Bios 740- Chapter 4.. Sequence Modeling: RNNs, LSTM, and GRU

Acknowledgement: Thanks to Mr. Shuai Huang for preparing some of the slides! I copied some pictures from the lecture presentations of StanfordCS224n. I also use some materials generated by chapgpt.

Content

Content

Motivation

Recurrent Neural Networks (RNNs) are motivated by their ability to address challenges inherent in sequential data.

Weather forecasting

Stock market trends

Autocomplete for texting

Genetic sequencing

Many real-world datasets are inherently sequential, where **the order of data points is crucial**.

- ❖ **Time series:** Stock prices, weather forecasts, and sensor data require capturing patterns over time.
- ❖ **Text:** The meaning of a sentence depends on word order (``The cat chased the dog" vs. ``The dog chased the cat").
- ❖ **Speech:** Phonemes and intonation must be processed sequentially to understand spoken language.
- ❖ **Video:** Frames in a video sequence have temporal relationships that determine the flow of events.

Speech recognition

Video frame prediction

Music composition

…

Sequences

A **sequence** is an **ordered list of elements**, where the order of the elements matters. They are fundamentally different from unordered data because each element is **dependent** or **influenced** by the previous elements.

Examples:

- Letters (words)
- Words (sentences)
- Sentences (documents)
- Frames (video)
- Amino-acids (genetic code)
- fMRI/ECG signals

"Hello, how are you?" (chatbot input).

e.g., A **security camera** capturing a person walking.

e.g., **"ACGTAGCTAGT"** represents a biological sequence.

Why Are Sequences Important?

Unlike independent data points, sequences contain **temporal or contextual dependencies**:

- **Future values depend on past values** (e.g., predicting tomorrow's weather).
- **Words in a sentence rely on context** (e.g., in "bank deposit" vs. "river bank").
- **Biological sequences determine genetic functions**.

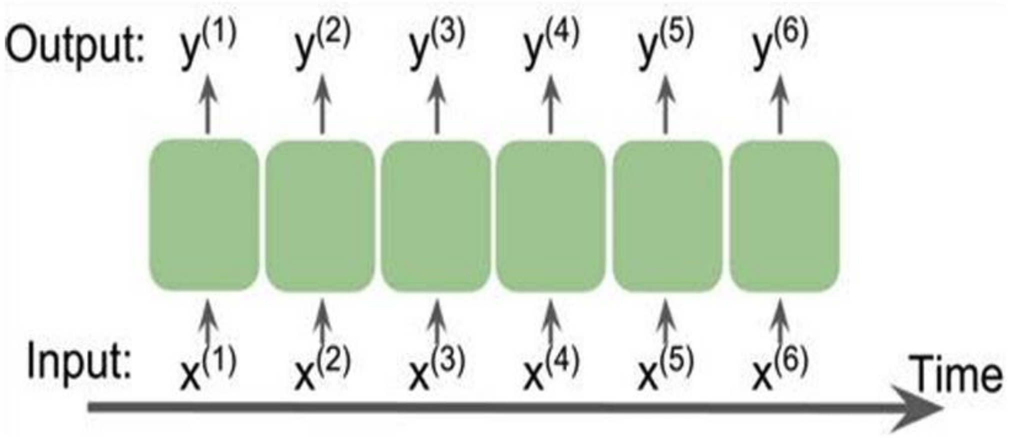
Sequences

Challenges in modeling sequential data

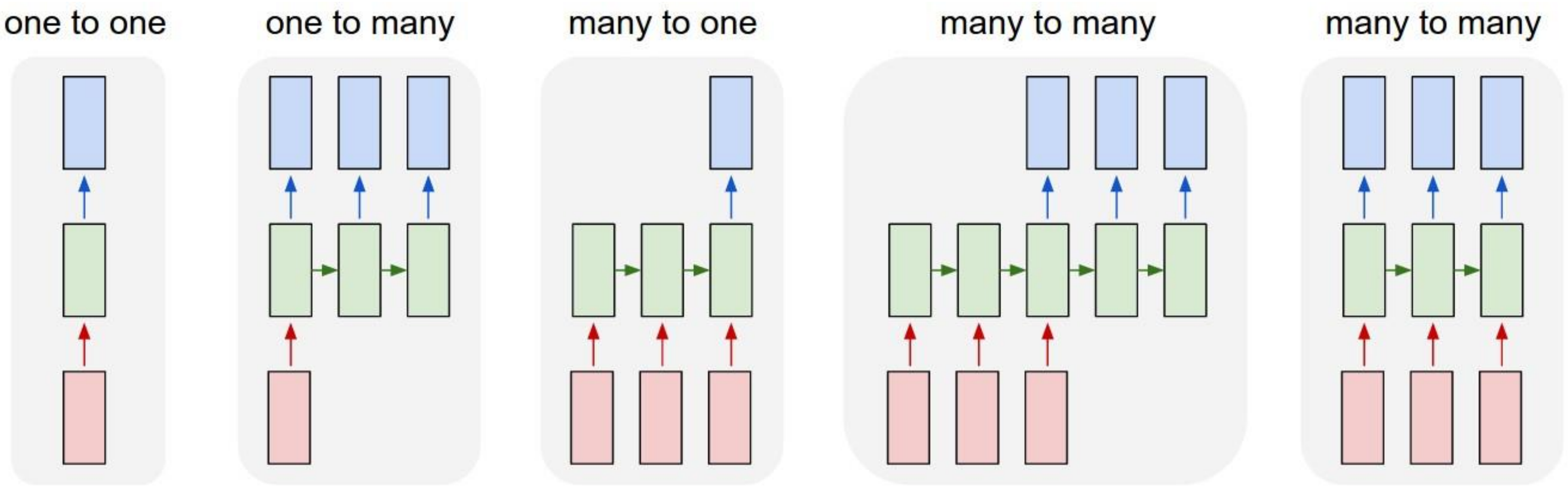
- **Infinite number of possible sequences:**
- ❖ Sequences can vary in **length** (short vs. long sequences).
- ❖ Sequences can have **variable patterns** (e.g., DNA sequences, language models).
- ❖ Order **matters**, meaning different orders of the same elements can have different meanings.
- **Need for Probability Distributions Over Sequences:**
- ➢ Since an infinite number of sequences exist, we **cannot store all possible sequences explicitly**.
- ➢ Instead, we model a **probabilistic function** that assigns a likelihood to each possible sequence.
- ➢ Example: Given a sequence **S=(x1,x2,...,xT)**, we want to learn a **probability distribution** P(S).

RNNs are designed for modeling sequences

- Sequences of any length in the input, in the output, or in both
- They can remember past information
- Apply the same weights on each step



Different Categories of Sequence Modeling



Vanilla mode without RNN
e.g. image classification

Sequence output
e.g., image captioning

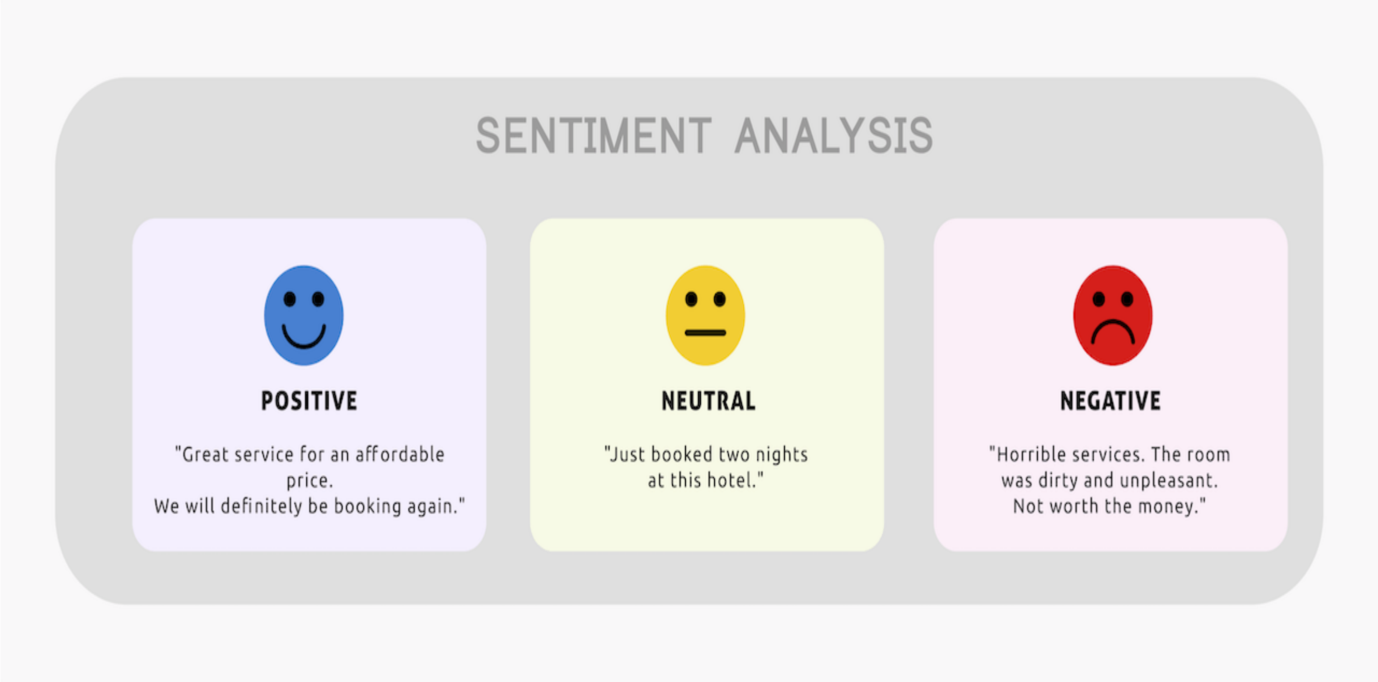
Sequence input
e.g., sentiment analysis

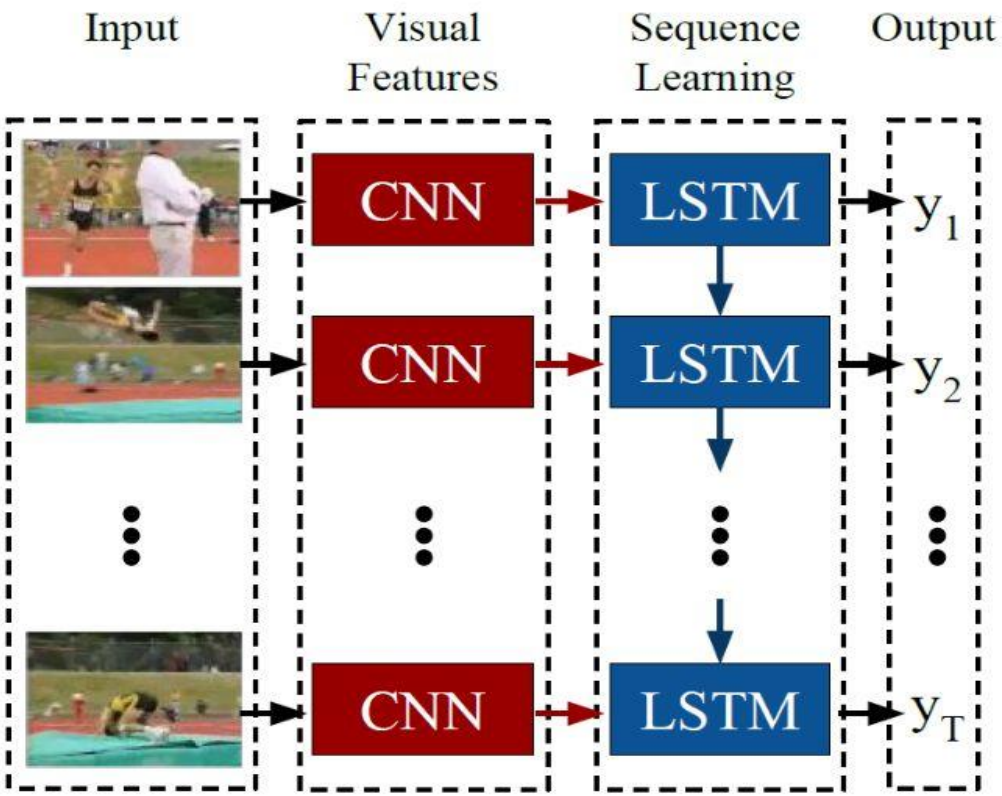
Sequence input and output
e.g., machine translation

Synced sequence input and output
e.g., video classification

Source: https://karpathy.github.io/2015/05/21/rnn-effectiveness/

Sentiment Analysis and Video Analytics





Sentiment Analysis is a **NLP** technique used to determine the emotional tone of a given text. It helps identify whether the sentiment of the text is **positive, negative, or neutral**.

https://www.gosmar.eu/machinelearning/2020/08/23/recurrent-neural-networks-for-sentiment-analysis/

Video analytics enable machines to **recognize actions, objects, and scenes** in videos.

https://imerit.net/blog/using-neural-networks-for-video-classification-blog-all-pbm/

Machine Translation

Machine Translation (MT) is the task of translating a sentence *x* from one language (the source language) to a sentence *y* in another language (the target language).

Goal: Produce translations that are both fluent and faithful to the meaning of the source text.

Applications: Global communication, localization cross-lingual information retrieval, etc.

x: *L'homme est né libre, et partout il est dans les fers*

y:

English:
Man is born free, but everywhere he is in chains

Chinese:
“人生而自由，但无处不在被枷锁束缚。”

Japanese:
「人間は自由に生まれるが、どこにいても鎖に縛られている。」

The early history of MT: 1950s

Rule-Based MT: Early systems used manually-crafted rules and bilingual dictionaries.

1990s-2010s: Statistical Machine Translation

Core idea: Utilized probabilistic models (e.g., IBM Models) and phrase-based translation. Relied on large parallel corpora to learn translation probabilities

- Suppose we're translating French → English.
- We want to find best English sentence y, given French sentence x

$$\mathrm{argmax}_y P(y|x)$$

- Use Bayes Rule to break this down into two components to be learned separately:

$$= \mathrm{argmax}_y \underbrace{P(x|y)}_{\text{Translation Model}} \underbrace{P(y)}_{\text{Language Model}}$$

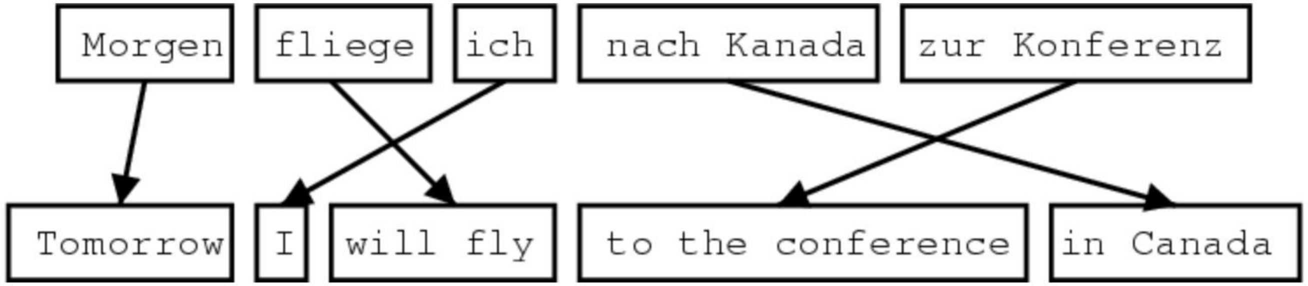
Translation Model

Models how words and phrases should be translated (*fidelity*). Learned from parallel data.

Language Model

Models how to write good English (*fluency*). Learned from monolingual data.

Not trivial to model!



1519年600名西班牙人在墨西哥登陆，去征服几百万人口的阿兹特克帝国，初次交锋他们损兵三分之二。

In 1519, six hundred Spaniards landed in Mexico to conquer the Aztec Empire with a population of a few million. They lost two thirds of their soldiers in the first clash.

translate.google.com (2009): 1519 600 Spaniards landed in Mexico, millions of people to conquer the Aztec empire, the first two-thirds of soldiers against their loss.

translate.google.com (2013): 1519 600 Spaniards landed in Mexico to conquer the Aztec empire, hundreds of millions of people, the initial confrontation loss of soldiers two-thirds.

translate.google.com (2015): 1519 600 Spaniards landed in Mexico, millions of people to conquer the Aztec empire, the first two-thirds of the loss of soldiers they clash.

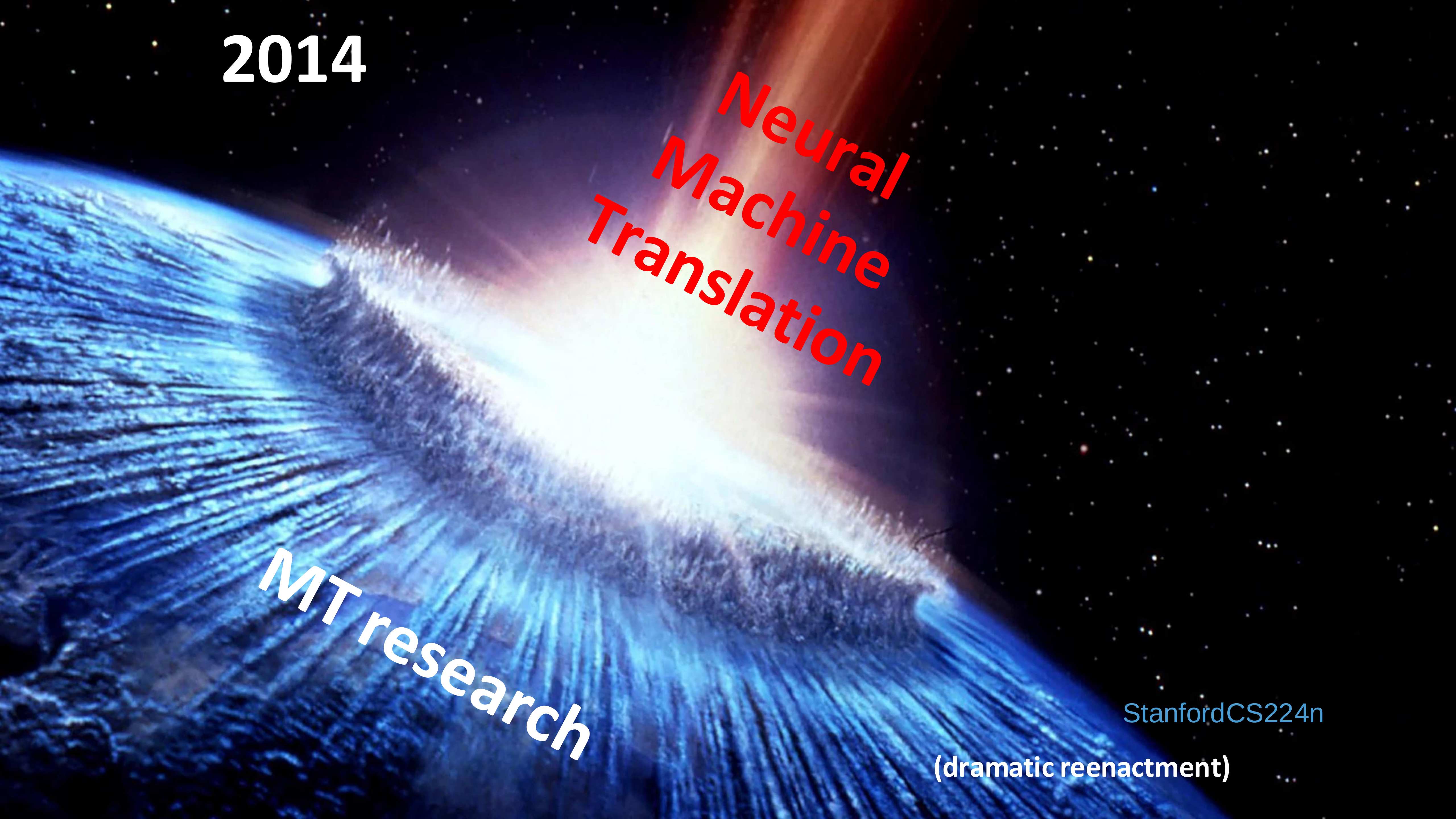
1990s–2010s: Statistical Machine Translation

- SMT was once a huge research field aimed at automatically translating text between languages. It relied on probabilistic models and large parallel corpora to learn translation patterns.

- Despite its success in the past, SMT required extensive manual design and engineering. The best SMT systems were extremely complex, involving hundreds of important details. Systems were built from many separately-designed subcomponents, each addressing a specific aspect of translation. Every component was carefully engineered to optimize the overall translation quality.

- SMT required extensive feature engineering to capture specific language phenomena: Designing features to model syntax, semantics, and context. Crafting features to capture idiomatic expressions and local linguistic patterns. Each feature was manually designed, tested, and fine-tuned. This process was both time-consuming and highly dependent on expert knowledge.

- SMT systems often required compiling and maintaining extra resources: Tables of equivalent phrases, bilingual dictionaries, and syntactic rules. Language-specific resources had to be built and maintained. A large amount of human effort was needed to manage these resources. The process had to be repeated for each language pair, making it labor-intensive and costly.


2014
Neural Machine Translation
MT research
Stanford CS224n
(dramatic reenactment)

NMT: the first big success story

Neural Machine Translation (NMT): Uses deep learning and end-to-end training to model translation and offers improved fluency and the ability to capture complex dependencies.

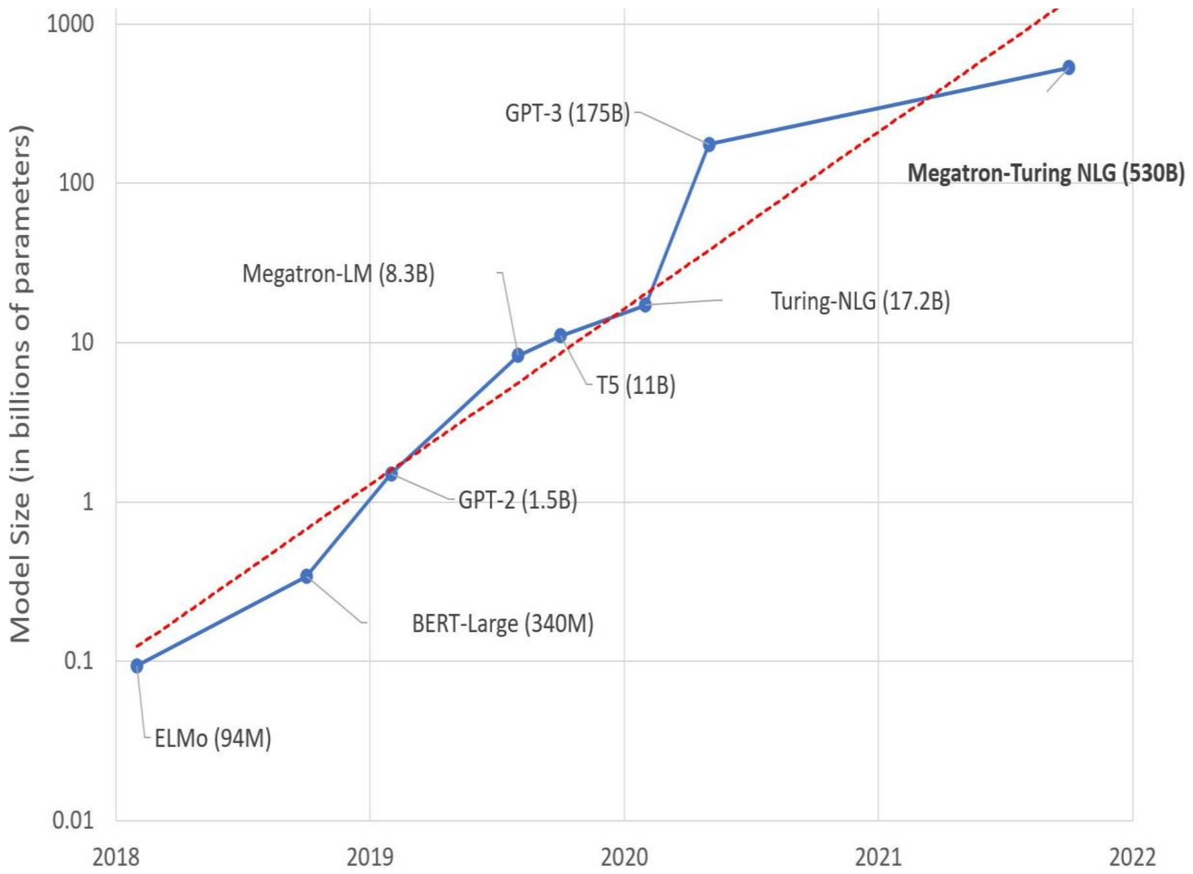
Neural Machine Translation went from a fringe research attempt in **2014** to the leading standard method in **2016**

- **2014**: First seq2seq paper published [Sutskever et al. 2014]
- **2016**: Google Translate switches from SMT to NMT – and by 2018 everyone had
  - https://www.nytimes.com/2016/12/14/magazine/the-great-ai-awakening.html

Microsoft SYSTRAN beyond language Google facebook
Baidu 百度 網易 NETEASE www.163.com Tencent 腾讯 搜狗搜索

- **This was amazing!**
  - **SMT** systems, built by hundreds of engineers over many years, were outperformed by NMT systems trained by small groups of engineers in a few months

Deepseek v.s. OpenAI



Modern NLP Systems

https://ai.plainenglish.io/deepseek-r1-vs-chatgpt-01-my-experience-ddbe09e80aa9

https://huggingface.co/blog/large-language-models

Challenges

- Standard NN models (MLPs, CNNs) are not able to handle sequences of data
- ❖ They accept a **fixed-sized vector** as input and produce a **fixed-sized vector** as output.
- ❖ The **weights are updated independently**, meaning there is **no memory** of past computations.
- ❖ The models **do not have recurrence**, so they cannot learn patterns across time steps.

- Many real-world problems require capturing **context over time**:
- ❖ **Speech Recognition** – Words depend on previous words.
- ❖ **Time-series Prediction** – Future values depend on past observations.
- ❖ **DNA Sequencing** – Genetic patterns unfold over long sequences.
- ❖ **Natural Language Processing (NLP)** – Meaning depends on word order.

- **Example: Simple Context-Dependent Problem:** Output YES if the number of 1s in the sequence is even; otherwise, output NO.
  - Input: 1000010101 → **YES**; Input: 100011 → **NO**

Challenges

High Dimensionality and Complexity - Sequential data often involves high-dimensional inputs with complex interdependencies:

- **Text:** Words and phrases have semantic and syntactic relationships across sentences.
- **Time Series:** Multivariate time series data (e.g., temperature, humidity, and pressure) exhibit interdependencies between variables over time.
- **Biological Data:** DNA sequences and protein structures involve intricate, sequential patterns.

Solution: RNNs address this by learning hierarchical representations through their recurrent structure, encoding both local and global patterns.

Noise and Missing Data - Sequential data often contains noise or missing values:

- **Noise:** Sensor readings and time series data may have irregularities or anomalies.
- **Missing values:** Gaps in sequences arise from interruptions in data collection.

Solution: RNNs aggregate information over time, making them robust to noise and capable of interpolating missing values using contextual information.

Challenges

Temporal Dependencies

- **Short-term dependencies:** In text, the current word depends on immediately preceding words (e.g., ``I want to eat a...'').
- **Long-term dependencies:** Distant elements in the sequence can influence the current state (e.g., in a paragraph, the topic sentence affects subsequent sentences).

Solution: RNNs maintain memory through hidden states, enabling them to model temporal dependencies. Variants like Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU) address challenges such as vanishing gradients, allowing effective modeling of long-term dependencies.

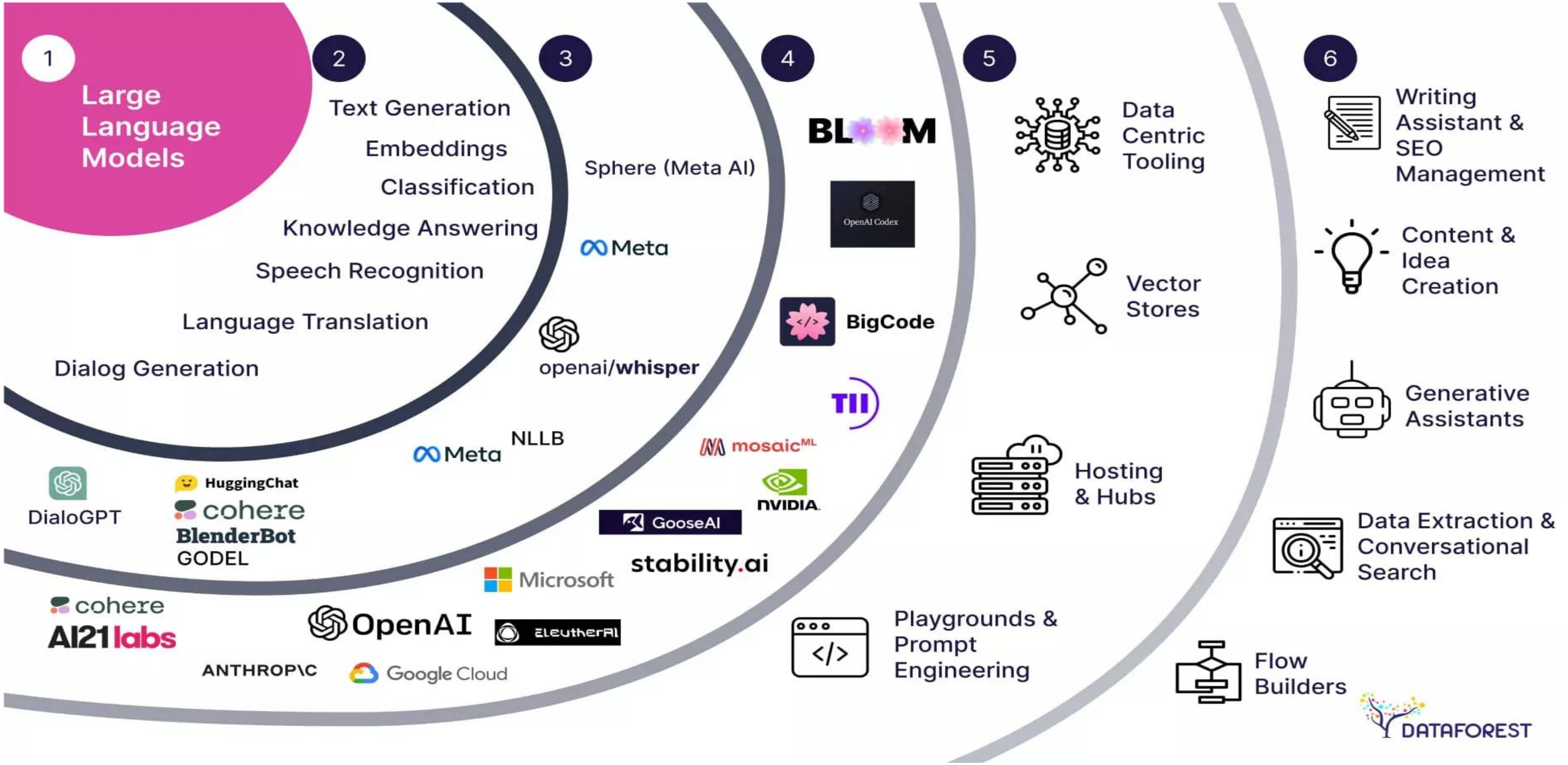
Variable-Length Inputs and Outputs - Many real-world tasks involve sequences of varying lengths, which traditional models struggle to handle. RNNs process inputs dynamically, making them ideal for tasks with variable-length data.

Examples:

- **Natural Language Processing (NLP):** Sentences have varying word counts, and RNNs can process each word without requiring fixed input dimensions.
- **Speech Recognition:** Audio recordings vary in duration depending on the speaker or content.
- **Time Series:** Data collected over irregular time intervals often results in sequences of differing lengths.

Content

Large Language Models

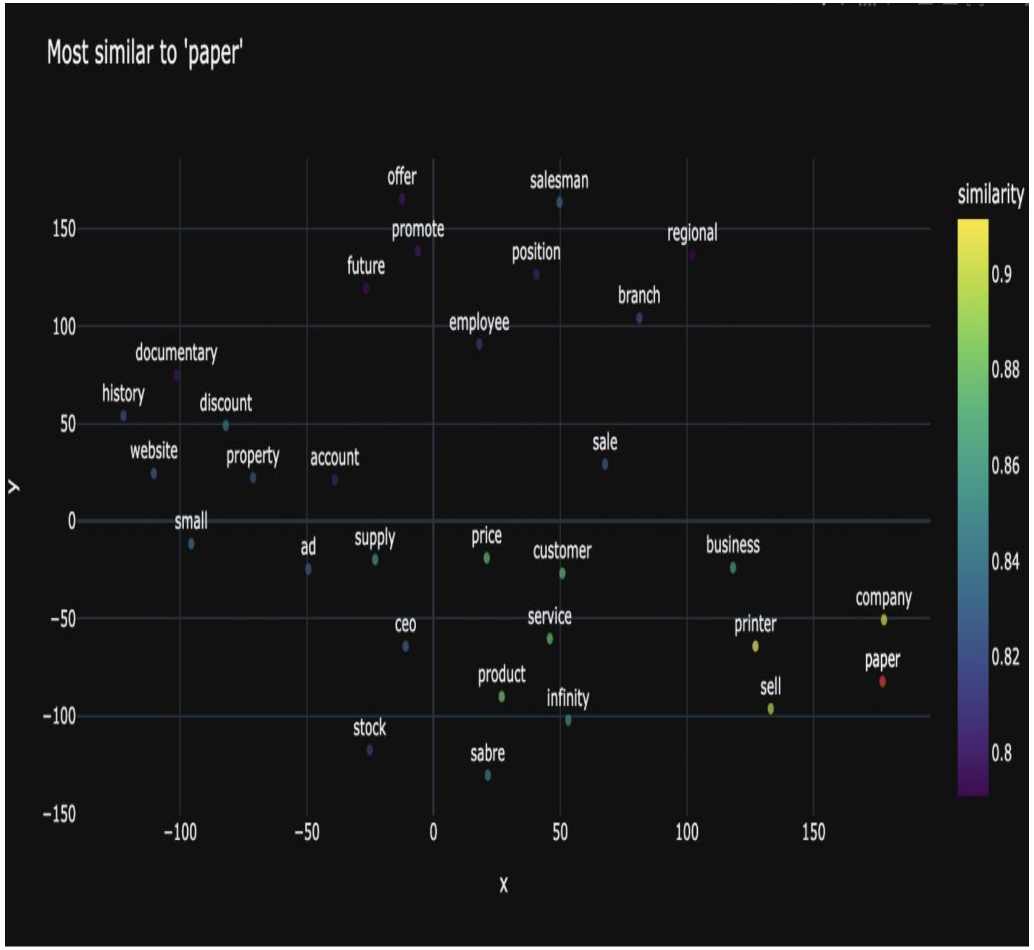


https://dataforest.ai/blog/large-language-models-advanced-communication

Text Processing for RNN

Step	Task	Description
1	Preprocessing Text	Clean text, tokenize, remove stopwords, and normalize case using NLTK.
2	Build Vocabulary	Assign a unique index to each word using a frequency-based vocabulary.
3	Convert Text to Sequences	Map tokenized words to their corresponding integer indices.
4	Padding Sequences	Standardize input sequence lengths by adding padding tokens where necessary.
5	Dataloader Preparation	Create PyTorch dataset and dataloader for mini-batch training.
6	Load Pretrained Word Embeddings	Use GloVe embeddings (100D) for better semantic representation.
7	Define RNN Model	Construct an RNN with an embedding layer, hidden layers, and output layer.
8	Loss and Optimization	Use Binary Cross-Entropy Loss (‘BCEWithLogitsLoss‘) and Adam optimizer.
9	Train Model	Train the RNN model using mini-batches from the dataloader.
10	Make Predictions	Preprocess new text, convert it to sequences, and run inference using the trained model.

Word Embeddings



Word embeddings are a fundamental technique in NLP. They convert words into dense, continuous vector representations. Word embeddings place similar words closer in vector space. Unlike traditional one-hot encoding, embeddings preserve:

- **Semantic relationships** between words.
- **Contextual meaning** of words in sentences.
- **Word similarity and analogies.**

Types of Word Embeddings

- **Frequency-Based Methods**
  - TF-IDF (Term Frequency-Inverse Document Frequency)
  - LSA (Latent Semantic Analysis)
- **Prediction-Based Methods (Neural Networks)**
  - ✓ Word2Vec (CBOW & Skip-gram)
  - ✓ GloVe (Global Vectors for Word Representation)
  - ✓ FastText (Subword Embeddings)
  - ✓ Transformer-Based (BERT, GPT)

Word Embeddings Dimensions

```python
from transformers import BertTokenizer, BertModel
import torch

# Load BERT model
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
model = BertModel.from_pretrained("bert-base-uncased")

# Tokenize and get embedding
text = "Hello world"
tokens = tokenizer(text, return_tensors="pt")
with torch.no_grad():
    output = model(**tokens)

print("BERT Embedding Dimension:", output.last_hidden_state.shape[-1])
```

Word Embeddings	Dimension	Key Features
Word2Vec	50-300	Trained on large corpora like Google News
GloVe	50-300	Uses word co-occurrence statistics
FastText	50-300	Handles subword information
ELMo	1024	Contextual embeddings from bidirectional LSTMs
BERT (base)	768	Transformer-based, context-aware
BERT (large)	1024	More parameters than BERT base
GPT-2 (small)	768	Transformer-based generative model
GPT-2 (medium)	1024	More layers and parameters
GPT-3	12288	High-dimensional transformer model

🔷 **For small models or mobile applications →** Use **50-300 dimensions** (Word2Vec, GloVe).

🔷 **For NLP applications with context-awareness →** Use **512-1024 dimensions** (BERT, ELMo).

🔷 **For large-scale generative AI →** Use **1024+ dimensions** (GPT-3, Transformers).

Train an RNN Language Model

Task 1: *How to import this paragraph for training the language model?*

Obama *"I stand here today humbled by the task before us, grateful for the trust you have bestowed, mindful of the sacrifices borne by our ancestors. I thank President Bush for his service to our nation, as well as the generosity and cooperation he has shown throughout this transition. Forty-four Americans have now taken the presidential oath. The words have been spoken during rising tides of prosperity and the still waters of peace. Yet, every so often the oath is taken amidst gathering clouds and raging storms. At these moments, America has carried on not simply because of the skill or vision of those in high office, but because We the People have remained faithful to the ideals of our forbearers, and true to our founding documents. "*

```python
import re
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from gensim.models import Word2Vec
import numpy as np
import random

# For reproducibility
torch.manual_seed(42)
np.random.seed(42)
random.seed(42)

# ================================
# 1. Define and Preprocess the Text
# ================================
text = (
"I stand here today humbled by the task before us, grateful for the trust you have bestowed, "
"mindful of the sacrifices borne by our ancestors. I thank President Bush for his service to our nation, "
"as well as the generosity and cooperation he has shown throughout this transition.\n\n"
"Forty-four Americans have now taken the presidential oath. The words have been spoken during rising tides "
"of prosperity and the still waters of peace. Yet, every so often the oath is taken amidst gathering clouds "
"and raging storms. At these moments, America has carried on not simply because of the skill or vision of "
"those in high office, but because We the People have remained faithful to the ideals of our forbearers, and "
"true to our founding documents."
)
```

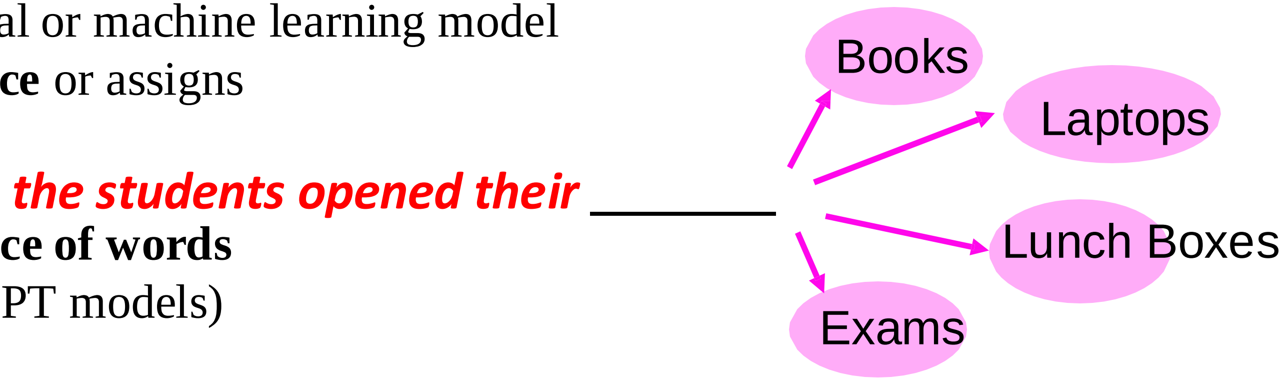
```python
print("\n--- Original Text ---\n")
print(text)

# Split the text into sentences using a simple regex
sentences = re.split(r'(?<=[.!?])\s+', text.strip())
print("\n--- Split Sentences ---")
for i, s in enumerate(sentences):
print(f"Sentence {i+1}: {s}")

def tokenize(text):
# Convert to lowercase and split using regex for word boundaries
return re.findall(r'\b\w+\b', text.lower())

# Tokenize each sentence
tokenized_sentences = [tokenize(sentence) for sentence in sentences]
print("\n--- Tokenized Sentences ---")
for i, tokens in enumerate(tokenized_sentences):
print(f"Sentence {i+1} Tokens: {tokens}")

# Build vocabulary (set of unique words)
vocab = sorted(set(word for sentence in tokenized_sentences for word in sentence))
word_to_idx = {word: idx for idx, word in enumerate(vocab)}
idx_to_word = {idx: word for word, idx in word_to_idx.items()}
vocab_size = len(vocab)

print("\n--- Vocabulary ---")
print(word_to_idx)
print(f"Vocabulary Size: {vocab_size}")
```

```python
# ==================================
# 2. Train Word2Vec and Build Embedding Matrix
# ==================================
embedding_dim = 100

w2v_model = Word2Vec(sentences=tokenized_sentences,
vector_size=embedding_dim, window=5, min_count=1, workers=4)

# Build the embedding matrix (vocab_size x embedding_dim)
embedding_matrix = torch.zeros(vocab_size, embedding_dim)
for word, idx in word_to_idx.items():
embedding_vector = w2v_model.wv[word]
if embedding_vector is not None:
embedding_matrix[idx] = torch.tensor(embedding_vector)
# ==================================
# 3. Prepare Input and Target Sequences for Language Modeling
# ==================================
# For language modeling, we treat the text as one continuous
sequence.
indices = [word_to_idx[word] for sentence in
tokenized_sentences for word in sentence]

# Create input sequence and target sequence.
# The target is the next word (shifted by one position). The
last target is omitted.
input_indices = indices[:-1] # all except last
target_indices = indices[1:] # all except first

# Convert to PyTorch tensors. We use batch_size = 1 for
simplicity.
input_seq = torch.tensor([input_indices], dtype=torch.long) #
Shape: (1, seq_len)
target_seq = torch.tensor([target_indices], dtype=torch.long)
# Shape: (1, seq_len)
```

```
Sentence 1: I stand here today humbled by the task before us,
grateful for the trust you have bestowed, mindful of the
sacrifices borne by our ancestors.
--- Sentence 1 Tokens: ['i', 'stand', 'here', 'today',
'humbled', 'by', 'the', 'task', 'before', 'us', 'grateful',
'for', 'the', 'trust', 'you', 'have', 'bestowed', 'mindful',
'of', 'the', 'sacrifices', 'borne', 'by', 'our', 'ancestors']
--- Vocabulary --- {'america': 0, 'americans': 1, 'amidst': 2,
'ancestors': 3, 'and': 4, 'as': 5, 'at': 6, 'because': 7,
'been': 8, 'before': 9, 'bestowed': 10, 'borne': 11, 'bush':
12, 'but': 13, . . .  'we': 88, 'well': 89, 'words': 90,
'yet': 91, 'you': 92}
 Vocabulary Size: 93
--- Training Word2Vec Model --- Word2Vec training completed. -
-- Embedding Matrix ---
Embedding Matrix Shape: torch.Size([93, 100]) Sample
Embeddings: america: tensor([-0.0081, -0.0009, 0.0064, 0.0087,
-0.0050])... americans: tensor([ 0.0010, 0.0086, -0.0040,
0.0030, 0.0032])... amidst: tensor([-0.0065, 0.0073, 0.0061, -
0.0049, -0.0017
--- Token Indices --- [37, 68, 33, 81, 36, 14, 74, 72, 9, 85,
29, 22, 74, 84, 92, 31, 10, 41, 47, 74, 61, 11, 14, 52, 3, 37,
73, 55, 12, 22, 35, 62, 80, 52, 43, 5, 89, 5, 74, 28, 4, 17,
32, 30, 63, 78, 76, 82, 24, 26, 1, 31, 45, 71, 74, 56, 46, 74,
90, 31, 8, 67, 19, 60, 79, 47, 57, 4, 74, 69, 87, 47, 53, 91,
20, 66, 49, 74, 46, 40, 71, 2, 27, 16, 4, 58, 70, 6, 75, 42,
0, 30, 15, 50, 44, 64, 7, 47, 74, 65, 51, 86, 47, 77, 39, 34,
48, 13, 7, 88, 74, 54, 31, 59, 21, 80, 74, 38, 47, 52, 23, 4,
83, 80, 52, 25, 18] --- Input and Target Indices --- Input
Indices (first 20): [37, 68, 33, 81, 36, 14, 74, 72, 9, 85,
29, 22, 74, 84, 92, 31, 10, 41, 47, 74] Target Indices (first
20): [68, 33, 81, 36, 14, 74, 72, 9, 85, 29, 22, 74, 84, 92,
31, 10, 41, 47, 74, 61] Input Sequence Shape: torch.Size([1,
126]) Target Sequence Shape: torch.Size([1, 126])
```

Formulation: Language Modeling (LM)

A **language model (LM)** is a statistical or machine learning model that **predicts the next word in a sequence** or assigns **probabilities** to sequences of words.



- ✅ **Predicts the likelihood of a sequence of words**
- ✅ **Generates human-like text** (e.g., GPT models)
- ✅ **Understands context and meaning**
- ✅ **Enables AI systems to process and generate natural language**

More formally: given a sequence of words $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(t)}$,

compute the probability distribution of the next word $\boldsymbol{x}^{(t+1)}$:

$$P(\boldsymbol{x}^{(t+1)} \mid \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)})$$

where $\boldsymbol{x}^{(t+1)}$ can be any word in the vocabulary $V = \{\boldsymbol{w}_1, \ldots, \boldsymbol{w}_{|V|}\}$

Example:

- **Input:** *"I am going to the"*
- **Model prediction:** *"store" (80%), "beach" (15%), "moon" (5%)*
- The model assigns probabilities and selects the most likely next word.

Language Modeling

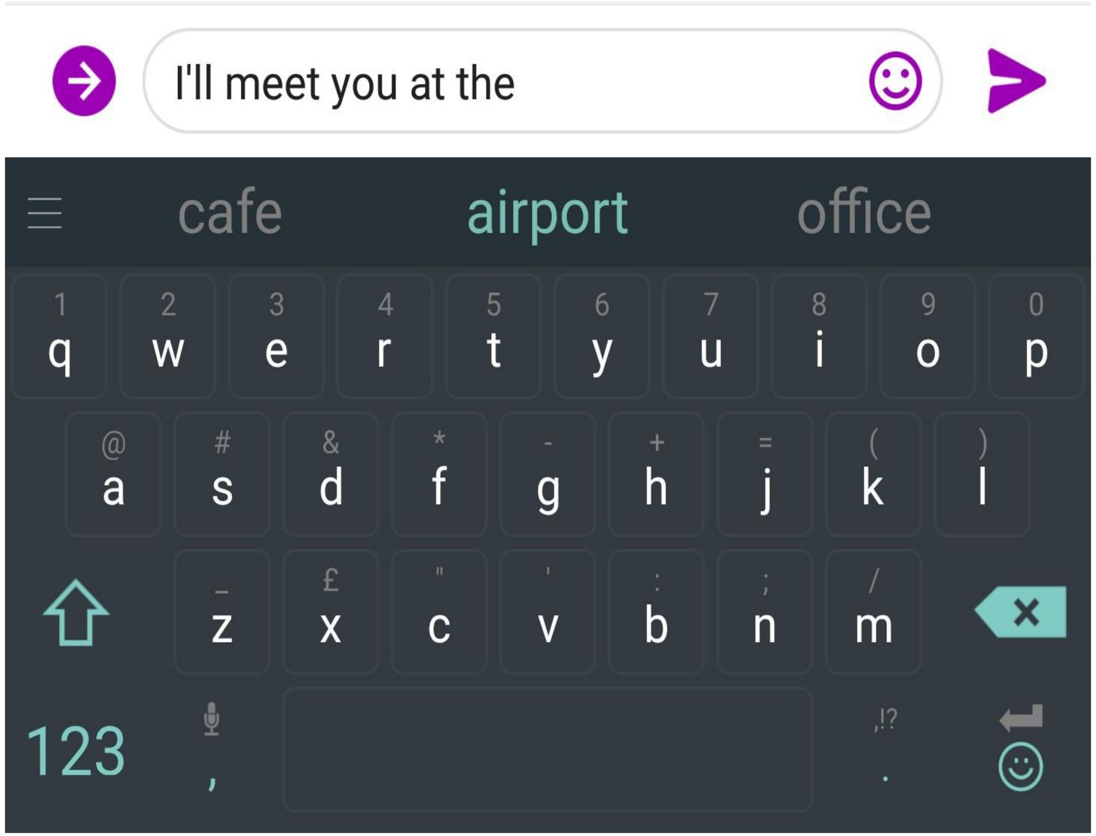
- You can also think of a Language Model as a system that assigns a probability to a piece of text
- For example, if we have some text $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}$ then the probability of this text (according to the Language Model) is:

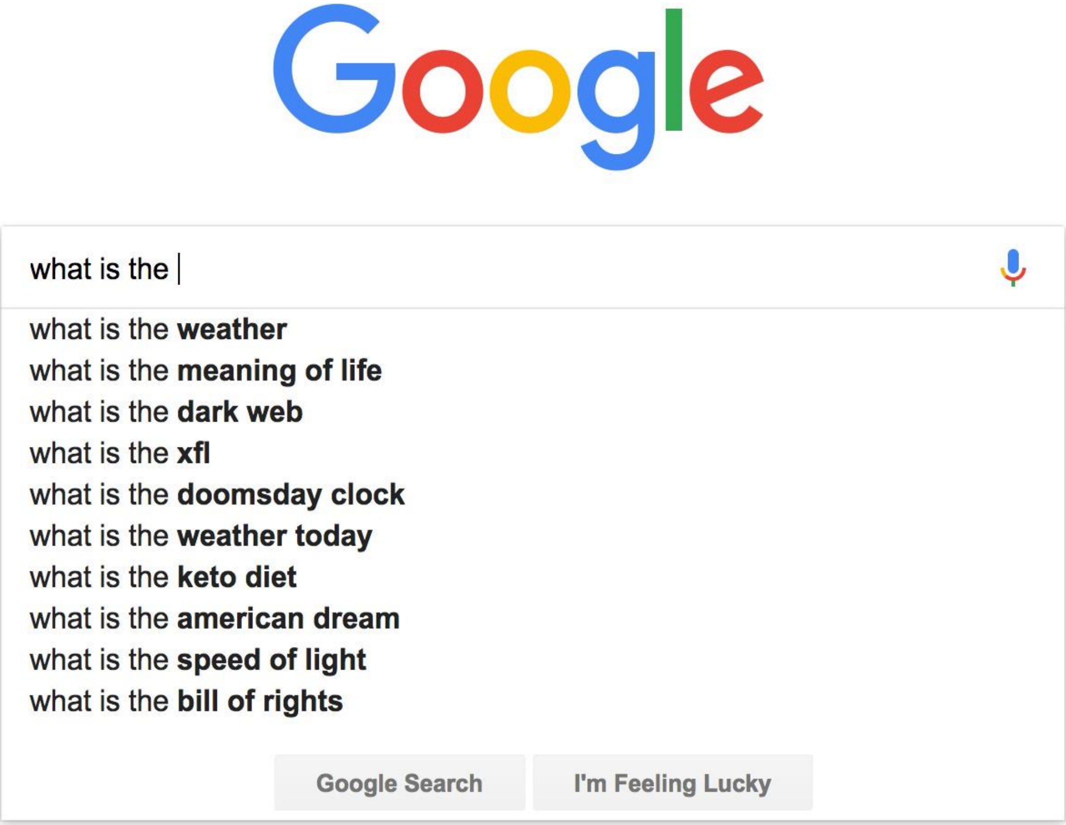
$$P(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}) = P(\boldsymbol{x}^{(1)}) \times P(\boldsymbol{x}^{(2)} | \boldsymbol{x}^{(1)}) \times \cdots \times P(\boldsymbol{x}^{(T)} | \boldsymbol{x}^{(T-1)}, \ldots, \boldsymbol{x}^{(1)})$$

$$= \prod_{t=1}^{T} \underbrace{P(\boldsymbol{x}^{(t)} | \boldsymbol{x}^{(t-1)}, \ldots, \boldsymbol{x}^{(1)})}_{\text{This is what our LM provides}}$$

StanfordCS224n

You use Language Models every day!





StanfordCS224n

n-gram Language Models

the students opened their ______

- **Question**: How to learn an n-gram Language Model during the pre-DL period?
- **Answer:** An **n-gram** is a sequence of **n consecutive words** from a text. The larger the **n**, the more context the model considers when making predictions.
- **Definition:** An *n*-gram is a chunk of *n* consecutive words.
  - unigrams: “the”, “students”, “opened”, ”their”
  - bigrams: “the students”, “students opened”, “opened their”
  - trigrams: “the students opened”, “students opened their”
  - four-grams: “the students opened their”

Idea:

- **Collect statistics** on how frequently different **n-grams appear in a corpus**.
- The **probability of the next word** is estimated using the **previous (n-1) words**.

StanfordCS224n

n-gram Language Models

- Under the Markov assumption: $x^{(t+1)}$ depends only on the preceding *n*-1 words

$$P(\boldsymbol{x}^{(t+1)}|\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(1)}) = P(\boldsymbol{x}^{(t+1)}|\overbrace{\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)}}^{n\text{-1 words}})$$

(assumption)

prob of a n-gram

prob of a (n-1)-gram

$$= \frac{\boxed{P(\boldsymbol{x}^{(t+1)},\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})}}{\boxed{P(\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})}}$$

(definition of conditional prob)

- **Question:** How do we get these *n*-gram and (*n*-1)-gram probabilities?
- **Answer:** By counting them in some large corpus of text!

$$\approx \frac{\text{count}(\boldsymbol{x}^{(t+1)},\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})}{\text{count}(\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})}$$

(statistical approximation)

StanfordCS224n

n-gram Language Models: Example

Suppose we are learning a 4-gram Language Model.

~~*as the proctor started the clock, the*~~ $\underbrace{\textit{students opened their}}_{\text{condition on this}}$ ______

discard

$$P(\boldsymbol{w}|\text{students opened their}) = \frac{\text{count}(\text{students opened their } \boldsymbol{w})}{\text{count}(\text{students opened their})}$$

For example, suppose that in the corpus:

- "students opened their" occurred 1000 times
- "students opened their books" occurred 400 times
  - ➔ P(books | students opened their) = 0.4
- "students opened their exams" occurred 100 times
  - ➔ P(exams | students opened their) = 0.1

Should we have discarded the "proctor" context?

StanfordCS224n

Major Challenges

🚧 **Data Sparsity** – Rare n-grams may not appear frequently in training data.
🚧 **Fixed Context Window** – Cannot capture long-range dependencies beyond n words.
🚧 **Poor Generalization** – Cannot understand unseen word sequences.

MC1

Problem: What if *"students opened their w"* never occurred in data? Then w has probability 0!

(Partial) Solution: Add small δ to the count for every $w \in V$. This is called *smoothing*.

$$P(\boldsymbol{w}|\text{students opened their}) = \frac{\boxed{\text{count}(\text{students opened their } \boldsymbol{w})}}{\boxed{\text{count}(\text{students opened their})}}$$

MC2

Problem: What if *"students opened their"* never occurred in data? Then we can't calculate probability for *any w*!

(Partial) Solution: Just condition on *"opened their"* instead.
This is called *backoff*.

MC3

Storage: Need to store count for all *n*-grams you saw in the corpus. Increasing *n* or increasing corpus increases model size!

MC4

Increasing *n* makes sparsity problems *worse*. Typically, we can't have *n* bigger than 5.

Generating text with a n-gram Language Model

You can also use a Language Model to generate text

today the ______

condition on this

get probability distribution

company	0.153
bank	0.153
price	0.077
italian	0.039
emirate	0.039
…	

sample

today the price ______

condition on this

get probability distribution

of	0.308
for	0.050
it	0.046
to	0.046
is	0.031
…	

sample

today the price ______

condition on this

get probability distribution

the	0.072
18	0.043
oil	0.043
its	0.036
gold	0.018
…	

sample

StanfordCS224n

Generating text with a n-gram Language Model

You can also use a Language Model to 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…

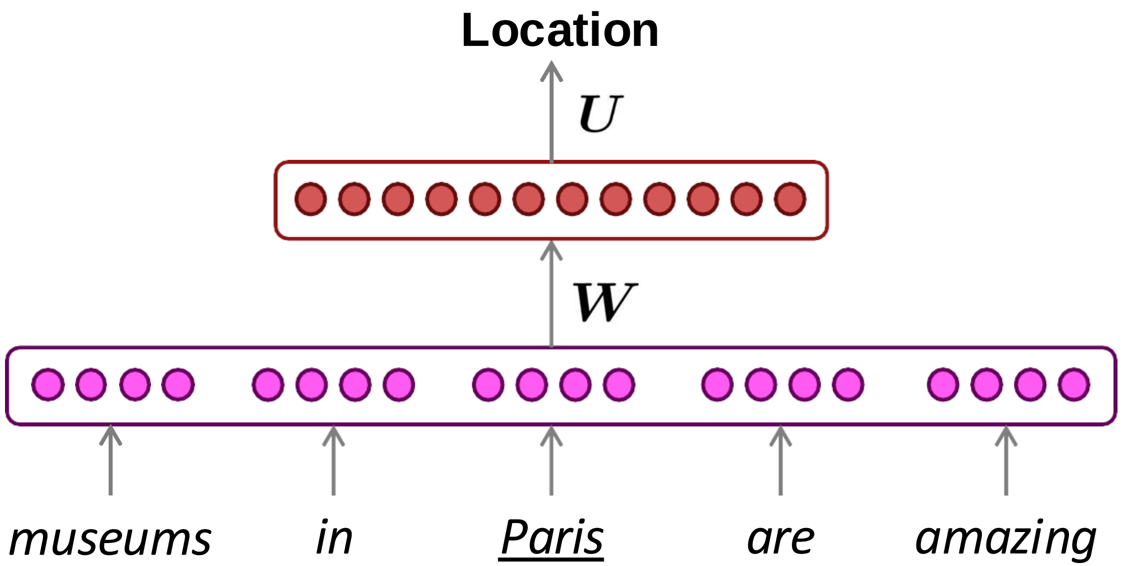
StanfordCS224n

How to build a *neural* language model?

- Recall the Language Modeling task:
  - Input: sequence of words $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(t)}$
  - Output: prob. dist. of the next word $P(\boldsymbol{x}^{(t+1)} | \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)})$

- How about a window-based neural model?
  - We saw this applied to Named Entity Recognition (NER):

NER is a fundamental **NLP** task that involves identifying and classifying **specific entities** in a given text into predefined categories such as **names of people, organizations, locations, dates, monetary values, and more**.



StanfordCS224n

A Fixed-window Neural Language Model

Output Distribution

$$\hat{y} = \mathrm{softmax}(\boldsymbol{U}\boldsymbol{h} + \boldsymbol{b}_2) \in \mathbb{R}^{|V|}$$

Hidden Layer

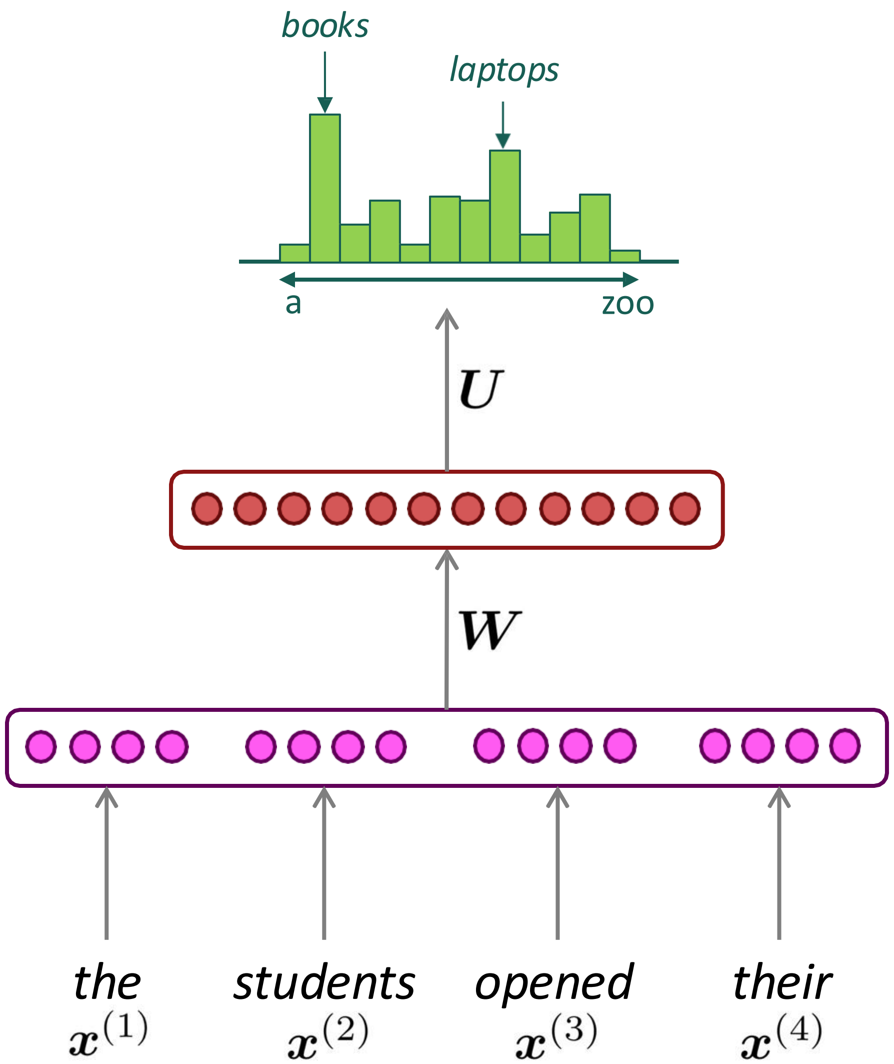
$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{e} + \boldsymbol{b}_1)$$

Concatenated Word Embeddings

$$\boldsymbol{e} = [\boldsymbol{e}^{(1)}; \boldsymbol{e}^{(2)}; \boldsymbol{e}^{(3)}; \boldsymbol{e}^{(4)}]$$

Words /One-hot vectors

$$\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(3)}, \boldsymbol{x}^{(4)}$$



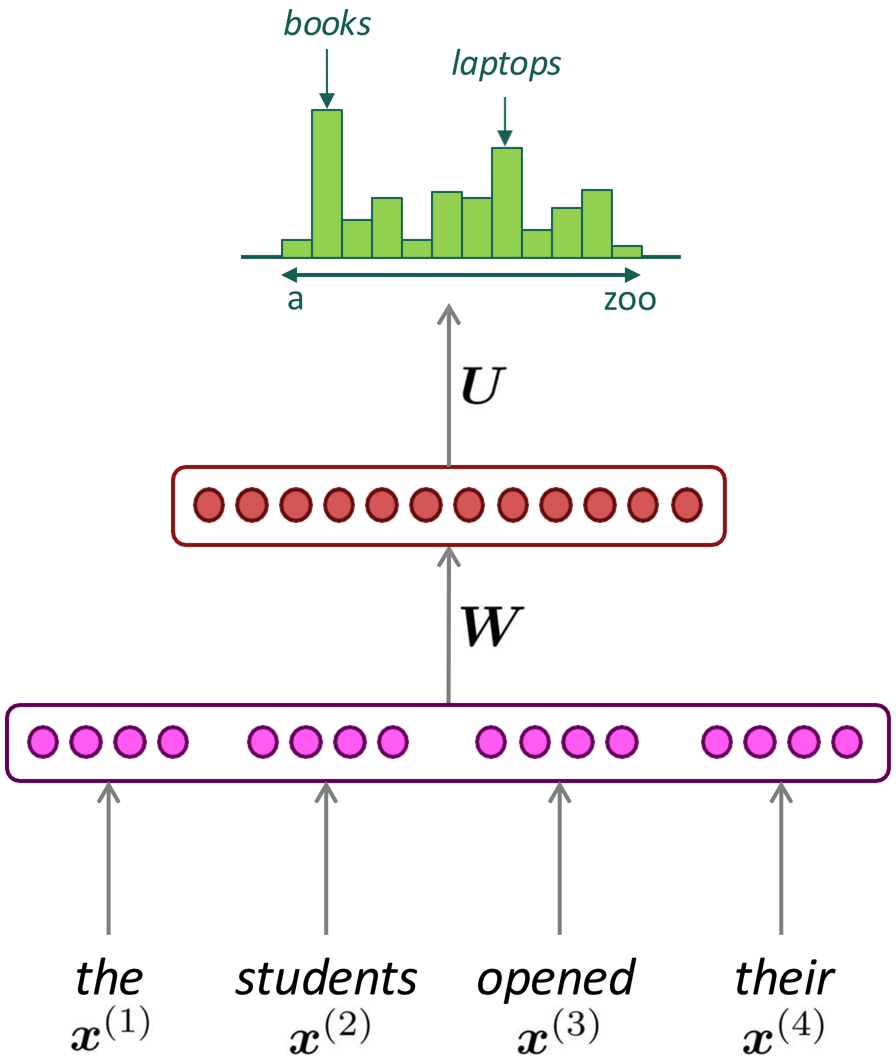
StanfordCS224n

A fixed-window neural Language Model

Y. Bengio, et al. (2000/2003): A Neural Probabilistic Language Model

Improvements over *n*-gram LM:
- No sparsity problem
- Don't need to store all observed *n*-grams

Remaining **problems**:
- Fixed window is too small
- Enlarging window enlarges W
- Window can never be large enough!
- $x^{(1)}$ and $x^{(2)}$ are multiplied by completely different weights in W. No symmetry in how the inputs are processed.



We need a neural architecture that can process *any length input*

StanfordCS224n

Content

History of Deep RNNs

The Rise of Deep RNNs (2010s - Present)

🔷 RNNs in NLP and AI

- **2013** – Google used LSTM for **speech recognition**.
- **2014** – **Seq2Seq Models** (Sutskever et al.) used LSTMs for **machine translation**.
- **2015** – Google Translate adopted LSTMs for **neural machine translation (NMT)**.

🔷 Attention and Transformers Change the Game

- **2015** – **Bahdanau et al.** introduced **Attention Mechanisms**, improving Seq2Seq models.
- **2017** – **Vaswani et al.** introduced **Transformers**, replacing RNNs with a more parallelizable model.

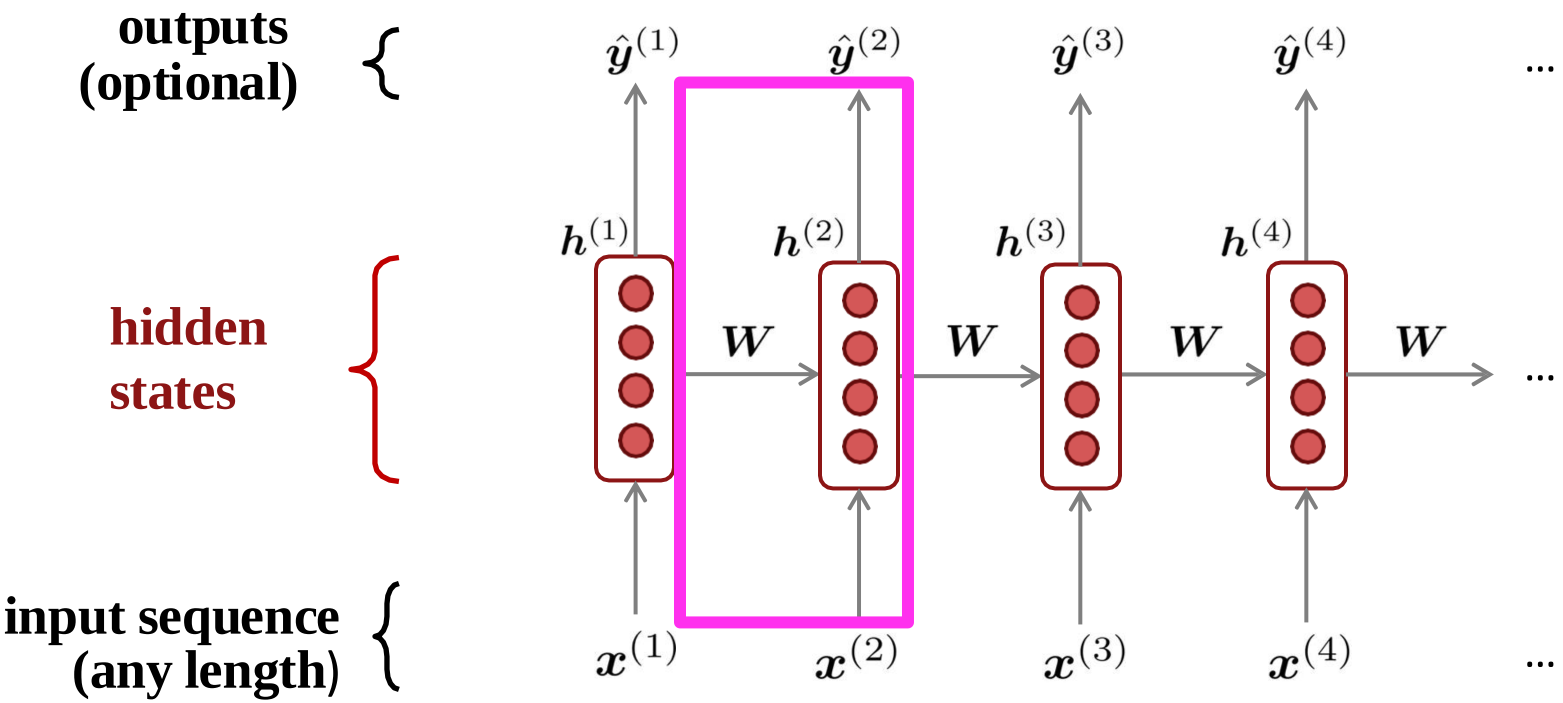
🔍 Key Concept:

Transformers like **BERT (2018), GPT-3 (2020), and ChatGPT (2022) outperformed RNNs**, leading to their decline in NLP.

Recurrent Neural Networks (RNN)

A family of neural architectures

Core idea: Apply the same weights W *repeatedly*




```
nn.RNN(input_size, hidden_size, num_layers, batch_first=True)
```

StanfordCS224n

A Simple RNN Language Model

output distribution

$$\hat{y}^{(t)} = \mathrm{softmax}\left(Uh^{(t)} + b_2\right) \in \mathbb{R}^{|V|}$$

hidden states

$$h^{(t)} = \sigma\left(W_h h^{(t-1)} + W_e e^{(t)} + b_1\right)$$

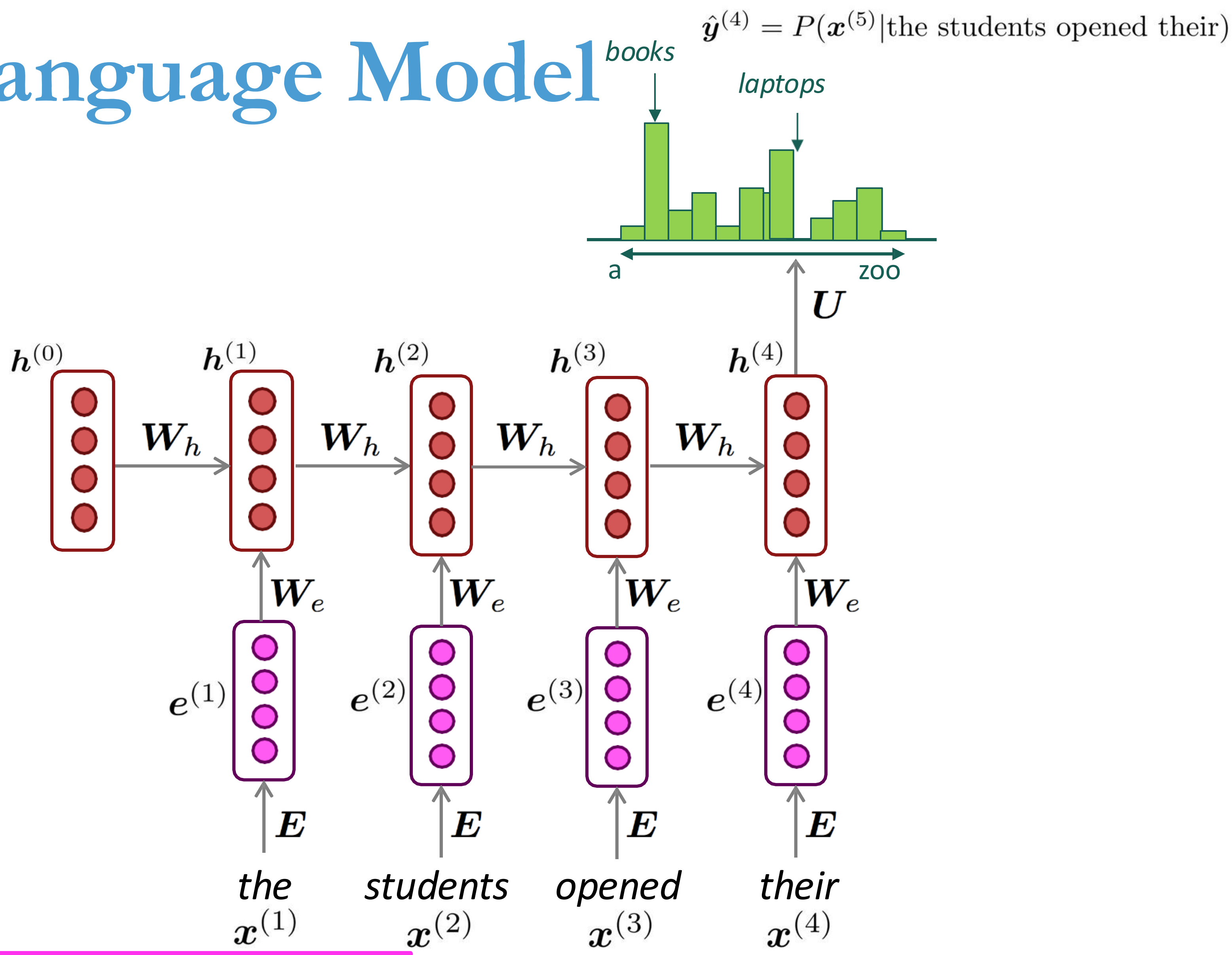
$h^{(0)}$ is the initial hidden state

word embeddings

$$e^{(t)} = Ex^{(t)}$$

words / one-hot vectors

$$x^{(t)} \in \mathbb{R}^{|V|}$$



***Note**: this input sequence could be much longer now!*

StanfordCS224n

RNN Language Models

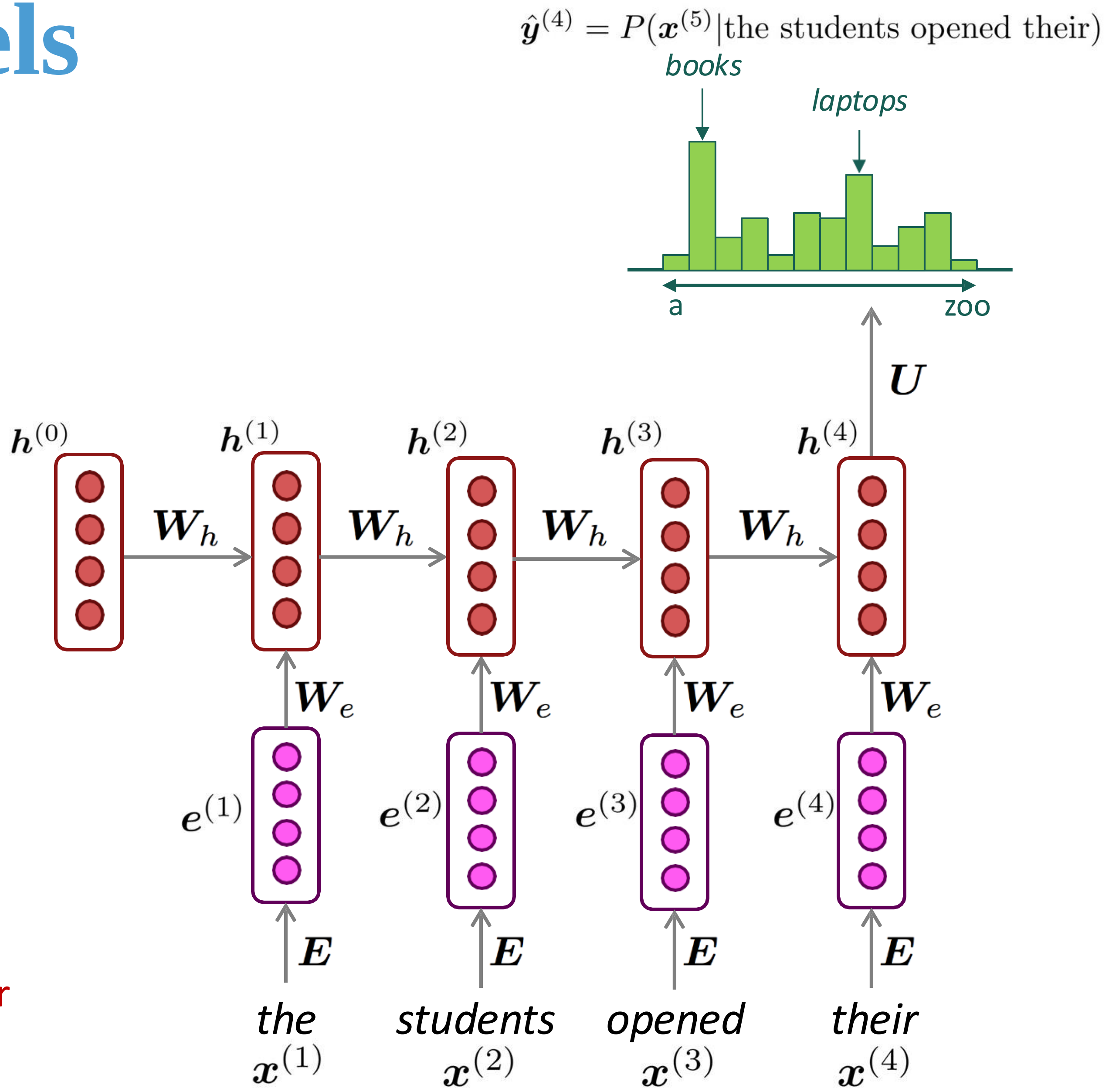
RNN **Advantages**:

- Can process any length input
- Computation for step *t* can (in theory) use information from many steps back
- Model size doesn't increase for longer input context
- Same weights applied on every timestep, so there is symmetry in how inputs are processed.

RNN **Disadvantages**:

- Recurrent computation is slow
- In practice, difficult to access information from many steps back

More on these later



StanfordCS224n

Training an RNN Language Model

- Prepare the dataset (tokenize text into words) and convert words to numerical tensors (word embeddings).
- Build an RNN-LM and compute output distribution $\hat{\boldsymbol{y}}^{(t)}$ *for every step t*.
- Loss function on step *t* is cross-entropy loss between predicted probability distribution $\hat{\boldsymbol{y}}^{(t)}$ and the true next word $\boldsymbol{y}^{(t)}$ (one-hot for $\boldsymbol{x}^{(t+1)}$):

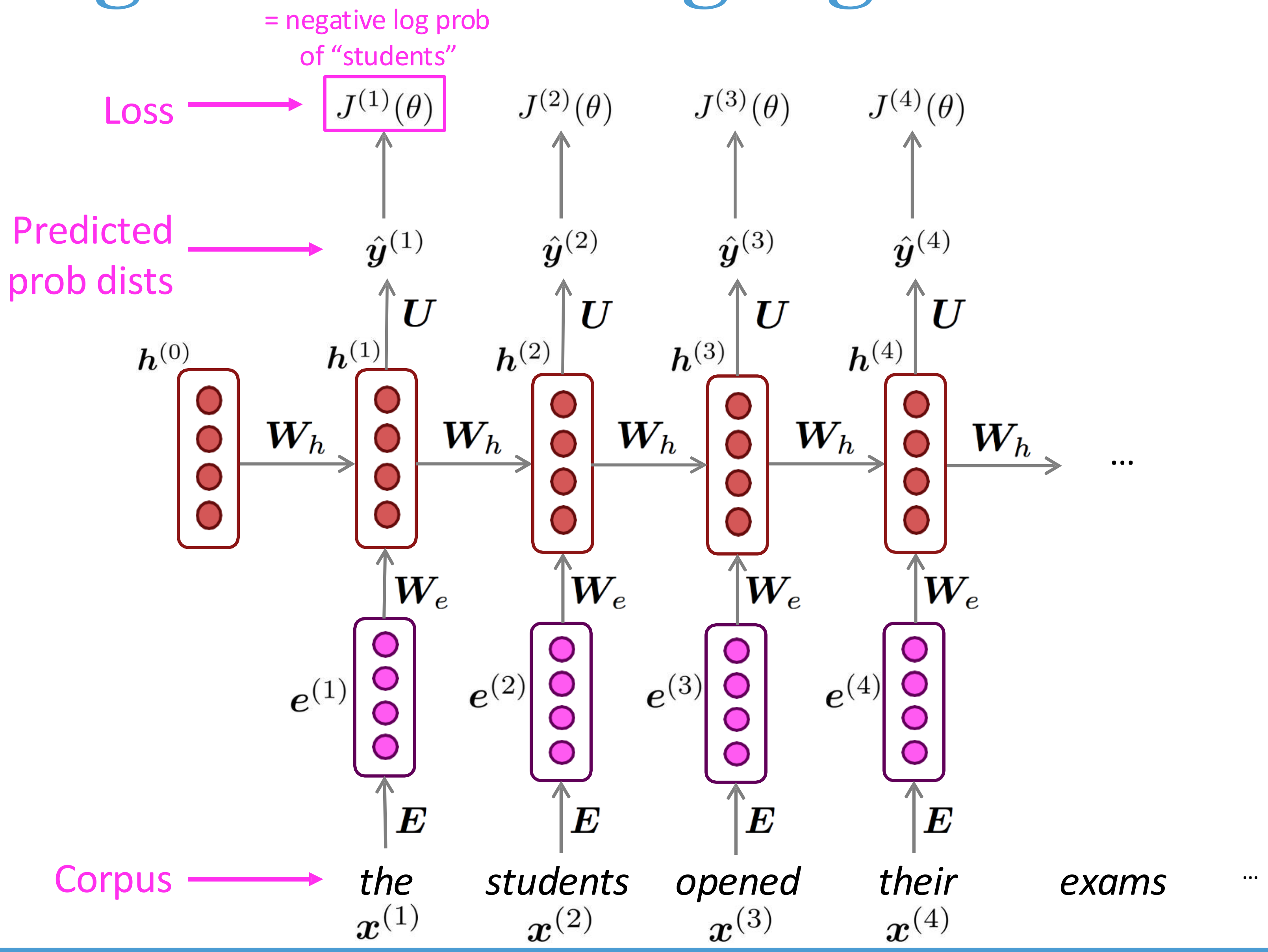
$$J^{(t)}(\theta) = CE(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) = -\sum_{w \in V} \boldsymbol{y}_w^{(t)} \log \hat{\boldsymbol{y}}_w^{(t)} = -\log \hat{\boldsymbol{y}}_{\boldsymbol{x}_{t+1}}^{(t)}$$

- Average this to get the overall loss for entire training set:

$$J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J^{(t)}(\theta) = \frac{1}{T}\sum_{t=1}^{T} -\log \hat{\boldsymbol{y}}_{\boldsymbol{x}_{t+1}}^{(t)}$$

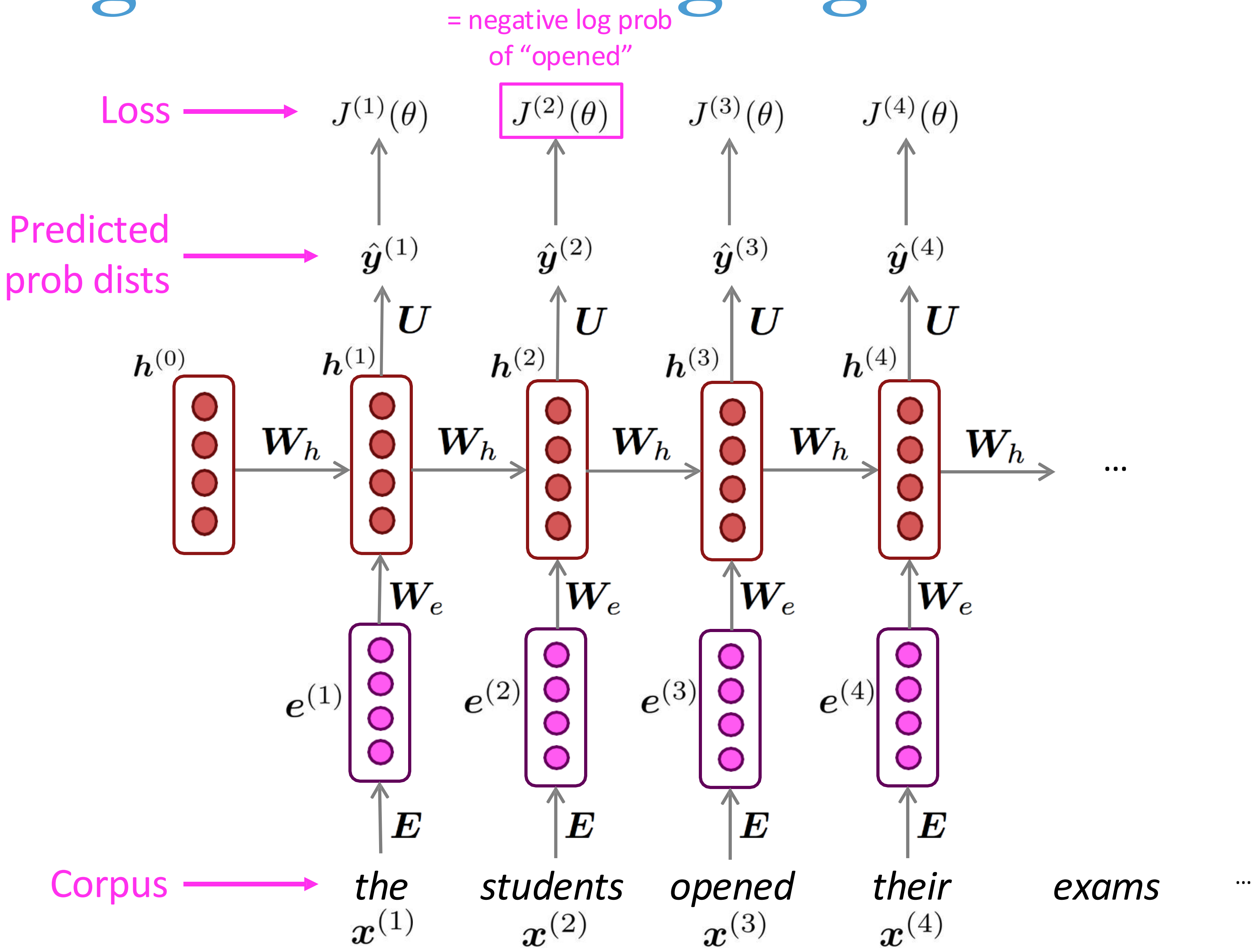
StanfordCS224n

Training an RNN Language Model



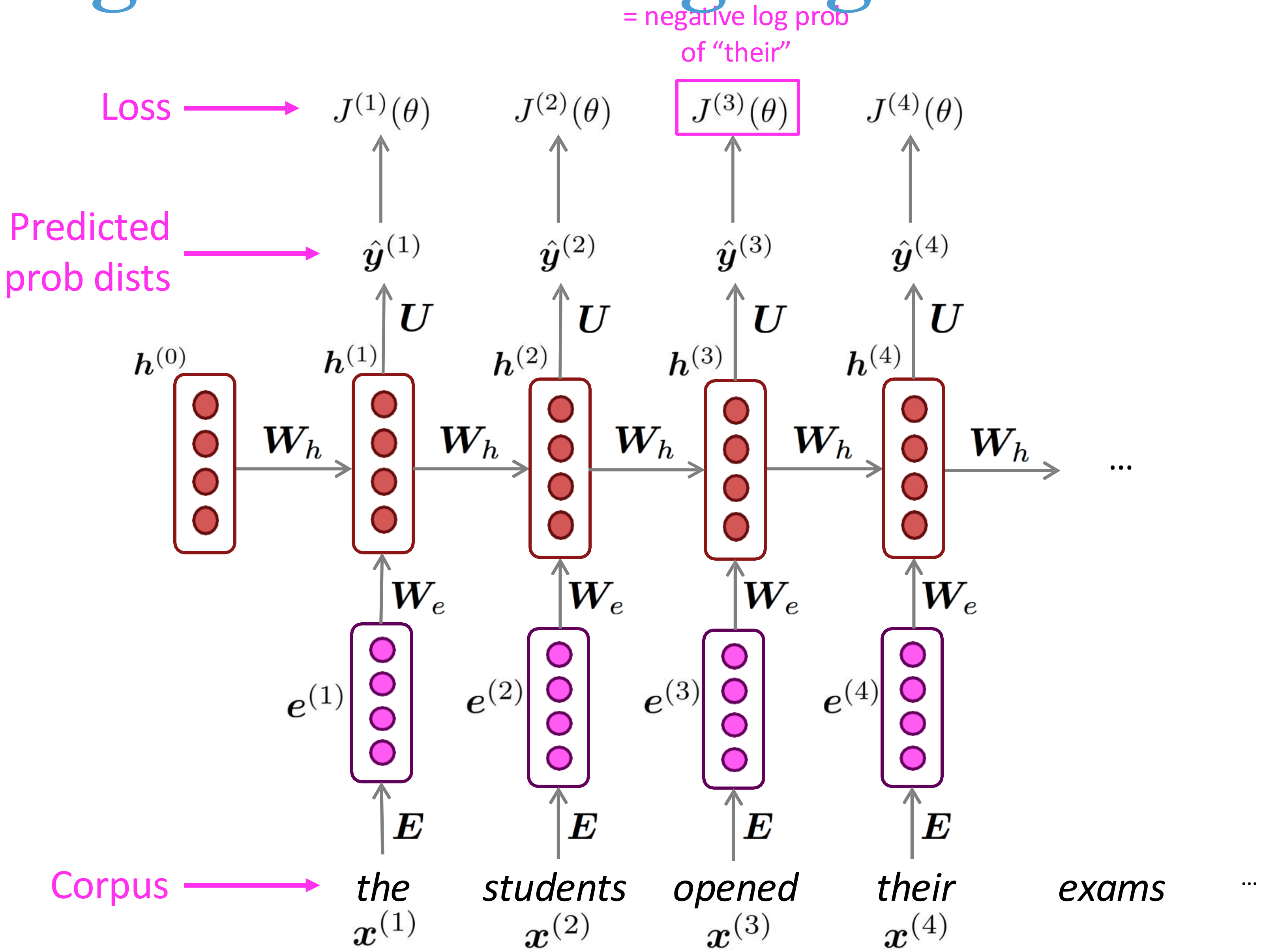
StanfordCS224n

Training an RNN Language Model



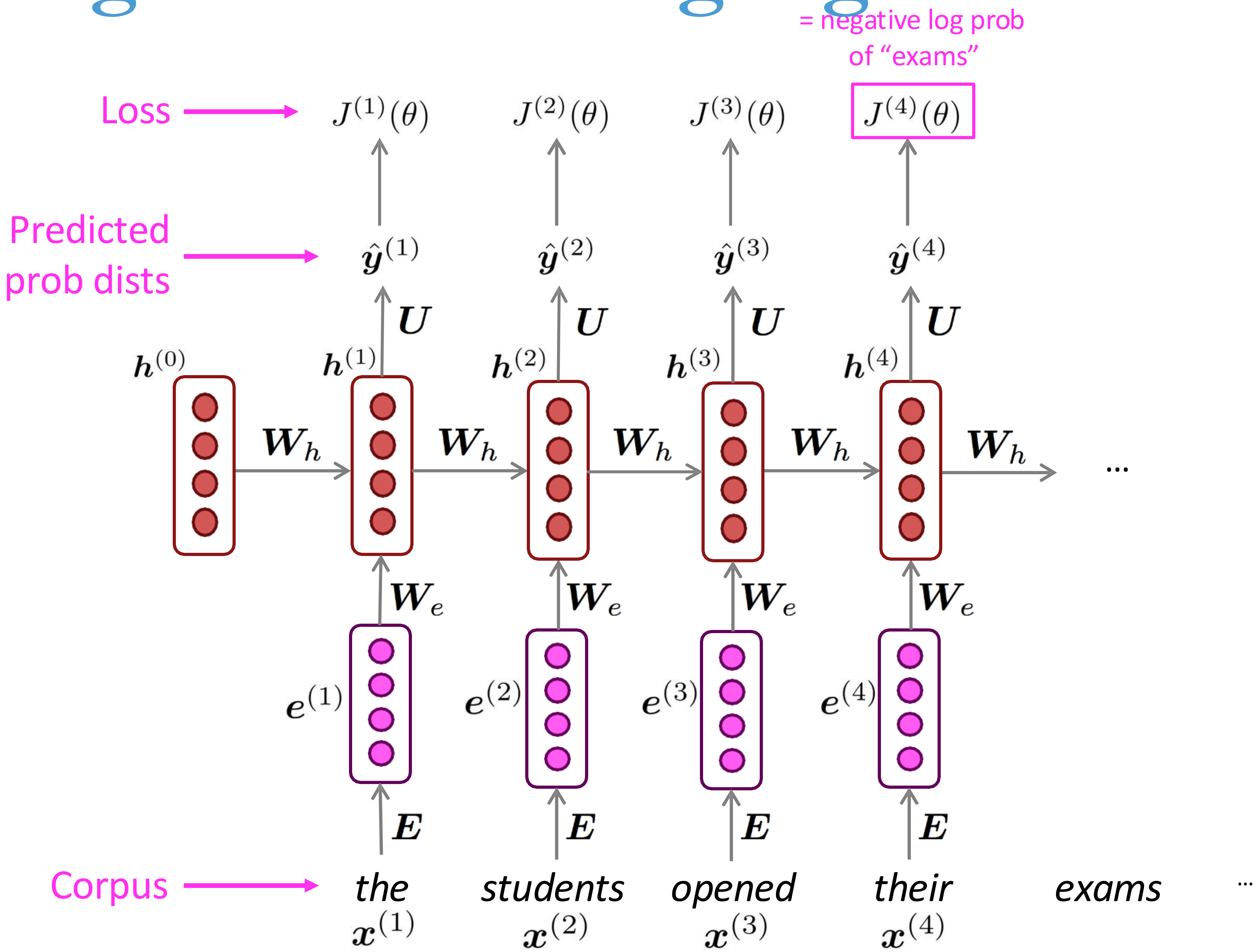
StanfordCS224n

Training an RNN Language Model



StanfordCS224n

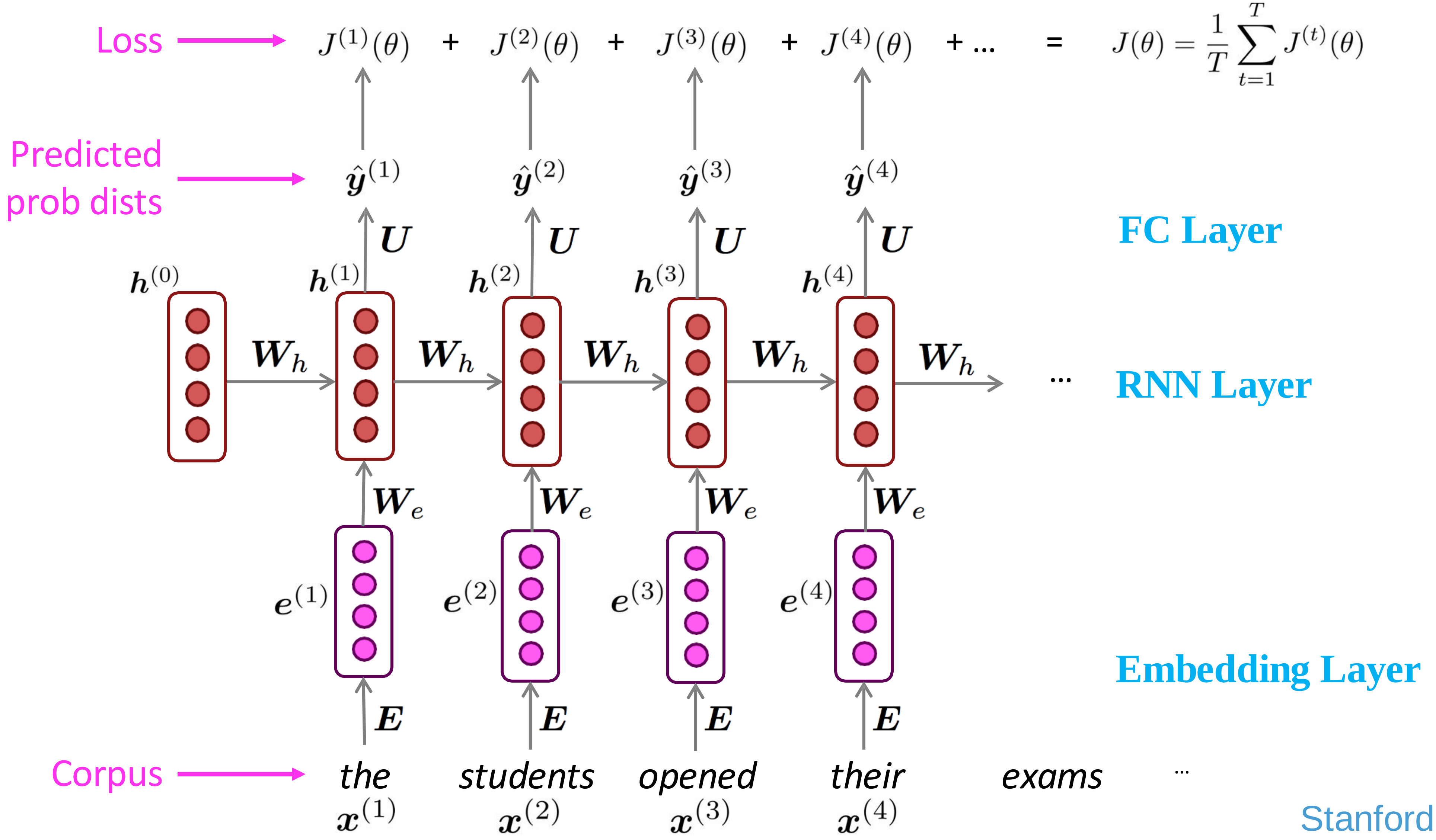
Training an RNN Language Model



StanfordCS224n

Training an RNN Language Model

"Teacher forcing"



StanfordCS224n

Dimensions and Parameters

Input dimension (D): Size of each input vector (e.g., 100 for Word2Vec embeddings).
Hidden dimension (H): Size of the hidden state vector (e.g., 64).
Batch size (B): Number of sequences processed in parallel.
Sequence length (S): Number of time steps (tokens) per sequence.
Output dimension (V): For example, vocabulary size in language modeling or the number of classes in classification.

1. **Embedding Layer**
   Maps input tokens to dense vectors (dimension: D).
2. **RNN Layer**
   Processes the sequence of embeddings and produces hidden states $h^{(t)}$ at each time step.

   Equation:

   $$a_t = W_e\,\mathbf{e}_t + b_e + W_h\,h^{(t-1)} + b_h$$

   $$h^{(t)} = \tanh(a_t)$$
3. **Fully Connected (FC) Layer**
   Maps hidden state $h^{(t)}$ to output $y^{(t)}$ (dimension: V).

   Equation:

   $$y^{(t)} = W_{hy}\,h^{(t)} + b_y$$

Components:

- **Input-to-Hidden:** $W_e \in \mathbb{R}^{H\times D} \to \#$ Parameters: $H \times D$
- **Hidden-to-Hidden:** $W_h \in \mathbb{R}^{H\times H} \to \#$ Parameters: $H \times H$
- **Biases:** $b_e \in \mathbb{R}^H$ and $b_h \in \mathbb{R}^H \to$ Total: $2H$

Total RNN Cell Parameters:

$$(H \times D) + (H \times H) + 2H$$

Dimensions:

- $h^{(t)} \in \mathbb{R}^H$; $\quad W_{hy} \in \mathbb{R}^{V\times H}$
- $b_y \in \mathbb{R}^V$; $\quad y^{(t)} \in \mathbb{R}^V$ (e.g., logits over vocabulary)

Parameter Count:

$$\text{FC Parameters} = (V \times H) + V = V \times (H + 1)$$

Training Challenges

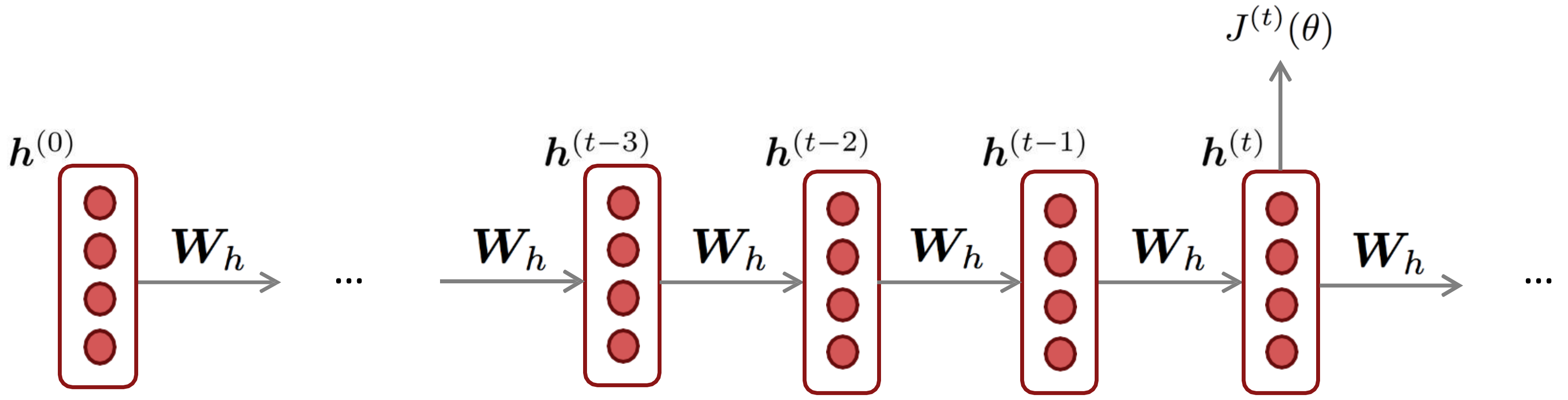
- **Memory Constraints**: Storing all word embeddings, gradients, and activations requires enormous memory.
- **Computational Cost**: Performing **backpropagation over the entire dataset** in a single step is impractical.
- **Batching is Required**: Instead of processing all data at once, models use **mini-batches** to update weights efficiently.

$$J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J^{(t)}(\theta)$$

Challenges	Solution	
Too much memory usage	✅ Mini-batch training	Divide the **entire dataset into mini-batches** (e.g., **batch size = 32**).
Long sequences overflow memory	✅ Truncated BPTT (TBPTT)	**Truncate** the sequence into smaller **sub-sequences** (e.g., **20 time steps** at a time).
Exploding gradients	✅ Gradient clipping	Clip gradients to a maximum norm (**e.g., 5**).
Slow training	✅ Efficient batching and parallelization	

$$g = g \times \frac{C}{\max(||g||, C)}$$

Backpropagation for RNNs



Question: What's the derivative of $J^{(t)}(\theta)$ w.r.t. the repeated weight matrix $\boldsymbol{W}_h$?

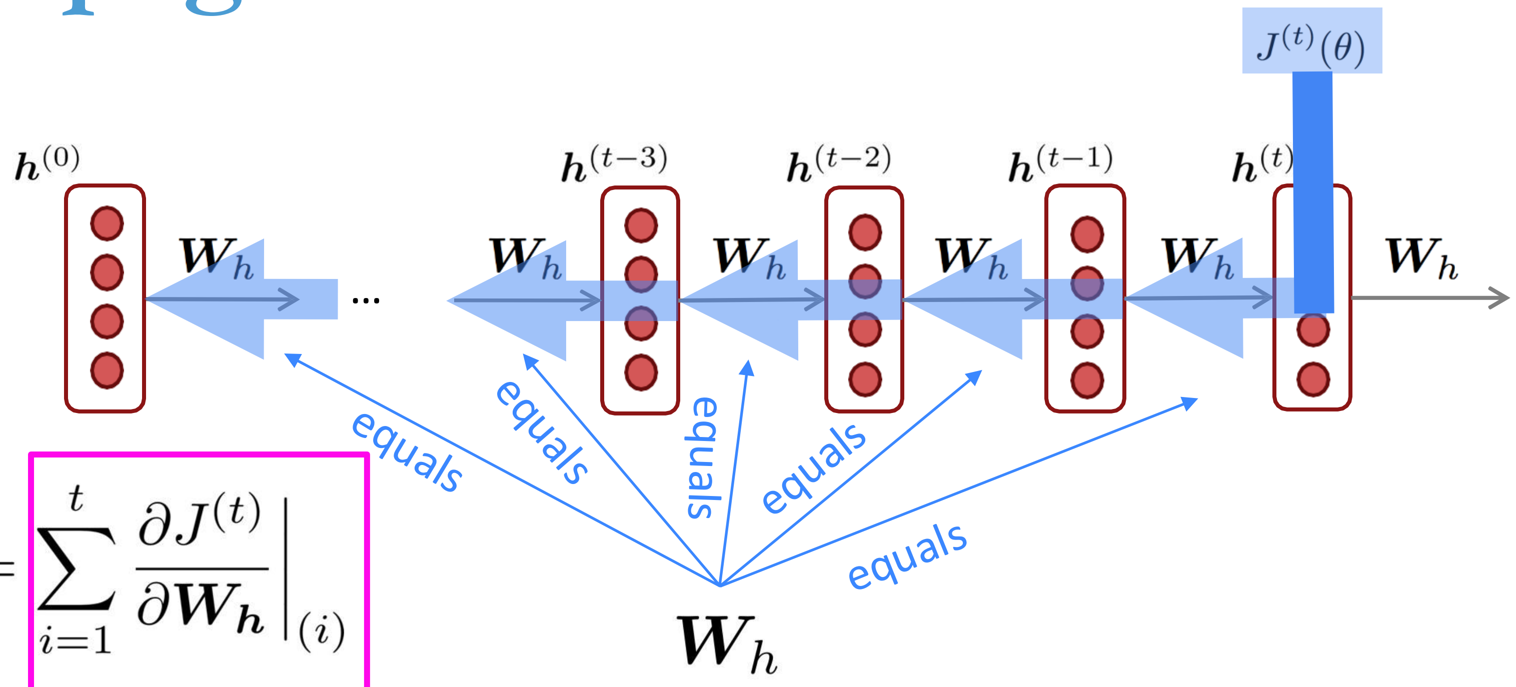
Answer: $$\frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}} = \sum_{i=1}^{t} \left. \frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}} \right|_{(i)}$$

"The gradient w.r.t. a repeated weight is the sum of the gradient w.r.t. each time it appears"

Why?

StanfordCS224n

Backpropagation for RNNs



In practice, often "truncated" after ~20 timesteps for training efficiency reasons

$$\frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}} = \boxed{\sum_{i=1}^{t} \left. \frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}} \right|_{(i)}}$$

Question: How do we calculate this?

Answer: Backpropagate over timesteps *i* = *t*, … ,0, summing gradients as you go. This algorithm is called **"backpropagation through time"** [Werbos, P.G., 1988, *Neural Networks* **1**, and others]

Apply the multivariable chain rule:

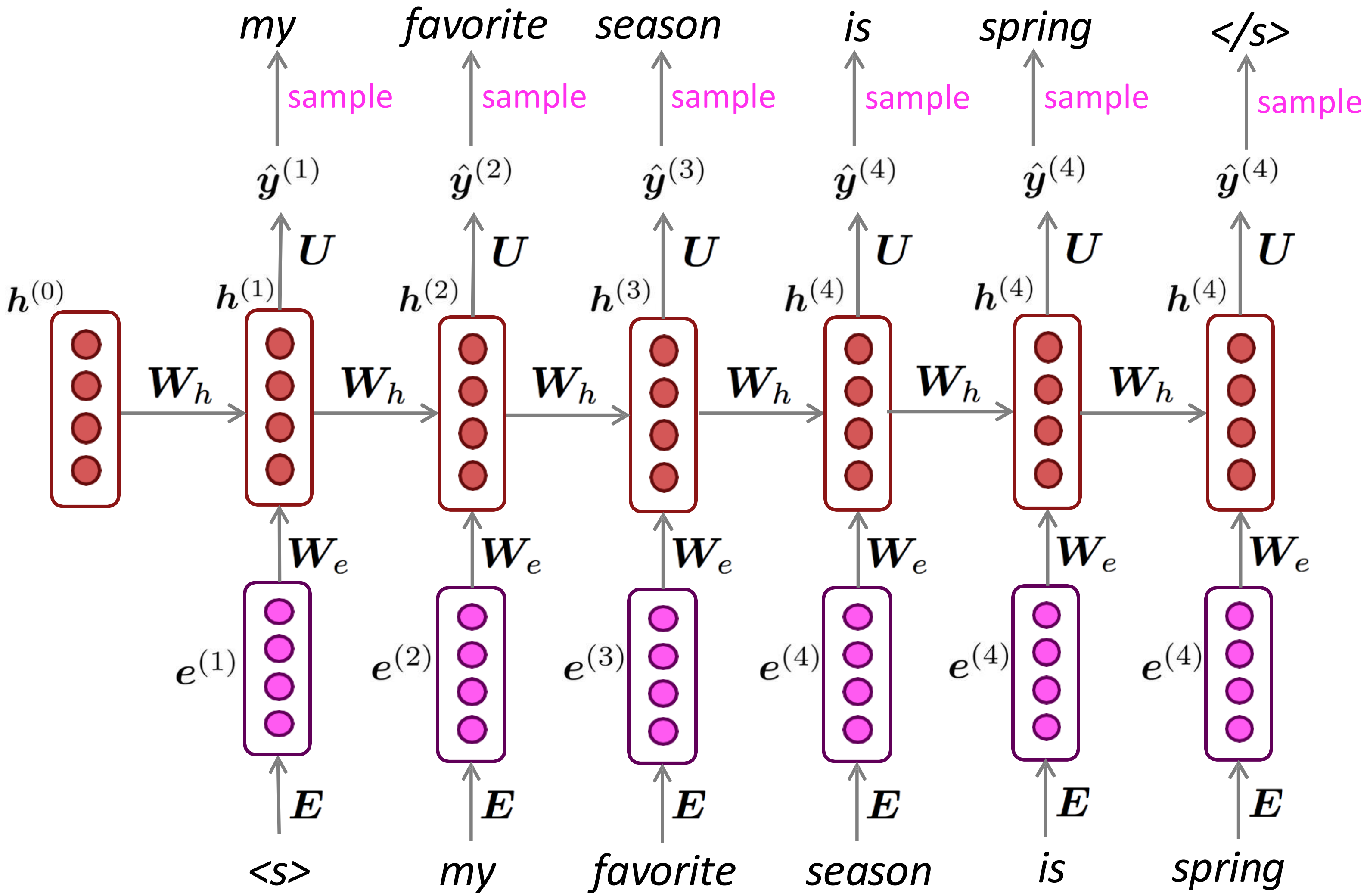
$$\frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h} = \sum_{i=1}^{t} \left. \frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h} \right|_{(i)} \underbrace{\boxed{\frac{\partial \boldsymbol{W}_h |_{(i)}}{\partial \boldsymbol{W}_h}}}_{=1}$$

$$= \sum_{i=1}^{t} \left. \frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h} \right|_{(i)}$$

StanfordCS224n

Generating roll outs

Just like an n-gram Language Model, you can use a RNN Language Model to generate text by repeated sampling. Sampled output becomes next step's input.



StanfordCS224n

Train an RNN Language Model

Tasks 2 & 3: *How to train an RNN-LM on this paragraph and then generate text in that style?*

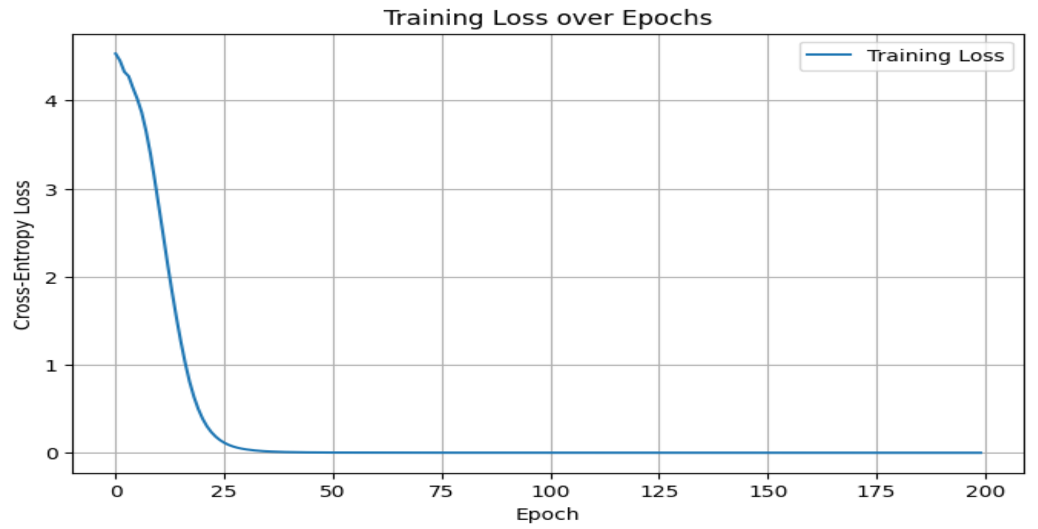
Obama "*I stand here today humbled by the task before us, grateful for the trust you have bestowed, mindful of the sacrifices borne by our ancestors. I thank President Bush for his service to our nation, as well as the generosity and cooperation he has shown throughout this transition. Forty-four Americans have now taken the presidential oath. The words have been spoken during rising tides of prosperity and the still waters of peace. Yet, every so often the oath is taken amidst gathering clouds and raging storms. At these moments, America has carried on not simply because of the skill or vision of those in high office, but because We the People have remained faithful to the ideals of our forbearers, and true to our founding documents.* "

```python
# ================================
# 4. Define the RNN Language Model in PyTorch
# ================================
class RNNLanguageModel(nn.Module):
def __init__(self, vocab_size, embedding_dim, hidden_size,
num_layers, embedding_matrix):
super(RNNLanguageModel, self).__init__()
self.embedding = nn.Embedding(num_embeddings=vocab_size,
embedding_dim=embedding_dim)
self.embedding.weight.data.copy_(embedding_matrix)
# Optionally freeze the embeddings if desired:
# self.embedding.weight.requires_grad = False
self.rnn = nn.RNN(input_size=embedding_dim,
hidden_size=hidden_size, num_layers=num_layers,
nonlinearity='tanh', batch_first=True)
# Fully connected layer to map hidden state to vocabulary
logits
self.fc = nn.Linear(hidden_size, vocab_size)
def forward(self, x, h0):
# x: (batch_size, seq_len)
x_embed = self.embedding(x) # (batch_size, seq_len,
embedding_dim)
out, hn = self.rnn(x_embed, h0) # out: (batch_size, seq_len,
hidden_size)
logits = self.fc(out) # (batch_size, seq_len, vocab_size)
return logits, hn

hidden_size = 64
num_layers = 1

model = RNNLanguageModel(vocab_size, embedding_dim,
hidden_size, num_layers, embedding_matrix)
```

```python
# ================================
# 5. Define Loss Function, Optimizer, and a Learning Rate
Scheduler
# ================================
criterion = nn.CrossEntropyLoss() # for next-word prediction
optimizer = optim.Adam(model.parameters(), lr=0.01)
scheduler = optim.lr_scheduler.StepLR(optimizer,
step_size=50, gamma=0.5)

# ================================
# 6. Training (Fitting) Process
# ================================
num_epochs = 200
model.train()
loss_history = []

batch_size, seq_len = input_seq.shape
for epoch in range(1, num_epochs + 1):
h0 = torch.zeros(num_layers, batch_size, hidden_size)
optimizer.zero_grad()
# Forward pass: get logits over vocabulary
logits, hn = model(input_seq, h0)
# logits shape: (1, seq_len, vocab_size)
# Reshape logits and target for loss computation
logits = logits.view(-1, vocab_size) # shape:
(batch_size*seq_len, vocab_size)
targets = target_seq.view(-1) # shape: (batch_size*seq_len)
loss = criterion(logits, targets)
loss.backward()
optimizer.step()
scheduler.step() # update learning rate
loss_history.append(loss.item())

print("\n--- Training Completed ---")
```

```python
# ================================
# 7. Text Generation Based on the Trained Model
# ================================
def generate_text(model, seed_text, length, word_to_idx,
idx_to_word, hidden_size, num_layers):

model.eval()
seed_tokens = tokenize(seed_text)
seed_indices = [word_to_idx.get(word, 0) for word in
seed_tokens] # default to 0 if not found
input_seq = torch.tensor([seed_indices], dtype=torch.long)
h = torch.zeros(num_layers, 1, hidden_size)
generated_indices = seed_indices.copy()

# Generate tokens one by one
for _ in range(length):
logits, h = model(input_seq, h)
last_logits = logits[:, -1, :]
next_token = last_logits.argmax(dim=1).item()
generated_indices.append(next_token)
# Prepare input for next iteration: the newly generated token
becomes the next input.
input_seq = torch.tensor([[next_token]], dtype=torch.long)
generated_words = [idx_to_word[idx] for idx in
generated_indices]
return " ".join(generated_words)
# Generate text from a seed
seed_text = "Obama"
generated_text = generate_text(model, seed_text, length=50,
word_to_idx=word_to_idx,
idx_to_word=idx_to_word, hidden_size=hidden_size,
num_layers=num_layers)
print("\n--- Generated Text ---")
print(generated_text)
```

```
--- RNN Language Model Architecture ---
RNNLanguageModel(  (embedding): Embedding(93, 100)  (rnn): RNN(100, 64, batch_first=True)  (fc): Linear(in_features=64, out_features=93, bias=True) )
--- Loss Function and Optimizer ---
Loss Function: CrossEntropyLoss()
Optimizer: Adam ( Parameter Group 0 amsgrad: False betas: (0.9, 0.999) capturable: False differentiable: False eps: 1e-08 foreach: None fused: None initial_lr: 0.01 lr: 0.01 maximize: False weight_decay: 0 ) Scheduler: <torch.optim.lr_scheduler.StepLR object at 0x7fc1615ae210>
--- Training Started ---
Epoch 1/200 | Loss: 4.5318 Epoch 20/200 | Loss: 0.5003 Epoch 40/200 | Loss: 0.0104 Epoch 60/200 | Loss: 0.0037 Epoch 80/200 | Loss: 0.0029 Epoch 100/200 | Loss: 0.0026 Epoch 120/200 | Loss: 0.0024 Epoch 140/200 | Loss: 0.0023 Epoch 160/200 | Loss: 0.0022 Epoch 180/200 | Loss: 0.0021 Epoch 200/200 | Loss: 0.0020
--- Training Completed ---
```



Outputs

- --- Generated Text ---

bush for his service to our nation as well as the generosity and cooperation he has shown throughout this transition forty four americans have now taken the presidential oath the words have been spoken during rising tides of prosperity and the still waters of peace yet every so often the oath

--- Generated Text ---

america has carried on not simply because of the skill or vision of those in high office but because we the people have remained faithful to the ideals of our forbearers and true to our founding documents for the trust you have bestowed mindful of the sacrifices borne by our ancestors

Evaluating Language Models

- **Perplexity:** Measures how well a model predicts a sample.
- **BLEU, ROUGE, METEOR:** Compare generated text to reference texts.
- **Accuracy and F1 Score:** Used in tasks with classification elements.
- **Human Evaluation:** Judges quality, fluency, and relevance.
- **Task-Specific Metrics:** Tailored metrics for particular applications.
  - **Fluency:** Is the generated text grammatically correct and natural?
  - **Coherence:** Does the text flow logically from one sentence to the next?
  - **Relevance:** How well does the generated text answer a prompt or capture key details?
  - **Engagement and Creativity:** Particularly important in creative writing or dialogue systems.

Normalized by.
number of words

$$\text{perplexity} = \underbrace{\prod_{t=1}^{T} \left(\frac{1}{P_{\text{LM}}(\boldsymbol{x}^{(t+1)} | \, \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)})} \right)^{1/T}}_{\text{Inverse probability of corpus, according to Language Model}} = \prod_{t=1}^{T} \left(\frac{1}{\hat{\boldsymbol{y}}^{(t)}_{\boldsymbol{x}_{t+1}}} \right)^{1/T} = \exp\left(\frac{1}{T} \sum_{t=1}^{T} -\log \hat{\boldsymbol{y}}^{(t)}_{\boldsymbol{x}_{t+1}} \right) = \exp(J(\theta))$$

$\boxed{\text{Lower perplexity is better!}}$

RNNs greatly improved perplexity

n-gram model →

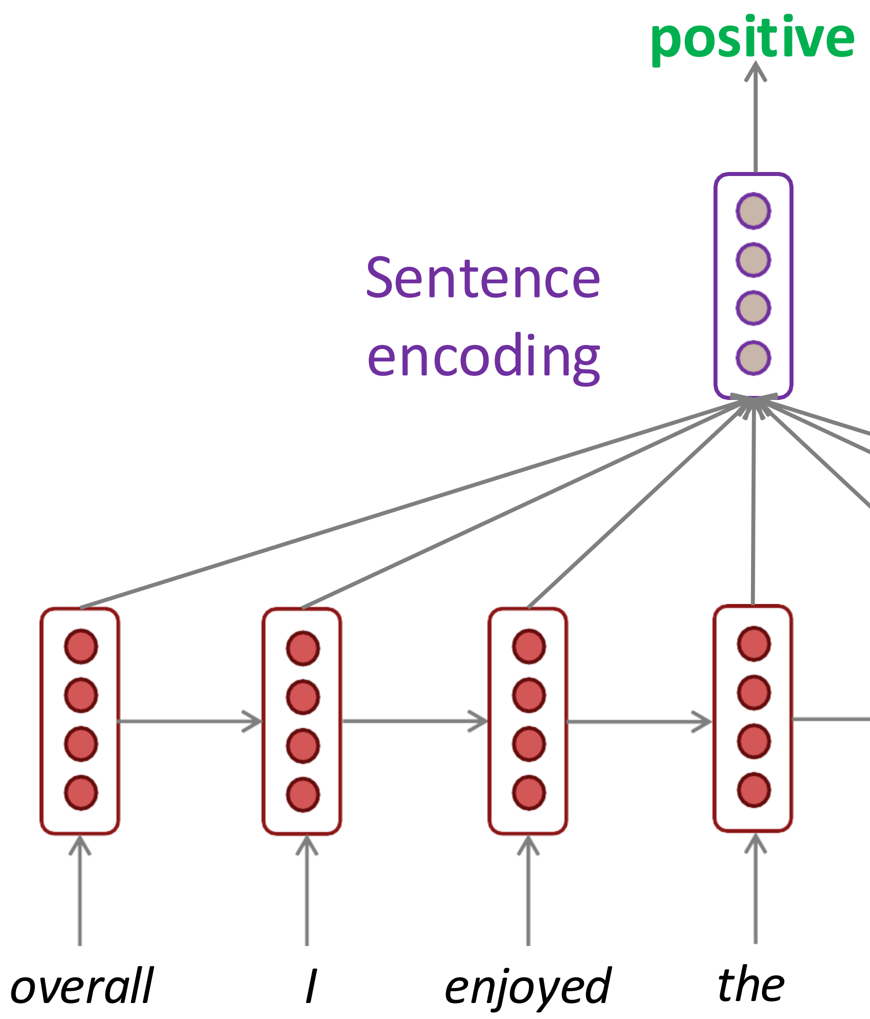
Increasingly complex RNNs

Model	Perplexity
Interpolated Kneser-Ney 5-gram (Chelba et al., 2013)	67.6
RNN-1024 + MaxEnt 9-gram (Chelba et al., 2013)	51.3
RNN-2048 + BlackOut sampling (Ji et al., 2015)	68.3
Sparse Non-negative Matrix factorization (Shazeer et al., 2015)	52.9
LSTM-2048 (Jozefowicz et al., 2016)	43.7
2-layer LSTM-8192 (Jozefowicz et al., 2016)	30
Ours small (LSTM-2048)	43.9
Ours large (2-layer LSTM-2048)	39.8

Perplexity improves (lower is better)

Source: https://research.fb.com/building-an-efficient-neural-language-model-over-a-billion-words/

Sentiment Classification



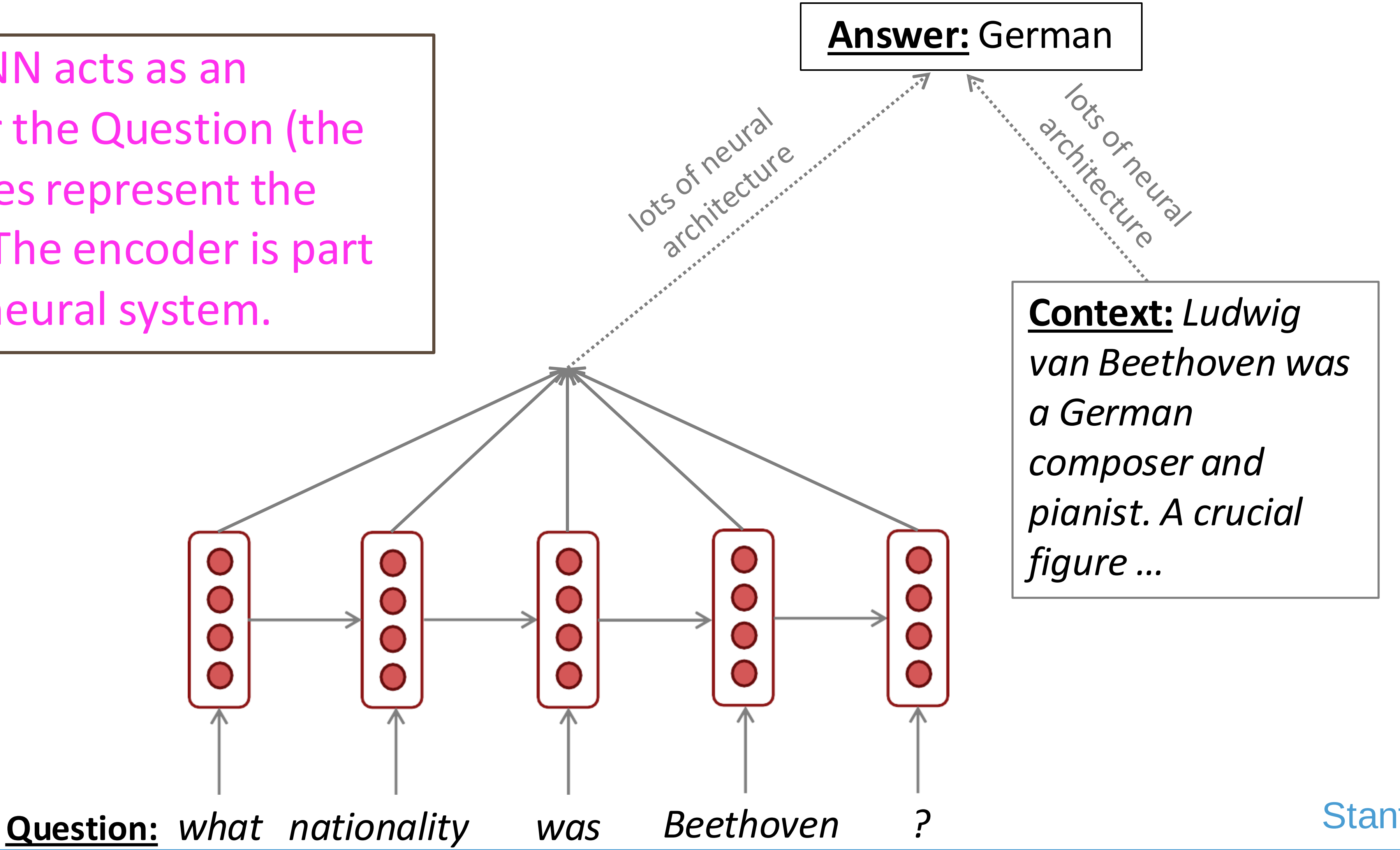
How to compute sentence encoding?

Usually better: Take element-wise max or mean of all hidden states

StanfordCS224n

Question Answering

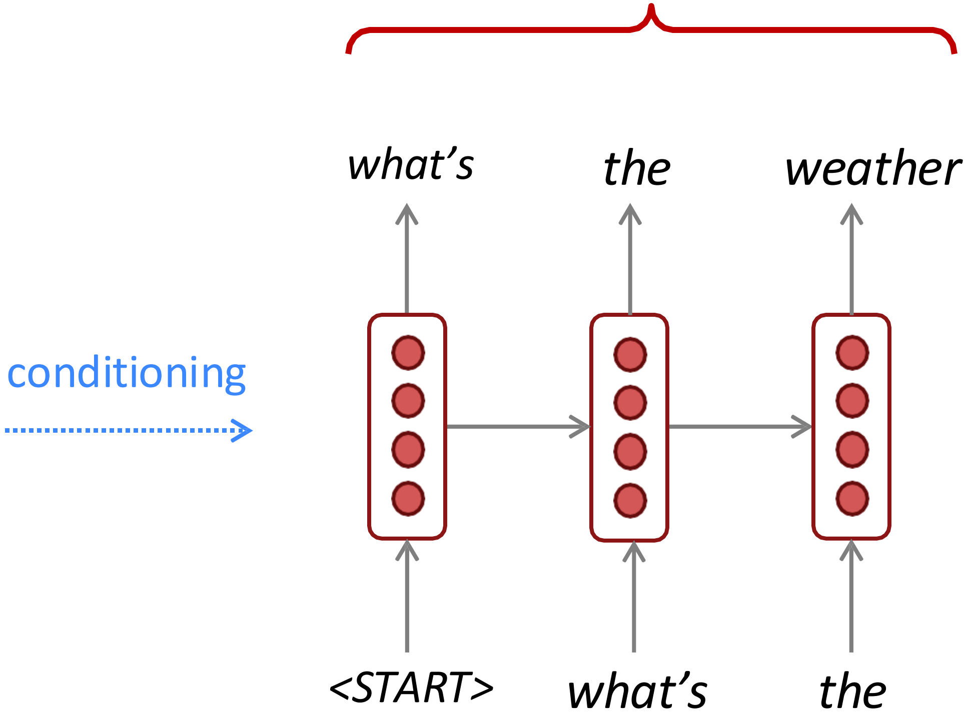
Here the RNN acts as an **encoder** for the Question (the hidden states represent the Question). The encoder is part of a larger neural system.



StanfordCS224n

Speech Recognition

Input (audio)



This is an example of a *conditional language model*.

StanfordCS224n

Example: Classifying Names

```python
import torch.nn as nn
import torch.nn.functional as F

class CharRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(CharRNN, self).__init__()

        self.rnn = nn.RNN(input_size, hidden_size)
        self.h2o = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, line_tensor):
        rnn_out, hidden = self.rnn(line_tensor)
        output = self.h2o(hidden[0])
        output = self.softmax(output)

        return output
```

```python
n_hidden = 128
rnn = CharRNN(n_letters, n_hidden, len(alldata.labels_uniq))
print(rnn)
```

This tutorial serves as an introduction to **sequence modeling with RNNs** and shows how character-level information can be used for **text classification**.

- **Input:** A name (e.g., "Schmidt") is provided as a sequence of characters.
- **Processing:** Each character is converted into a numerical tensor (one-hot encoding or embeddings). The RNN processes the character sequence, updating its hidden state at each step.
- **Output:** The model predicts the **nationality/language** of the name (e.g., "German").

This is from the official PyTorch tutorial: https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

Example: Classifying Names

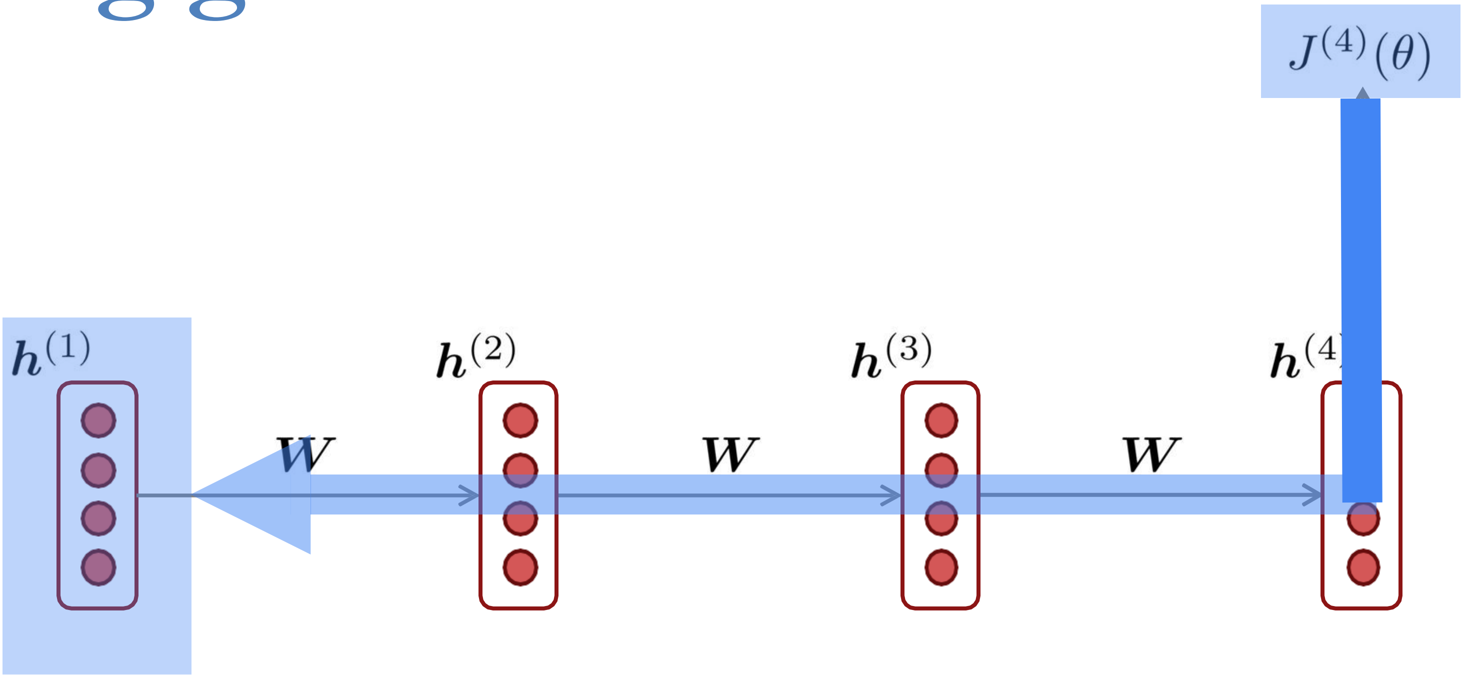
```python
import random
import numpy as np

def train(rnn, training_data, n_epoch = 10, n_batch_size = 64,
report_every = 50, learning_rate = 0.2, criterion = nn.NLLLoss()):
    """
    Learn on a batch of training_data for a specified number of
    iterations and reporting thresholds
    """
    # Keep track of losses for plotting
    current_loss = 0
    all_losses = []
    rnn.train()
    optimizer = torch.optim.SGD(rnn.parameters(), lr=learning_rate)

    start = time.time()
    print(f"training on data set with n = {len(training_data)}")

    for iter in range(1, n_epoch + 1):
        rnn.zero_grad() # clear the gradients

        batches = list(range(len(training_data)))
        random.shuffle(batches)
        batches = np.array_split(batches, len(batches) //n_batch_size )
```

```python
        for idx, batch in enumerate(batches):
            batch_loss = 0
            for i in batch: #for each example in this batch
                (label_tensor, text_tensor, label, text) = training_data[i]
                output = rnn.forward(text_tensor)
                loss = criterion(output, label_tensor)
                batch_loss += loss

            # optimize parameters
            batch_loss.backward()
            nn.utils.clip_grad_norm_(rnn.parameters(), 3)
            optimizer.step()
            optimizer.zero_grad()

            current_loss += batch_loss.item() / len(batch)
        all_losses.append(current_loss / len(batches) )
        if iter % report_every == 0:
            print(f"{iter} ({iter / n_epoch:.0%}): = {all_losses[-1]}")
        current_loss = 0

    return all_losses

all_losses = train(rnn, train_set, n_epoch=27, learning_rate=0.15)
```

File: char_rnn_classification_tutorial.ipynb
This is from the official PyTorch tutorial: https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

Content

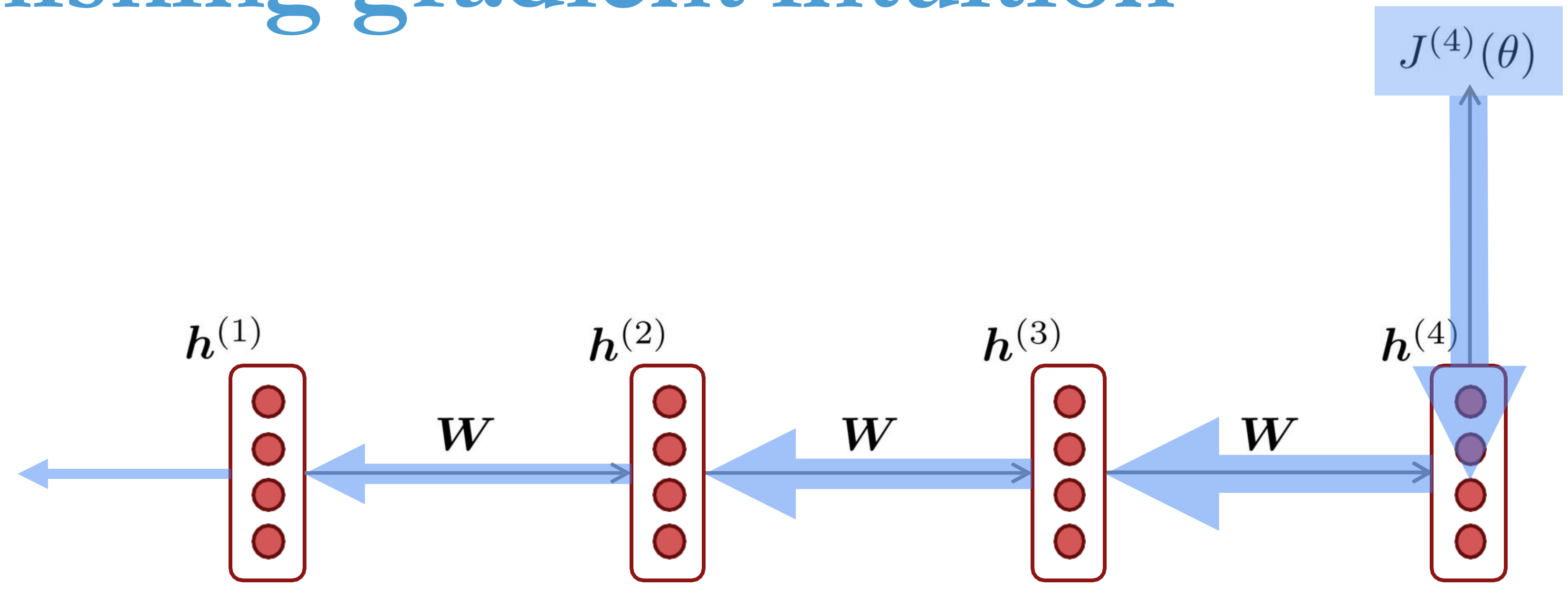
Vanishing gradient intuition



$$\frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(1)}} = \ ?$$

- This decay makes it difficult for RNNs to learn long-term dependencies.
- Empirical evidence shows rapid gradient norm decay.

Vanishing gradient intuition



$$\frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(1)}} = \boxed{\frac{\partial \boldsymbol{h}^{(2)}}{\partial \boldsymbol{h}^{(1)}}} \times \boxed{\frac{\partial \boldsymbol{h}^{(3)}}{\partial \boldsymbol{h}^{(2)}}} \times \boxed{\frac{\partial \boldsymbol{h}^{(4)}}{\partial \boldsymbol{h}^{(3)}}} \times \frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(4)}}$$

What happens if these are small?

Vanishing gradient problem:
When these are small, the gradient signal gets smaller and smaller as it backpropagates further

Vanishing gradient intuition

- Recall: $$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}_1\right)$$
- What if σ were the identity function, $\sigma(x) = x$?

$$\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} = \mathrm{diag}\left(\sigma'\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}_1\right)\right)\boldsymbol{W}_h \qquad \text{(chain rule)}$$

$$= \boldsymbol{I}\ \boldsymbol{W}_h = \boldsymbol{W}_h$$

- Consider the gradient of the loss $J^{(i)}(\theta)$ on step i, with respect to the hidden state $\boldsymbol{h}^{(j)}$ on some previous step j. *Let* $\ell = i - j$

$$\frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(j)}} = \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \prod_{j<t\leq i} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} \qquad \text{(chain rule)}$$

$$= \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \prod_{j<t\leq i} \boldsymbol{W}_h = \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \boxed{\boldsymbol{W}_h^{\ell}} \qquad \left(\text{value of } \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}}\right)$$

If W_h is "small", then this term gets exponentially problematic as ℓ becomes large

StanfordCS224n

Source: "On the difficulty of training recurrent neural networks", Pascanu et al, 2013. http://proceedings.mlr.press/v28/pascanu13.pdf (and supplemental materials), at http://proceedings.mlr.press/v28/pascanu13-supp.pdf

Vanishing gradient intuition

- What's wrong with $\boldsymbol{W}_h^\ell$?
- Consider if the eigenvalues of $\boldsymbol{W}_h$ are $\overbrace{\text{all less than}}^{\text{sufficient but not necessary}}$ 1:

$$\lambda_1, \lambda_2, \ldots, \lambda_n < 1$$
$$\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_n \quad \text{(eigenvectors)}$$

- We can write $\frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \boldsymbol{W}_h^\ell$ using the eigenvectors of $\boldsymbol{W}_h$ as a basis:

$$\frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \boldsymbol{W}_h^\ell = \sum_{i=1}^{n} c_i \boxed{\lambda_i^\ell} \boldsymbol{q}_i \approx \boldsymbol{0} \text{ (for large } \ell)$$

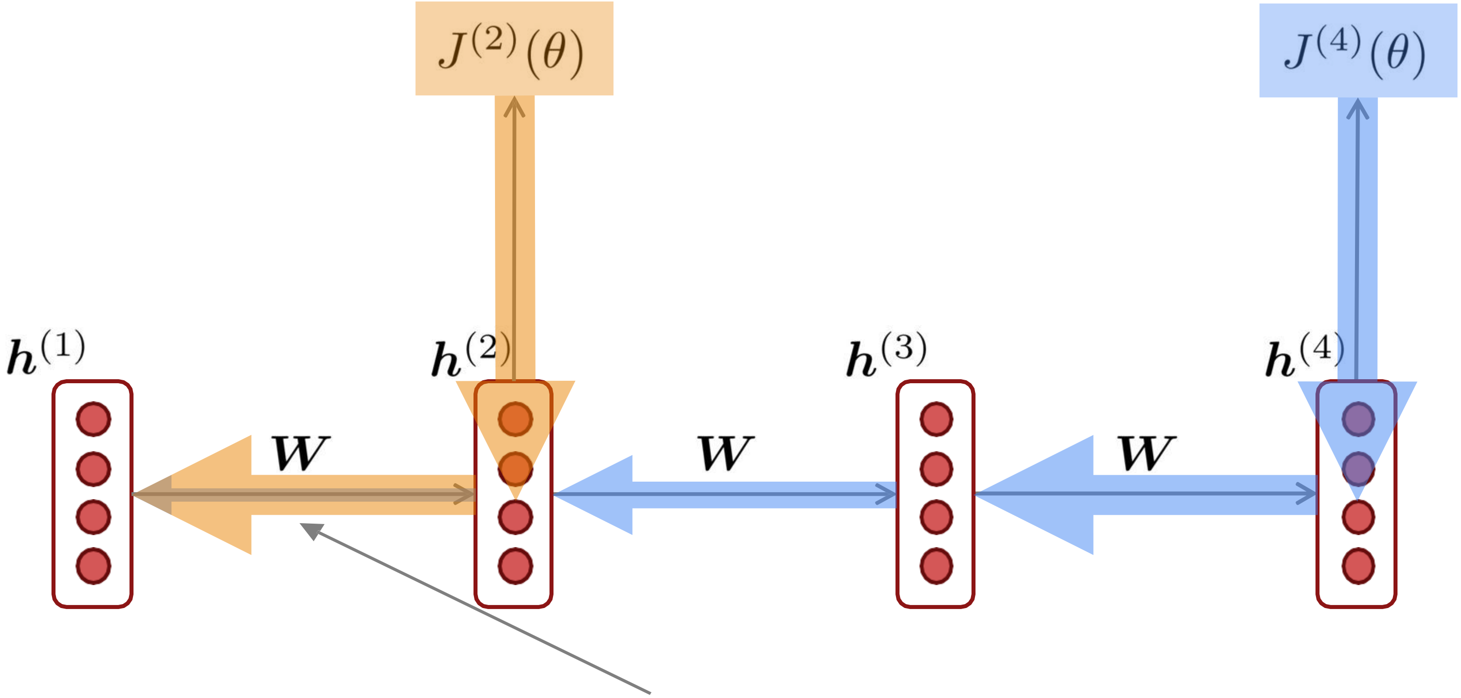
Approaches 0 as ℓ grows, so gradient vanishes

- What about nonlinear activations σ (i.e., what we use?)
  - Pretty much the same thing, except the proof requires $\lambda_i < \gamma$ for some γ dependent on dimensionality and σ

StanfordCS224n

Source: "On the difficulty of training recurrent neural networks", Pascanu et al, 2013. http://proceedings.mlr.press/v28/pascanu13.pdf (and supplemental materials), at http://proceedings.mlr.press/v28/pascanu13-supp.pdf

Why is vanishing gradient a problem?



- **Short-Range vs. Long-Range:** Imagine a long chain of dominos where each domino represents a layer or time step. If the force transferred from one domino to the next diminishes (say by a constant factor each time), then after a long chain, the force reaching the first domino becomes nearly zero. This means the first few dominos (or layers) barely "feel" the impact of the initial force (or error), and thus they do not adjust effectively based on long-term dependencies.
- **Resulting Behavior:** The model ends up "paying attention" only to the parts of the sequence that are immediately relevant (the nearby gradient signals) while ignoring distant context. This leads to challenges such as:
  - Inability to learn relationships or dependencies that span many time steps.
  - Poor performance on tasks that require integrating information over long sequences.

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 ________*

Problem:
The training example requires the model to understand that the first occurrence of **"tickets"** should influence the final output. However, due to the vanishing gradient, the error signal indicating that **"tickets"** was the correct prediction does not effectively travel back to where it is needed.

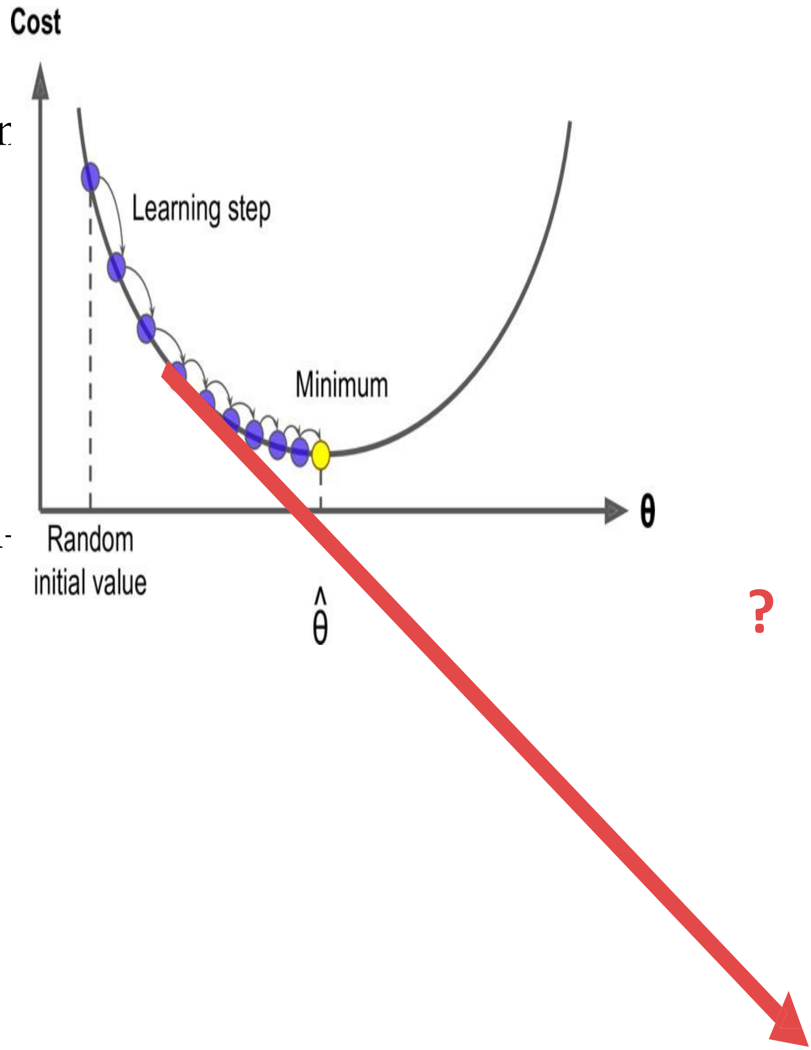
Outcome:
The model primarily updates its weights based on more immediate context (such as the recent words about installing toner), and as a result, it struggles to predict long-range dependencies like the repeated **"tickets"** at the end.

Key Takeaway:
When gradients vanish over long sequences, the model learns only the short-term (local) dependencies and fails to capture the long-term (global) context required to accurately model relationships spanning many time steps.

Why is exploding gradient a problem?

- **Big Gradients = Big Updates:**
  If the gradient becomes too large, then multiplying it by the learning rate yields an update step that is far too big. This can drastically change the model's parameters in single update, causing the model to jump to a region in parameter space where the loss is extremely high.

$$\theta^{new} = \theta^{old} - \alpha \nabla_{\theta} J(\theta)$$



- **Overshooting and Divergence:**
  The model may overshoot any potential minima and end up in a poor configuration-figuratively, you might think you're following a path upward (finding a local minimum), but instead, you are suddenly in an entirely different and suboptimal region (like ending up in Iowa, far from your intended destination).

- **Numerical Problems:**
  If updates are too extreme, you might encounter numerical issues like Inf or NaN values, which can halt training and require you to restart from a safe checkpoint.

Gradient clipping

- **Gradient clipping**: if the norm of the gradient is greater than some threshold, scale it down before applying SGD update

Algorithm 1 Pseudo-code for norm clipping

$\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$
if $\|\hat{\mathbf{g}}\| \geq threshold$ **then**
  $\hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|} \hat{\mathbf{g}}$
end if

Stability:
With gradient clipping, you are less likely to encounter the situation where your network's parameters become so large that they result in numerical overflow (Inf or NaN values), which would require you to restart training.

Convergence:
Although exploding gradients can disrupt convergence, clipping helps maintain a controlled learning process where the update steps are kept within a safe range, thereby supporting steady convergence.

Focus on Other Challenges:
Since exploding gradients can be managed relatively easily through clipping, you can often shift your focus to more challenging issues like vanishing gradients or designing architectures that capture long-term dependencies.

Source: "On the difficulty of training recurrent neural networks", Pascanu et al, 2013. http://proceedings.mlr.press/v28/pascanu13.pdf

How to fix the vanishing gradient problem?

- **Gated Architectures:**
  LSTM and GRU models include gates that help regulate the flow of information and gradients over long time sequences, mitigating the vanishing gradient problem.
- **Activation Functions:**
  ReLU and similar activations help reduce the saturation effect (common in sigmoid or tanh), which can cause gradients to vanish.
- **Normalization Techniques:**
  Batch or layer normalization helps to standardize activations and gradients, ensuring that they remain within a reasonable range.
- **Residual Connections:**
  Adding skip connections enables gradients to flow directly from later layers to earlier layers, bypassing some of the multiplicative effects that cause vanishing.
- **Weight Initialization:**
  Using initialization strategies like Xavier or He initialization keeps the scale of the activations and gradients more controlled.

Content

LSTMs: Apple WWDC Keynote 2016

Apple WWDC 2016: Apple is tentatively dipping its toe into the AI waters with technologies that can analyze your photos for faces and context - all done locally - and by applying LSTM deep learning technologies to Messaging.

Long Short-Term Memory RNNs (LSTMs)

- The original proposal by Hochreiter and Schmidhuber in 1997 as a solution to the vanishing gradient problem.

- Although that paper is widely cited, the modern LSTM architecture also owes much to innovations by Gers et al. (2000).

- The work of Alex Graves around 2006, who not only helped demonstrate the potential of LSTM models but also invented CTC for speech recognition.

- LSTM gained widespread attention when Hinton introduced it at Google in 2013, with Graves contributing significantly as his postdoc.

Hochreiter and Schmidhuber, 1997. Long short-term memory. https://www.bioinf.jku.at/publications/older/2604.pdf
Gers, Schmidhuber, and Cummins, 2000. Learning to Forget: Continual Prediction with LSTM. https://dl.acm.org/doi/10.1162/089976600300015015
Graves, Fernandez, Gomez, and Schmidhuber, 2006. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural nets. https://www.cs.toronto.edu/~graves/icml_2006.pdf

LSTMs: real-world success

- In 2013–2015, LSTMs started achieving state-of-the-art results
  - Successful tasks include handwriting recognition, speech recognition, machine translation, parsing, and image captioning, as well as language models
  - LSTMs became the dominant approach for most NLP tasks

- Now (2019–2024), Transformers have become dominant for all tasks
  - For example, in **WMT** (a Machine Translation conference + competition):
    - In WMT 2014, there were 0 neural machine translation systems (!)
    - In WMT 2016, the summary report contains "RNN" 44 times (and these systems won)
    - In WMT 2019: "RNN" 7 times, "Transformer" 105 times

__Source:__ "Findings of the 2016 Conference on Machine Translation (WMT16)", Bojar et al. 2016, http://www.statmt.org/wmt16/pdf/W16-2301.pdf
__Source:__ "Findings of the 2018 Conference on Machine Translation (WMT18)", Bojar et al. 2018, http://www.statmt.org/wmt18/pdf/WMT028.pdf
__Source:__ "Findings of the 2019 Conference on Machine Translation (WMT19)", Barrault et al. 2019, http://www.statmt.org/wmt18/pdf/WMT028.pdf

Long Short-Term Memory RNNs (LSTMs)

- At each time step t, the LSTM maintains **a hidden state h(t)** and **a cell state c(t)**, both vectors of length n.
- The cell state stores **long-term information**, while the hidden state represents **the immediate output**.
- Three gates—the **forget gate**, **input gate**, and **output gate**—dynamically control the erasing, **reading, and writing** of information in the cell state.
- Each gate is computed as a vector with values between 0 and 1, where **values indicate the proportion of information to keep or update.**
- This gating mechanism allows the LSTM to **selectively maintain important information over long sequences**, effectively addressing the vanishing gradient problem encountered in traditional RNNs.

This dynamic gating system is a key innovation that makes LSTMs powerful for tasks requiring long-term memory, such as language modeling, speech recognition, and time-series prediction.

Long Short-Term Memory (LSTM)

A sequence of inputs $x^{(t)}$. Compute a sequence of hidden states $h^{(t)}$ and cell states $c^{(t)}$. 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

$$f^{(t)} = \boxed{\sigma}\left(W_f h^{(t-1)} + U_f x^{(t)} + b_f\right)$$

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

$$i^{(t)} = \sigma\left(W_i h^{(t-1)} + U_i x^{(t)} + b_i\right)$$

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

$$o^{(t)} = \sigma\left(W_o h^{(t-1)} + U_o x^{(t)} + b_o\right)$$

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

$$\tilde{c}^{(t)} = \tanh\left(W_c h^{(t-1)} + U_c x^{(t)} + b_c\right)$$

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

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

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

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

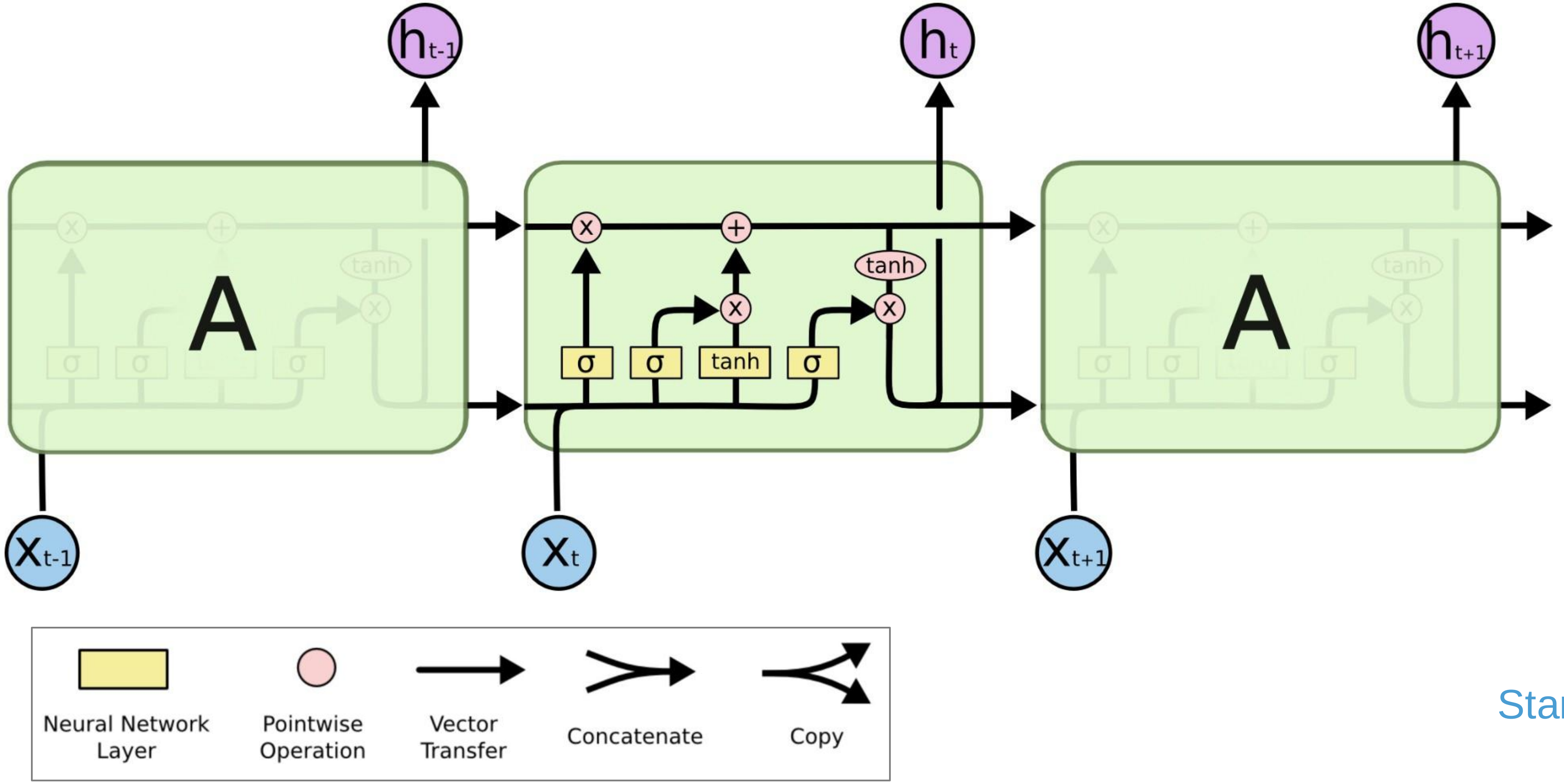
All these are vectors of same length *n*

Gates are applied using element-wise (or Hadamard) product: $\odot$

StanfordCS224n

Long Short-Term Memory (LSTM)

You can think of the LSTM equations visually like this:



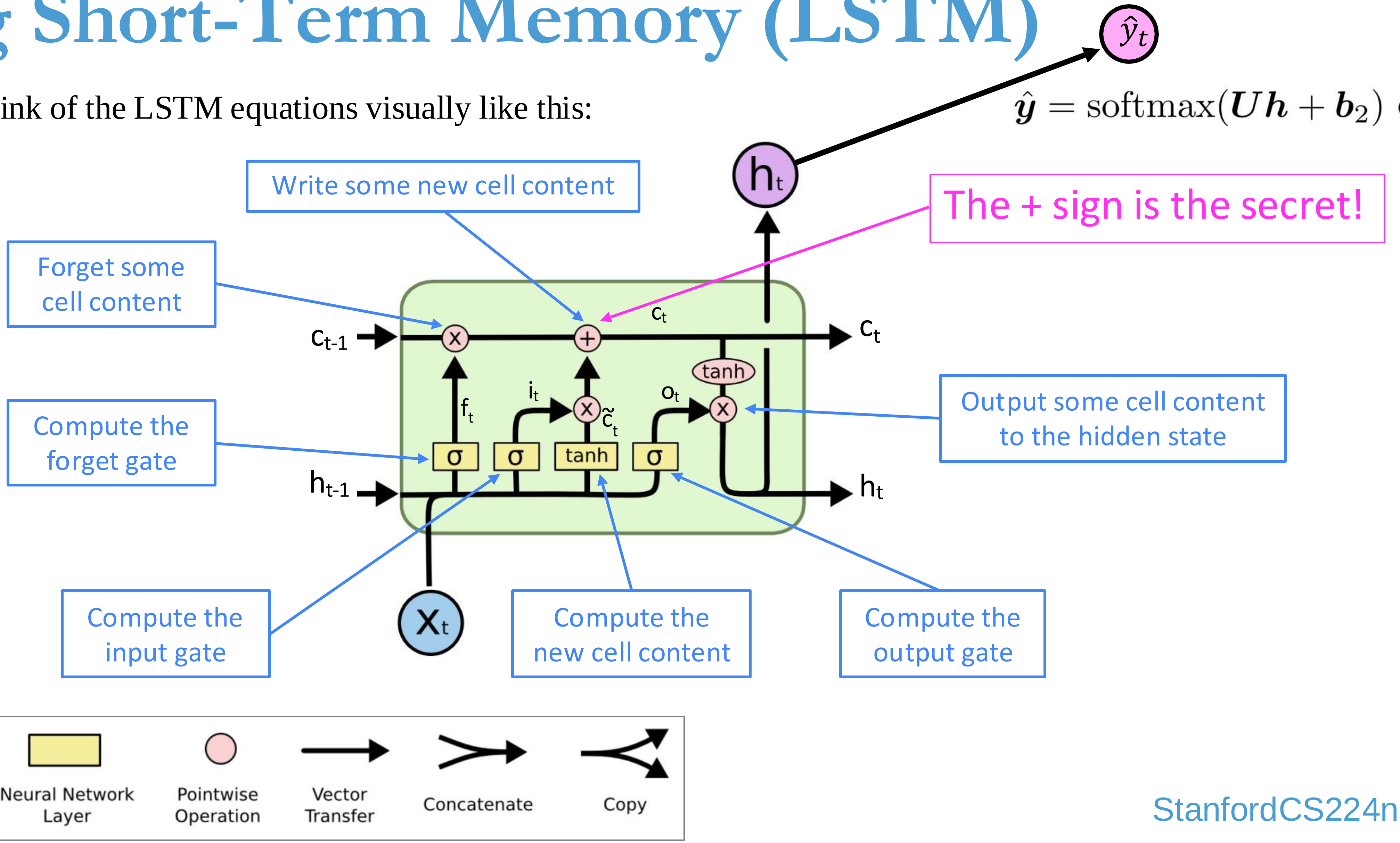
StanfordCS224n

Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long Short-Term Memory (LSTM)

You can think of the LSTM equations visually like this:

$$\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{U}\boldsymbol{h} + \boldsymbol{b}_2) \in \mathbb{R}^{|V|}$$



StanfordCS224n

Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Preserving Long-Term Dependencies

LSTMs vs. Vanilla RNNs

- **LSTM Advantage:**

The LSTM architecture makes it much easier for an RNN to preserve information over many timesteps. Example: If the forget gate is set to 1 and the input gate to 0 for a cell dimension, the corresponding cell state is preserved indefinitely.

- **Vanilla RNN Limitation:**

A vanilla RNN must learn a recurrent weight matrix that preserves information in the hidden state. In practice, vanilla RNNs typically manage to preserve information over only about 7 timesteps.

- **Extended Memory with LSTMs:**

LSTMs can effectively preserve information for around 100 timesteps, greatly enhancing the model's ability to capture long-term dependencies.

- **Alternative Approaches:**

There are other methods to create direct, linear pass-through connections that capture long-distance dependencies.

Common Problems

- **Universality of the Problem:**
  Vanishing gradients are not unique to a single type of network—they affect all architectures, especially as depth increases.
- **Root Cause:**
  It explains how the chain rule and activation function choices cause gradients to decay, which is the mathematical reason behind the problem.
- **Impact on Learning:**
  With vanishing gradients, lower layers learn very slowly, affecting the overall performance and convergence of the network.
- **Architectural Remedies:**
  Modern networks counteract this problem by adding direct connections (skip/residual connections) that help maintain gradient flow, with examples like ResNets and DenseNets.

For example:

- Residual connections aka "ResNet"
- Also known as skip-connections
- The identity connection preserves information by default
- This makes deep networks much easier to train

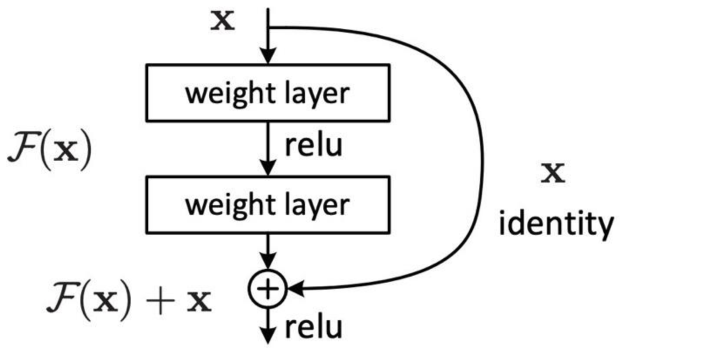


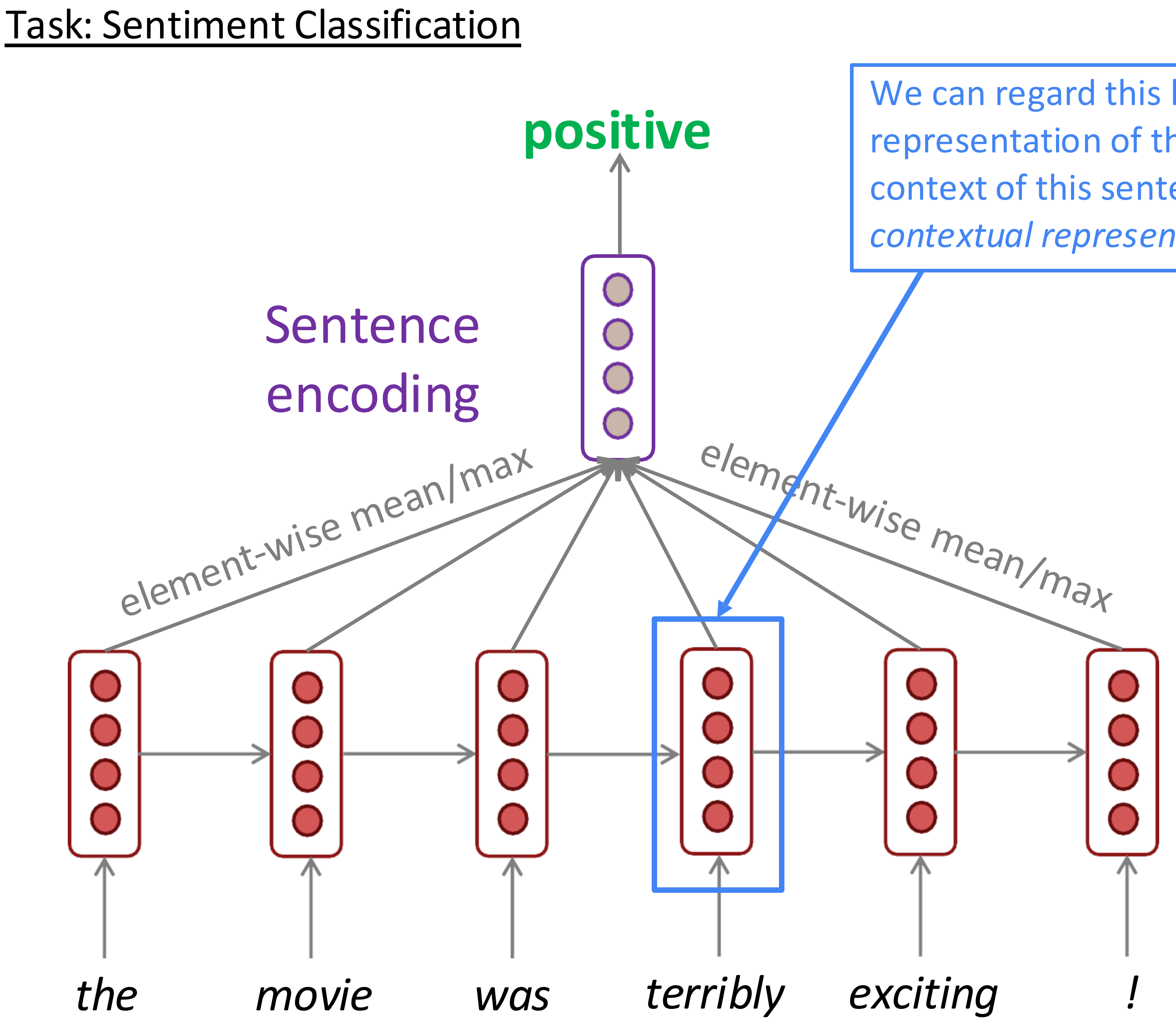
Figure 2. Residual learning: a building block.

"Deep Residual Learning for Image Recognition", He et al, 2015. https://arxiv.org/pdf/1512.03385.pdf

Code: Classifying Names with a LSTM

```
import torch.nn as nn
import torch.nn.functional as F

class CharLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(CharLSTM, self).__init__()

        # Replace RNN with LSTM
        self.lstm = nn.LSTM(input_size, hidden_size)
        self.h2o = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, line_tensor):
        # LSTM produces two outputs: output and (hidden_state, cell_state)
        lstm_out, (hidden_state, cell_state) = self.lstm(line_tensor)
        # Use the last hidden state to produce output
        output = self.h2o(hidden_state[-1])
        output = self.softmax(output)

        return output

lstm = CharLSTM(n_letters, n_hidden=128, len(alldata.labels_uniq))
```

Bidirectional and Multi-layer RNNs



We can regard this hidden state as a representation of the word "*terribly*" in the context of this sentence. We call this a *contextual representation.*

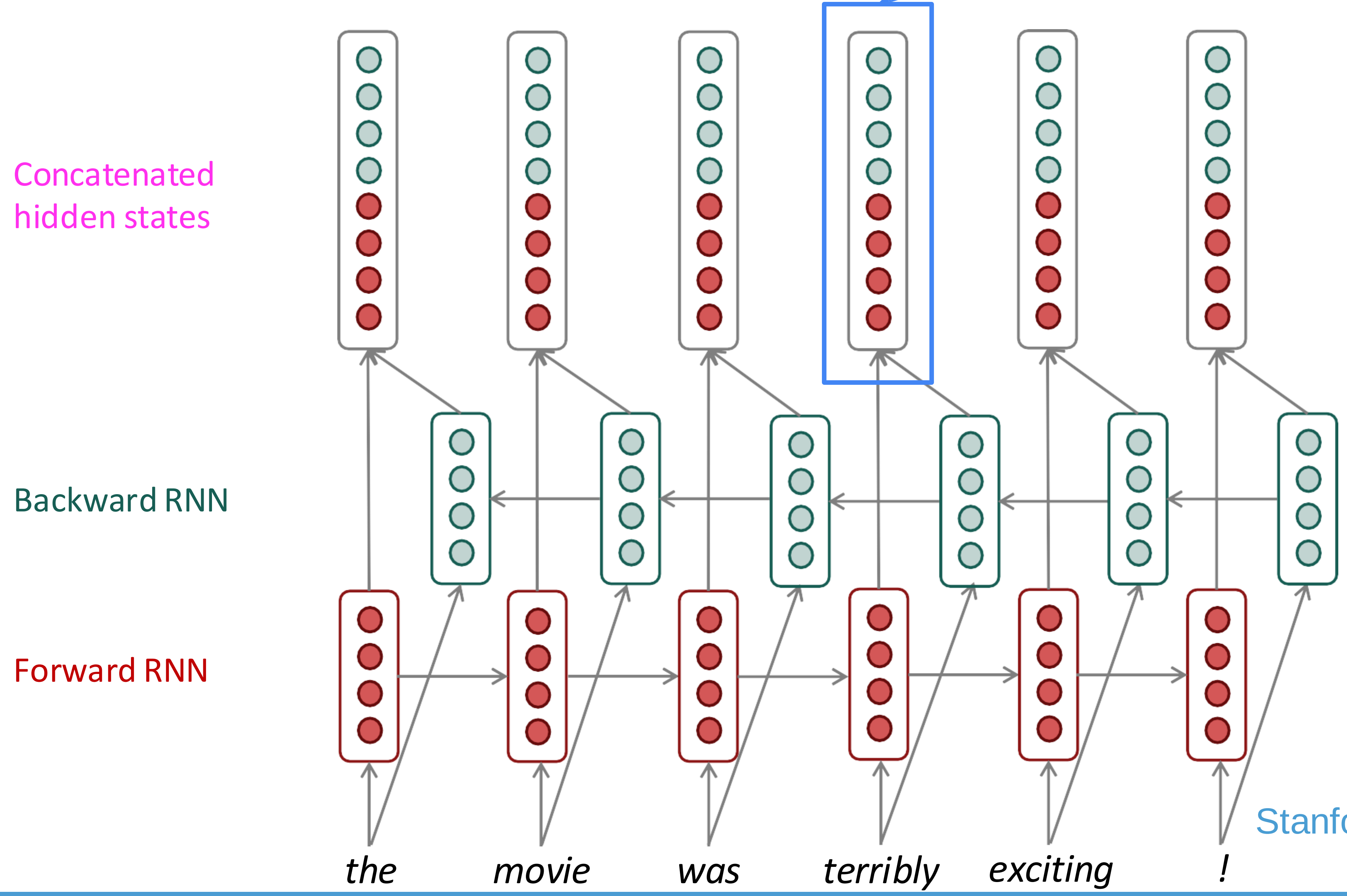
These contextual representations only contain information about the *left* context (e.g. *"the movie was"*).

What about *right* context?

In this example, *"exciting"* is in the right context and this modifies the meaning of *"terribly"* (from negative to positive)

Bidirectional RNNs

This contextual representation of “terribly” has both left and right context!



StanfordCS224n

Bidirectional RNNs

On timestep *t*:

This is a general notation to mean "compute one forward step of the RNN" – it could be a simple RNN or LSTM computation.

Forward RNN $\overrightarrow{\boldsymbol{h}}^{(t)} = \boxed{\mathrm{RNN}_{\mathrm{FW}}}(\overrightarrow{\boldsymbol{h}}^{(t-1)}, \boldsymbol{x}^{(t)})$

Backward RNN $\overleftarrow{\boldsymbol{h}}^{(t)} = \mathrm{RNN}_{\mathrm{BW}}(\overleftarrow{\boldsymbol{h}}^{(t+1)}, \boldsymbol{x}^{(t)})$

Generally, these two RNNs have separate weights

Concatenated hidden states $\boxed{\boldsymbol{h}^{(t)}} = [\overrightarrow{\boldsymbol{h}}^{(t)}; \overleftarrow{\boldsymbol{h}}^{(t)}]$

We regard this as "the hidden state" of a bidirectional RNN. This is what we pass on to the next parts of the network.

StanfordCS224n

Pros and Cons of Bidirectional RNNs

Pros Section:

- **Enhanced Context:** Emphasizes that BiRNNs incorporate both past and future context, leading to richer representations.
- **Improved Accuracy:** Highlights the improved performance in various tasks.
- **Better Long-Term Dependency Modeling:** Points out that the dual-direction approach can capture dependencies over longer sequences.

Cons Section:

- **Increased Computational Cost:** Notes the extra resources required for running both forward and backward RNNs.
- **Not Suitable for Real-Time Applications:** Explains that access to future context makes BiRNNs less practical in scenarios where such information isn’t available.
- **Complexity:** Mentions that the increased complexity can make training and tuning more challenging.

Bidirectional RNNs

- Bidirectional RNNs require access to the entire input sequence to compute both forward and backward passes. They are best suited for tasks where the complete sequence is available in advance (e.g., sequence encoding, text classification, and machine translation).

- In Language Modeling (LM), the model generates text one token at a time. At each generation step, only the left context (previous tokens) is available. Bidirectional RNNs are not applicable to LM because future context is unknown during generation.

- When the entire input sequence is available (e.g., for encoding), bidirectionality is very powerful. It provides a richer representation by capturing both past and future context for each token.

- Each token's embedding in BERT is contextualized with both left and right information. Typically, BERT models produce embeddings of 768 or 1024 dimensions. BERT has set new standards for many NLP tasks by providing rich, pretrained representations.

Code: Bidirectional RNNs

```
class CharBiRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(CharBiRNN, self).__init__()

        # Bidirectional RNN
        self.rnn = nn.RNN(input_size, hidden_size, bidirectional=True)
        # Adjust output size to account for bidirectional hidden states (2 * hidden_size)
        self.h2o = nn.Linear(hidden_size * 2, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, line_tensor):
        # rnn_out contains outputs for all timesteps, hidden contains the last hidden state
        rnn_out, hidden = self.rnn(line_tensor)
        # Combine the last forward and backward hidden states
        hidden_cat = torch.cat((hidden[-2], hidden[-1]), dim=1)
        output = self.h2o(hidden_cat)
        output = self.softmax(output)
        return output

bidir_rnn = CharBiRNN(n_letters, n_hidden, len(alldata.labels_uniq))
```

Stacked RNNs

Stacked RNNs (also known as deep RNNs): multiple recurrent layers are placed on top of each other.

- The output of each RNN layer serves as the input to the next, creating a hierarchical representation of the sequential data. The lower RNNs should compute lower-level features and the higher RNNs should compute higher-level features.

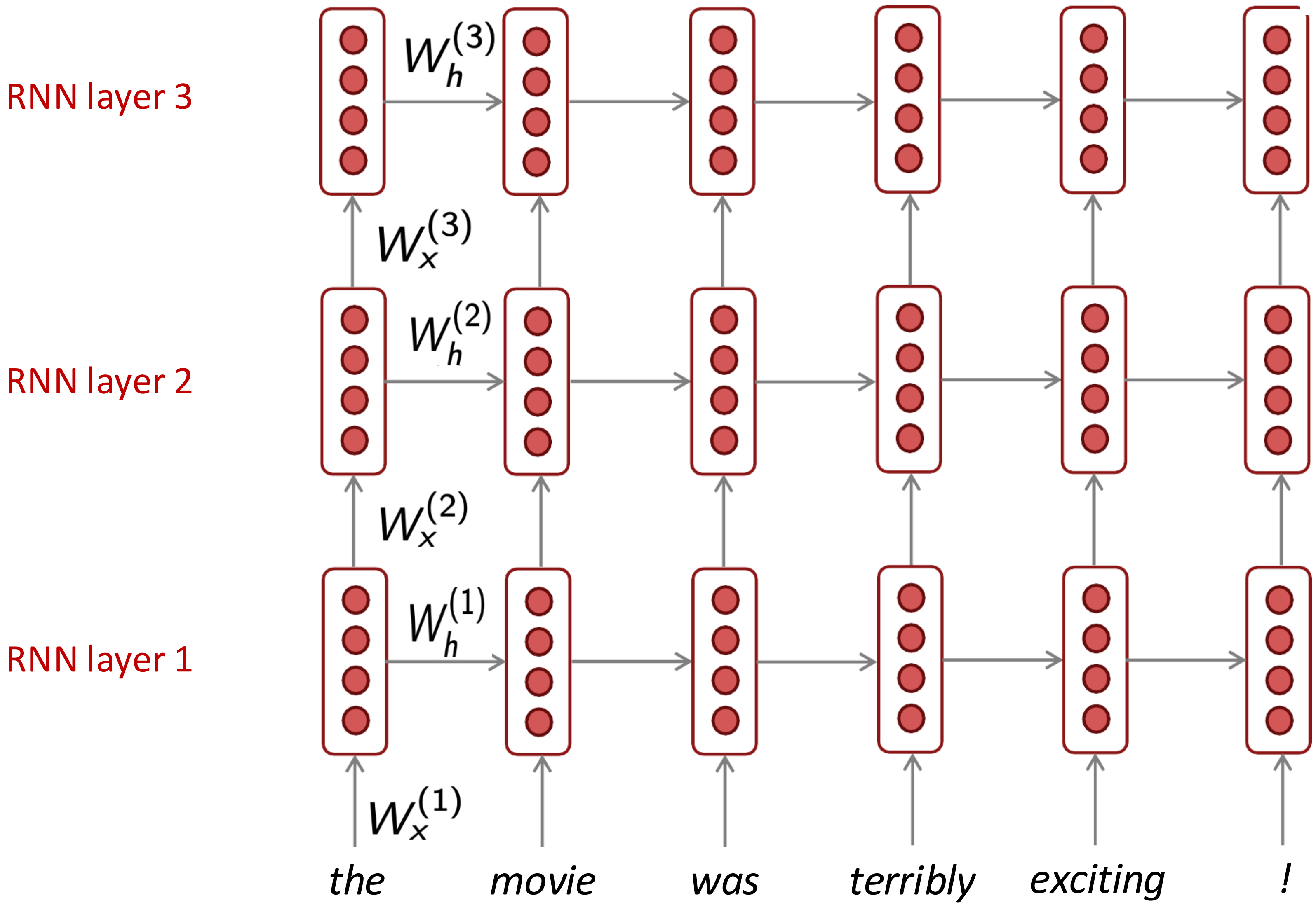
- This architecture is used to capture complex, abstract temporal patterns. Hierarchical layers can learn more abstract features. Better capture complex temporal dependencies.

- Improved Performance: Often achieve higher accuracy on tasks such as language modeling, speech recognition, and time series prediction.

- However, they also introduce challenges such as increased training complexity and computational cost.

- With careful design and tuning, stacked RNNs are a powerful tool for sequence modeling.

Three-layer RNNs



$$\mathbf{y}_t = g\left(W_o \mathbf{h}_t^{(3)} + \mathbf{b}_o\right)$$

$$\mathbf{h}_t^{(3)} = f\left(W_x^{(3)} \mathbf{h}_t^{(2)} + W_h^{(3)} \mathbf{h}_{t-1}^{(3)} + \mathbf{b}^{(3)}\right)$$

$$\mathbf{h}_t^{(2)} = f\left(W_x^{(2)} \mathbf{h}_t^{(1)} + W_h^{(2)} \mathbf{h}_{t-1}^{(2)} + \mathbf{b}^{(2)}\right)$$

$$\mathbf{h}_t^{(1)} = f\left(W_x^{(1)} \mathbf{x}_t + W_h^{(1)} \mathbf{h}_{t-1}^{(1)} + \mathbf{b}^{(1)}\right)$$

StanfordCS224n

Multi-layer RNNs in practice

- **Stacked RNNs allow for more complex, hierarchical representations:** Lower layers capture basic, low-level features, while higher layers integrate these into more abstract, high-level features.’

- **Performance improvements are observed with moderate stacking:** Empirical results (e.g., from Britz et al., 2017) indicate that 2–4 layers can be optimal for tasks such as machine translation.

- **Deep RNNs require architectural innovations:** For very deep RNNs (e.g., 8 layers or more), skip-connections or dense connections are necessary to maintain gradient flow and facilitate training.

- **Transformers push depth further with built-in residual connections:** Models like BERT, which can have 12–24 layers, offer a different approach to capturing long-range dependencies via self-attention and deep stacking.

Code: Muti-layer Bidirectional RNNs

```
class MultiLayerBiRNNClassifier(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers):
        super(MultiLayerBiRNNClassifier, self).__init__()
        # Multi-layer bidirectional RNN
        self.rnn = nn.RNN(
            input_size,
            hidden_size,
            num_layers=num_layers,
            bidirectional=True,
            batch_first=True
        )
        # Adjust the linear layer to account for bidirectional hidden states
        self.h2o = nn.Linear(hidden_size * 2, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input_tensor):
        # rnn_out: Outputs from all time steps, hidden: Hidden states for all layers
        rnn_out, hidden = self.rnn(input_tensor)
        # Combine the last hidden states from both directions in the last layer
        hidden_cat = torch.cat((hidden[-2], hidden[-1]), dim=1)
        output = self.h2o(hidden_cat)
        output = self.softmax(output)
        return output
```

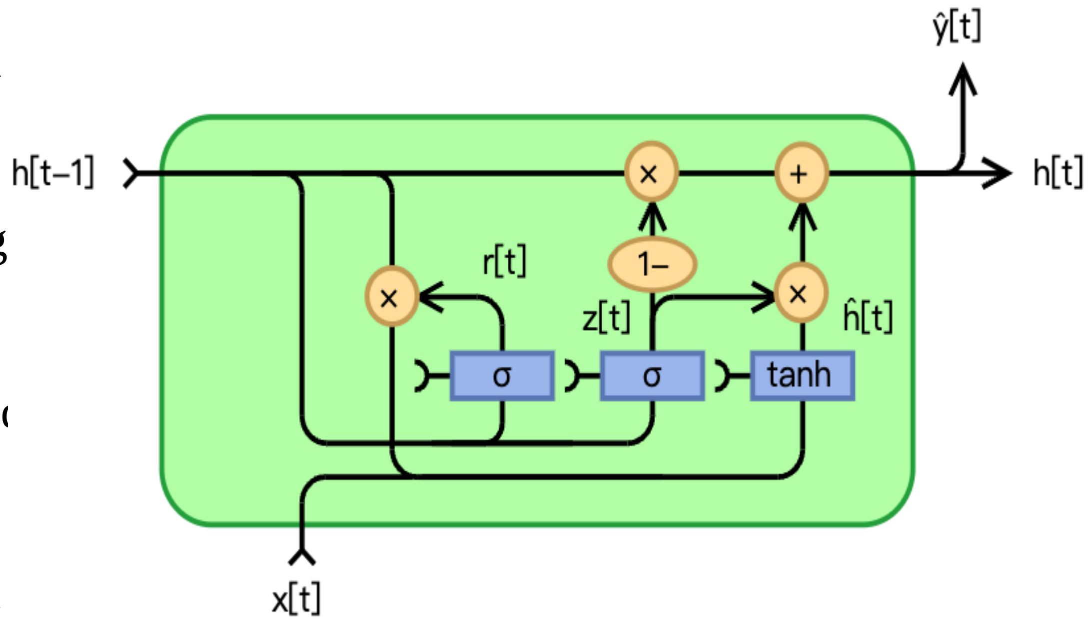
GRU (Gate Recurrent Unit)

- Gate Recurrent Unit is one of the ideas that has enabled RNN to become much better at capturing very long-range dependencies and made RNN much more effective.

- The GRU is like a LSTM with a gating mechanism to input or forg certain features, but lacks a context vector or output gate, resulting fewer parameters than LSTM. Proposed as a simpler alternative to LSTM, the GRU merges the forget and input gates into a single upd gate.

- GRUs are known for having fewer parameters than LSTMs, which lead to faster training and similar performance in many tasks.

  Each GRU cell has two main gates:



- ▶ Reset Gate (r(t)): Controls how much of the previous hidden state to forget.
- ▶ Update Gate (z(t)): Decides how much of the candidate activation to use.
- ▶ The GRU combines these gates to update its hidden state without a separate cell state.

1. Update Gate (z_t): Determines how much of the past hidden state h_{t-1} should be retained and how much of the new candidate state $\hat{h}_{t-1}$ should be added to form the current hidden state.

$$z_t = \sigma(W_z \cdot x_t + U_z \cdot h_{t-1} + b_z)$$

2. Reset Gate (r_t): Determines how much of the past hidden state h_{t-1} contributes to the computation of the new candidate state $\hat{h}_{t-1}$.

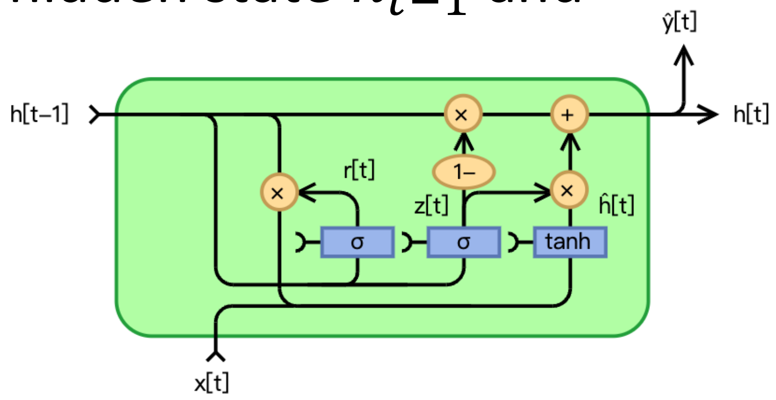
$$r_t = \sigma(W_r \cdot x_t + U_r \cdot h_{t-1} + b_r)$$

3. Candidate State ($\hat{h}_t$): New information computed at the current time step.

$$\hat{h}_t = \tanh(W_h \cdot x_t + U_h \cdot (r_t \odot h_{t-1}) + b_h)$$

4. Hidden State Update: Combines contributions from the past hidden state h_{t-1} and the new candidate state $\hat{h}_{t-1}$ using the update gate z_t.

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



StanfordCS224n

Classifying Names with a Character-Level GRU

```
import torch.nn as nn
import torch.nn.functional as F

class CharGRU(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(CharGRU, self).__init__()

        # Replace RNN with GRU
        self.gru = nn.GRU(input_size, hidden_size)
        self.h2o = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, line_tensor):
        # GRU produces outputs and hidden states
        gru_out, hidden = self.gru(line_tensor)
        # Use the last hidden state for classification
        output = self.h2o(hidden[-1])
        output = self.softmax(output)

        return output
```

Content

1 Introduction to Sequence Modeling

2 Introduction to Recurrent Neural Networks (RNNs)

3 Problems with RNNs

4 Extensions of RNNs

5 Genomic Sequence Analysis

Functional Effects of Noncoding Variants

- Most disease-associated variants identified by GWAS are located in noncoding regions of the genome.

- These variants can have significant regulatory effects. For example, they may modify transcription factor binding sites, alter chromatin accessibility, or impact epigenetic marks such as DNA methylation and histone modifications. These changes can, in turn, affect gene expression patterns and contribute to the development of complex diseases such as cancer, diabetes, and autoimmune disorders.

- Predicting the functional effects of these noncoding variants is therefore crucial for understanding the genetic basis of complex diseases and for identifying potential therapeutic targets. Traditional approaches relying on hand-crafted features or statistical methods often struggle to capture the intricate, nonlinear relationships within genomic data.

- Deep learning offers a powerful alternative by automatically learning hierarchical representations directly from raw genomic sequences. With the ability to model both local and long-range dependencies, deep learning models can identify subtle sequence patterns and interactions that are critical for gene regulation. This capability has led to significant advances in predicting regulatory functions and understanding the mechanisms by which noncoding variants contribute to disease pathology.

Deep Learning-based Sequence Analyzer

❖ DeepSEA Model Architecture

- ➢ Incorporates wide sequence context (1 kbp) and use one-hot to encode the final sequence.
- ➢ Uses hierarchical convolutional neural networks (CNNs) to learn sequence dependencies.
- ➢ Employs multitask learning for sharing predictive features across chromatin factors.
- ➢ Optimized for accuracy in functional variant prediction.

One-Hot Encoding

- ▶ A $\rightarrow [1, 0, 0, 0]$
- ▶ C $\rightarrow [0, 1, 0, 0]$
- ▶ G $\rightarrow [0, 0, 1, 0]$
- ▶ T $\rightarrow [0, 0, 0, 1]$

❖ Training and Data Sources

- ➢ Training data from ENCODE and Roadmap Epigenomics projects.
- ➢ 690 TF binding profiles, 125 DNase I hypersensitivity profiles, 104 histone-mark profiles.
- ➢ Covers 17% of the human genome (521.6 Mbp).
- ➢ Trained using CNN.

Zhou, J., & Troyanskaya, O. G. (2015)

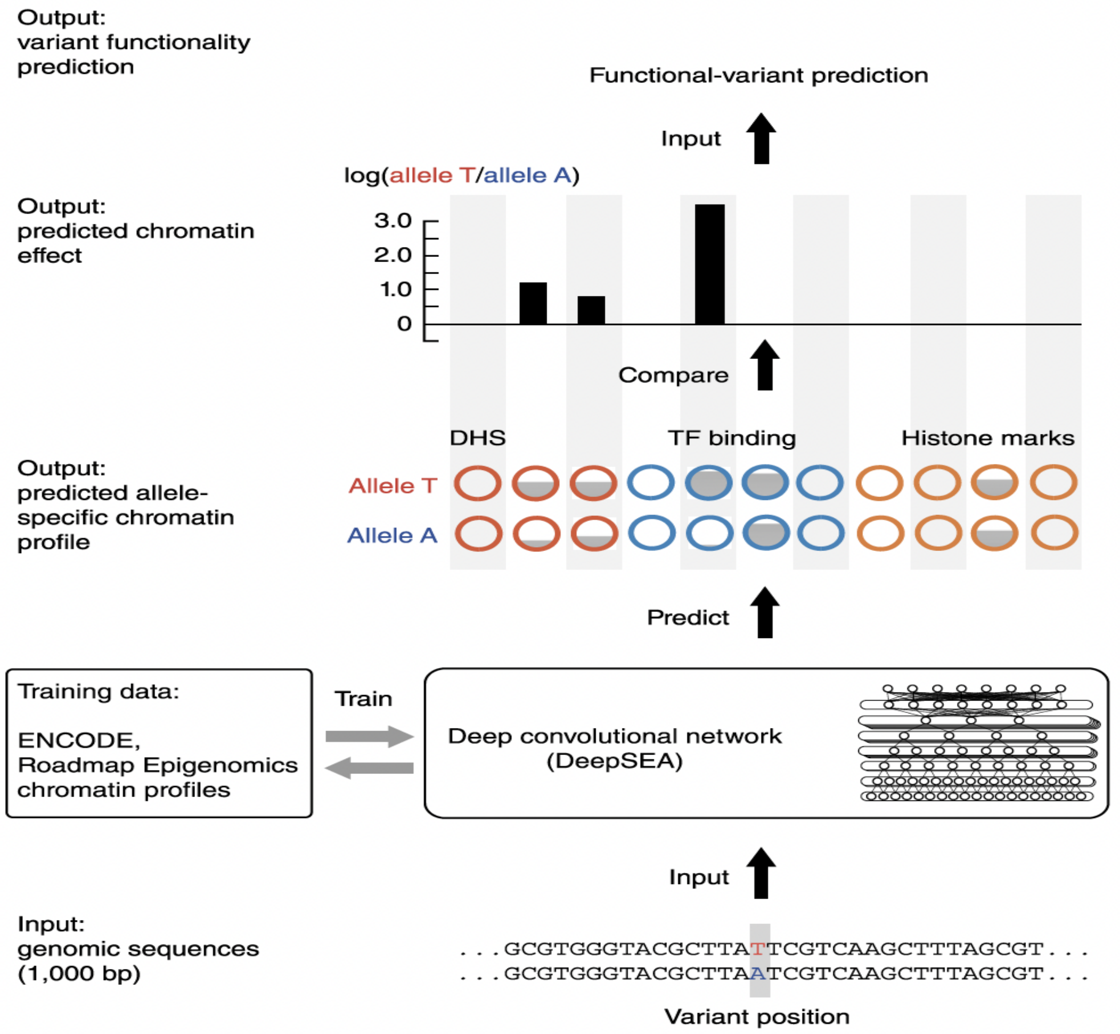
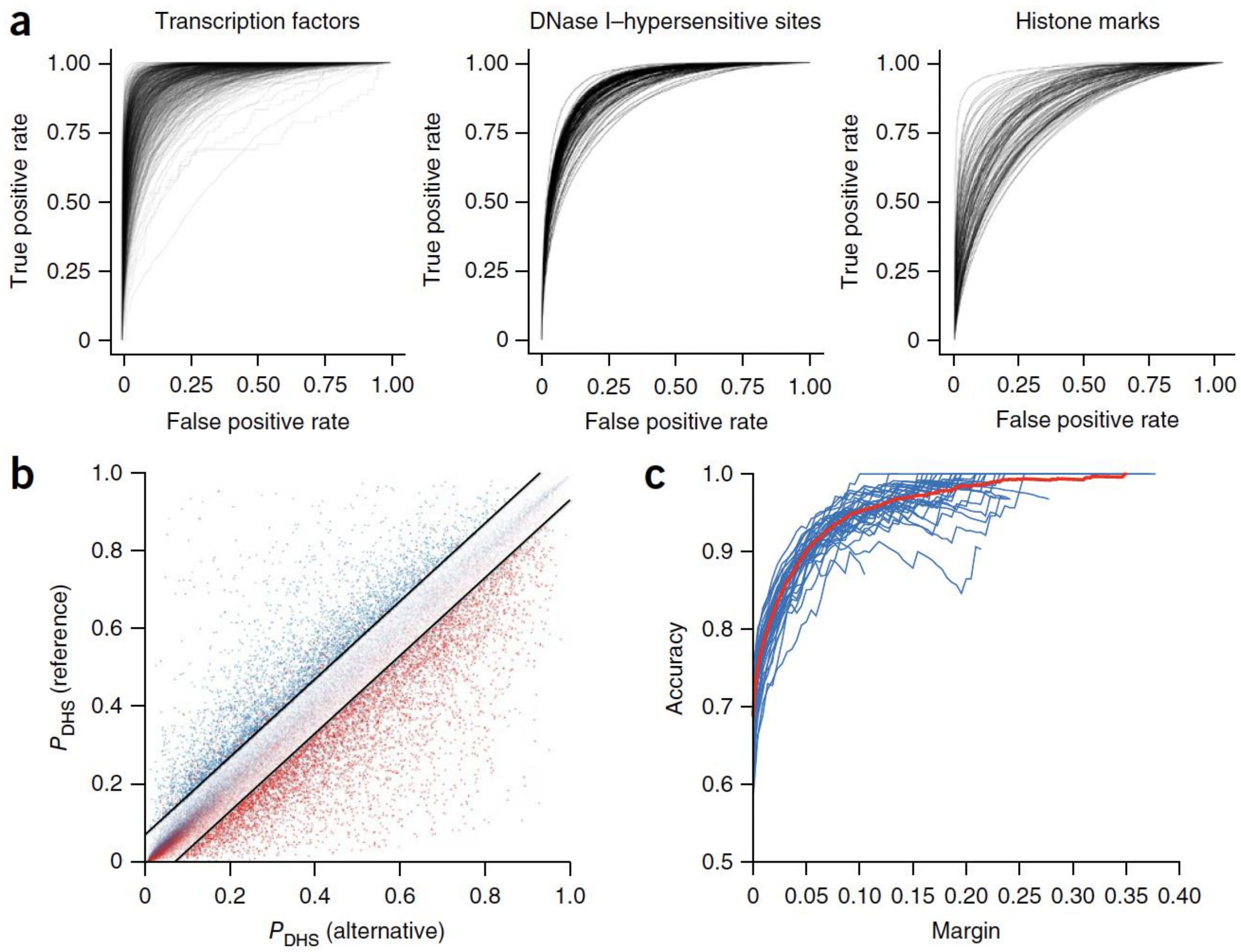


Figure 1 | Schematic overview of the DeepSEA pipeline, a strategy for predicting chromatin effects of noncoding variants.

Performance and Sensitivity



- DeepSEA achieves high accuracy in predicting chromatin features:
  - **TF binding: AUC = 0.958**
  - **DNase I sensitivity: AUC = 0.923**
  - **Histone marks: AUC = 0.856**
- Outperforms gkm-SVM on nearly all TFs.
- Enables accurate sequence-based functional predictions.

- Uses 'in silico saturated mutagenesis' to assess sequence feature importance.
- Evaluated on 57,407 allelically imbalanced SNPs from 35 cell types.
- Achieves >95% accuracy for high-confidence predictions.
- Consistently predicts known SNP effects on TF binding (e.g., FOXA1, GATA1, FOXA2).

DanQ

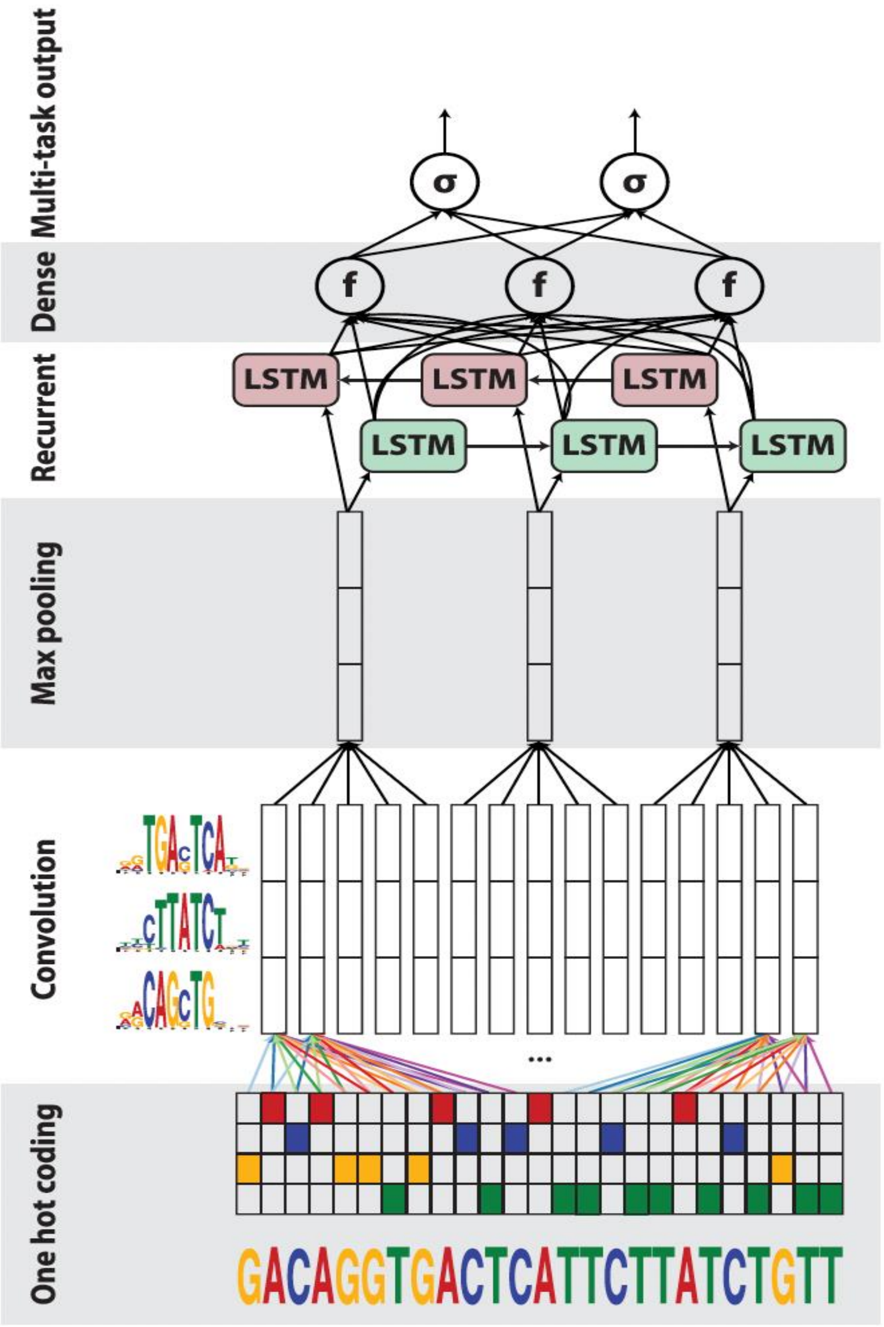
❖ Motivations

- ➢ Previous models like DeepSEA and gkm-SVM predict regulatory function but lack recurrent components.
- ➢ Need for a model that integrates motif discovery and long-term dependencies in DNA sequences.

❖ DanQ Model Framework

- ➢ **Input Data:** One-hot encoded 1000-bp DNA sequences from GRCh37 genome.
- ➢ **Training Data:** 919 chromatin features from ENCODE & Roadmap Epigenomics datasets.
- ➢ **Architecture:**
  - CNN layer for motif scanning.
  - Max pooling layer to reduce spatial size.
  - BLSTM layer to capture motif relationships.
  - Fully connected layer with sigmoid outputs.

Quang, D., & Xie, X. (2016).



Performance and Sensitivity

❖ **Comparisons:**

- ➢ ROC AUC: DanQ outperforms DeepSEA for 94.1% of targets.
- ➢ PR AUC: DanQ achieves over 50% relative improvement on some markers.
- ➢ Enhanced motif discovery compared to previous CNN-based models.

❖ **Results**

- ➢ ROC and PR curves demonstrate significant performance gain over DeepSEA.
- ➢ • Motif analysis shows DanQ effectively learns biologically relevant patterns.
- ➢ • Functional SNP prioritization shows improved detection of regulatory variants

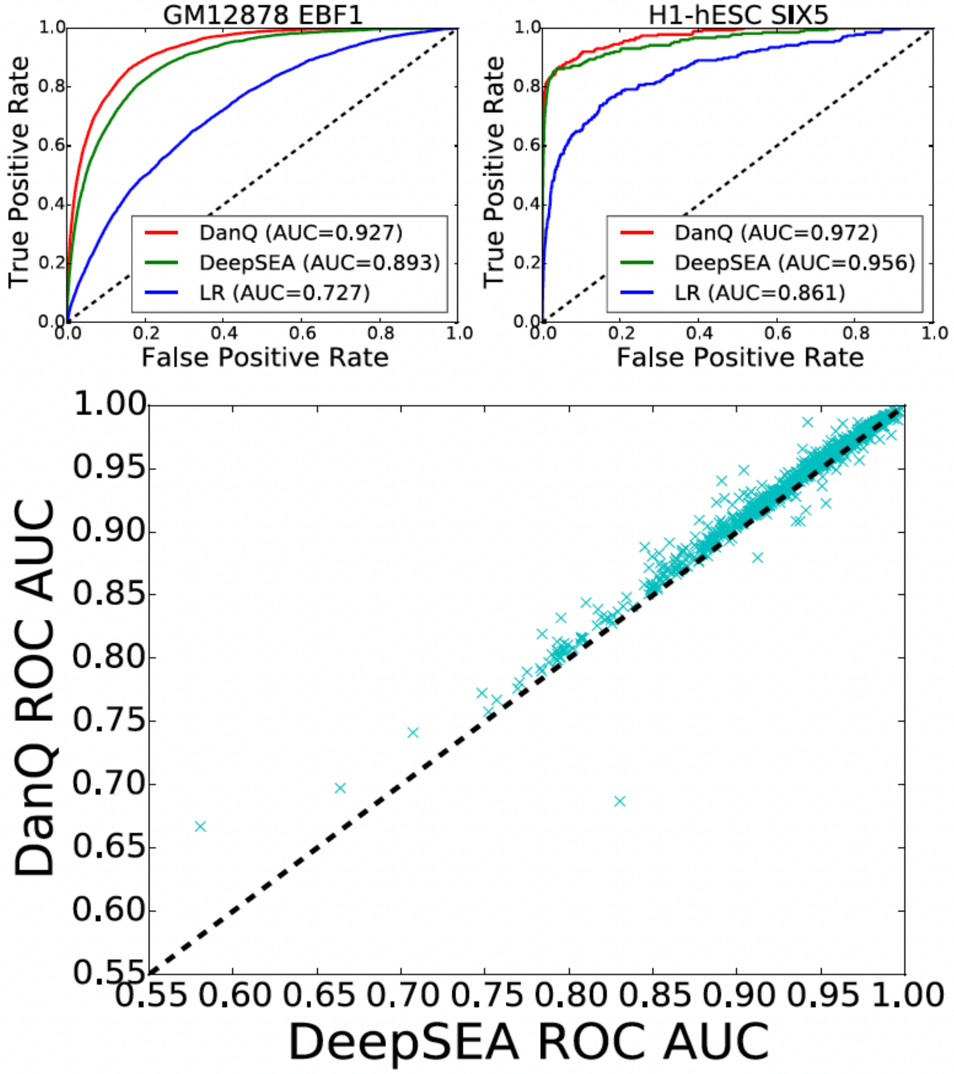
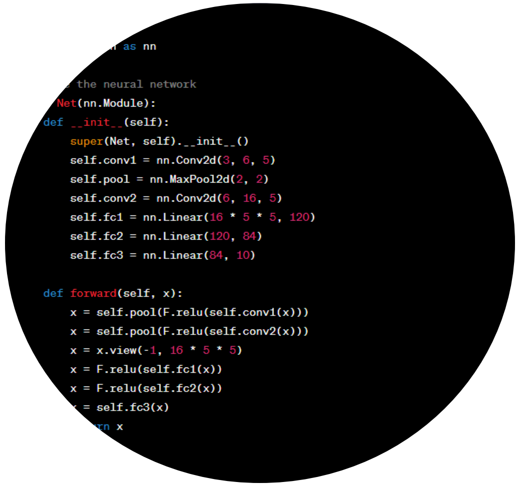


Figure 2. (Top) ROC curves for the GM12878 EBF1 and H1-hESC SIX5 targets comparing the performance of the three models. (Bottom) Scatterplot comparing DanQ and DeepSEA ROC AUC scores. DanQ outperforms DeepSEA for 94.1% of the targets in terms of ROC AUC.

References

- Levy, O., Goldberg, Y., & Dagan, I. (2015). Improving distributional similarity with lessons learned from word embeddings. *Transactions of the association for computational linguistics*, *3*, 211-225.
- Lipton, Z. C. (2015). A Critical Review of Recurrent Neural Networks for Sequence Learning. *arXiv Preprint, CoRR, abs/1506.00019*.
- Mienye, I. D., Swart, T. G., & Obaido, G. (2024). Recurrent neural networks: A comprehensive review of architectures, variants, and applications. *Information*, *15*(9), 517.
- Novakovsky, G., Dexter, N., Libbrecht, M. W., Wasserman, W. W., & Mostafavi, S. (2023). Obtaining genetics insights from deep learning via explainable artificial intelligence. *Nature Reviews Genetics*, *24*(2), 125-137.
- Pennington, J., Socher, R., & Manning, C. D. (2014, October). Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)* (pp. 1532-1543).
- Quang, D., & Xie, X. (2016). DanQ: a hybrid convolutional and recurrent deep neural network for quantifying the function of DNA sequences. *Nucleic acids research*, *44*(11), e107-e107.
- Sapoval, N., Aghazadeh, A., Nute, M. G., Antunes, D. A., Balaji, A., Baraniuk, R., ... & Treangen, T. J. (2022). Current progress and open challenges for applying deep learning across the biosciences. *Nature Communications*, *13*(1), 1728.
- Sherstinsky, A. (2020). Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. *Physica D: Nonlinear Phenomena*, *404*, 132306.
- Staudemeyer, R. C., & Morris, E. R. (2019). Understanding LSTM--a tutorial into long short-term memory recurrent neural networks. *arXiv preprint arXiv:1909.09586*.
- Yu, Y., Si, X., Hu, C., & Zhang, J. (2019). A review of recurrent neural networks: LSTM cells and network architectures. *Neural computation*, *31*(7), 1235-1270.
- Zhou, J., & Troyanskaya, O. G. (2015). Predicting effects of noncoding variants with deep learning–based sequence model. *Nature methods*, *12*(10), 931-934.

How to succeed in this course?



Practice

Explore

Visualize

Discuss

Ask