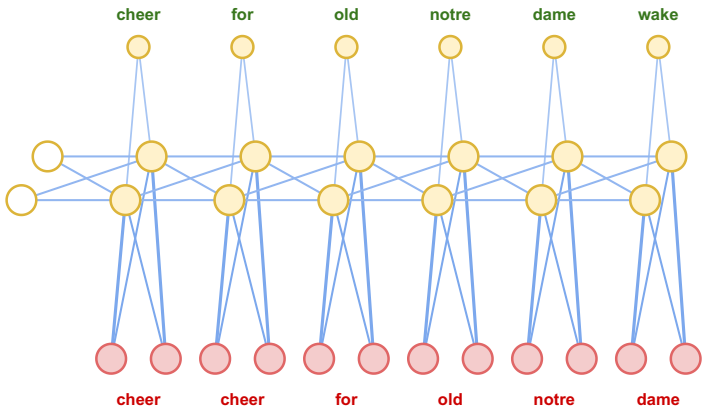
UNIT 03 · CHAPTER 3

Recurrent Neural Networks

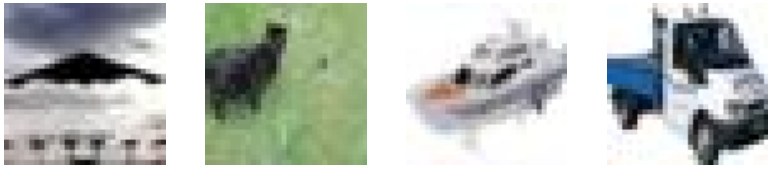


CSE 30124 · Introduction to Artificial Intelligence
William Theisen · University of Notre Dame

So Far

So far we've seen networks that take in **single samples** that lack _____

photos in order



airplane cat ship truck

shuffled: same labels



ship airplane truck cat

words in order

cheer cheer for old notre dame

shuffled: a different sentence

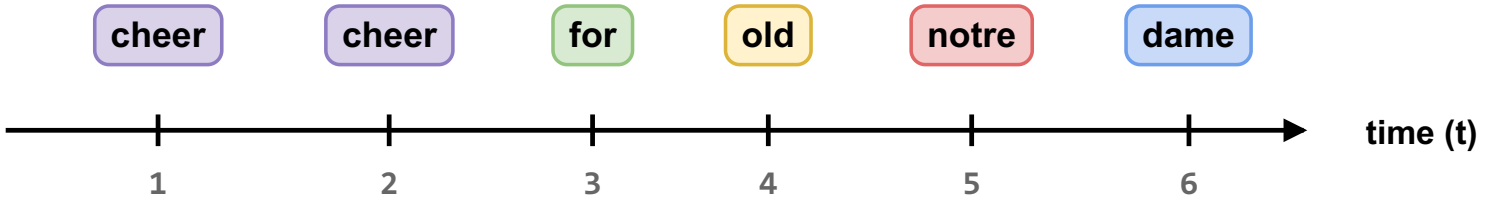
notre cheer for cheer dame old

When our CNN labels a photo, **it doesn't matter what photo came before it or comes after**. This is not the case for _____ like language

Sequence Data

Examples of **sequence data:**

- _____
- _____
- _____
- _____



How do we handle data where the **order** of what we're seeing **matters**? How do we _____ over time?

Language as Input

Computers understand only numbers so we need to somehow _____ _____ in such a way as to **enable computer understanding.** We call this subfield of **AI**: _____ **(NLP)**

Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr
32	20	SP	48	30	0	64	40	@	80	50	P	96	60	`	112	70	p
33	21	!	49	31	1	65	41	A	81	51	Q	97	61	a	113	71	q
34	22	"	50	32	2	66	42	B	82	52	R	98	62	b	114	72	r
35	23	#	51	33	3	67	43	C	83	53	S	99	63	c	115	73	s
36	24	$	52	34	4	68	44	D	84	54	T	100	64	d	116	74	t
37	25	%	53	35	5	69	45	E	85	55	U	101	65	e	117	75	u
38	26	&	54	36	6	70	46	F	86	56	V	102	66	f	118	76	v
39	27	'	55	37	7	71	47	G	87	57	W	103	67	g	119	77	w
40	28	(	56	38	8	72	48	H	88	58	X	104	68	h	120	78	x
41	29	)	57	39	9	73	49	I	89	59	Y	105	69	i	121	79	y
42	2A	*	58	3A	:	74	4A	J	90	5A	Z	106	6A	j	122	7A	z
43	2B	+	59	3B	;	75	4B	K	91	5B	[	107	6B	k	123	7B	{
44	2C	,	60	3C	<	76	4C	L	92	5C	\	108	6C	l	124	7C	|
45	2D	-	61	3D	=	77	4D	M	93	5D	]	109	6D	m	125	7D	}
46	2E	.	62	3E	>	78	4E	N	94	5E	^	110	6E	n	126	7E	~
47	2F	/	63	3F	?	79	4F	O	95	5F	_	111	6F	o	127	7F	DEL

NLP starts with two preprocessing steps: _____ and _____

Tokenization

Tokenization is the process of _____ a piece of **text** into **tokens**.

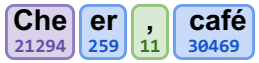
```python
import re
text = """Cheer, cheer for old Notre Dame,
Wake up the echoes cheering her name,
Send a volley cheer on high,
Shake down the thunder from the sky."""
clean = re.sub(r"[^a-z ]", " ", text.lower())
tokens = clean.split()
vocab = list(dict.fromkeys(tokens))  # first appearance
ids = [vocab.index(t) for t in tokens]
print(len(tokens), "tokens,", len(vocab), "words")
print(tokens[:9])
print(ids[:9])
```

OUTPUT

```
26 tokens, 22 words
['cheer', 'cheer', 'for', 'old', 'notre', 'dame', 'wake', 'up', 'the']
[0, 0, 1, 2, 3, 4, 5, 6, 7]
```

Tokenization Levels

There are several **tokenization levels**. Our four lines at each one, "Cheer, café" in tiles:

level	one token is	tokens	vocabulary
	a character	136	25
	a word	26	22
	a word or a piece of one	32	200,019

é isn't one of our 25 characters, but it is 2 of the 256 **bytes**

We want to **keep the vocabulary small** and **keep tokenized sequences short**

Subword Tokenizers

Common words get _____, rarer ones are _____.

Che er , cheer for old Notre Dame ,↵

Wake up the echoes cheering her name ,↵

Send a volley cheer on high ,↵

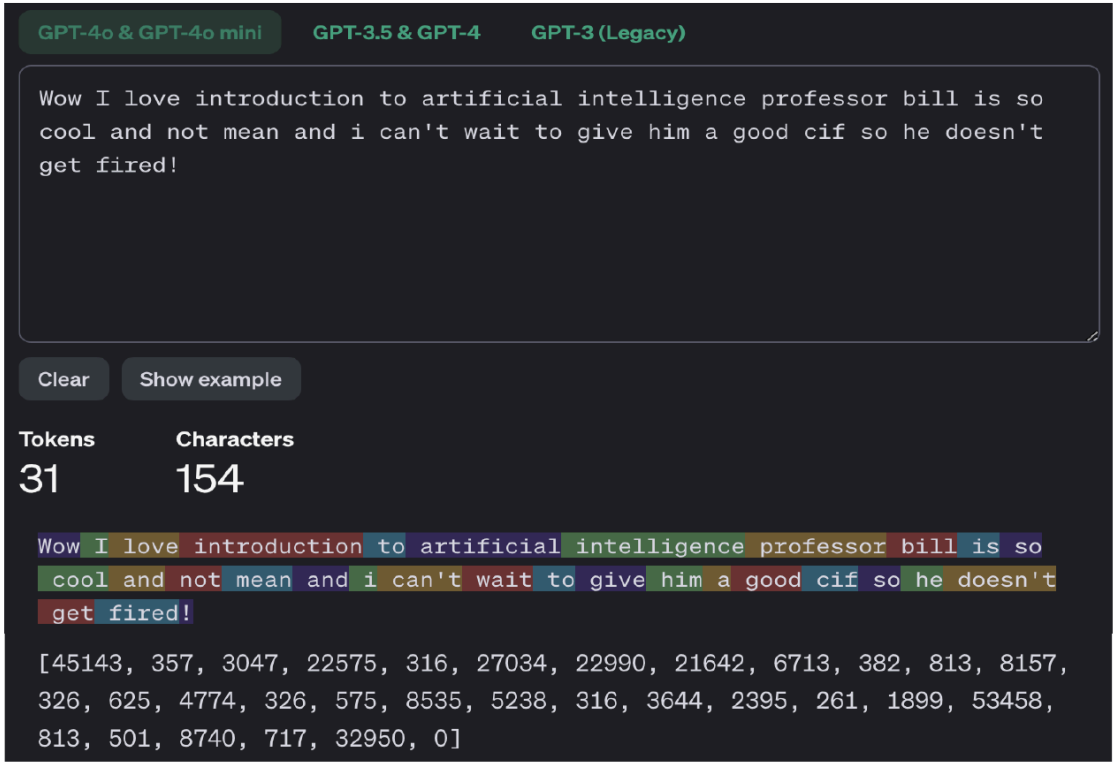
Shake down the thunder from the sky .

GPT-4o has about `200,000` **tokens**, so almost every word is a token of its own, with its space. "Cheer" at the start of a line is `2` pieces, Che er, but cheer after a space is one.

This implies that cheer at the start of a sentence is pretty rare, but it appears enough within sentences to deserve its own **token**.

Tokenization (exercise) - Openai tokenizer

OpenAI actually still has their tokenizer public, and you can play around with it!



Subword Tokenizers - Text as Bytes

Computers store text as **bytes**: 8 **bits**, so 256 possible values. "Cheer" is 5 **bytes**, one per character, in decimal, hex and binary:

	C	h	e	e	r
decimal	67	104	101	101	114
hex	43	68	65	65	72
binary	01000011	01101000	01100101	01100101	01110010

bytes starting with a 1 are used in **UTF-8** as pieces of longer characters instead: é is 2 **bytes**, an emoji is 4

	decimal	**hex**	**binary**
é	195 169	c3 a9	11000011 10101001
🍀	240 159 141 128	f0 9f 8d 80	11110000 10011111 10001101 10000000

Text as Bytes in Python

`.encode("utf-8")` turns a string into its bytes:

```python
text = "Cheer, café 🍀"
data = text.encode("utf-8")
print(len(text), "characters,", len(data), "bytes")
print(list(data))
print([bytes([b]) for b in data])
print([hex(b) for b in data])
print([bin(b) for b in data])
```

OUTPUT

```
13 characters, 17 bytes
[67, 104, 101, 101, 114, 44, 32, 99, 97, 102, 195, 169, 32, 240, 159, 141, 128]
[C, h, e, e, r, ,,  , c, a, f, \xc3, \xa9,  , \xf0, \x9f, \x8d, \x80]
['0x43', '0x68', '0x65', '0x65', '0x72', '0x2c', '0x20', '0x63', '0x61', '0x66', '0xc3', '0xa9',
'0x20', '0xf0', '0x9f', '0x8d', '0x80']
['0b1000011', '0b1101000', '0b1100101', '0b1100101', '0b1110010', '0b101100', '0b100000',
'0b1100011', '0b1100001', '0b1100110', '0b11000011', '0b10101001', '0b100000', '0b11110000',
'0b10011111', '0b10001101', '0b10000000']
```

Subword Tokenization Algorithm: Byte-Level BPE

Byte-level BPE starts from the 256 bytes (ID = the byte's value), then:

1. scans the text and _____
2. adds the most frequent pair to the vocabulary as a _____ with the next ID
3. replaces that pair everywhere in the text, then goes back to 1

Counts of pairs of our four lines: h e 8, e r 6, e e 4, c h 4, ␣ t 4, t h 4 …

ID	vocabulary	tokens
0–255	the bytes (C 67 is 01000011)	
256	h + e → he 256	128
257	e + r → er 257	123
258	he + er → heer 258	119
259	␣ + t → t 259	115
260	␣ + c → c 260	112
261	c + heer → cheer 261	109

136 bytes, 109 tokens after 6 merges. GPT-4o keeps going to about 200,000 tokens

Token IDs

A token's place in the **vocabulary** (in order of first appearance) is its **token ID**

cheer 0 | cheer 0 | for 1 | old 2 | notre 3 | dame 4

wake 5 | up 6 | the 7 | echoes 8 | cheering 9 | her 10 | name 11

send 12 | a 13 | volley 14 | cheer 0 | on 15 | high 16

shake 17 | down 18 | the 7 | thunder 19 | from 20 | the 7 | sky 21

token	ID	token	ID
cheer	0	name	11
for	1	send	12
old	2	a	13
notre	3	volley	14
dame	4	on	15
wake	5	high	16
up	6	shake	17
the	7	down	18
echoes	8	thunder	19
cheering	9	from	20
her	10	sky	21

Are we actually gaining anything from this?

The numbers say nothing about meaning of the words

thunder 19 isn't more like sky 21 than cheer 0

Word Embeddings

After **tokenization**, our next step is to **"embed"** each **token**. The **embedding table** has one row per **token ID**: for our `22` words, `GloVe's` is `<22, 300>`. **Embedding** a **token** is simply looking up its row.

token	ID	0	1	2	3	…	299
cheer	0	0.41	−0.15	−0.04	−0.37	…	0.19
for	1	−0.24	0.12	0.19	−0.27	…	0.16
old	2	−0.38	−0.07	−0.37	−0.34	…	1.12
notre	3	0.01	0.34	0.02	0.32	…	0.42
dame	4	−0.09	0.34	0.11	0.26	…	0.48
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
sky	21	−0.03	−0.22	0.72	0.20	…	−0.08

This gives us a `feature vector` for each **token**, similar to the `4,096` features our **CNN** made for each photo. The **embedding** is like the _____ of a token

GloVe Embeddings

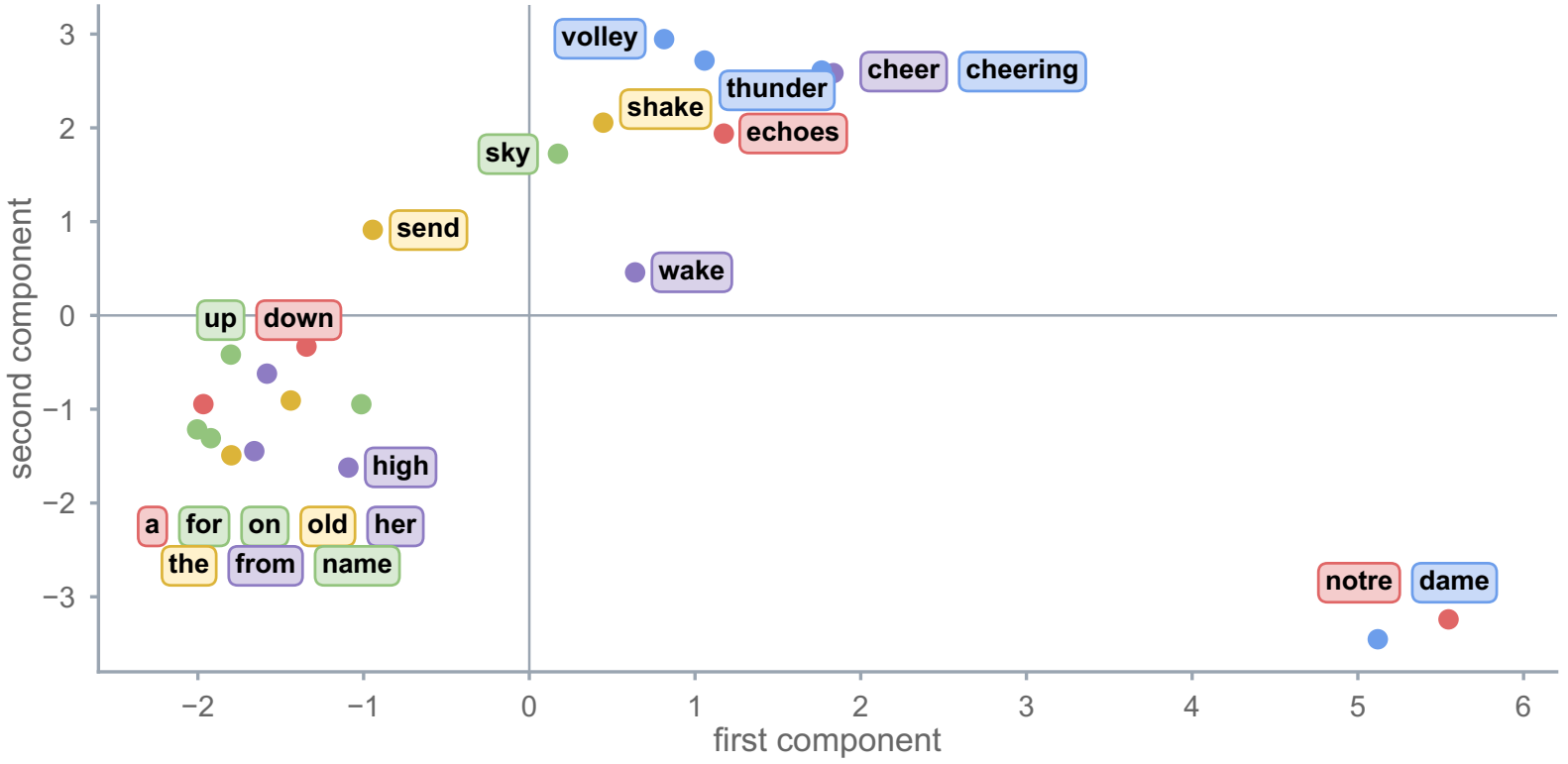
```python
import torch
vocab, rows = [], []
for line in open("glove-victory-march.txt"):
    word, *nums = line.split()  # a word, 300 numbers
    vocab.append(word)
    rows.append([float(n) for n in nums])
E = torch.tensor(rows)  # our 22 words, in song order
print(E.shape)
E = E / E.norm(dim=1, keepdim=True)  # length 1
S = E @ E.T  # cosine similarity of every pair
S = S - 2 * torch.eye(22)  # not its own neighbor
for w in ["cheer", "notre", "up", "thunder"]:
    s = S[vocab.index(w)]
    best = vocab[s.argmax()]
    print(f"{w!r:9} {best!r:10} {s.max():.2f}")
```

OUTPUT

```
torch.Size([22, 300])
'cheer'   'cheering' 0.61
'notre'   'dame'     0.91
'up'      'down'     0.75
'thunder' 'sky'      0.34
```

Word Embeddings

Converting **tokens** to **embeddings** allows us to interpret them as **points in high dimensional space**, which allows us to _____!

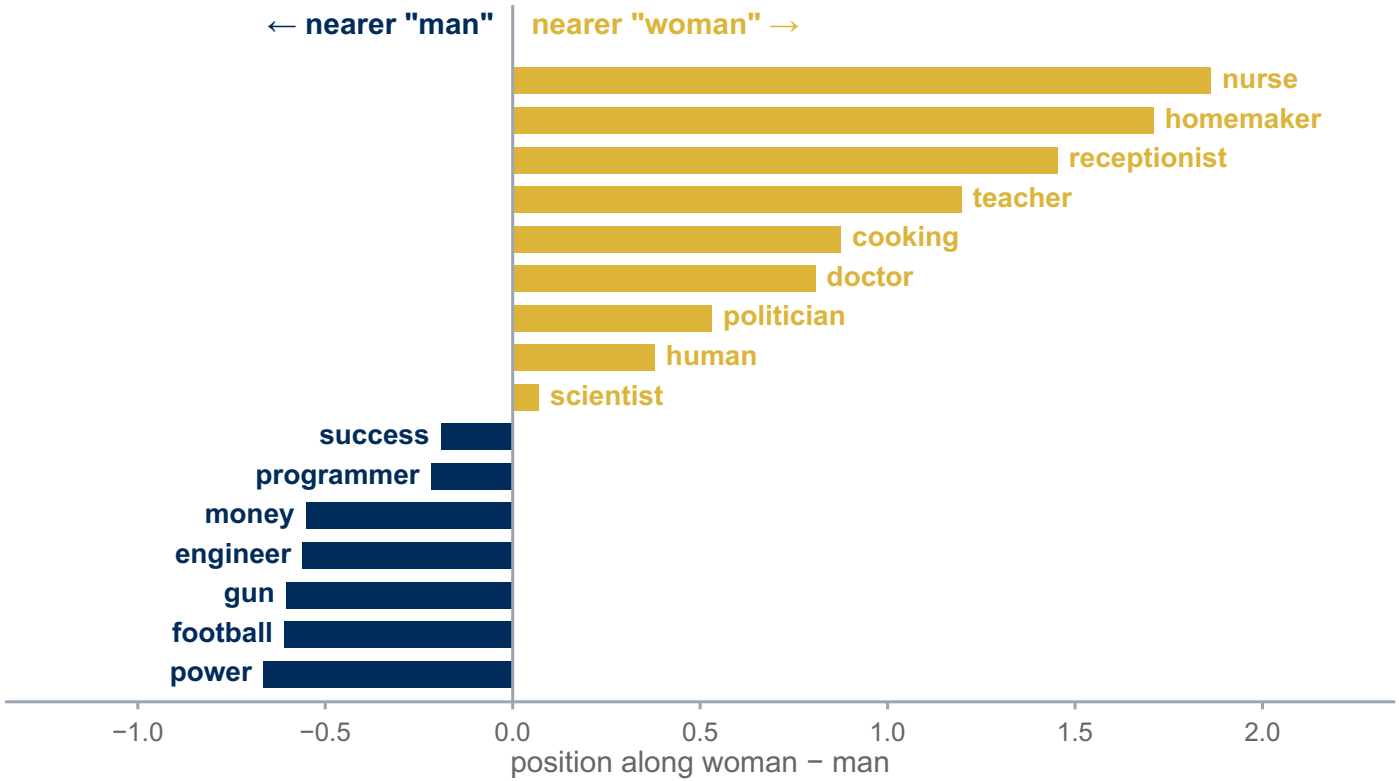


NOTE: In this plot we use **PCA** to view our **embeddings** in 2 dimensions but `GloVe` embeddings are actually `<1, 300>` for each **token**

Embeddings

GloVe embeddings are based on _____, even though **up** and **down** mean opposite things, they appear next to the same words, so they sit close together (`0.75`).

Embeddings learn whatever patterns the text has, **biases included**.

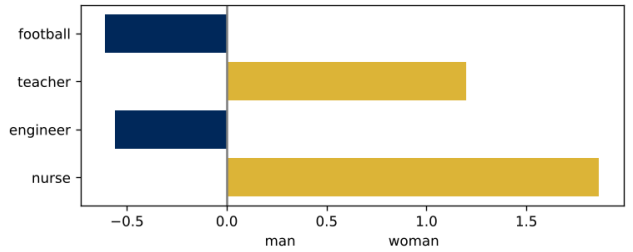


Exploration of Biases in Embeddings

```python
import numpy as np
import torch
import matplotlib.pyplot as plt
# type any words here, then Run,
# NOTE: it will only work if GloVe knows the word
words = ["nurse", "engineer", "teacher", "football"]
vocab = open("glove-20k-words.txt").read().split()
E = torch.tensor(np.load("glove-20k.npy")).float()
def vec(w):
    return E[vocab.index(w)]
axis = vec("woman") - vec("man")
axis = axis / axis.norm()  # the man-to-woman direction
words = [w for w in words if w in vocab]  # 20,002 words
pos = [float(vec(w) @ axis) for w in words]
for w, p in sorted(zip(words, pos), key=lambda t: t[1]):
    print(f"{w:12} {p:6.2f}")
cols = ["#002B5B" if p < 0 else "#DCB439" for p in pos]
plt.figure(figsize=(7, 0.4 * len(words) + 1))
plt.barh(words, pos, color=cols)
plt.axvline(0, color="gray")
plt.xlabel('man                    woman')
plt.show()
```

OUTPUT

```
football      -0.61
engineer      -0.56
teacher        1.20
nurse          1.86
```



Python RNN by Hand - Embedding Layer

An embedding layer is just _____

```python
import torch

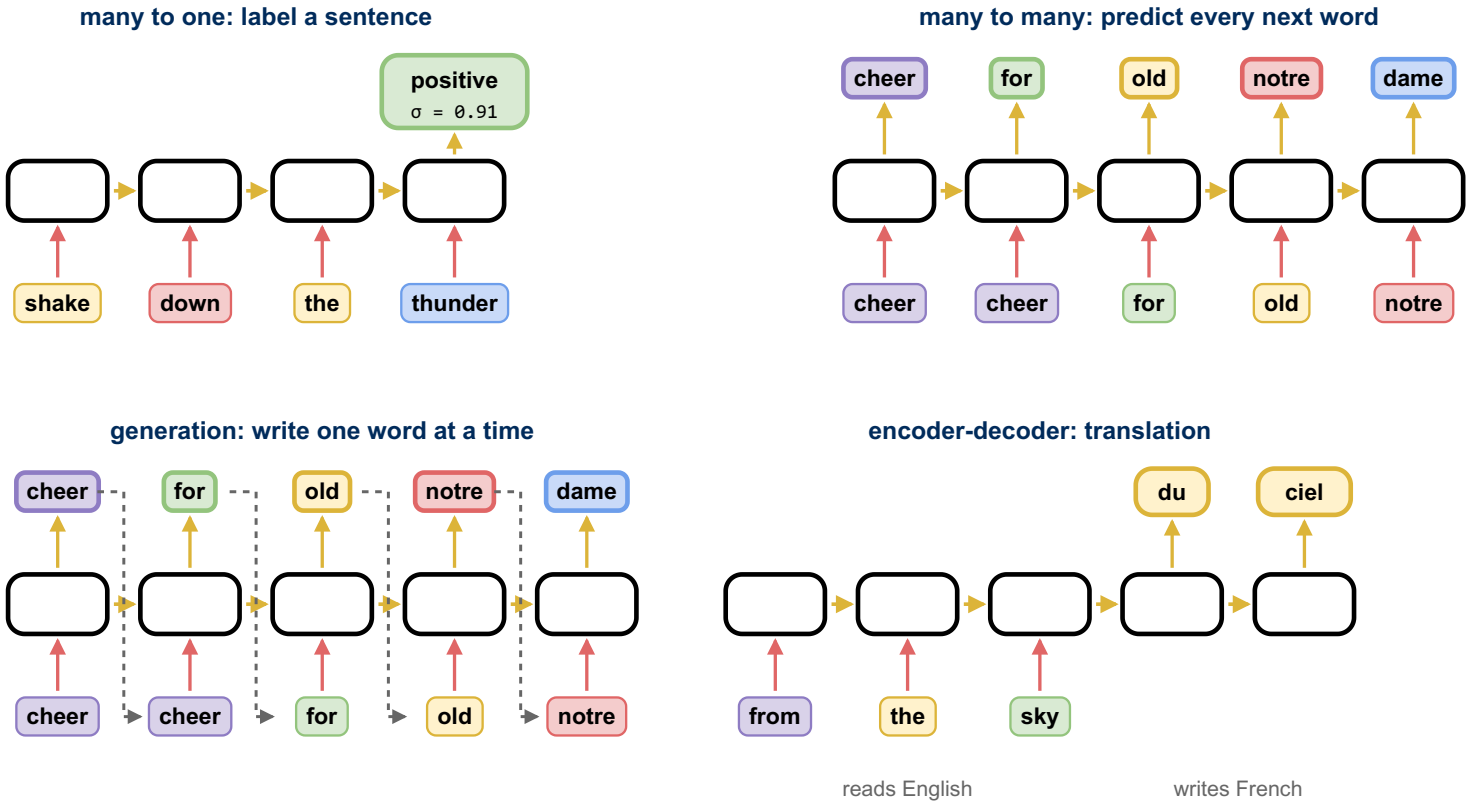
class Embedding:
    def __init__(self, n_tokens, d):
        self.E = torch.randn(n_tokens, d) * 0.1

    def forward(self, ids):  # (B, T) token IDs
        self.ids = ids       # keep for backward
        return self.E[ids]   # (B, T, d): one row per ID
```

Basically all **NLP models**, including **LLMs** like **Claude** and **ChatGPT** share the same two preprocessing steps: **tokenization** and **embedding**

NLP Tasks

Tokenization and **embedding** are only the `preprocessing` steps we need before we use an **NLP** model, they only give us the _____ for our model.



Some **tasks** we've seen so far are **regression** and **classification**, but **NLP** has some new **tasks** of its own.

NLP Task: Generation

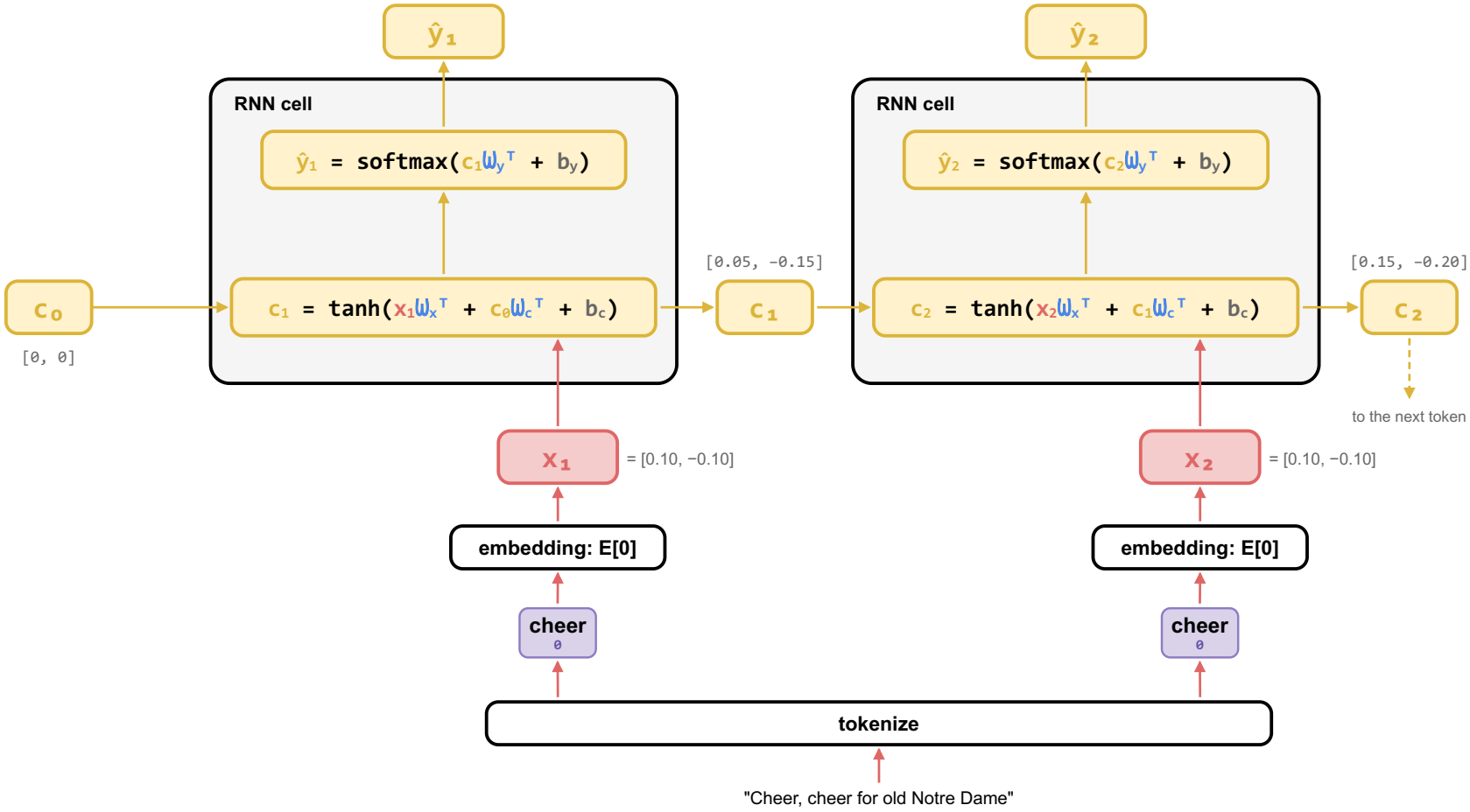
All four of the NLP tasks require _____. If we are predicting the next token after the current one, after notre always comes dame, but picking the next token after the is harder



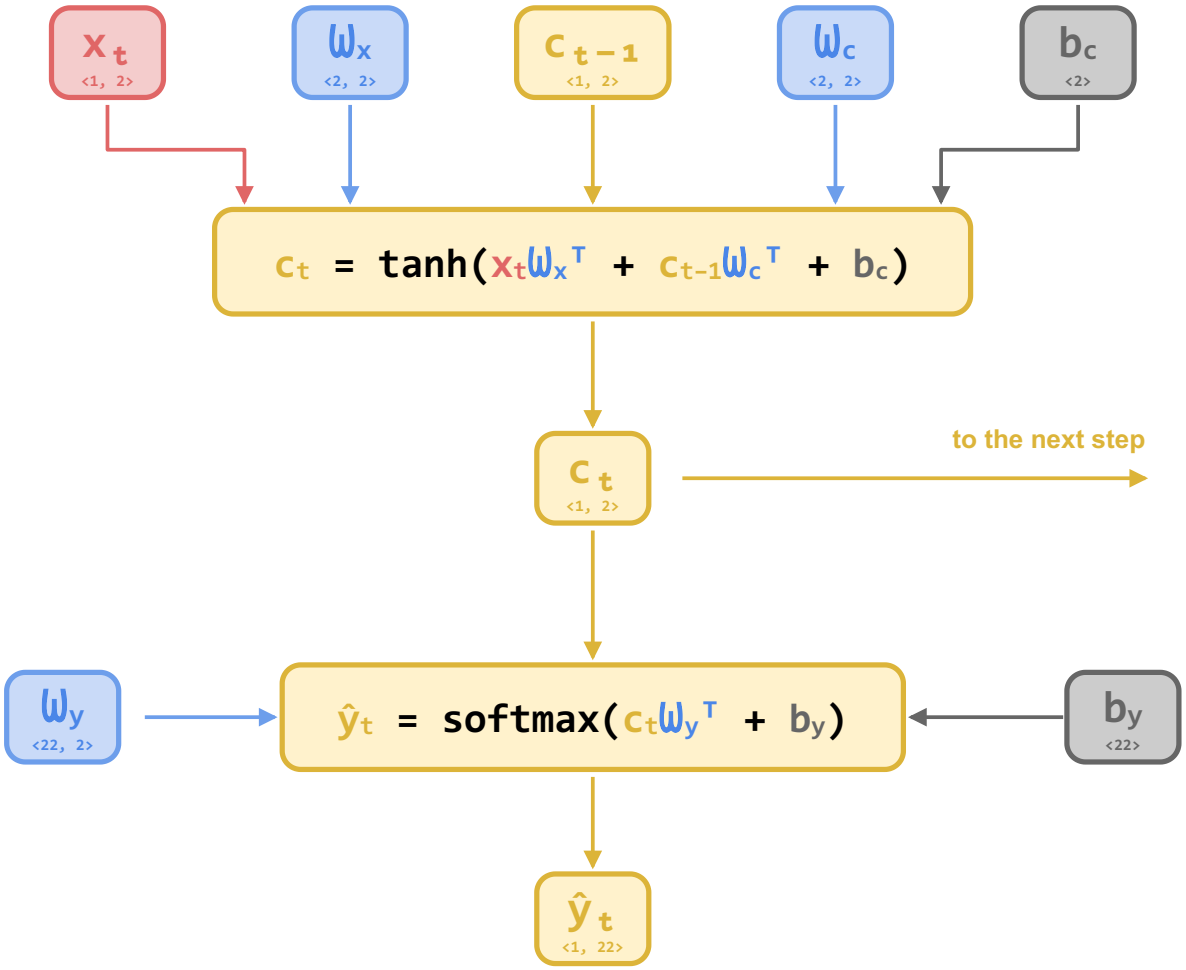
We need a model that reads _____ at a time but also maintains the _____ of what we've seen so far over `time`

Sequence Networks - RNNs (1986)

We can begin to solve the problem of **context** by using a new model, called a _____ **(RNN)**



Exploring the RNN

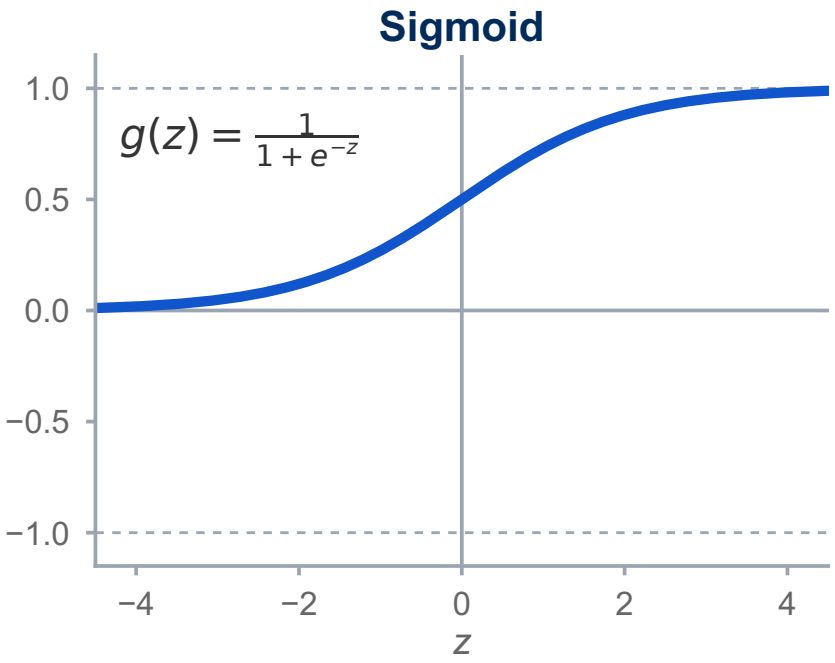


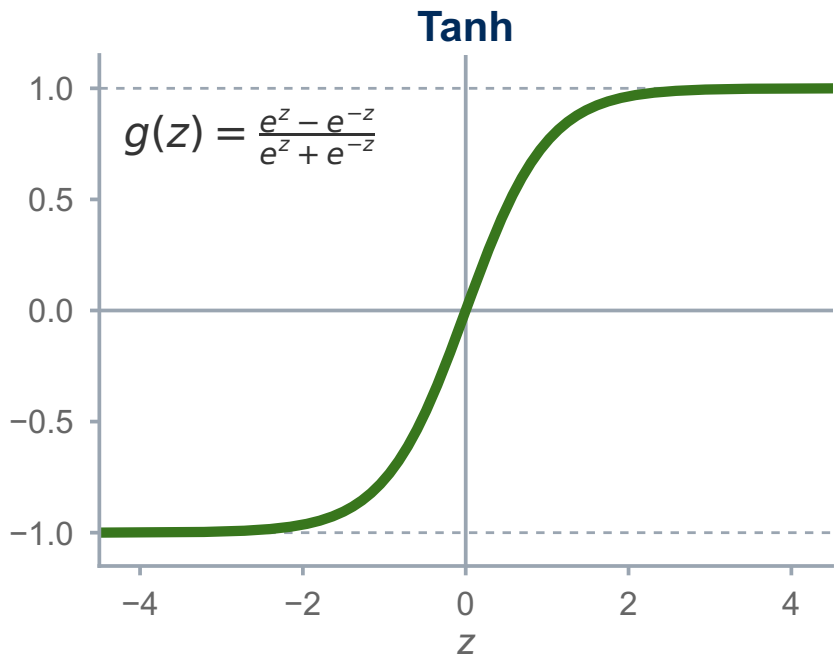
Fully Connected Layers vs. RNNs (Old Exam Question)

Old exam question: Write the "equations" for a fully connected layer and an RNN. Explain the reasoning behind the difference between the two.

Activation Functions - tanh

While we've seen the _____ and _____, in **RNNs** the **tanh function** is used.

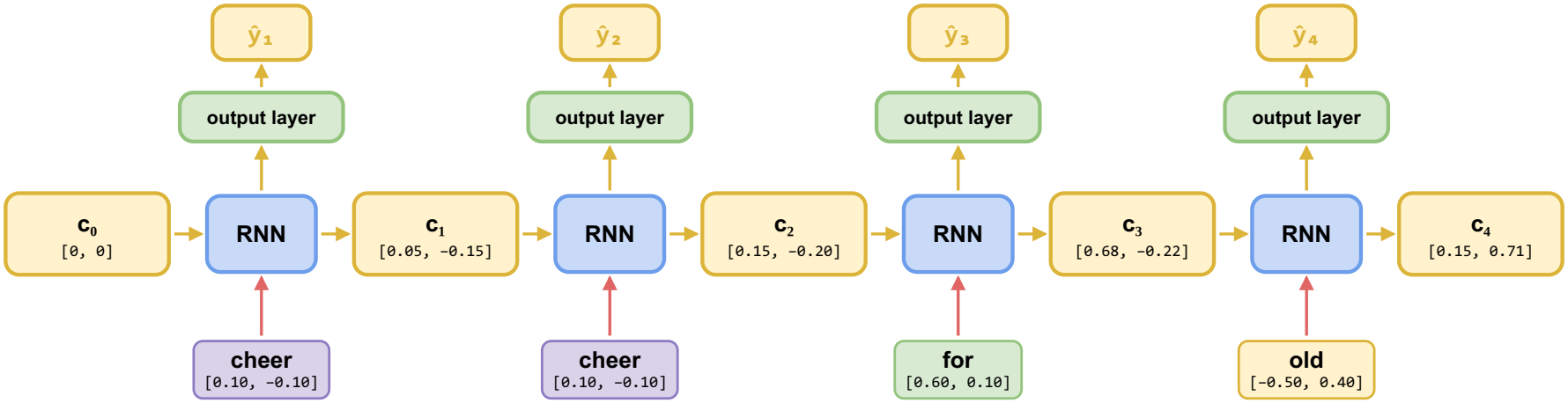




Why we use the **tanh function** will become clear when we starting training **RNNs** next lecture

Time After Time

Most **RNN** diagrams depict them in their "unrolled" state, which can be confusing

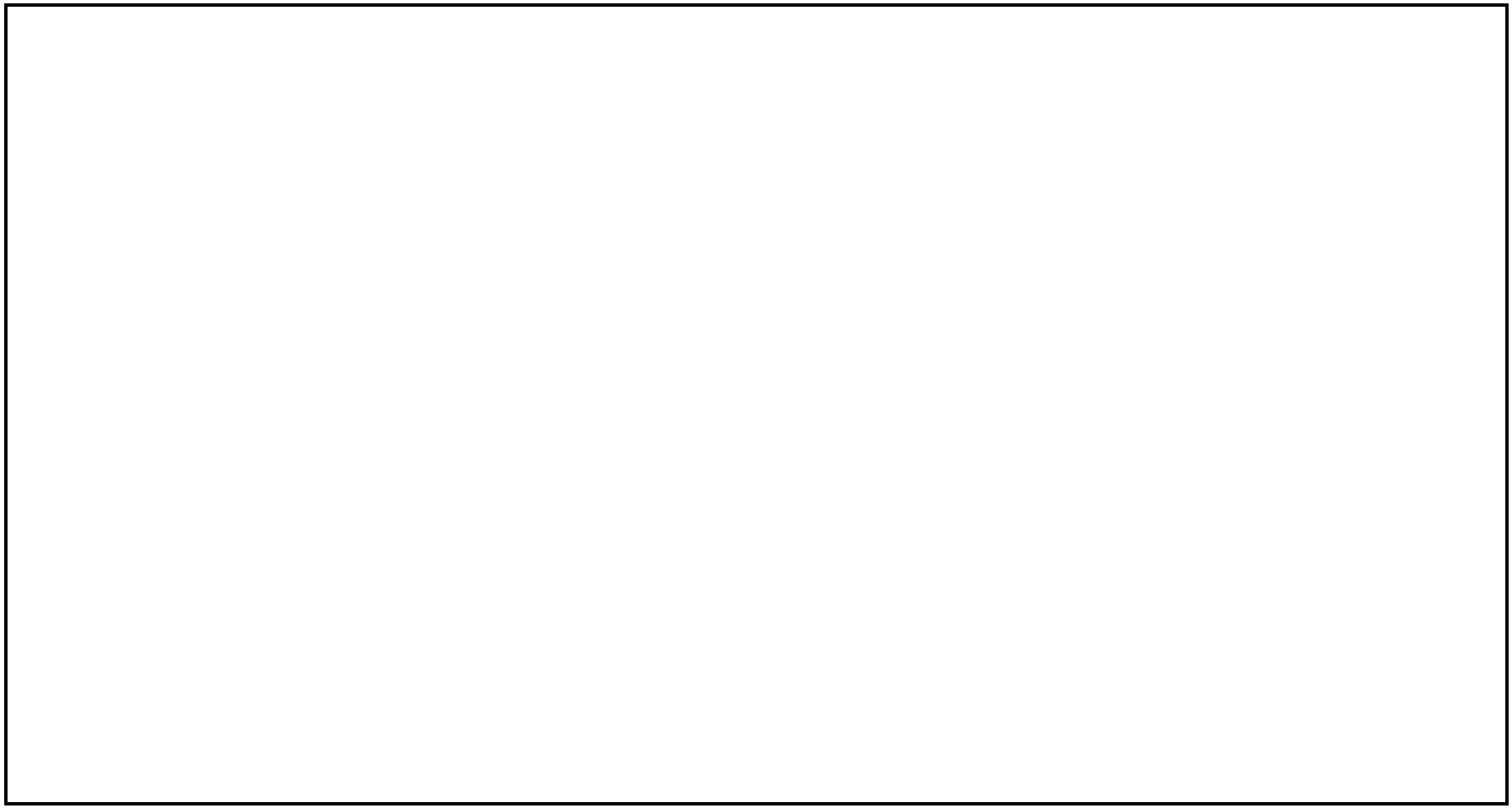


Unlike an **FFN**, this isn't four different layers:

every **RNN** cell and **output layer** shares _____.

RNNs (Old Exam Questions)

Old exam question: Draw the computation graph for an **RNN**, for the sentence "Good Luck", with a word level **tokenizer**. Only output for the final **token**.



Python RNN by Hand - Recurrent Layer

One **step** is the **RNN** equation: $c_t = \tanh(x_t W_x^T + c_{t-1} W_c^T + b_c)$

```python
class RNNLayer:
    def __init__(self, n_in, n_h):
        self.Wx = torch.randn(n_h, n_in) / n_in**0.5
        self.Wc = torch.randn(n_h, n_h) / n_h**0.5
        self.b = torch.zeros(n_h)

    def _step(self, x_t, c_prev):  # one word, c_t
        return torch.tanh(x_t @ self.Wx.T + c_prev @ self.Wc.T + self.b)

    def forward(self, X):  # (B, T, n_in)
        B, T, _ = X.shape
        c = torch.zeros(B, self.Wc.shape[0])  # c_0
        self.states = [c]  # keep every c_t for BPTT

        for t in range(T):  # one token at a time
            c = self._step(X[:, t], c)  # same weights, C is our "context"
            self.states.append(c)

        return torch.stack(self.states[1:], dim=1)
```

The `forward pass` is _____, reusing the **same weights**

What Makes Them "Recurrent"? (Old Exam Question)

Old exam question: Why do we call them recurrent neural networks? What makes them "recurrent"?

The Forward Pass

Our **tokens**, a random **embedding table** and **output layer**, and our two weight matrices:

```
import re
import numpy as np
import torch
text = """Cheer, cheer for old Notre Dame,
Wake up the echoes cheering her name,
Send a volley cheer on high,
Shake down the thunder from the sky."""
tokens = re.sub(r"[^a-z ]", " ", text.lower()).split()
vocab = list(dict.fromkeys(tokens))
ids = [vocab.index(t) for t in tokens]
rng = np.random.default_rng(0)
E = torch.tensor(rng.normal(size=(22, 2)).round(1),
                 dtype=torch.float32)  # embeddings
W_y = torch.tensor(rng.normal(size=(22, 2)).round(1),
                   dtype=torch.float32)  # output layer
W_x = torch.tensor([[1.0, 0.5], [-0.5, 1.0]])
W_c = torch.tensor([[0.5, -0.5], [0.5, 0.5]])
```

The Forward Pass

Every word through the **RNN**, in order: _____

```
b_c, c = torch.zeros(2), torch.zeros(2)  # bias, c₀
for t, i in enumerate(ids):
    c = torch.tanh(E[i] @ W_x.T + c @ W_c.T + b_c)
    if tokens[t] == "the":
        print(tokens[t-2:t+1], c.round(decimals=3))
```

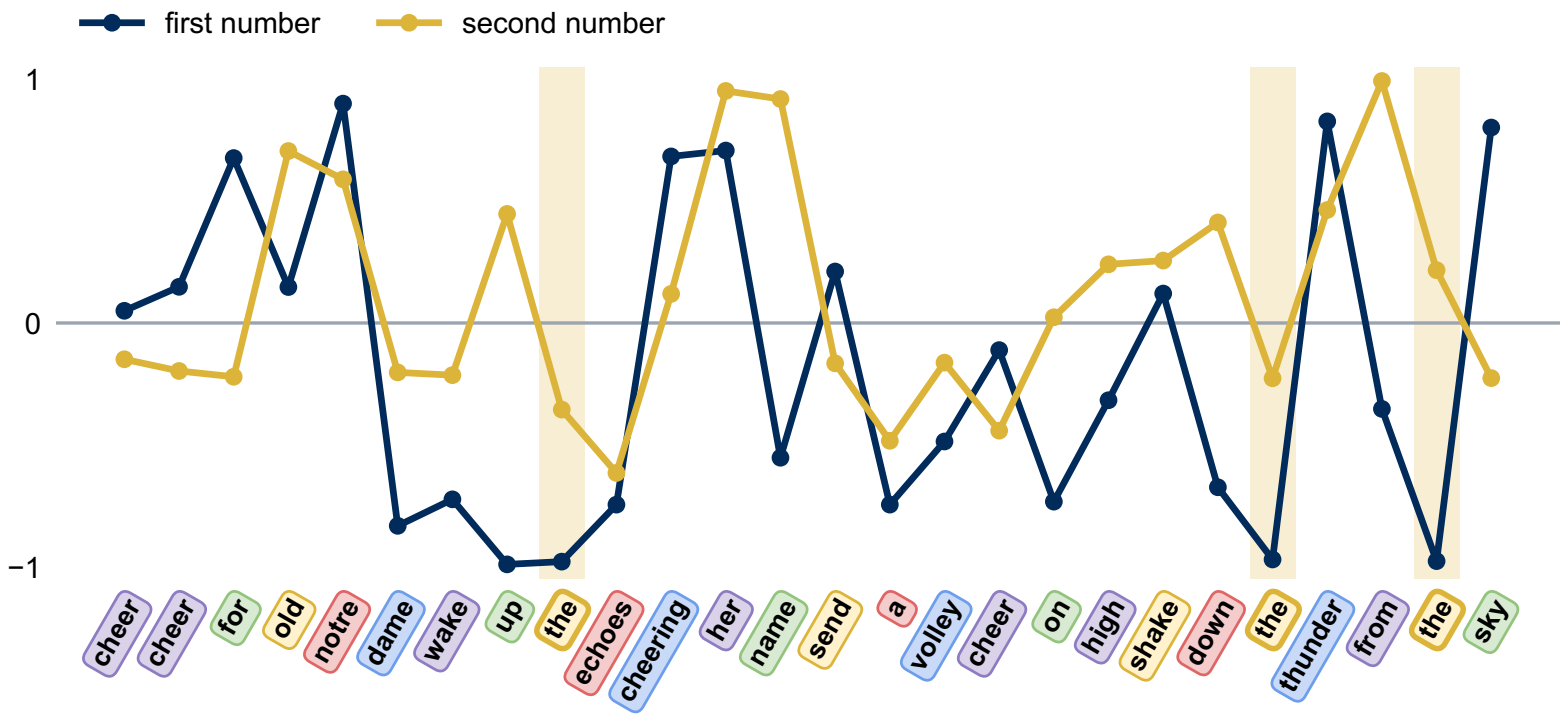
OUTPUT

```
[wake, up, the] tensor([-0.9790, -0.3550])
[shake, down, the] tensor([-0.9700, -0.2270])
[thunder, from, the] tensor([-0.9770,  0.2170])
```

The three **the**s have different _____, so a later `_step` can tell them apart

Context of a Token

Our simple **RNN's** **context matrix** is a **<1, 2>** matrix



NOTE: The **context** of what we've seen so far is often called the **hidden state**

RNNs in PyTorch

`nn.Embedding` is _____, `nn.RNN` runs _____, `nn.Linear` is _____:

```
from torch import nn
emb = nn.Embedding(22, 2) # Embedding layer
rnn = nn.RNN(2, 2, batch_first=True) # RNN block
fc = nn.Linear(2, 22)  # the output layer
with torch.no_grad():  # our weights, not random ones
    emb.weight.copy_(E)
    rnn.weight_ih_l0.copy_(W_x)
    rnn.weight_hh_l0.copy_(W_c)
    rnn.bias_ih_l0.zero_()
    rnn.bias_hh_l0.zero_()
    fc.weight.copy_(W_y)
    fc.bias.zero_()
    x = emb(torch.tensor([ids]))  # 1 song, 26 words
    out, c = rnn(x)
    y_hat = fc(out).softmax(dim=-1)
print(tuple(out.shape), tuple(y_hat.shape))
print(out[0, 8].round(decimals=3))  # wake up the
e = vocab.index("echoes")  # the real next word
print(y_hat[0, 8, e].round(decimals=3))
```

OUTPUT

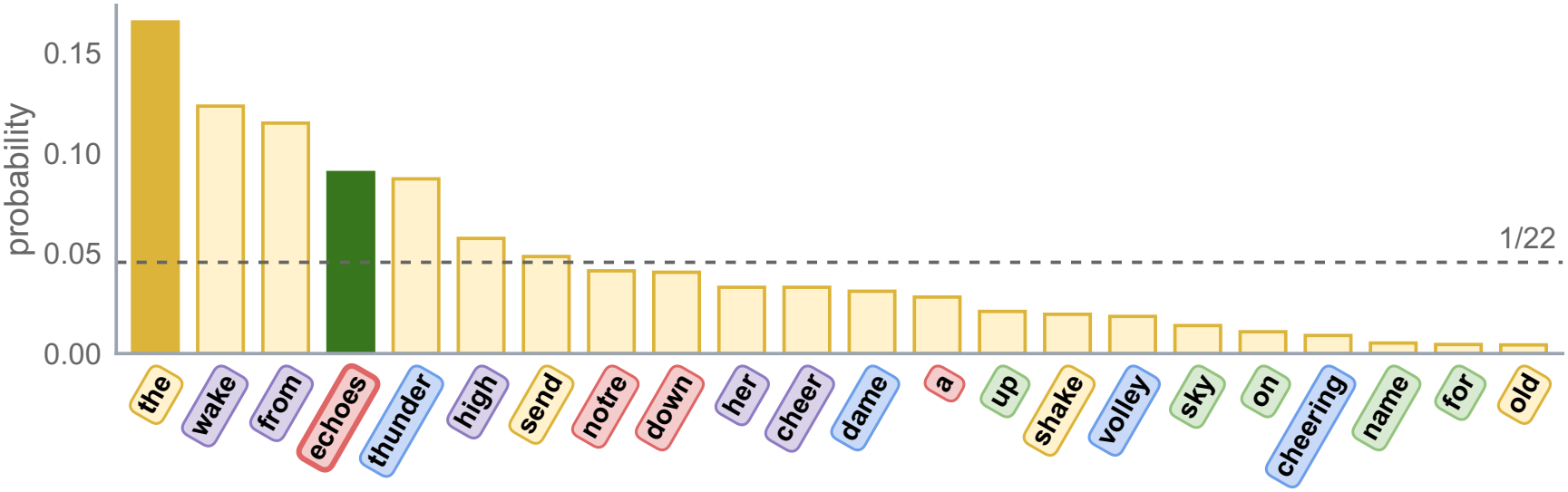
```
(1, 26, 2) (1, 26, 22)
tensor([-0.9790, -0.3550])
tensor(0.0900)
```

Output from an RNN

For our task of predicting the next **token** in a sequence, the output from our **RNNs output layer** is a probability of every **token** in our vocabulary being the next **token**

the prediction at step t

$$\hat{y}_t = \text{softmax}(c_t W_y^T + b_y)$$

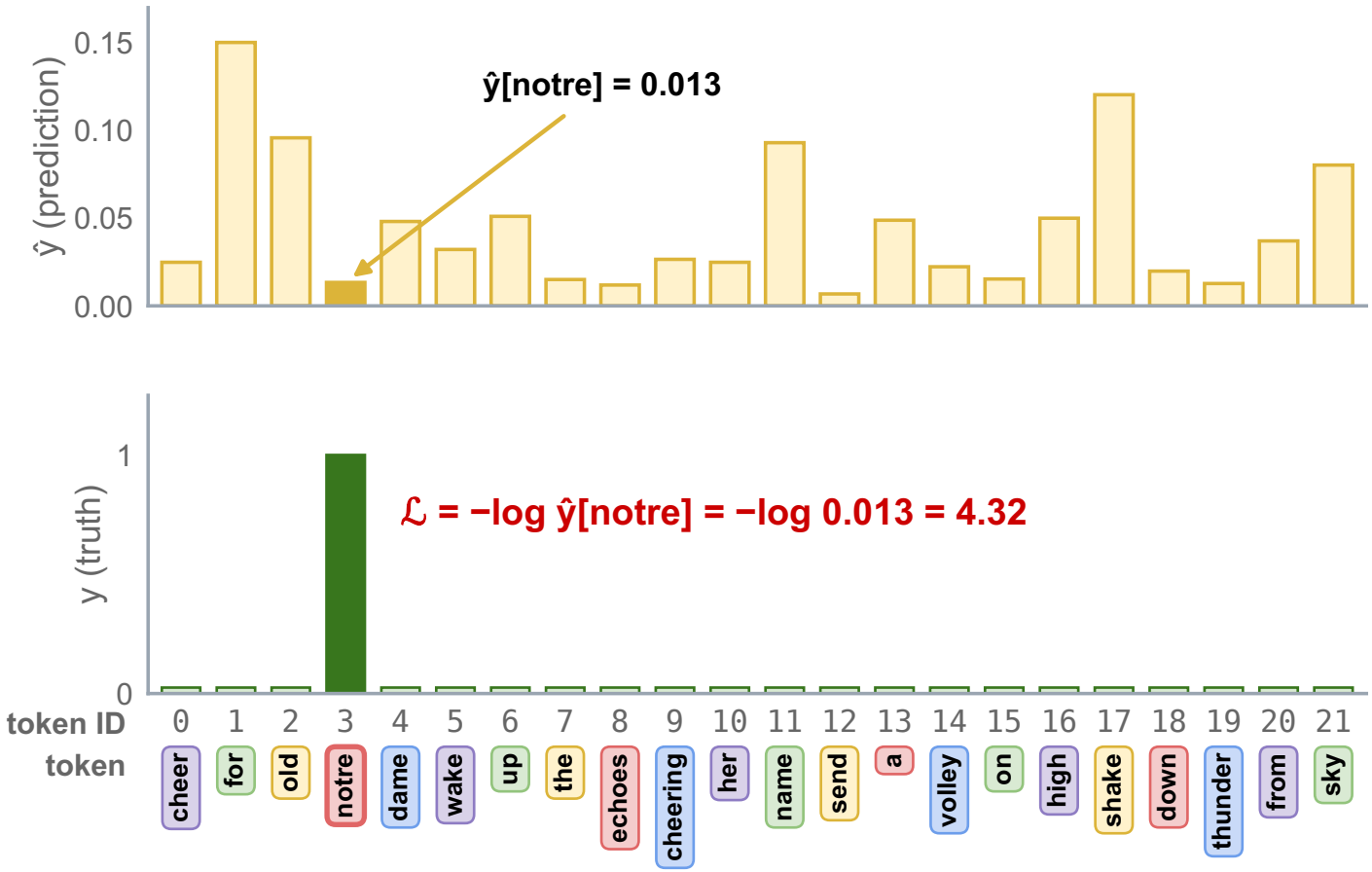


After **wake** **up** **the**: **echoes** gets `0.090`

This is identical to how an LLM like **ChatGPT** or **Claude** creates output too!

Output from an RNN

To train a text generation model, we can re-use cross entropy loss and think of text generation almost as a classification task.



Python RNN by Hand - Linear Output Layer and Network

We can use the same `LinearLayer` and `Network` as last time.

```
class LinearLayer:
    def __init__(self, n_in, n_out):
        self.W = torch.randn(n_out, n_in) * (2/n_in)**0.5
        self.b = torch.zeros(n_out)

    def forward(self, X):  # every c_t at once
        self.X = X         # keep for backward
        return X @ self.W.T + self.b

class Network:
    def __init__(self, layers):
        self.layers = layers

    def forward(self, X):
        for layer in self.layers:
            X = layer.forward(X)
        return X  # logits
```

Running Our RNN by Hand

```python
import re
text = """Cheer, cheer for old Notre Dame,
Wake up the echoes cheering her name,
Send a volley cheer on high,
Shake down the thunder from the sky."""

tokens = re.sub(r"[^a-z ]", " ", text.lower()).split()
vocab = list(dict.fromkeys(tokens))
X = torch.tensor([[vocab.index(t) for t in tokens]])
torch.manual_seed(0)
model = Network([Embedding(22, 2), RNNLayer(2, 2), LinearLayer(2, 22)])
logits = model.forward(X)
emb, rnn, out = model.layers
print("IDs", tuple(X.shape), "-> logits", tuple(logits.shape))
print("hidden states:", len(rnn.states))
n = [emb.E, rnn.Wx, rnn.Wc, rnn.b, out.W, out.b]
print("weights:", sum(w.numel() for w in n))
```

OUTPUT

```
IDs (1, 26) -> logits (1, 26, 22)
hidden states: 27
weights: 120
```

The number of weights **doesn't depend on how long the input sequence is**.

Text Generation

Predict the next **token**, then _____. We call this process **autoregressive** text generation

```python
import re
import torch

song = """Cheer, cheer for old Notre Dame,
Wake up the echoes cheering her name,
Send a volley cheer on high,
Shake down the"""

split_song = re.sub(r"[^a-z ]", " ", text.lower()).split() # split song into tokens
num_new_tokens = 10 # generate 3 new tokens

for t in range(num_new_tokens):
    # get the token IDs for each token in our song
    X = torch.tensor([[vocab.index(token) for token in split_song]])
    logits = model.forward(X) # run our RNN on the token IDs to produce prediction logits
    split_song.append(vocab[logits[0, -1].argmax()]) # append the most likely next token to the song

print(split_song)  # print original song with the 5 new predicted tokens on the end
```

OUTPUT

```
[cheer, cheer, for, old, notre, dame, wake, up, the, echoes, cheering, her, name, send, a,
volley, cheer, on, high, shake, down, the, thunder, from, the, sky, the, name, the, name, the,
name, the, name, the, name]
```