CS190C Lec1

Word Embedding & Language Modeling

Overview

- How to embed words
- What is language modeling
- Some naive and early language models
  - N-grams
  - RNN
  - LSTM

PART1：How to embed words?

Problem: Computer cannot understand natural language…

- We should try to convert them, such as words, into digital form.
- How to represent words formally?

A naive idea

- Like a dictionary, each word has its position in it
- Can we represent a word using a certain number?
  - That is: each word has its "one-to-one mapping value"
- Just like this:

Word	I	love	Natural	Language	Processing	" "	.
Value	0	1	2	3	4	5	6

`"I love Natural Language Processing."` $\Rightarrow$ `0,5,1,5,2,5,3,5,4,6`

Mapping has its defect

- Can we have a method to infer this word's part of speech or meaning, just based on the mapping value?
- For example: If you only receive a string of numbers: `0,5,1,5,2,5,3,5,4,6`, can you successfully infer the meaning of this sentence?
- Mathematically, one-to-one mapping can be understood as: **The semantics of different words constitute a one-dimensional vector space!**
  - Each word is a 1-dim vector in this 1-dim space
  - Using 1-dim vector space to represent natural language is obviously not enough!

Enlarge the dimension?

- If we use 2-dim space to represent some words?
    - x-aixs represent "area", y-axis represent "population"
    - Can we use 2-dim vectors to approximatly represent: `China` `India` `Canada` `Luxembourg`? (Plot a graph of it)
- Similarly, the higher the dimension, the richer the semantics it can represent.
- For example, GPT-3 uses 2048-dim to represent words.

What is word embedding?

So far, we know that:

- We can use n-dim vector to represent the meaning of the word formally.
- Each dimension can be understood as a semantic feature, much like a spatial dimension in linear algebra.

We call these word vectors **Word Embedding**.

- Encoding words with appropriate word embeddings is a key concept in natural language processing.
- In later lectures, we will introduce some important encoder models.

PART2: What is language modeling?

Look at an LLM

What is NLP?

Qwen3-Max

NLP stands for **Natural Language Processing**. It is a subfield of **artificial intelligence (AI)** and **computational linguistics** that focuses on enabling computers to understand, interpret, generate, and interact with **human language** in a meaningful and useful way.

Key Goals of NLP:

1. Understanding: Extract meaning from human language (e.g., sentiment analysis, intent recognition).
2. Generation: Produce human-like text (e.g., chatbots, automated reports).
3. Translation: Convert text or speech from one language to another (e.g., Google Translate).
4. Interaction: Enable seamless communication between humans and machines using natural language (e.g., voice assistants like Siri or Alexa).

Common NLP Tasks:

- Tokenization: Splitting text into words, phrases, or symbols.
- Part-of-Speech Tagging: Identifying nouns, verbs, adjectives, etc.
- Named Entity Recognition (NER): Detecting entities like people, organizations, dates.
- Sentiment Analysis: Determining if a sentence expresses positive, negative, or neutral emotion.
- Machine Translation: Translating text between languages.
- Text Summarization: Condensing long documents into shorter versions.
- Question Answering: Systems that answer questions posed in natural language.
- Speech Recognition & Synthesis: Converting speech to text and vice versa.

- Input a "prompt" (a string of words).
- Give an answer based on the prompt.

How does it generate words to finally form an answer?

- Similar to how humans speak, each word is generated logically based on the previous words.
- That is: based on past words, and generate new words.
- This is called **Language Modeling**!

Tips: Difference between `Language Model` and `Language Modeling`

- Language Model: a tool to generate certain answer based on prompts.
- Language Modeling: the methods to generate "new words" based on "old words".

So, what's the "**methods**"?

General methods

There might be several suitable options for the new generated word...

- But different word may have different levels of suitability.
- We can try to model "levels of suitability" into probabilistic distribution.
- That is, to find a way to calculate the probability of generating a certain word as the "next word," given the "previous words."

Language Modeling Methods...

There are different models, using different certain methods to calculate the probabilistic distribution.

- N-grams
- RNN, LSTM (An optimized architecture based on RNN)
- Transformer
- GPT, BERT .etc (Based on Transformer)

PART3： N-grams

Another naive idea

Simply consider:

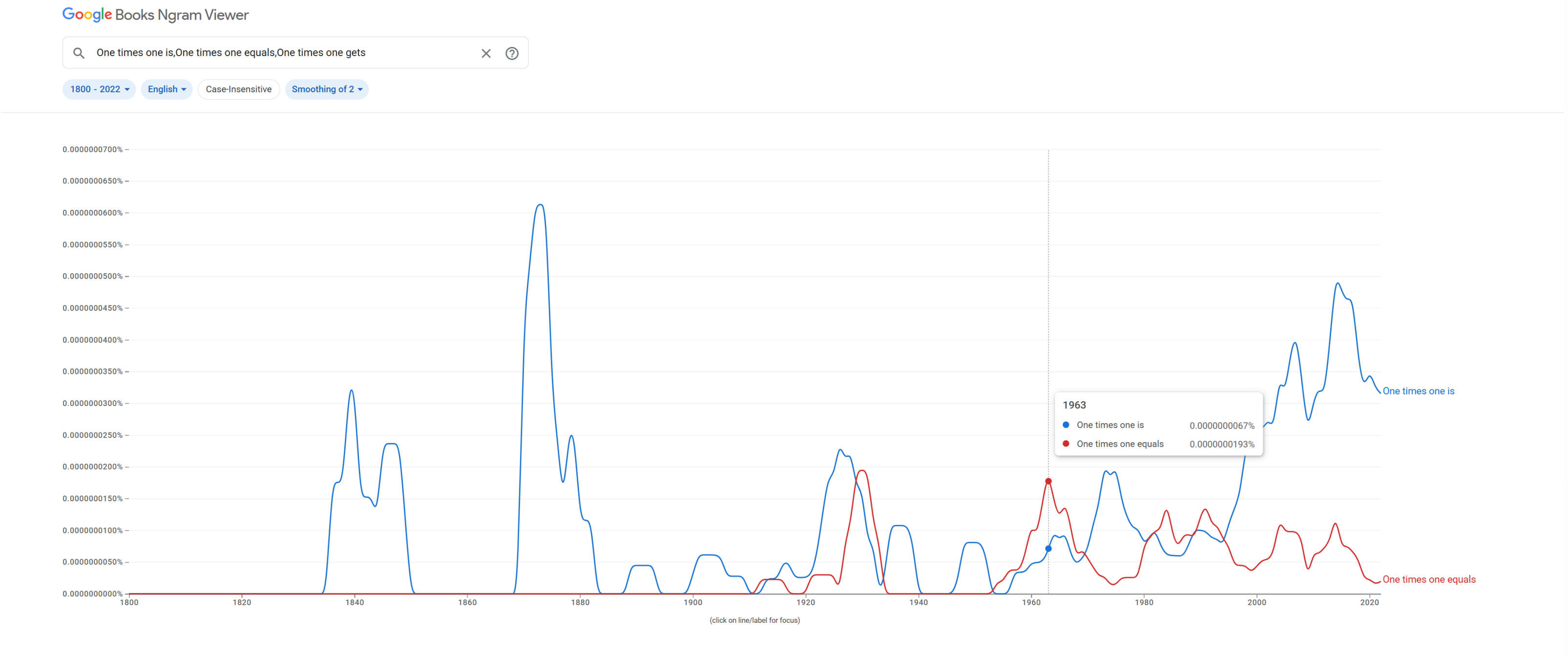
- If we just focus on a fix window of previous words to generate a new word, it can still work at most of time.

For example:

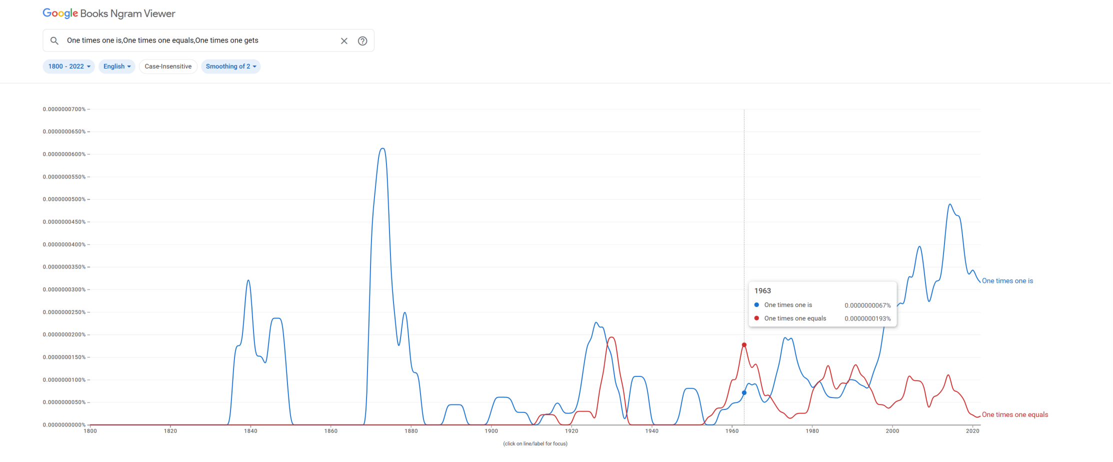
- Look at only the 3 previous words to generate the next word.
- We call this a `4-grams`, meaning our fixed window (3 previous words + 1 predicted word) has a total length of 4 words.
- Prompt: "Let's calculate simple multiplication! One times one..."
- Previous words focused: `One times one`

A naive 4-grams

- Previous words focused: `One times one`
- Generate new word directly use statistical laws!
- We can use https://books.google.com/ngrams/



A naive 4-grams



- Try all words in vocabulary
- We can also get $P(\text{new}, \text{old})$ (marginal probability of 4-grams)
- What we want to model is $P(\text{new} \mid \text{old}) \propto P(\text{new}, \text{old})$

- Suppose we've modeled $P(w_i \mid \text{old})$.
- We choose $\arg\max_{w_i} P(w_i \mid \text{old})$ or sample words according to the distribution.

- Suppose we decide `is` to be the new word.
- Sentence now: `One times one is`
- Next turn: use `times one is` to generate a new word, and so on.

Pros and cons?

- Quite simple, and requires very little calculation (especially on training)
- If we actually use N-grams for text generation, what will happen?

Generate text?

today the price of gold per ton , while production of shoe lasts and shoe industry , the bank intervened just after it considered and rejected an imf demand to rebuild depleted european stocks , sept 30 end primary 76 cts a share .

Surprisingly grammatical!

…but **incoherent.** We need to consider more than three words at a time if we want to model language well.

But increasing n worsens sparsity problem, and increases model size…

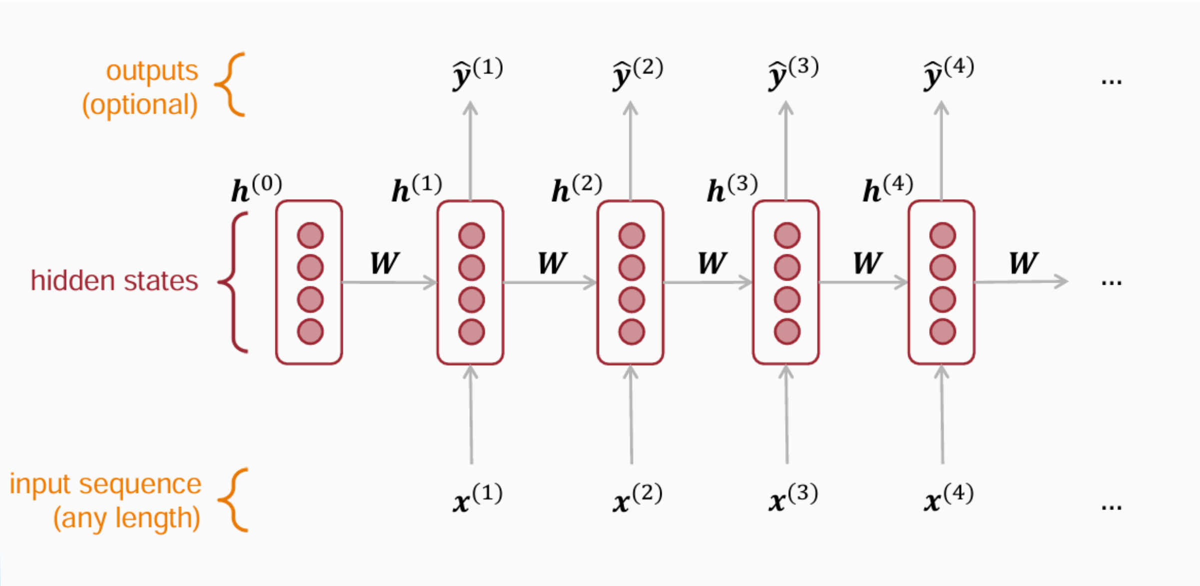
- For **grammar**…
  - Local part-of-speech collocations (e.g., subject-predicate, verb-object) follow statistical rules successfully.
- But for **semantics**…
  - Missing of global context.
  - One mistake will cause an accumulation of subsequent errors.

PART4: Recurrent Nerual Network (RNN)

How to make use of global context?

By Encoding:

- We've already encoded words as vectors.
- Can we encode entire sentences or texts as vectors too?
- If we get an encoded vector of text, we can generate new words with an "understanding" of the text!



How to encode words and text?

- For words:
  - At first, each word is still encoded with single number.
  - We aim to learn a matrix $W_e \in \mathbb{R}^{d \times |V|}$, where each column is the embedding of the i-th word.
  - Then for i-th word, we can get its embedding by computing $W_e e$,
    - e is the **one-hot vector** of this word (a vector of all zeros except for a 1 at the i-th position)
  - $\Rightarrow$ Therefore, $W_e e$ perfectly extracts the i-th column of W_e

How to encode words and text?

- For the text:
  - Similar to reading word by word, our understanding of the text becomes more substantial as we process each new word.
  - When a new word is read, the "understanding" of the text is a mixture of: the **previous understanding** of the text, and the **information** of the new **new word**.
  - That is: the text embedding at the current time step should combine the **previous text embedding** and the **embedding of the new word**.

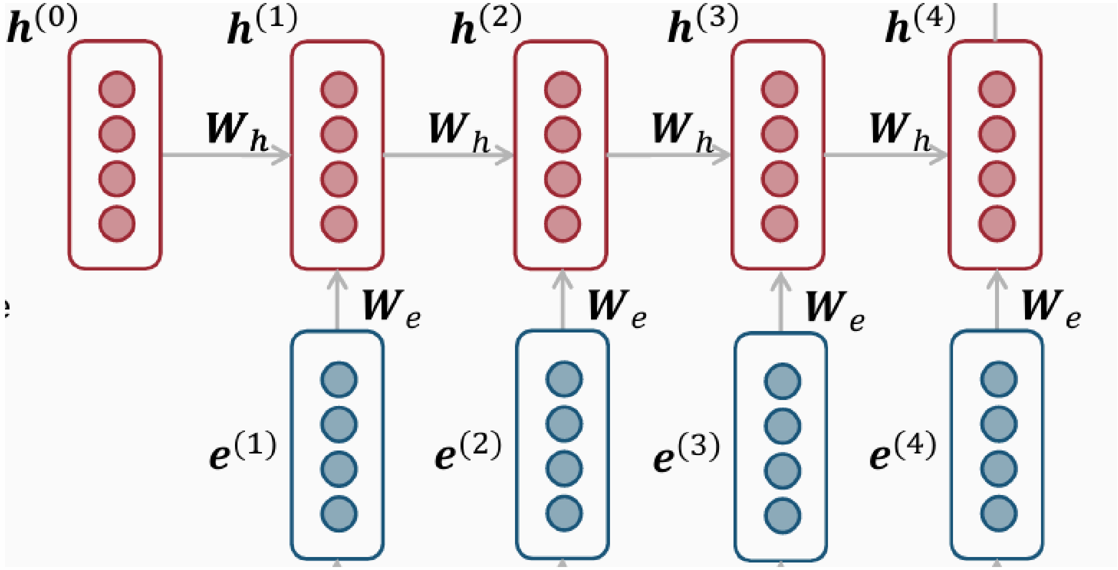
How to encode words and text?

Let embedding of text in step-t be h^t:

- Previous Context: $W_h h^{t-1}$.
  - Learn W_h to properly inherit previous information.
- Current Input: $W_e e^t$
  - Learn W_e to efficiently extract new word's information.
- Combination: $W_h h^{t-1} + W_e e^t + b_1$. ($b_1$ is the optional bias term)
- Apply a nonlinear activation (usually use sigmoid)
  - Introduce non-linearity to model complex patterns.

How to encode words and text?

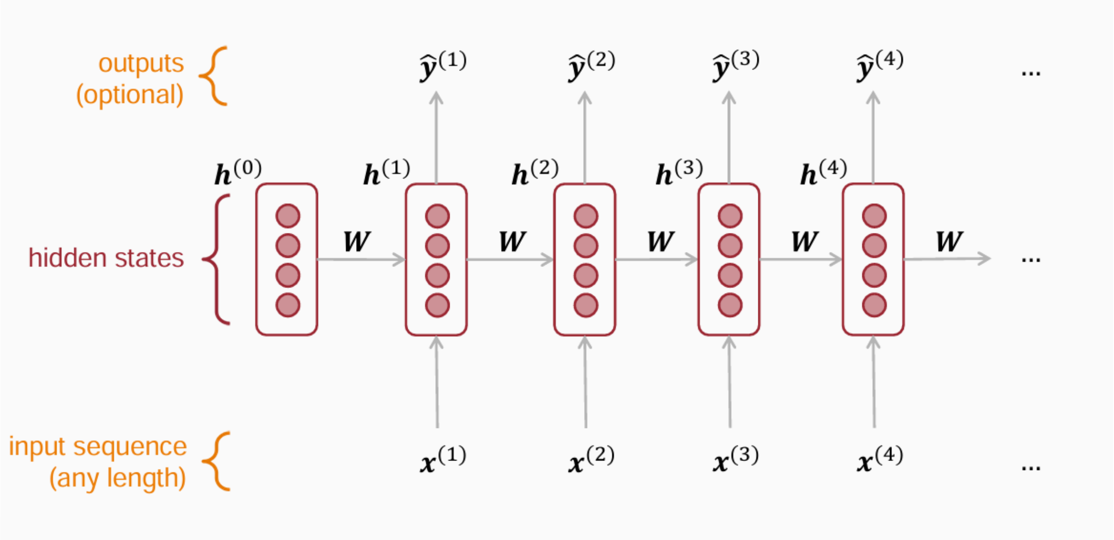
- $h^t = \sigma(W_h h^{t-1} + W_e e^t + b_1)$
- Parameters to train:
  - W_e
  - W_h
  - b_1



How to language modeling?

Just use a linear activation to h^t, generate the probability distribution of the new words!

- $\hat{y}^t = \mathrm{Softmax}(Uh^t + b_2)$
- Parameters to train:
  - U: the linear activation matrix
  - b_2: the optional bias term



Implement a simple RNN

https://github.com/kuangpenghao/NLP_models_by_hand/blob/main/toy_RNN.py

- 3 training sentences
- Train RNN
- Input sentence except the last word
- Hope to output the correct word

```
epoch:0200,loss:0.715562
epoch:0400,loss:0.200380
epoch:0600,loss:0.097146
epoch:0800,loss:0.060034
epoch:1000,loss:0.064022
input:['i', 'like'],output:dog
input:['i', 'love'],output:coffee
input:['i', 'hate'],output:milk
```

Implement a simple RNN

- main function

```
if __name__=="__main__":

    config=TextRNNConfig()
    model=TextRNN(config)

    trainer=TextRNNTrainer(config,model)
    predictor=TextRNNPredictor(config,model)

    trainer.train()

    test_sentences=["i like dog", "i love coffee", "i hate milk"]
    for sentence in test_sentences:
        sentence=sentence.split()
        input_sentence=sentence[:-1]
        predicted_word=predictor.predict(input_sentence)
        print(f"input:{input_sentence},output:{predicted_word}")
```

Implement a simple RNN

```
class TextRNNConfig:
    def __init__(self):
        self.n_hidden=5
        self.sentences=["i like dog", "i love coffee", "i hate milk"]

        word_list=' '.join(self.sentences).split()
        word_list=list(set(word_list))
        self.word_dict={w:i for i,w in enumerate(word_list)}
        self.number_dict={i:w for i,w in enumerate(word_list)}

        self.n_class=len(word_list)

        self.batch_size=2
        self.learning_rate=0.001
        self.epochs=1000
        self.interval=200
```

Implement a simple RNN

```
class TextRNN(nn.Module):
    def __init__(self,config):
        super(TextRNN,self).__init__()
        self.config=config
        self.rnn=nn.RNN(self.config.n_class,self.config.n_hidden)
        self.W=nn.Linear(self.config.n_hidden,self.config.n_class,bias=True)

    def forward(self,X):
        X=X.transpose(0,1)
        ori_hidden=torch.zeros(1,X.shape[1],self.config.n_hidden)
        outputs,last_hidden=self.rnn(X,ori_hidden)
        output=outputs[-1]
        output=self.W(output)
        return output
```

Implement a simple RNN

```
class TextRNNDataset(Dataset):
    def __init__(self,config):
        super(TextRNNDataset,self).__init__()
        self.config=config

    def __len__(self):
        return len(self.config.sentences)

    def __getitem__(self,idx):
        sentence=self.config.sentences[idx]
        words=sentence.split()
        words=[self.config.word_dict[i] for i in words]

        input_idx=words[:-1]
        one_hot=np.eye(self.config.n_class)[input_idx]
        input_one_hot=torch.tensor(one_hot,dtype=torch.float32)

        target_idx=words[-1]
        target_idx=torch.tensor(target_idx,dtype=torch.int64)

        return input_one_hot,target_idx
```

Implement a simple RNN

```
class TextRNNTrainer:
    def __init__(self,config,model):
        self.config=config
        self.model=model
        self.loss_function=nn.CrossEntropyLoss()
        self.optimizer=torch.optim.SGD(model.parameters(),lr=self.config.learning_rate,momentum=0.9)

        self.datagetter=TextRNNDataset(config)
        self.dataloadder=DataLoader(self.datagetter,batch_size=self.config.batch_size,shuffle=True)

    def train(self):
        for epoch in range(self.config.epochs):
            for input_batch,target_batch in self.dataloadder:
                self.optimizer.zero_grad()
                output=self.model(input_batch)
                loss=self.loss_function(output,target_batch)
                loss.backward()
                self.optimizer.step()
            if (epoch+1)%self.config.interval==0:
                print(f"epoch:{epoch+1:04d},loss:{loss.item():.6f}")
```

Implement a simple RNN

```python
class TextRNNPredictor:
    def __init__(self,config,model):
        self.config=config
        self.model=model

    def predict(self,input_sentence):
        with torch.no_grad():
            word=[self.config.word_dict[i] for i in input_sentence]
            word=np.eye(self.config.n_class)[word]
            word=torch.tensor(word,dtype=torch.float32).unsqueeze(0)

            output=self.model(word)
            outcome=output.max(1,keepdim=False)[1]
            predicted_word=self.config.number_dict[outcome.item()]

            return predicted_word
```

Pros and cons?

- Actually, it can make use of the information of the global text.
- When the text is extremely long, step t is very large?
    - Gradient vanishing or exploding

Update parameters (W_h)

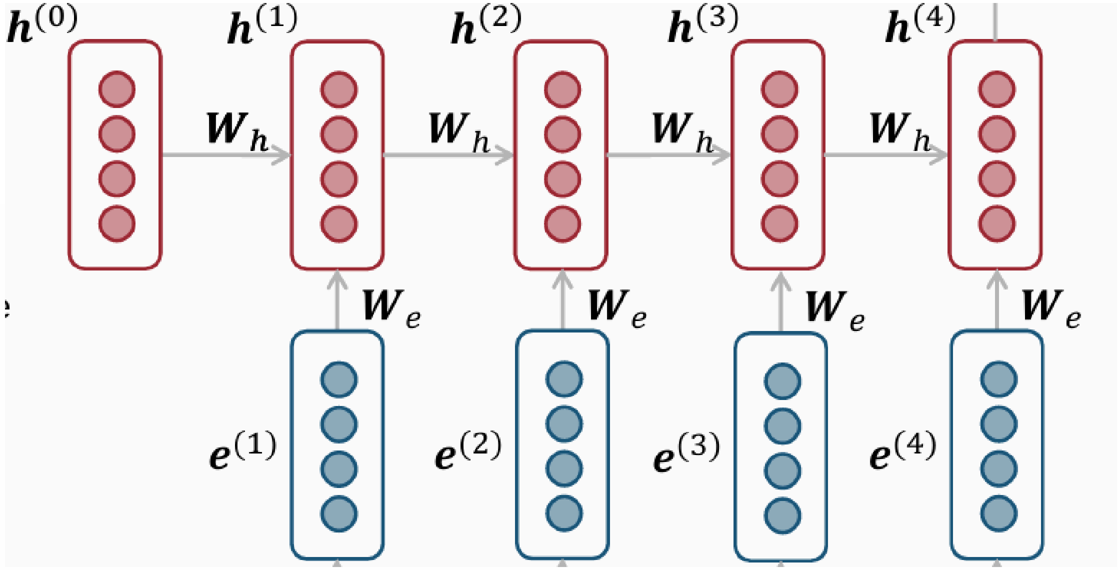
$$\frac{\partial L_T}{\partial W_h} = \sum_{t=1}^{T} \frac{\partial L_T}{\partial h^t} \frac{\partial h^t}{\partial W_h}$$

$$\frac{\partial L_T}{\partial h^t} = \frac{\partial L_T}{\partial h^T} \frac{\partial h^T}{\partial h^{T-1}} \cdots \frac{\partial h^{t+1}}{\partial h^t}$$

$$\frac{\partial h^{t+1}}{\partial h^t} = \frac{\partial \sigma(W_h h^t + W_e e^{t+1} + b_1)}{\partial h^t}$$

$$= \sigma'(W_h h^t + W_e e^{t+1} + b_1) W_h$$

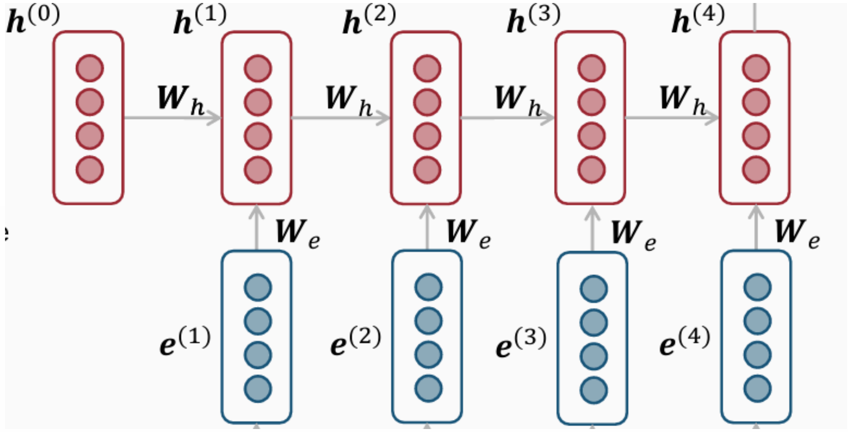
$$\Longrightarrow \frac{\partial L_T}{\partial h^t} = \frac{\partial L_T}{\partial h^T} \prod_{k=t+1}^{T} [\sigma'(z_k) W_h]$$



- We usually use sigmoid function as σ. Then $\sigma'(z) = \sigma(z)(1 - \sigma(z)) \in (0, 0.25]$

Update parameters (W_h)

- $$\frac{\partial L_T}{\partial h^t} = \frac{\partial L_T}{\partial h^T} \prod_{k=t+1}^{T} [\sigma'(z_k) W_h]$$
- If $\|W_h\| < 1$, each term must < 1
  $\Rightarrow \frac{\partial L_T}{\partial h^t} \to 0$, which may leads to $\frac{\partial L_T}{\partial W_h} \to 0$
  - **Vanishing gradient**
  - Causes inability to capture long-term dependencies.
- Likewise, if $\|W_h\|$ is large enough, $\frac{\partial L_T}{\partial W_h} \to \infty$
  - **Exploding gradient**
  - Causes bad updates and instability in training.



PART5: Long Short-Term Memory (LSTM)

Long Short-Term Memory RNNs (LSTMs)

- We have a sequence of inputs $\boldsymbol{x}^{(t)}$, and we will compute a sequence of hidden states $\boldsymbol{h}^{(t)}$ and cell states $\boldsymbol{c}^{(t)}$.
- On timestep t:

Sigmoid function: all gate values are between 0 and 1

Forget gate: controls what is kept vs forgotten, from previous cell state

$$\boldsymbol{f}^{(t)} = \sigma\left(\boldsymbol{W}_f \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_f \boldsymbol{x}^{(t)} + \boldsymbol{b}_f\right)$$

Input gate: controls what parts of the new cell content are written to cell

$$\boldsymbol{i}^{(t)} = \sigma\left(\boldsymbol{W}_i \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_i \boldsymbol{x}^{(t)} + \boldsymbol{b}_i\right)$$

Output gate: controls what parts of cell are output to hidden state

$$\boldsymbol{o}^{(t)} = \sigma\left(\boldsymbol{W}_o \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_o \boldsymbol{x}^{(t)} + \boldsymbol{b}_o\right)$$

New cell content: this is the new content to be written to the cell

$$\tilde{\boldsymbol{c}}^{(t)} = \tanh\left(\boldsymbol{W}_c \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_c \boldsymbol{x}^{(t)} + \boldsymbol{b}_c\right)$$

Cell state: erase ("forget") some content from last cell state, and write ("input") some new cell content

$$\boldsymbol{c}^{(t)} = \boldsymbol{f}^{(t)} \circ \boldsymbol{c}^{(t-1)} + \boldsymbol{i}^{(t)} \circ \tilde{\boldsymbol{c}}^{(t)}$$

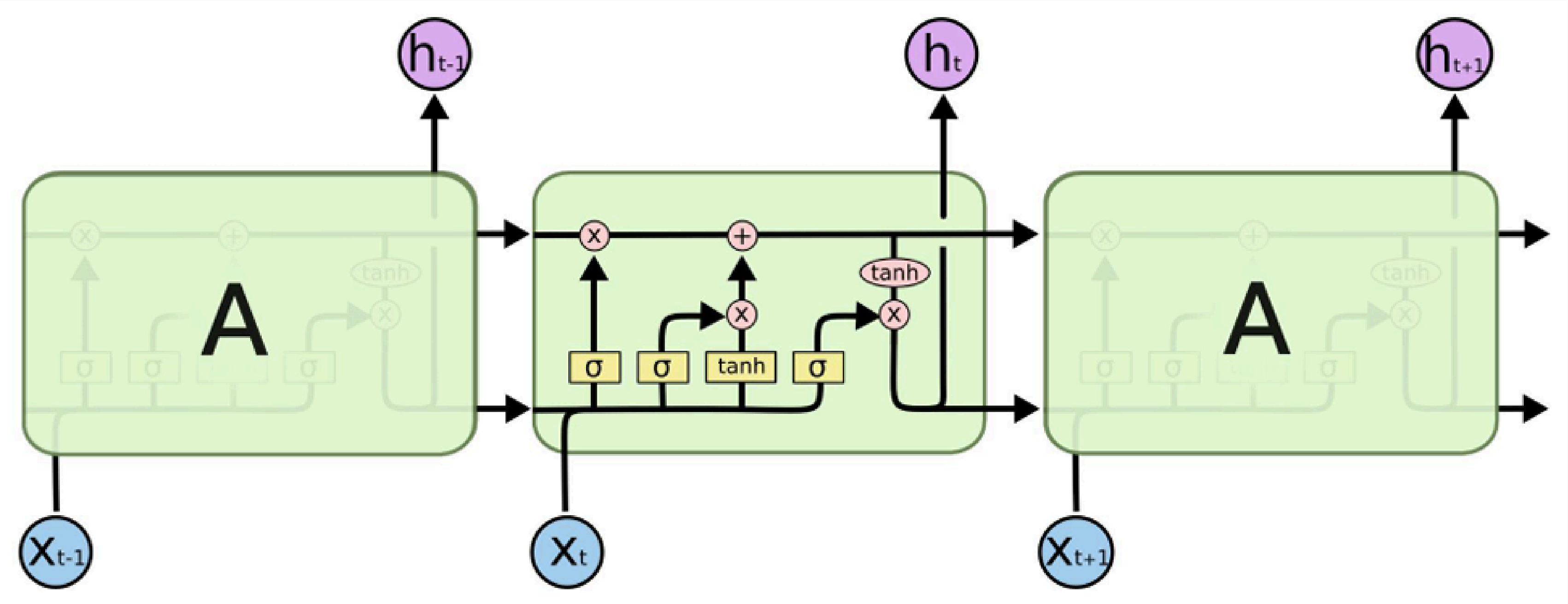
Hidden state: read ("output") some content from the cell

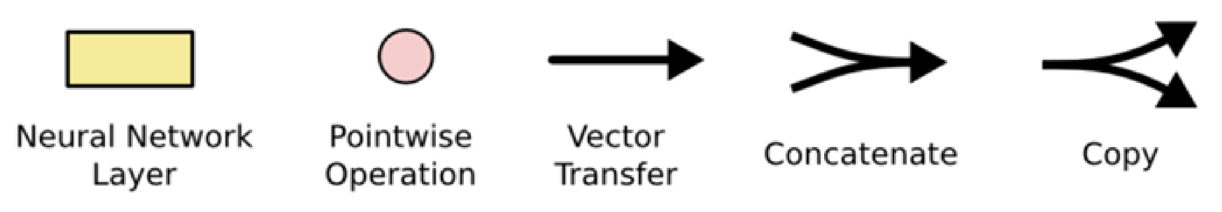
$$\boldsymbol{h}^{(t)} = \boldsymbol{o}^{(t)} \circ \tanh \boldsymbol{c}^{(t)}$$

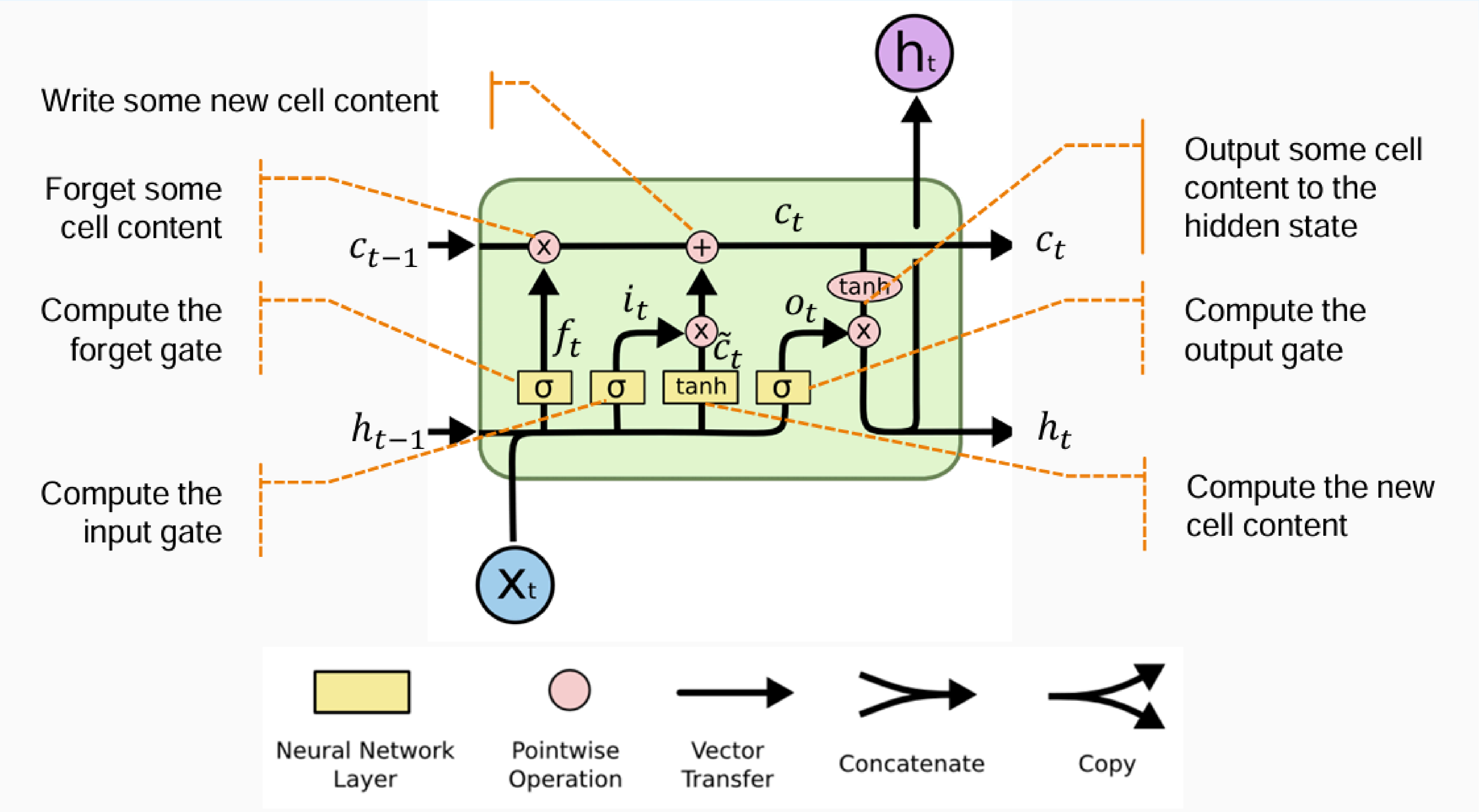
All these are vectors of same length n

Gates are applied using element-wise product

- You can think of the LSTM equations visually like this:






Write some new cell content
Forget some cell content
Compute the forget gate
Compute the input gate
Output some cell content to the hidden state
Compute the output gate
Compute the new cell content
c_{t-1}
h_{t-1}
c_t
h_t
f_t
i_t
$\tilde{c}_t$
o_t
σ
tanh
h_t
X_t
Neural Network Layer
Pointwise Operation
Vector Transfer
Concatenate
Copy

Think About It

- *N-gram models may not work well because of a lack of global context. Can we just make N very large to solve this problem?*

- **Sparsity**. You can easily find "gradient" in the corpus but hardly find a long phrase like "gradient may explode for RNN with large timesteps"
- **State space**. You can estimate the massive scale based on the number of possible combinations.