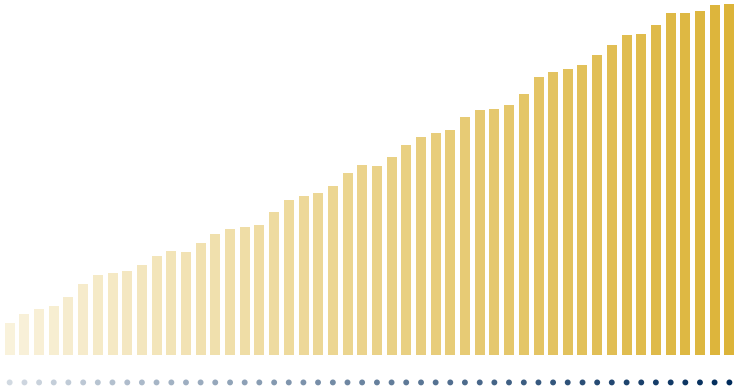
UNIT 03 · CHAPTER 4

Backpropagation Through Time



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

Feed-Forward Networks (old exam question)

```python
import torch
import torch.nn as nn
class FFN(nn.Module):
    def __init__(self):
        super(FFN, self).__init__()
        self.F1 = nn.Flatten()
        self.L1 = nn.Linear(28*28, 512)
        self.R1 = nn.Sigmoid()
        self.L2 = nn.Linear(512, 256)
        self.R2 = nn.Sigmoid()
        self.L3 = nn.Linear(256, 10)
    def forward(self, x):
        x = self.F1(x)
        x = self.L1(x)
        x = self.R1(x)
        x = self.L2(x)
        x = self.R2(x)
        x = self.L3(x)
        return x
```

a.) Draw the computation graph for the network.

Feed-Forward Networks (old exam question)

Note: You can assume `softmax` and `cross entropy loss` are used after the final layer, the two yielding a combined derivative of: $\hat{y} - y$

b.) Give the loss as a single nested equation.

$\mathcal{L} =$

c.) Give the derivative chain for calculating the gradient of the weights in the first layer.

$$\frac{\partial \mathcal{L}}{\partial w_1} =$$

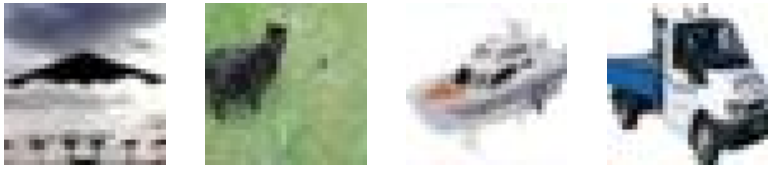
d.) Write the actual equation you'd use to calculate the gradient for the weights in the first layer.

$$\frac{\partial \mathcal{L}}{\partial w_1} =$$

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

Tokenization

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

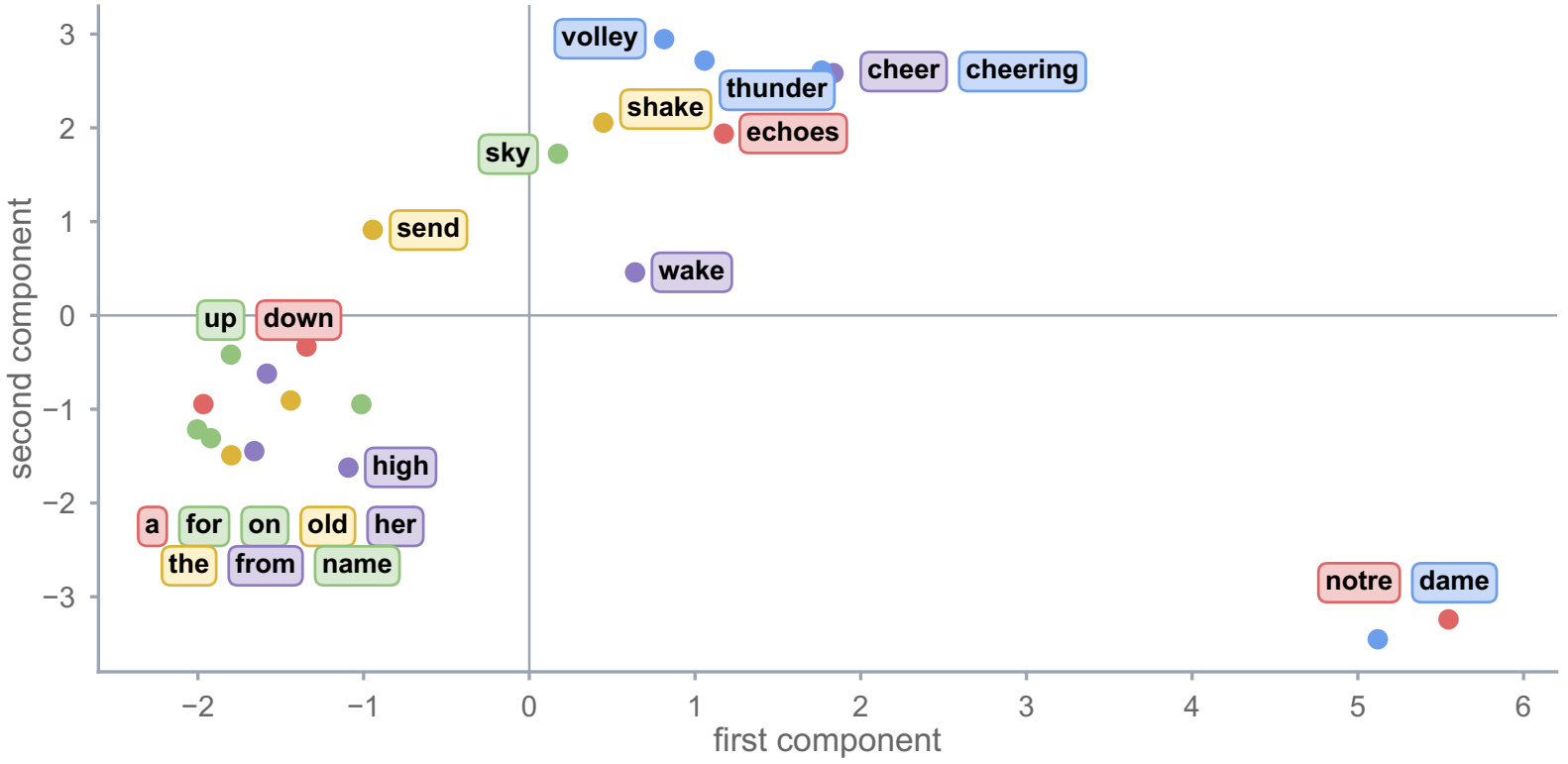
```python
import re
text = """Cheer, cheer for old Notre Dame,
Wake up the echoes cheering her name,
Send a volley cheer on high,
Shake down the thunder from the sky."""
clean = re.sub(r"[^a-z ]", " ", text.lower())
tokens = clean.split()
vocab = list(dict.fromkeys(tokens))  # first appearance
ids =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

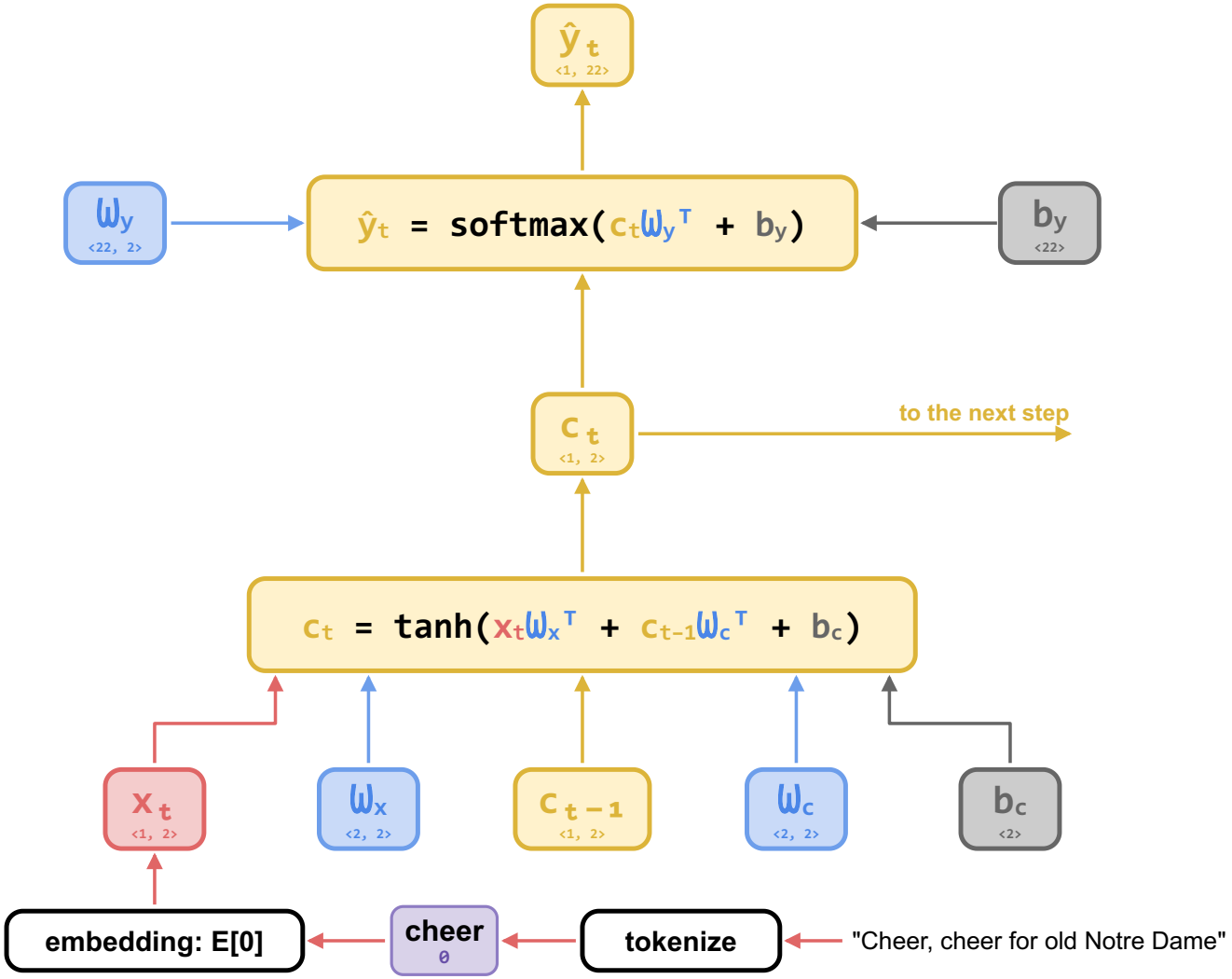
Word Embeddings

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**

Recurrent Neural Networks

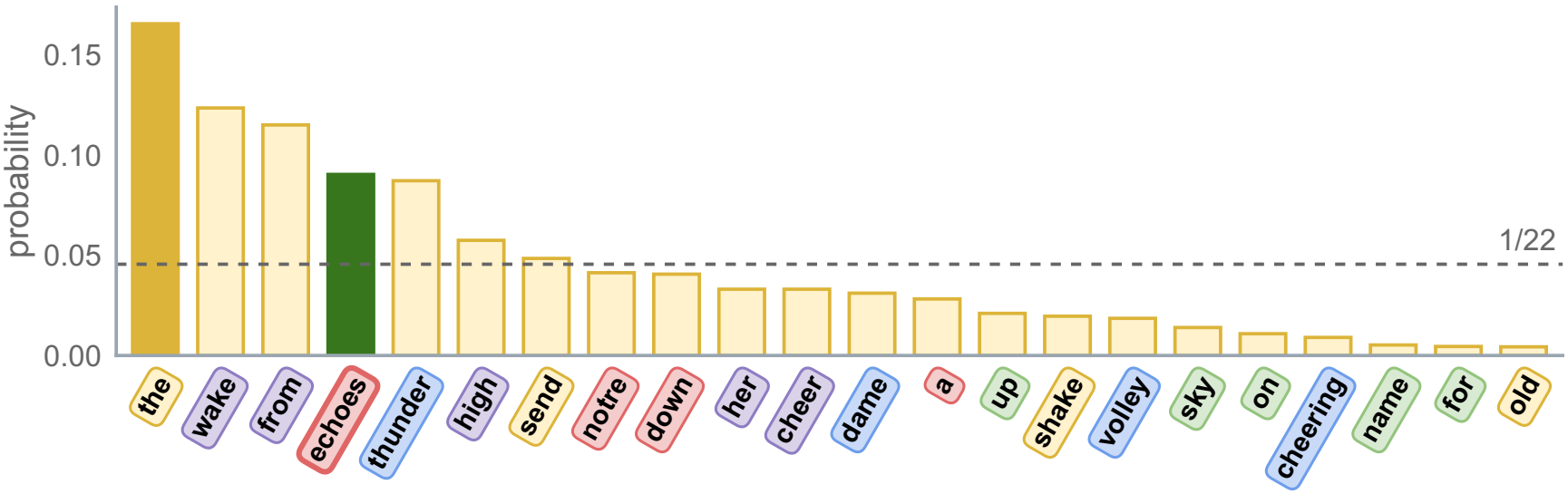


Output from an RNN

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

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

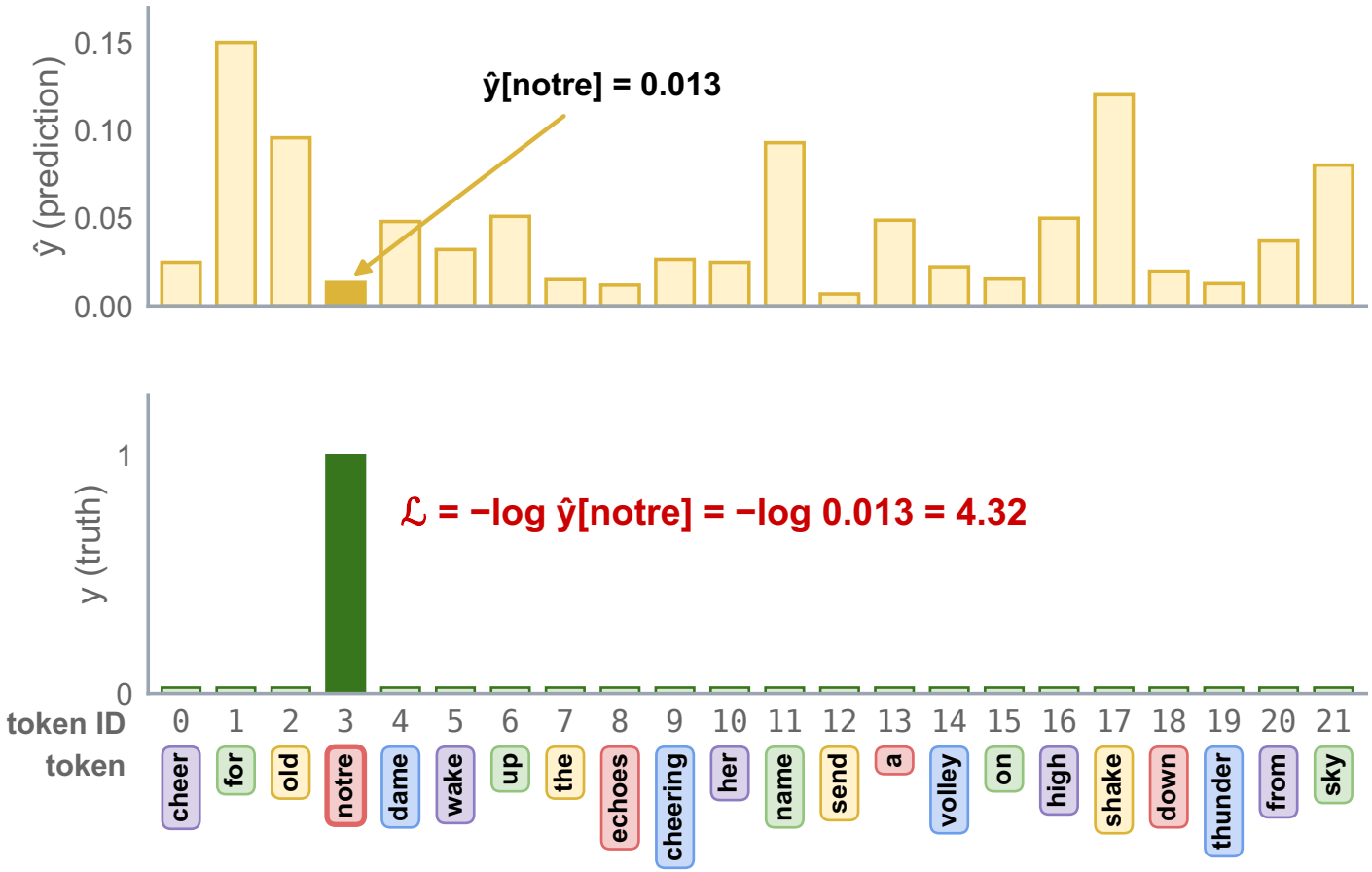


After **wake** **up** **the**: **echoes** gets `0.090`

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

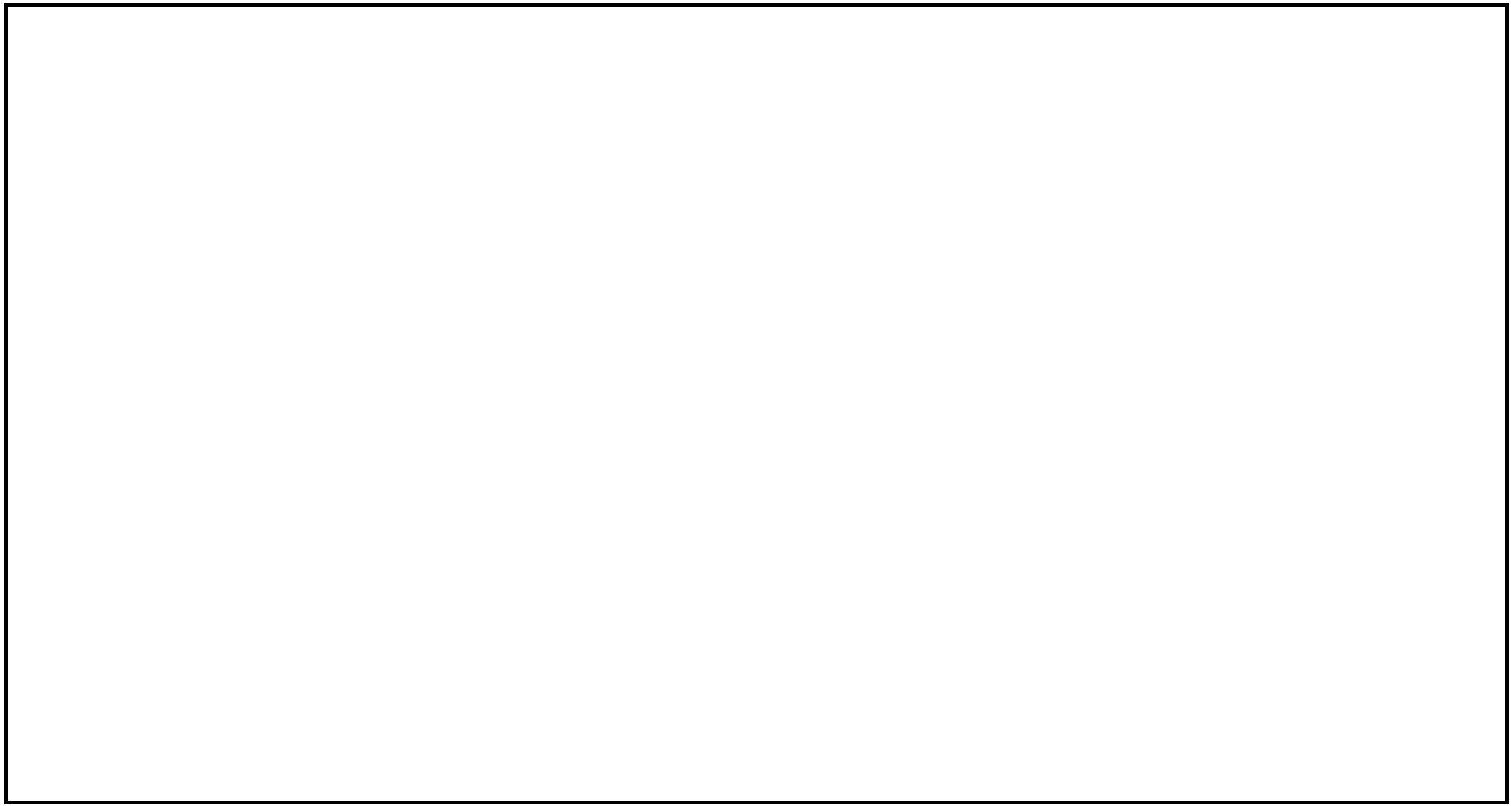
Output from an RNN

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



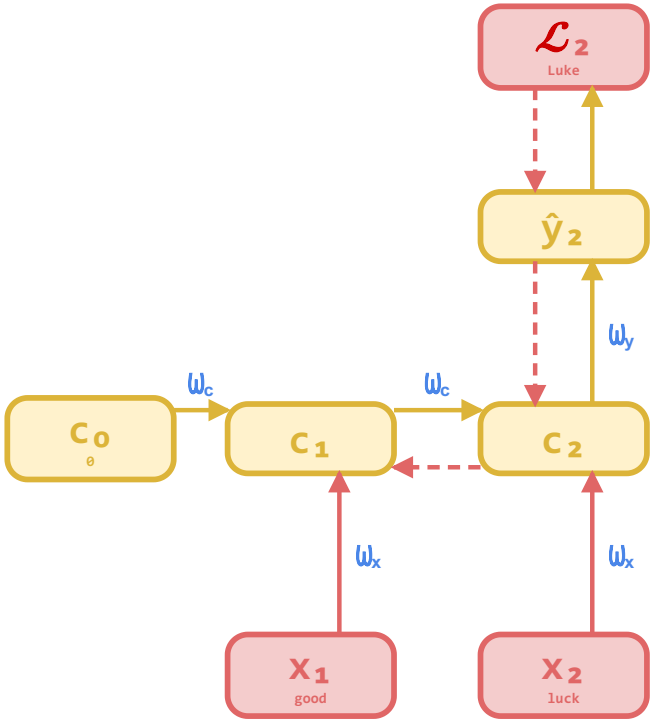
RNNs (Old Exam Questions)

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



Backpropagation Through Time

Backpropagation through time (BPTT) is just _____ _____, from the last word back to the first



Chain Rule - Multiplication and Addition

Along one path the chain rule _____ the derivatives. When something is used more than once, like w_c at both steps of "Good Luck", we ____ its paths:

b.) Give the loss for our "Good Luck" **RNN** (output only for the final token) as a single nested equation.

$\mathcal{L} =$

c.) Give the derivative chain for calculating the gradient of the input weights w_x.

$$\frac{\partial \mathcal{L}}{\partial w_x} =$$

d.) Write the actual equation you'd use to calculate the gradient for the input weights w_x.

$$\frac{\partial \mathcal{L}}{\partial w_x} =$$

BPTT by Hand - Embedding Layer

Last lecture's classes only went **forward**. Now each layer gets a **backward** that takes ___ _____ and returns _____.

```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
        return self.E[ids]   # (B, T, d)

    def backward(self, dX):  # (B, T, d): dℒ/dx_t
        d = self.E.shape[1]
        self.dE = torch.zeros_like(self.E)
        # each row we looked up gets its x_t's gradient
        self.dE.index_add_(0, self.ids.flatten(), dX.reshape(-1, d))

    def update(self, lr):
        self.E -= lr * self.dE
```

BPTT by Hand - Recurrent Layer

The same **forward pass** as last time, except it also keeps the inputs **X** for **backward**.

```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):
        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
        self.X = X  # keep the inputs for backward too
        c = torch.zeros(B, self.Wc.shape[0])  # c_0
        self.states = [c]  # every c_t, for BPTT
        for t in range(T):
            c = self._step(X[:, t], c)
            self.states.append(c)
        return torch.stack(self.states[1:], dim=1)
```

BPTT by Hand - Recurrent Layer Backward

We walk the steps _____: each c_t gets gradient from ___ and from ____, and the weight gradients are _____ over every copy.

```python
def backward(self, dC):  # (B, T, n_h): dℒ/dc_t from ŷ_t
    self.dWx = torch.zeros_like(self.Wx)
    self.dWc = torch.zeros_like(self.Wc)
    self.db = torch.zeros_like(self.b)
    dX = torch.zeros_like(self.X)
    dc_next = torch.zeros_like(self.states[0])

    for t in reversed(range(self.X.shape[1])):  # back
        dc = dC[:, t] + dc_next  # from ŷ_t and from c_t+1
        dz = dc * (1 - self.states[t + 1] ** 2)  # tanh'
        self.dWx += dz.T @ self.X[:, t]    # add up the
        self.dWc += dz.T @ self.states[t]  # gradient of
        self.db += dz.sum(dim=0)           # every copy
        dX[:, t] = dz @ self.Wx    # to the embedding
        dc_next = dz @ self.Wc     # to c_t-1
    return dX

RNNLayer.backward = backward  # add it to the class
```

BPTT by Hand - Updating the Weights

Every copy of the cell shares W_x, W_c and b_c

$$W_c \leftarrow W_c - \alpha \frac{\partial \mathcal{L}}{\partial W_c}$$

```python
def update(self, lr):  # one gradient descent step
    self.Wx -= lr * self.dWx
    self.Wc -= lr * self.dWc
    self.b -= lr * self.db

RNNLayer.update = update  # add it to the class
```

BPTT by Hand - Output Layer

```python
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
        return X @ self.W.T + self.b

    def backward(self, dZ):  # (B, T, n_out)
        self.dW = (dZ.transpose(1, 2) @ self.X).sum(dim=0)
        self.db = dZ.sum(dim=1).sum(dim=0)
        return dZ @ self.W  # dℒ/dc_t

    def update(self, lr):
        self.W -= lr * self.dW
        self.b -= lr * self.db
```

BPTT by Hand - Network

```python
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

    def backward(self, d):
        for layer in reversed(self.layers):
            d = layer.backward(d)

    def update(self, lr):
        for layer in self.layers:
            layer.update(lr)
```

`Network.backward` runs the layers in _____ order, each passing its gradient down, and `Network.update` steps _____ at once.

BPTT by Hand - Cross-Entropy

With **softmax** and **cross-entropy** together, the gradient of the logits is just _____ (the review's **d.)**), averaged over the `25` predictions.

```python
def cross_entropy(logits, y):  # (1, T, V), (1, T)
    p = torch.softmax(logits[0], dim=1)
    rows = torch.arange(y.shape[1])
    loss = -torch.log(p[rows, y[0]]).mean()
    d = p.clone()
    d[rows, y[0]] -= 1  # softmax + CE: ŷ − y
    return loss, (d / y.shape[1])[None]
```

Running BPTT by Hand

Last lecture's 2 hidden units are too few to remember the song, so we use 16. Each epoch: forward, **BPTT**, then _____.

```
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))
ids = torch.tensor([[vocab.index(t) for t in tokens]])
X, y = ids[:, :-1], ids[:, 1:]  # predict the next token
torch.manual_seed(0)
model = Network([Embedding(22, 8), RNNLayer(8, 16),
                 LinearLayer(16, 22)])
for epoch in range(200):
    loss, d = cross_entropy(model.forward(X), y)
    model.backward(d)  # BPTT, by hand
    model.update(0.5)  # one step of gradient descent
    if epoch % 50 == 0:
        print(f"epoch {epoch} loss: {loss.item():.4f}")
right = (model.forward(X)[0].argmax(dim=1) == y[0]).sum()
print(right.item(), "of", y.shape[1], "next words right")
```

OUTPUT

```
epoch 0 loss: 3.1958
epoch 50 loss: 0.2203
epoch 100 loss: 0.0775
epoch 150 loss: 0.0457
25 of 25 next words right
```

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 ]", " ", song.lower()).split() # split song into tokens
num_new_tokens = 4 # generate 10 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

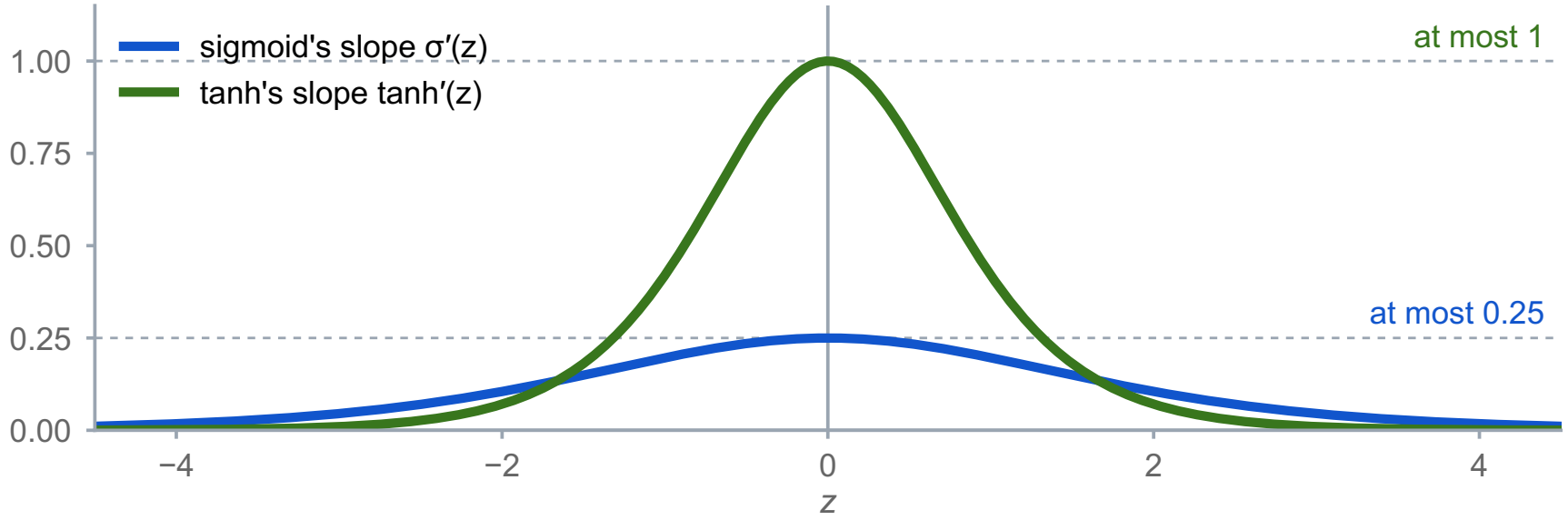
print(split_song)  # print original song with the 10 new predicted tokens on the end
```

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, old, notre, dame, wake, up,
the]
```

Vanishing/Exploding Gradients

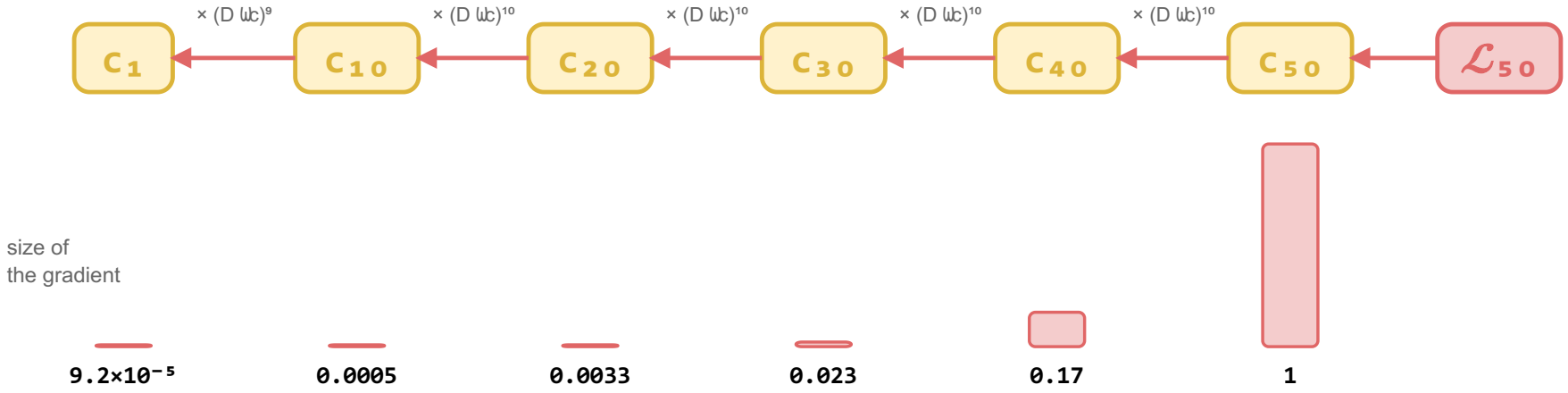
RNNs are particularly **vulnerable to the vanishing gradient problem** because we have to **compute the gradients of the loss** _____



This is why **RNNs** use **tanh**, not the **sigmoid**. The **sigmoid's** slope is at most `0.25`, so `50` steps back shrink the gradient at least $0.25^{50} \approx 8 \times 10^{-31}$ times. **Tanh's** slope reaches `1`, so it only shrinks the gradient where the cell is **saturated**.

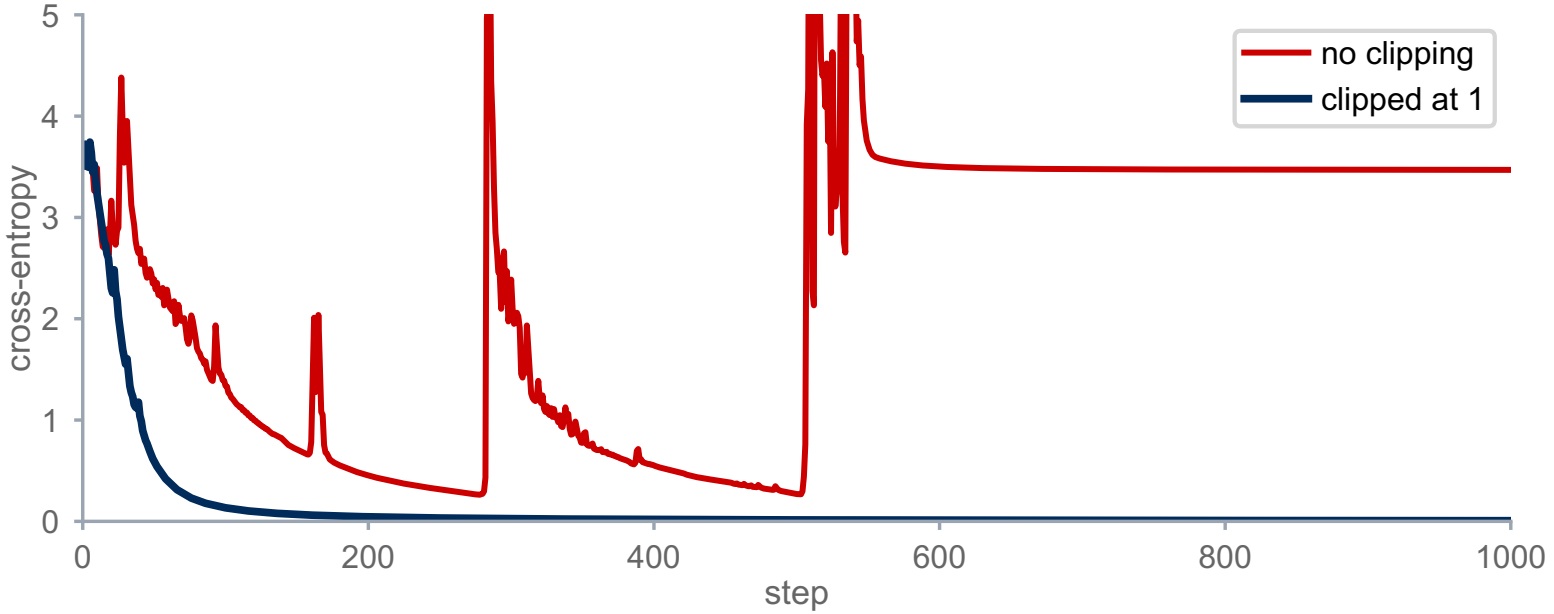
Vanishing Gradient

Since you are multiplying increasingly _____ values with each other during backprop, the gradient eventually trends to _, meaning **updates will stop happening**.



Exploding Gradient

Almost the opposite of the **vanishing gradient**, the **exploding gradient** is when the gradients become _____

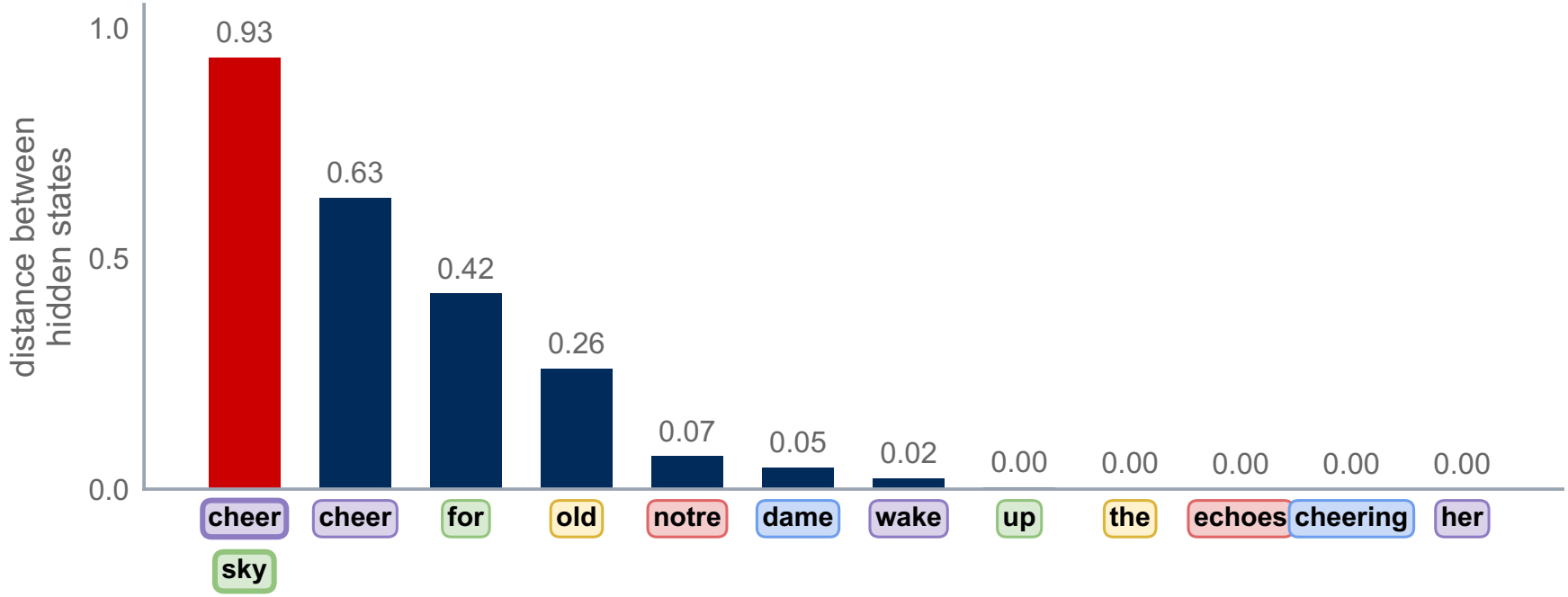


An **exploding gradient** has a simple fix: **clip, or put a ceiling on, the size of the gradient** we'll accept.

Vanishing gradients are harder: they need a new kind of cell.

Another Issue with RNNs: Long-Term Dependencies

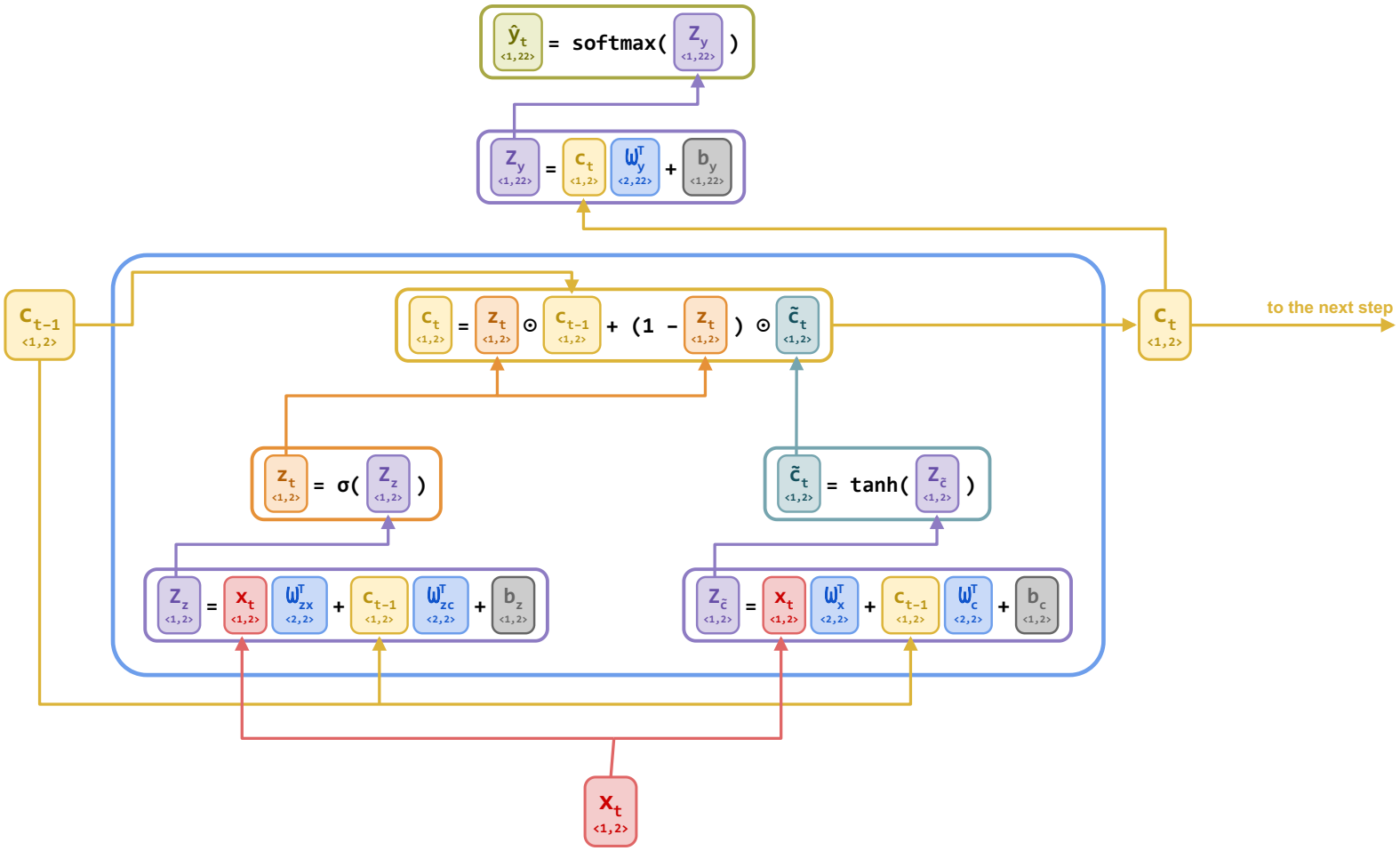
In last chapter's **RNN**, change the first word from **cheer** to **sky** and measure how far apart the hidden states are after each word. A word that depends on one far back is a **long-term dependency**.



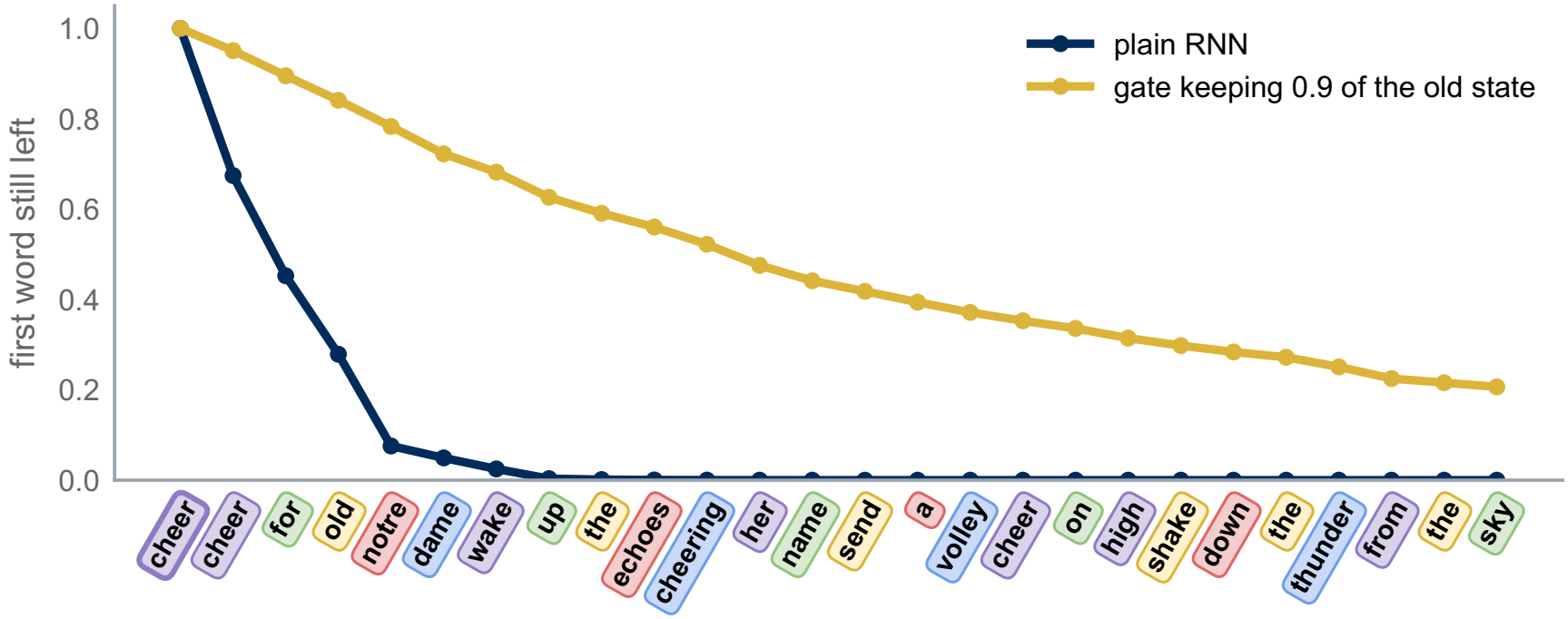
By the ninth word it's `0.001`: the **RNN** _____ how the song started. One fix: a _____ that lets each step choose what to keep

Gates and Computation Graphs

Add a new component called a **gate** that allows each step choose how much of the _____ to keep

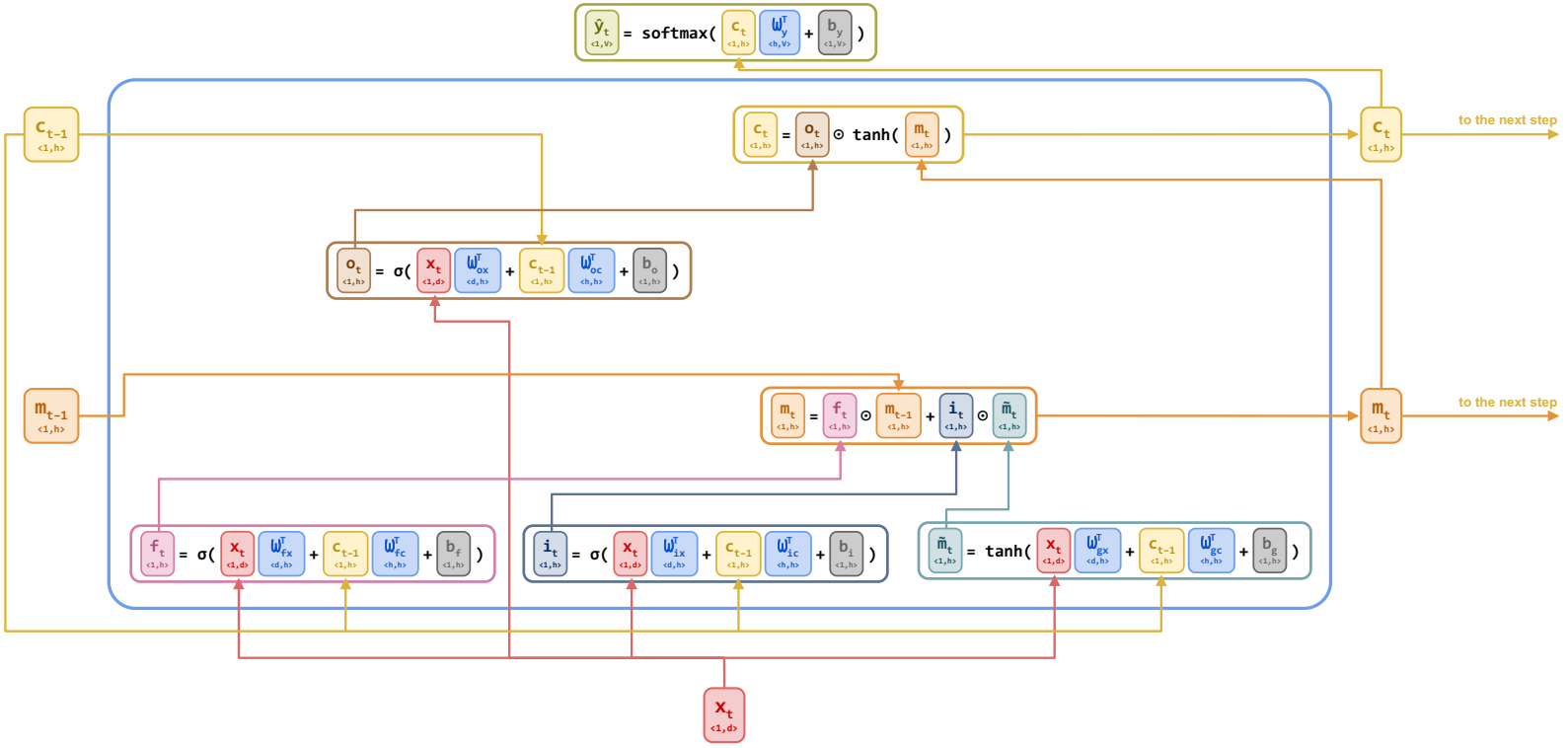


Effect of Gates



Long Short-Term Memory units - LSTMs (1997)

LSTMs introduce the idea of _____ to help solve the _____ gradient problem.



LSTMs and GRUs

A real gate is its own _____ layer, so the network **learns when to hold on and when to let go**

- **LSTM** (1997): _____ gates and a separate _____
- **GRU** (2014): an _____ and a _____

```python
from torch import nn
for cell in [nn.RNN(2, 2), nn.GRU(2, 2), nn.LSTM(2, 2)]:
    n = sum(p.numel() for p in cell.parameters())
    print(type(cell).__name__, n)
```

OUTPUT

```
RNN 12
GRU 36
LSTM 48
```

Breaking Down LSTMs

[4.1]

forget gate: how much of the old cell to keep

$$f_t = \sigma\left(x_t W_{fx}^\top + c_{t-1} W_{fc}^\top + b_f\right)$$

input gate: how much of the candidate to write

$$i_t = \sigma\left(x_t W_{ix}^\top + c_{t-1} W_{ic}^\top + b_i\right)$$

candidate: what to write

$$\tilde{m}_t = \tanh\left(x_t W_{gx}^\top + c_{t-1} W_{gc}^\top + b_g\right)$$

output gate: how much of the cell to show

$$o_t = \sigma\left(x_t W_{ox}^\top + c_{t-1} W_{oc}^\top + b_o\right)$$

the new cell state

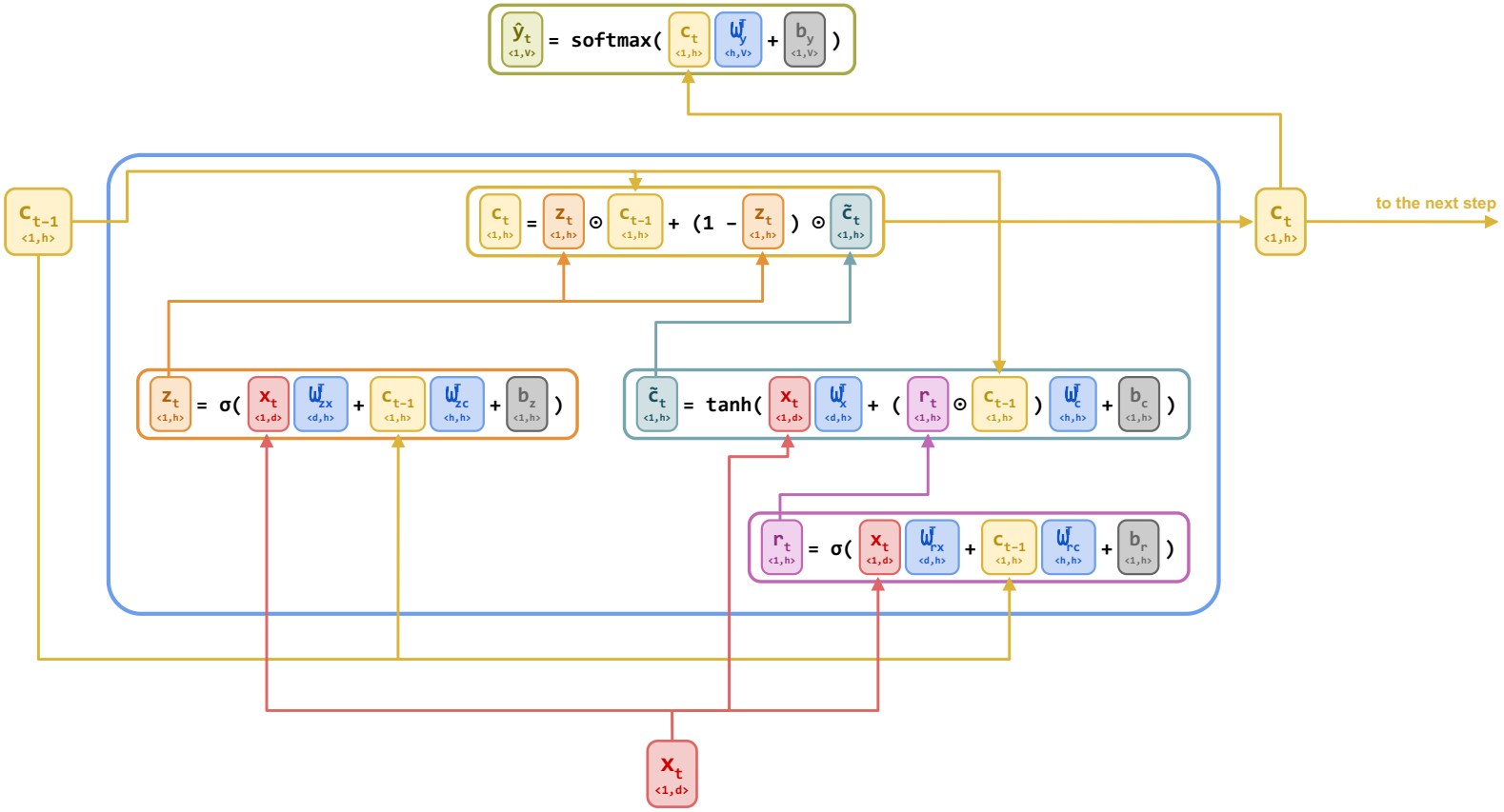
$$m_t = f_t \odot m_{t-1} + i_t \odot \tilde{m}_t$$

the new hidden state

$$c_t = o_t \odot \tanh(m_t)$$

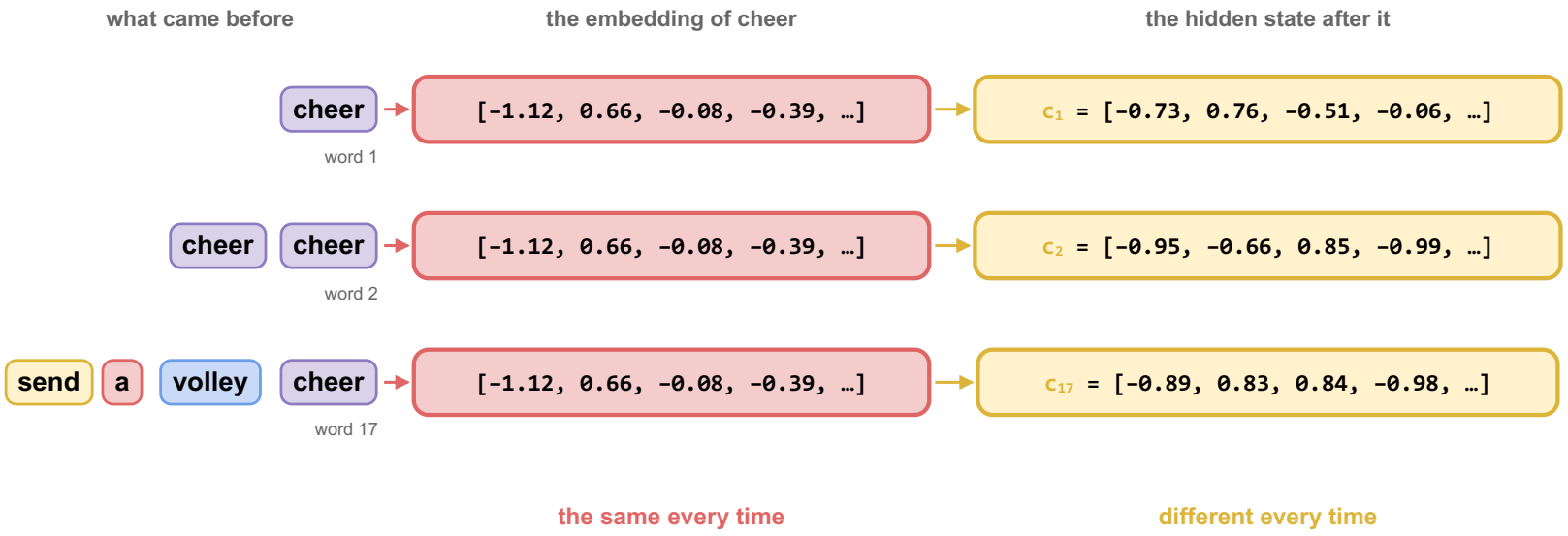
Gated Recurrent Units - GRUs (2014)

GRUs are a **simpler version** of **LSTMs** that simplify the gate structure



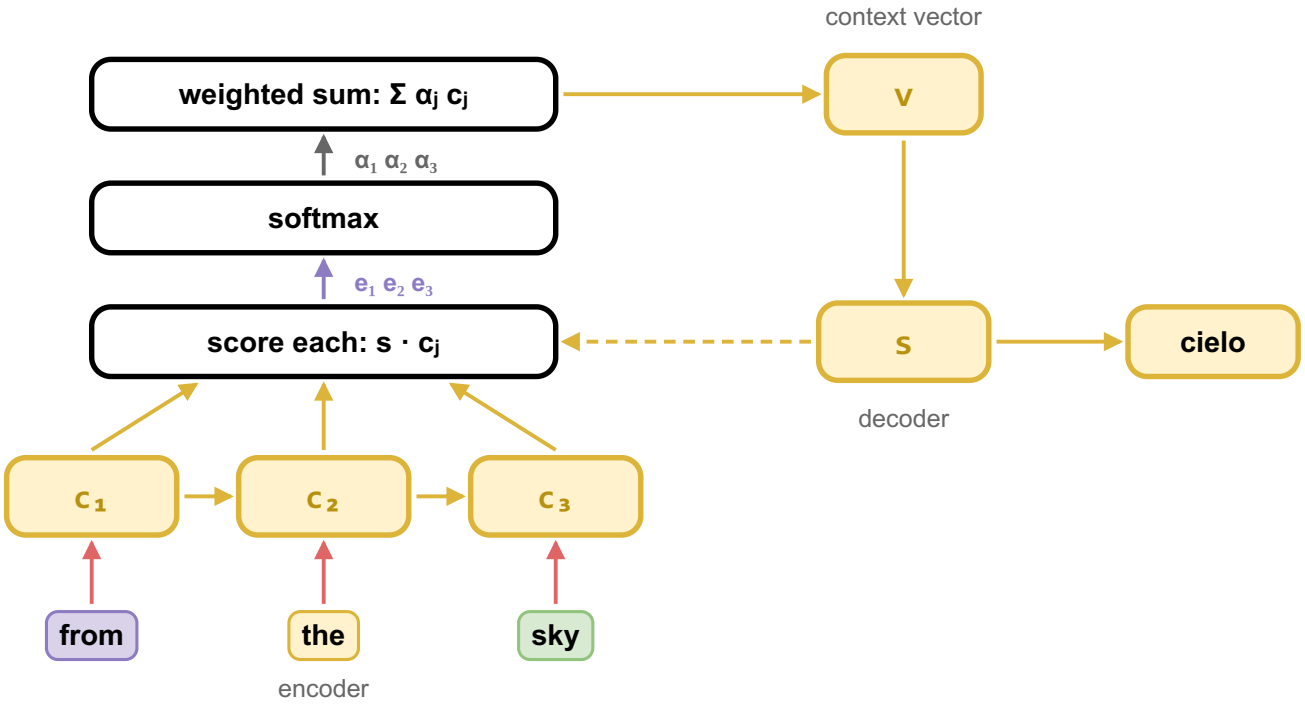
Static Embeddings

So far, the inputs to our model are what are called **static embeddings**, embeddings that are _____



Attention

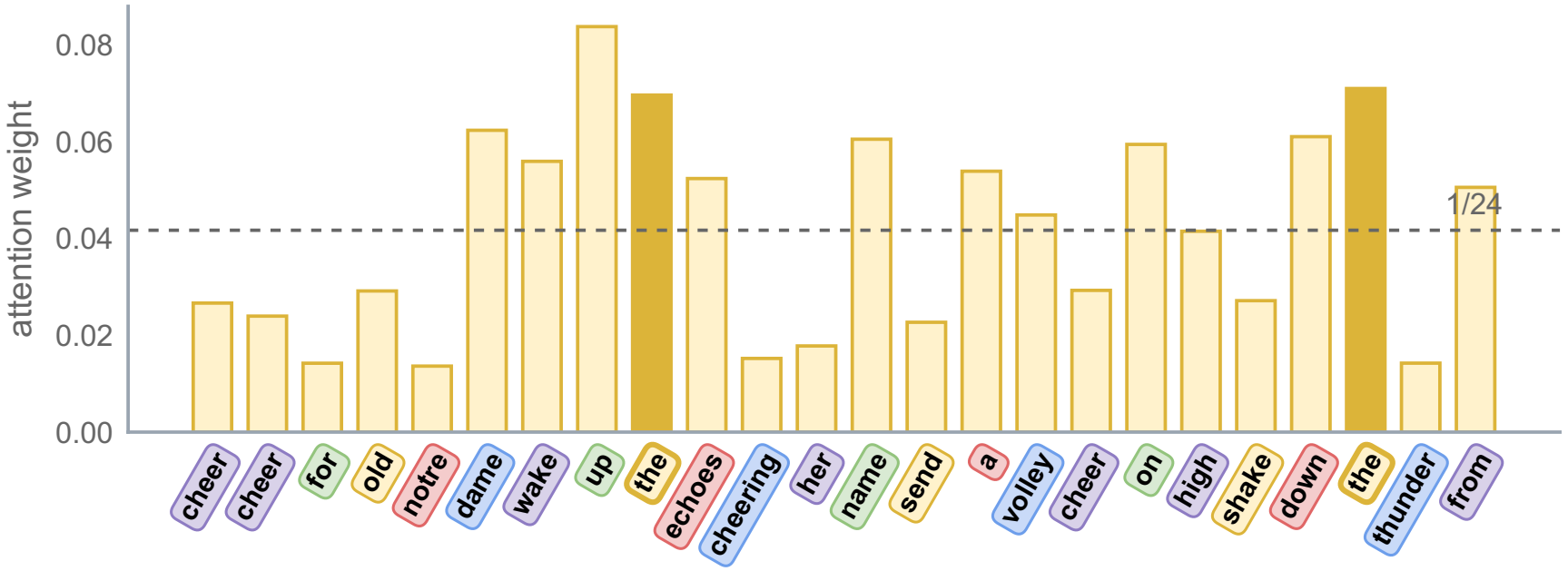
Bahdanau, Cho and Bengio (2014): before each token it writes, the decoder looks back at every encoder _____



Every word written gets _____

Attention Weights

Last chapter's network at the last **the**, looking back at the 24 words before it:



Most weight on **up** and the other **the**s: their hidden states have the biggest ____ _____ with this one. Next chapter: **attention** with **learned weights**

Attention

Some parts of our **context** are **more important than others**! **Attention** is a mechanism through which we can _____

_____

