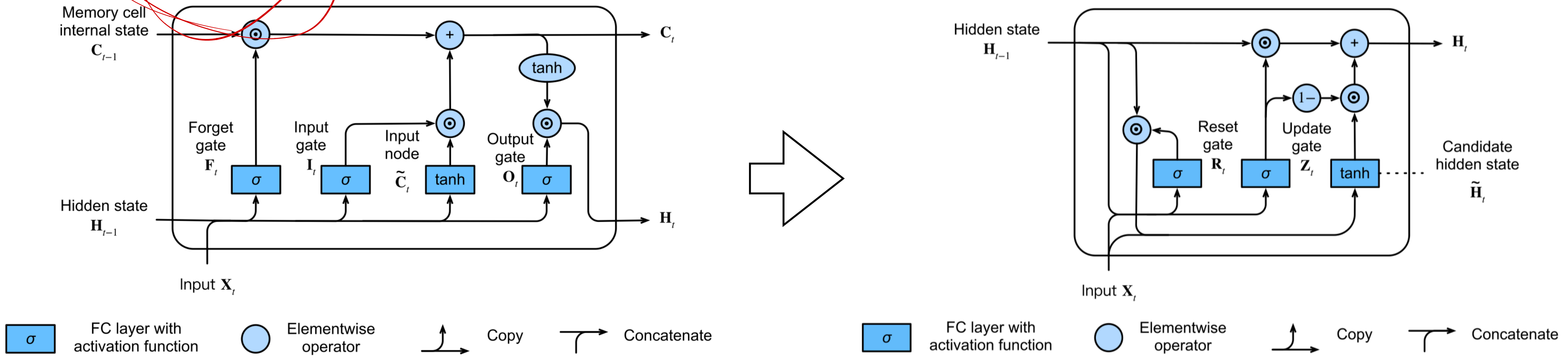
- As RNNs and particularly the LSTM architecture rapidly gained popularity during the 2010s, a number of papers began to experiment with simplified architectures in hopes of retaining the key idea of incorporating an internal state and gating mechanisms but with the aim of speeding up computation.
- The gated recurrent unit (Cho et al., 2014) offered a streamlined version of the LSTM memory cell that often achieves comparable performance but with the advantage of being faster to compute.

# GRU Model


Hidden state
$\mathbf{H}_{t-1}$
$\mathbf{H}_t$
Reset
gate
$\mathbf{R}_t$
Update
gate
$\mathbf{Z}_t$
Candidate
hidden state
$\tilde{\mathbf{H}}_t$
$\sigma$
$\sigma$
tanh
1−
+
Input $\mathbf{X}_t$
$\sigma$
FC layer with
activation function
Elementwise
operator
Copy
Concatenate

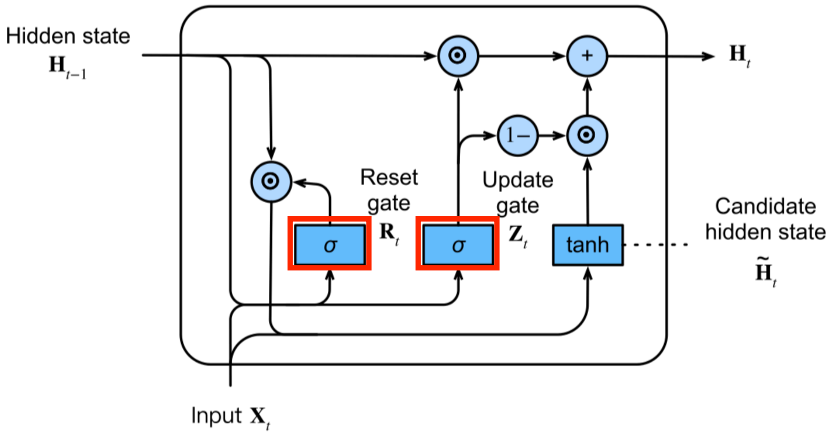
# LSTM to GRU

서술형



- In GRU, the LSTM's three gates (forget gate, input gate, and output gate) are replaced by two gates (reset gate and update gate).
  - Intuitively, the reset gate controls how much of the previous state we might still want to remember.
  - Likewise, the update gate would allow us to control how much of the new state is just a copy of the old one.
- Note that LSTMs maintain two streams (cell state $\mathbf{C}_t$ and hidden state $\mathbf{H}_t$), and the GRUs maintain a single stream (hidden state $\mathbf{H}_t$).

# Reset Gate and Update Gate



Hidden state $\mathbf{H}_{t-1}$
$\mathbf{H}_t$
1−
Reset gate $\mathbf{R}_t$
Update gate $\mathbf{Z}_t$
σ
σ
tanh
Candidate hidden state $\tilde{\mathbf{H}}_t$
Input $\mathbf{X}_t$

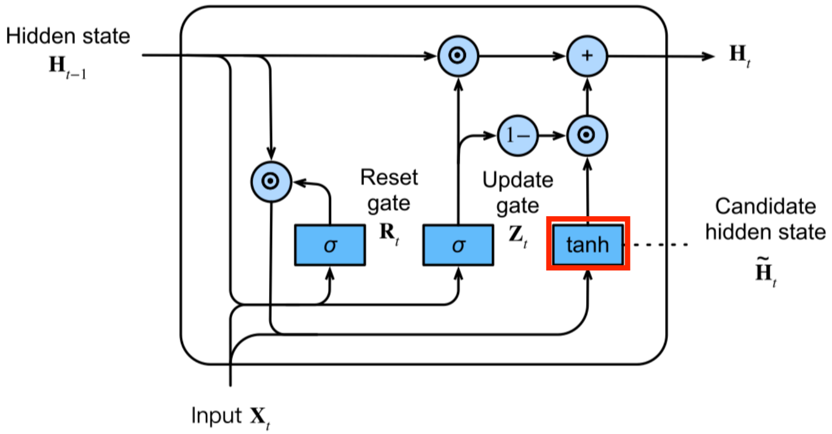
- For a given time step $t$, suppose that the input is a minibatch $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ and the hidden state of the previous time step is $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$.
- Then, the reset gate $\mathbf{R}_t \in \mathbb{R}^{n \times h}$ and the update gate $\mathbf{Z}_t \in \mathbb{R}^{n \times h}$ are computed as follows:

$$\mathbf{R}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xr} + \mathbf{H}_{t-1} \mathbf{W}_{hr} + \mathbf{b}_r)$$
$$\mathbf{Z}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xz} + \mathbf{H}_{t-1} \mathbf{W}_{hz} + \mathbf{b}_z)$$

  where $\mathbf{W}_{xr}, \mathbf{W}_{xz} \in \mathbb{R}^{d \times h}$ and $\mathbf{W}_{hr}, \mathbf{W}_{hz} \in \mathbb{R}^{h \times h}$ are weight parameters and $\mathbf{b}_r, \mathbf{b}_z \in \mathbb{R}^{1 \times h}$ are bias parameters.

- Both reset gate and update gate values are always in the interval $(0,1)$.

# Candidate Hidden State



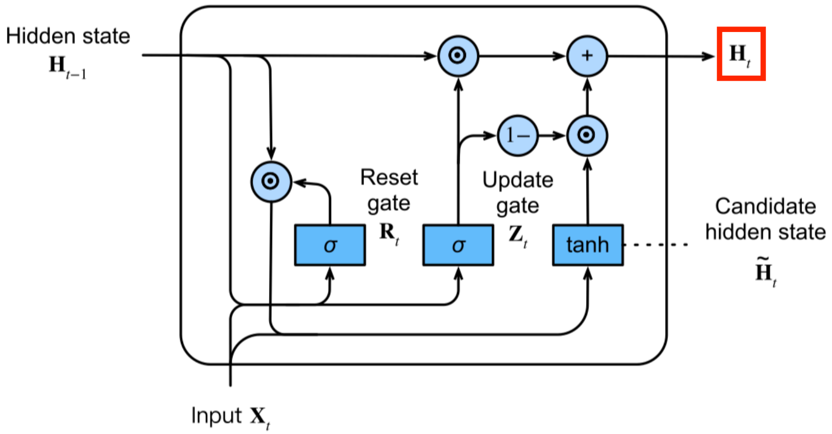
- Once we have computed the reset gate and update gate values, we compute the candidate hidden state $\tilde{\mathbf{H}}_t \in \mathbb{R}^{n\times h}$ at the time step $t$ as follows:

$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1})\mathbf{W}_{hh} + \mathbf{b}_h)$$

where $\mathbf{W}_{xh} \in \mathbb{R}^{d\times h}$ and $\mathbf{W}_{hh} \in \mathbb{R}^{h\times h}$ are weight parameters, $\mathbf{b}_h \in \mathbb{R}^{1\times h}$ is a bias parameter, and the symbol $\odot$ is the Hadamard (elementwise) product operator.

- The result is a candidate hidden state, since we still need to incorporate the action of the update gate.

# Hidden State



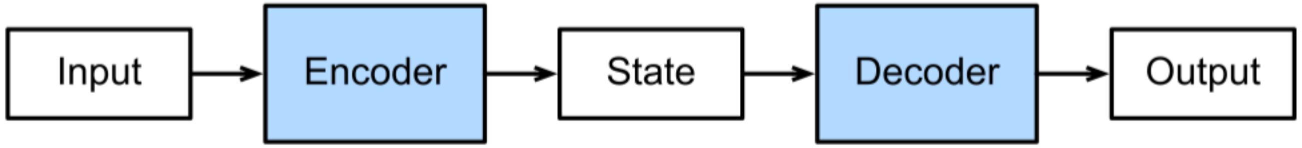
- Finally, we need to incorporate the effect of the update gate $\mathbf{Z}_t$.
- This determines the extent to which the new hidden state $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ matches the old state $\mathbf{H}_{t-1}$ versus how much it resembles the new candidate state $\tilde{\mathbf{H}}_t$.
- The update gate $\mathbf{Z}_t$ can be used for this purpose, leading to the final update equation:

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t.$$

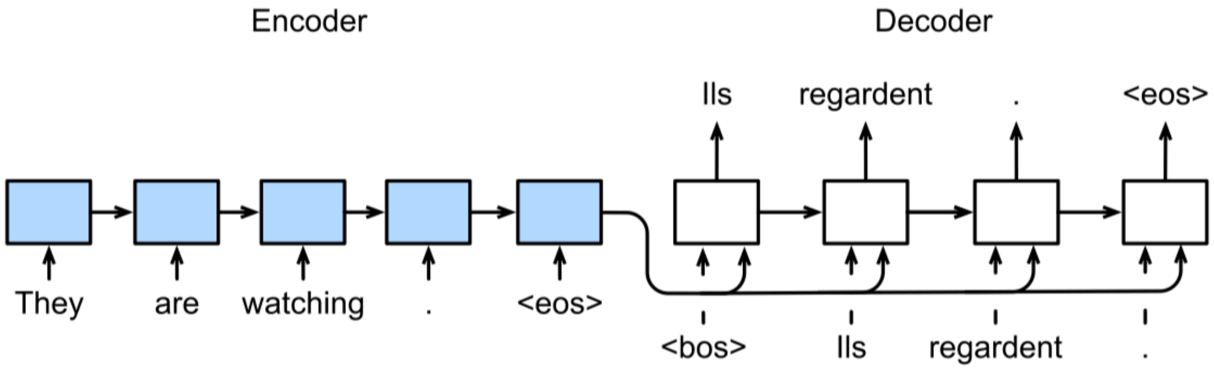
- In summary, GRUs have the following two distinguishing features:
    - Reset gates help capture short-term dependencies in sequences.
    - Update gates help capture long-term dependencies in sequences.

# The Encoder-Decoder Architecture

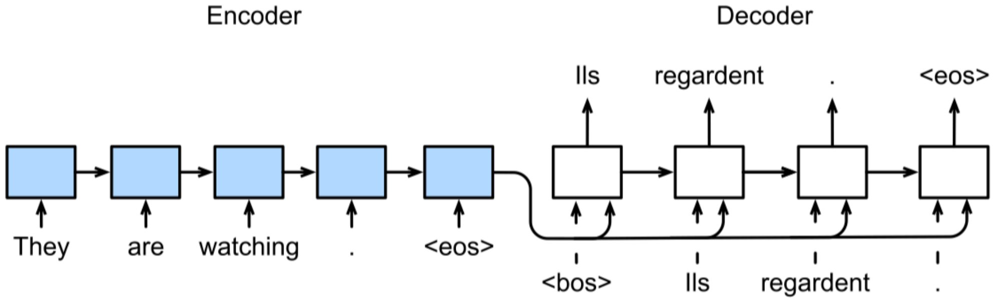
# The encoder-decoder architecture



- In general, seq2seq problems like machine translation, inputs and outputs are of varying lengths that are unaligned.
- The standard approach to handling this sort of data is to design an encode-decoder architecture consisting of two major components:
  - an encoder that takes a variable-length sequence as input, and
  - a decoder that acts as a conditional language model, taking in the encoded input and the leftwards context of the target sequence and predicting the subsequent token in the target sequence.



# Teacher Forcing



- While running the encoder on the input sequence is relatively straightforward, how to handle the input and output of the decoder requires more care.
- We will consider an encoder-decoder architecture for the machine translation task.
- Recall that the encoder transforms an input sequence of variable length into a fixed-shape context variable $c$.
- The decoder training matters:
  - Training without Teacher Forcing: At each time step, the RNN takes in an input token and predicts the next token. For the subsequent time step, the RNN uses its own prediction as an input.
  - Training with Teacher Forcing: Instead of using the RNN's own prediction as the next input, the actual ground truth is fed as the input for the next time step.

# Beam Search

# Test-time Prediction

- Previously, we introduced the encoder-decoder architectures and the standard techniques for training them end-to-end.
- However, when it comes to test-time prediction, we mentioned only the greedy strategy, where we select at each time step the token given the highest predicted probability.
- We first formalize this greedy search strategy and identify some problems that practitioners tend to run into.
- Subsequently, we compare this strategy with two alternatives: exhaustive search and beam search.
- Suppose that $\mathbb{Y}$ is the set of all tokens (i.e., vocabulary) and $T'$ is the maximum number of tokens of an output sequence.
- Then, our goal is to search for an ideal output from all $O(|\mathbb{Y}|^{T'})$ possible output sequences.

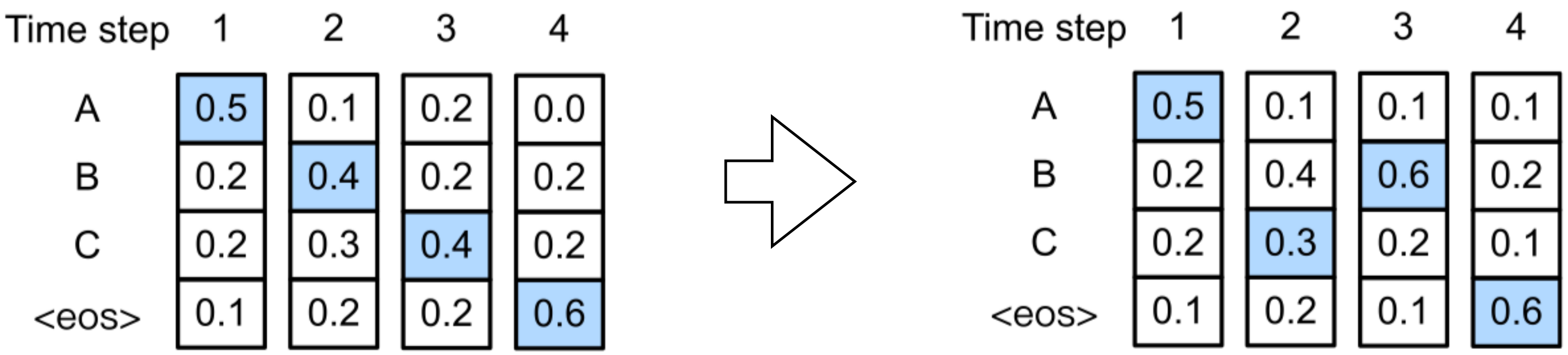
# Greedy Search

- Consider the simple greedy search strategy.
- At any time step $t'$, we simply select the token with the highest conditional probability from $\mathbb{Y}$:

$$y_{t'} = \underset{y \in \mathbb{Y}}{\arg\max}\, P(y \mid y_1, \ldots, y_{t'-1}, \mathbf{c})$$

where $\mathbf{c}$ is the context vector possibly computed from the encoder.

- Once the model output ⟨eos⟩ (or we reach the maximum length $T'$) the output sequence is completed.
- However, this does not always give us the most probable sequence.



| Time step | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| A | **0.5** | 0.1 | 0.2 | 0.0 |
| B | 0.2 | **0.4** | 0.2 | 0.2 |
| C | 0.2 | 0.3 | **0.4** | 0.2 |
| <eos> | 0.1 | 0.2 | 0.2 | **0.6** |

$0.5 \times 0.4 \times 0.4 \times 0.6 = 0.048$

| Time step | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| A | **0.5** | 0.1 | 0.1 | 0.1 |
| B | 0.2 | 0.4 | **0.6** | 0.2 |
| C | 0.2 | **0.3** | 0.2 | 0.1 |
| <eos> | 0.1 | 0.2 | 0.1 | **0.6** |

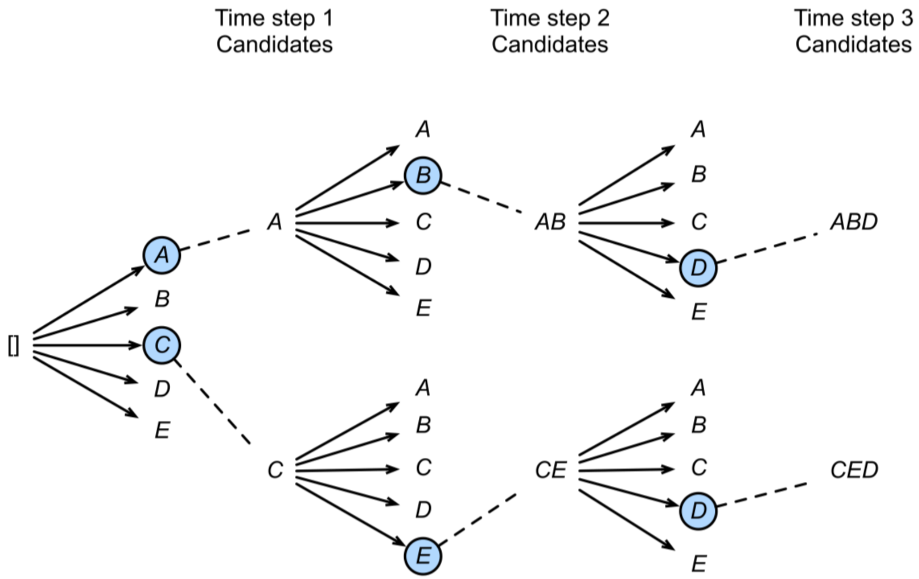
$0.5 \times 0.3 \times 0.6 \times 0.6 = 0.054$

# Exhaustive Search

- If the goal is to obtain the most likely sequence, we may consider using an exhaustive search: exhaustively enumerate all the possible output sequences with their conditional probabilities and then output the one that scores the highest.
- While this would certainly give us what we desire, it would come at a prohibitive computational cost of $O(|\mathbb{Y}|^{T'})$.
    - When $|\mathbb{Y}| = 10{,}000$ and $T' = 10$, we will need to evaluate $10{,}000^{10} = 10^{40}$ sequences.
- On the other hand, the computational cost of the greedy search is $O(|\mathbb{Y}|T')$.
    - When $|\mathbb{Y}| = 10{,}000$ and $T' = 10$, we only need to evaluate $10^5$ sequences.

# Beam Search

- The most straightforward version of beam search is characterized by a single hyperparameter, the beam size $k$.
  - At the time step $1$, we select the $k$ tokens with the highest predicted probabilities.
  - Each of them will be the first token of $k$ candidate output sequences, respectively.
  - As each subsequent time step, based on the $k$ candidate output sequences, we continue to select $k$ candidate output sequences with the highest predicted probabilities.



beam size $k = 2$, maximum length of an output sequence: $3$

서울과학기술대학교

SEOUL NATIONAL UNIV. OF SCIENCE & TECHNOLOGY

1910