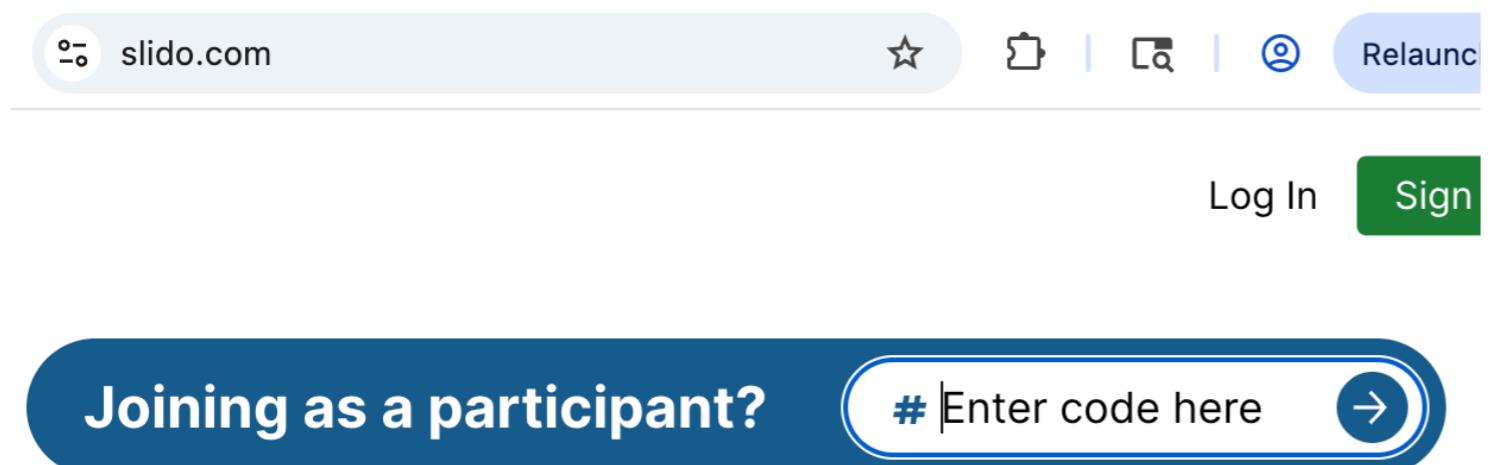


Slido

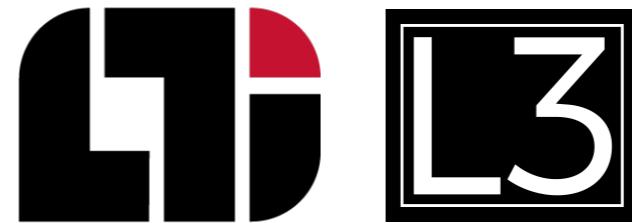
3542640



CS11-711 Advanced NLP Language Modeling

Sean Welleck

**Carnegie
Mellon
University**



<https://cmu-l3.github.io/anlp-spring2026/>

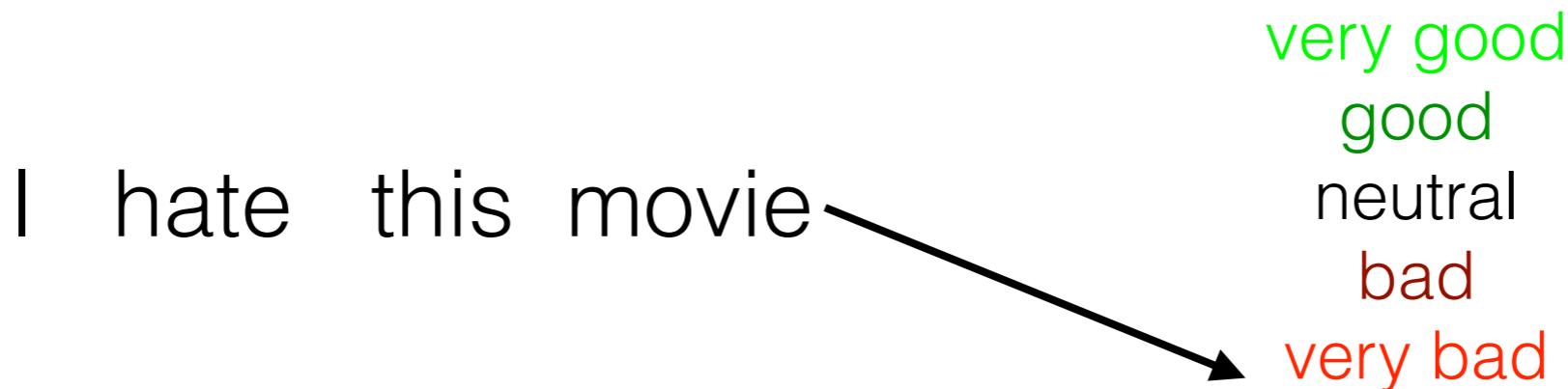
<https://github.com/cmu-l3/anlp-spring2026-code>

Types of Prediction: Binary, Multi-class, Structured

- Two classes (**binary classification**)



- Multiple classes (**multi-class classification**)



- Exponential/infinite labels (**structured prediction**)

I hate this movie → PRP VBP DT NN

I hate this movie → *kono eiga ga kirai*

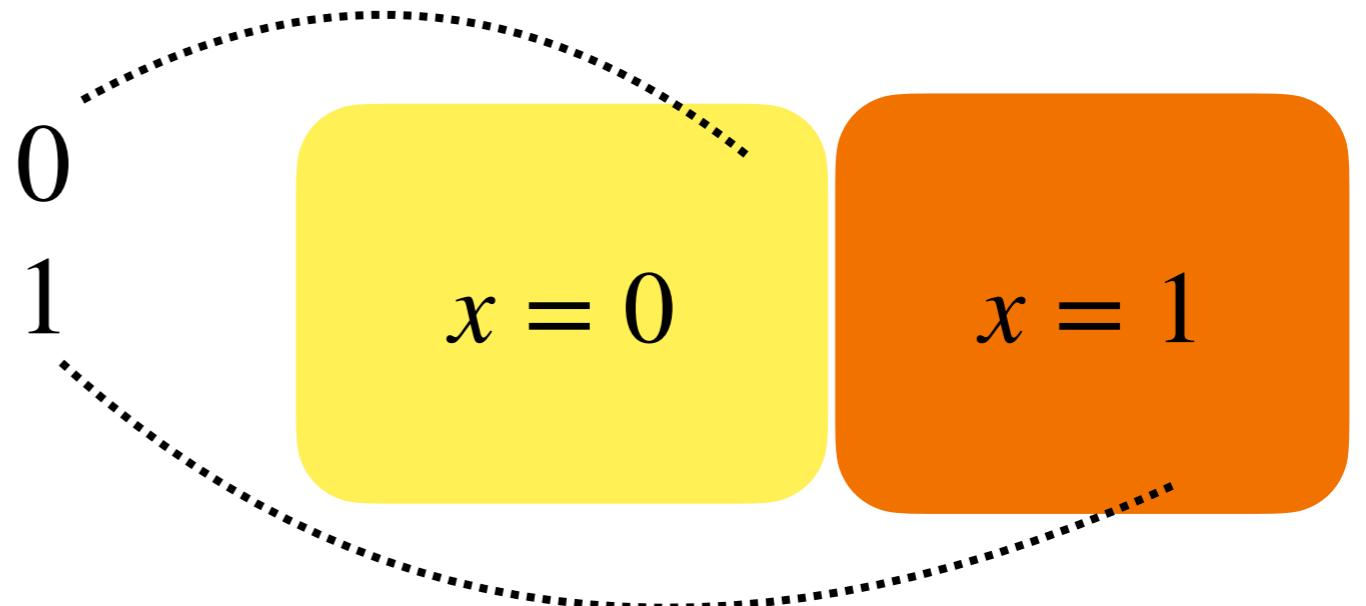
Language Modeling

S

What is a language model?

- A language model is a probability distribution over all sequences
 - $P(X)$
 - Example probability distribution: **biased coin**

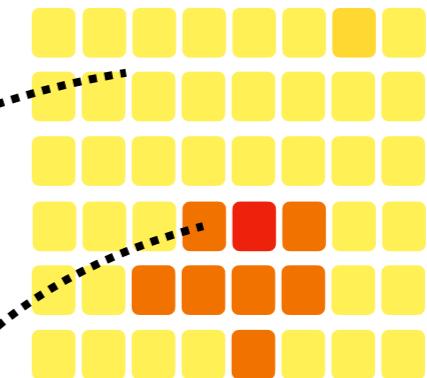
$$\bullet \quad P(X) = \begin{cases} 0.4 & x \text{ is } 0 \\ 0.6 & x \text{ is } 1 \end{cases}$$



What is a language model?

- A language model is a probability distribution over all sequences
 - $P(X)$
- Example language model:
 - $P(X) = 0.000013$ if x is a .
 - 0.000001 if x is aa .
 - ...
 - 0.019100 if x is a cat sat .
 - ...

One square = one sequence
All possible sequences — a lot!



What can we do with language models?

- **Score** sequences:

$P(\text{Jane went to the store .}) \rightarrow \text{high}$

$P(\text{store to Jane went the .}) \rightarrow \text{low}$

- **Generate** sequences:

$$\hat{x} \sim P(X)$$

What can we do with language models?

- **Conditional generation:** condition on an input context

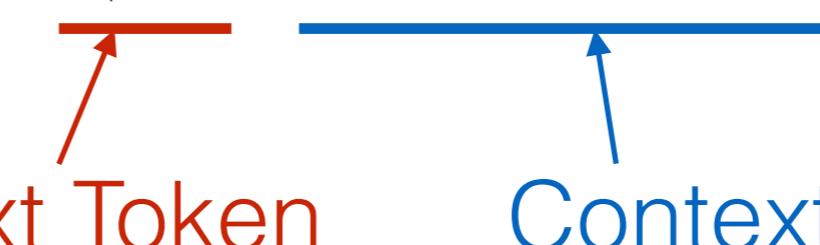
$$\hat{x}_{t+1:T} \sim P(X_{t+1:T} | x_{1:t})$$

- Machine translation:
 - Context: sentence in English
 - Continuation: sentence in Japanese
- General task:
 - Context: instructions, examples, start of output
 - Continuation: output

What can we do with language models?

- **Answer questions**
 - *Score* possible multiple choice answers
 - *Generate* a continuation of a question prompt
- **Classify text**
 - *Score* the text conditioned on a label
 - *Generate* a label given a classification prompt
- **Correct grammar**
 - *Score* each word and replace low-scoring ones
 - *Generate* a grammatical output
- ...

Auto-regressive Language Models

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


Next Token Context

Decomposes sequence modeling into next-token modeling

$P(X)$ is defined over \mathcal{X} (space of all sequences). **Very large**

$P(x_t \mid x_{<t})$ is defined over \mathcal{V} (token vocabulary). **Much smaller**

Auto-regressive Language Models

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

The diagram shows the formula for an auto-regressive language model. A red arrow points from the label 'Next Token' to the term x_t in the probability expression. A blue arrow points from the label 'Context' to the term x_1, \dots, x_{t-1} .

Key question: modeling

$$P(x_t \mid x_1, \dots, x_{t-1})$$

Roadmap

- Bigram models
- Ngram models
- Feedforward neural language model
- Practical deep learning considerations

Bigram models

$$P(X) \approx \prod_{t=1}^T p_{\theta} (x_t \mid x_{t-1})$$

Next Token 1-token context



Code: https://github.com/cmu-l3/anlp-fall2025-code/blob/main/03_Lm_fundamentals/Lm_basics_bigrams.ipynb

Training language models

Problem setup

- Goal: model a *data distribution*, i.e. $p_\theta \approx p_{data}$
- We only have a dataset of samples from p_{data} :
 - $D = \{x_n\}_{n=1}^N$
- Split the dataset into training, dev, and test sets

Training bigram models

- Set next-token probabilities based on how often each token x_t appears after x_{t-1} in the training dataset:

$$p(x_t \mid x_{t-1}) = \frac{\text{count}(x_{t-1}, x_t)}{\sum_{x'} \text{count}(x_{t-1}, x')}$$

- We can view this as training parameters $\theta_{i,j} = p(x_j \mid x_i)$

In Code

- Model a dataset of names. Character-level tokenization.

```
data = open('names.txt').read().splitlines()
data[:10]
✓ 0.0s
['emma',
 'olivia',
 'ava',
 'isabella',
 'sophia',
 'charlotte',
 'mia',
 'amelia',
 'harper',
 'evelyn']
```

```
bigram_counts = {}
for x in data:
    sequence = ['[S]'] + list(x) + ['[S]']
    for x1, x2 in zip(sequence, sequence[1:]):
        bigram = (x1, x2)
        bigram_counts[bigram] = bigram_counts.get(bigram, 0) + 1
```

```
[('n', '[S)'), 6763),
 ('a', '[S)'), 6640),
 ('a', 'n'), 5438),
 ('[S]', 'a'), 4410),
 ('e', '[S)'), 3983),
 ('a', 'r'), 3264),
 ('e', 'l'), 3248),
 ('r', 'i'), 3033),
 ('n', 'a'), 2977),
 ('[S]', 'k'), 2963)]
```

https://github.com/cmu-l3/anlp-spring2026-code/blob/main/03_lm_fundamentals/lm_basics_bigrams.ipynb

Based on Andrej Karpathy's lecture/video: <https://youtu.be/PaCmpygFfXo>

Model probabilities

ja

aa	1.64E-02	ab	1.60E-02	ac	1.39E-02	ad	3.08E-02	ae	2.04E-02	af	3.95E-03	ag	4.96E-03	ah	6.88E-02	ai	4.87E-02	aj	5.16E-03	ak	1.68E-02	al	7.46E-02	am	4.82E-02	an	1.60E-01	ao	1.86E-03	ap	2.42E-03	aq	1.77E-03	ar	9.63E-02	as	3.30E-02	at	2.03E-02	au	1.12E-02	av	2.46E-02	aw	4.75E-02	ax	5.1E-02	az	6.05E-02	br	1.28E-02	1.96E-01	[S]a	1.38E-01
ba	1.21E-01	bb	1.44E-02	bc	3.78E-04	bd	2.46E-02	be	2.48E-01	bf	0.00E+00	bg	0.00E+00	bh	1.55E-02	bi	8.20E-02	bj	3.78E-04	bk	0.00E+00	bl	3.89E-02	bm	0.00E+00	bn	1.51E-03	bo	3.97E-02	bp	0.00E+00	0.00E+00	br	3.18E-01	bt	3.02E-03	7.56E-04	bu	1.70E-02	bv	0.00E+00	0.00E+00	bx	3.14E-02	by	0.00E+00	bz	4.31E-02	[S]b	1.40E-01				
ca	2.31E-01	cb	0.00E+00	cc	1.19E-02	cd	2.83E-04	ce	1.56E-01	cf	0.00E+00	cg	5.66E-04	ch	1.88E-01	ci	7.67E-02	cj	8.49E-04	ck	8.95E-02	cl	3.28E-02	cm	0.00E+00	0.00E+00	co	1.08E-01	cp	2.83E-04	3.11E-02	cr	2.15E-02	cs	1.42E-03	ct	9.91E-03	cu	9.91E-03	cv	0.00E+00	0.00E+00	cx	8.49E-04	cy	2.94E-02	cz	1.13E-03	2.75E-02	[S]c	2.32E-01			
da	2.37E-01	db	1.82E-04	dc	5.46E-04	dd	2.71E-02	de	2.33E-01	df	9.10E-04	dg	4.55E-03	dh	2.15E-02	di	1.23E-01	dj	1.64E-03	dk	5.46E-04	dl	1.09E-02	dm	5.46E-03	dn	6.88E-02	do	0.00E+00	1.82E-04	dr	7.71E-02	ds	5.28E-03	dt	7.28E-04	du	1.67E-02	dv	3.09E-03	dw	4.18E-03	dx	0.00E+00	5.77E-02	dy	1.82E-04	dz	9.39E-02	[S]d	2.38E-01			
ea	3.32E-02	eb	5.92E-03	ec	7.49E-03	ed	1.88E-02	ee	6.22E-02	ef	4.02E-03	eg	6.12E-03	eh	7.44E-03	ei	4.01E-02	ej	2.69E-03	ek	8.72E-03	el	1.59E-01	em	3.77E-02	en	1.31E-01	eo	1.32E-02	ep	4.06E-03	eg	6.86E-04	er	9.59E-02	es	4.22E-02	et	2.84E-02	eu	3.38E-03	ev	2.27E-02	ew	2.45E-03	ex	6.46E-03	ey	5.24E-02	ez	8.86E-03	1.95E-01	[S]e	3.33E-02
fa	2.67E-01	fb	0.00E+00	fc	0.00E+00	fd	0.00E+00	fe	1.36E-01	ff	4.86E-02	fg	1.10E-02	fh	1.10E-03	fi	1.77E-01	fg	0.00E+00	fk	2.21E-03	fl	2.21E-02	fm	0.00E+00	4.42E-03	fo	6.63E-02	fp	0.00E+00	0.00E+00	fr	1.26E-01	fs	6.63E-03	ft	1.99E-02	fu	1.10E-02	fv	0.00E+00	4.42E-03	fx	0.00E+00	1.55E-02	fy	2.21E-03	fz	8.84E-02	[S]f	2.68E-01			
ga	1.71E-01	gb	1.56E-03	gc	0.00E+00	gd	9.86E-03	ge	1.73E-01	gf	5.19E-04	gg	1.30E-02	gh	1.87E-01	gi	9.86E-02	gj	1.56E-03	gk	0.00E+00	gl	1.66E-02	gm	3.11E-03	gn	1.40E-02	go	4.31E-02	gp	0.00E+00	0.00E+00	gr	1.04E-01	gs	1.56E-02	gt	1.61E-02	gu	4.41E-02	gv	5.19E-04	gw	1.35E-02	gx	0.00E+00	1.61E-02	gy	5.19E-04	gz	5.60E-02	[S]g	1.72E-01	
ha	2.95E-01	hb	1.05E-03	hc	2.63E-04	hd	3.15E-03	he	8.85E-02	hf	2.63E-04	hg	2.63E-04	hh	1.31E-04	hi	9.57E-02	hj	1.18E-03	hk	3.81E-03	hl	2.43E-02	hm	1.54E-02	hn	1.81E-02	ho	3.77E-02	hp	1.31E-04	hg	2.68E-02	hr	4.07E-03	hs	9.32E-03	ht	2.18E-02	hu	5.12E-03	hw	1.31E-03	hx	0.00E+00	2.80E-02	hy	2.63E-03	hz	3.16E-01	[S]h	2.96E-01		
ia	1.38E-01	ib	6.21E-03	ic	2.88E-02	id	2.49E-02	ie	9.34E-02	if	5.71E-03	ig	2.42E-02	ih	5.37E-03	ii	4.63E-03	ij	4.29E-03	ik	2.51E-02	il	7.60E-02	im	2.41E-02	in	1.20E-01	io	3.32E-02	ip	2.99E-03	iq	2.94E-03	ir	4.80E-02	is	7.43E-02	it	3.06E-02	iu	6.16E-03	iv	1.52E-02	iw	4.52E-04	ix	5.03E-03	iy	4.40E-02	iz	1.41E-01	[S]i	1.39E-01	
ja	5.08E-01	jb	3.45E-04	jc	1.36E-03	jd	1.38E-03	je	1.52E-01	jf	0.00E+00	0.00E+00	ji	1.55E-02	jk	4.10E-02	jl	6.90E-04	jk	6.90E-04	km	3.10E-03	kn	6.90E-04	ko	1.65E-01	kp	3.45E-04	0.00E+00	jr	3.79E-03	kt	2.41E-03	jt	6.90E-04	ju	6.97E-02	kv	1.72E-03	kw	2.07E-03	0.00E+00	3.45E-03	0.00E+00	2.45E-02	[S]j	5.09E-01							
ka	3.43E-01	kb	3.97E-04	kc	3.97E-04	kd	3.97E-04	ke	1.78E-01	kf	1.98E-04	kg	0.00E+00	kh	6.09E-02	ki	1.01E-01	kj	3.97E-04	kk	3.97E-03	kl	2.76E-02	km	1.79E-03	kn	5.16E-03	ko	6.83E-02	kp	0.00E+00	0.00E+00	kr	2.16E-02	ks	1.88E-02	kt	3.37E-03	ku	9.92E-03	kv	3.97E-04	kw	6.75E-03	0.00E+00	ky	7.52E-02	kz	3.97E-04	qz	7.20E-02	[S]k	3.44E-01	
la	1.88E-01	lb	3.73E-03	lc	1.79E-03	ld	9.89E-03	le	2.09E-01	lf	1.58E-03	lg	4.30E-04	lh	1.36E-03	li	1.78E-01	lj	4.30E-04	lk	1.72E-03	lm	9.64E-02	ln	1.00E-03	lo	4.96E-02	lp	1.07E-03	2.15E-04	lr	1.29E-03	ls	6.73E-03	lt	5.52E-03	lu	2.32E-02	lv	5.16E-03	lw	1.15E-03	lx	0.00E+00	1.14E-01	ly	7.16E-04	lz	9.41E-02	[S]l	1.87E-01			
ma	3.90E-01	mb	1.69E-02	mc	7.68E-03	md	3.61E-03	me	1.23E-01	mf	1.51E-04	mg	0.00E+00	mh	7.53E-04	mi	1.89E-01	mj	1.05E-03	mk	1.51E-04	ml	7.53E-04	mm	2.53E-02	mn	3.01E-03	mo	6.81E-02	mp	5.72E-03	mq	0.00E+00	1.46E-02	mr	5.27E-03	ms	6.02E-04	mt	2.09E-02	mu	4.52E-04	mv	3.01E-04	mw	0.00E+00	4.32E-02	my	1.66E-03	mz	7.77E-02	[S]m	3.91E-01	
na	1.62E-01	nb	4.37E-04	nc	1.16E-02	nd	3.84E-02	ne	7.42E-02	nf	6.00E-04	ng	1.49E-02	nh	9.41E-02	ni	2.40E-03	nj	3.16E-03	nk	1.06E-02	nn	1.04E-01	no	2.71E-02	np	2.73E-04	0.00E+00	nr	2.40E-03	ns	1.52E-02	nt	2.42E-02	nu	3.00E-03	nv	6.00E-04	nx	3.27E-04	ny	2.54E-02	nz	7.91E-03	o[S]	1.63E-01								
oa	1.88E-02	ob	1.76E-02	oc	1.44E-02	od	2.39E-02	oe	1.66E-02	of	4.29E-03	og	5.55E-03	oh	2.16E-02	oi	8.70E-03	oj	2.02E-03	ok	8.57E-03	ol	7.80E-02	om	3.29E-02	on	3.04E-01	oo	1.45E-02	op	1.20E-02	og	3.78E-04	or	1.33E-01	os	6.35E-02	ot	1.49E-02	ou	3.47E-02	ov	2.22E-02	ow	1.44E-02	ox	5.67E-04	oy	1.30E-02	oz	6.81E-03	o[S]	1.08E-02	
pa	2.04E-01	pb	1.95E-03	pc	9.75E-04	pd	0.00E+00	pe</																																														

Training : why counting?

- The counting procedure corresponds to **maximum likelihood** estimation for this model:

$$\max_{\theta} \sum_{x \in D_{train}} \log p_{\theta}(x)$$

- Idea: set the parameters so that the model assigns high probability to the training data D_{train}

Exercise: derive the update on the previous slide

Training: Why maximum likelihood?

- Makes p_θ match the data distribution p_{data} (p_* for brevity)

$$\min_{\theta} D_{KL}(p_* \parallel p_\theta) =$$

Dataset:
samples from p_*



S

Note: using log space

- Multiplication of probabilities can be re-expressed as addition of log probabilities

$$P(X) = \prod_{i=1}^{|X|} P(x_i) \longrightarrow \log P(X) = \sum_{i=1}^{|X|} \log P(x_i)$$

- **Why?:** numerical stability, other conveniences

Generation

- Generate from an autoregressive model by iteratively sampling a next token, then appending it to the context
 - Until [S] is generated:
$$\hat{x}_t \sim p_{\theta}(x_t | \hat{x}_{t-1})$$
- Equivalent to sampling from the model's joint distribution over full sequences! (*More in lecture 7*)

In Code

```
def generate_sequence():
    sequence = ['[S]']
    while True:
        current_char = sequence[-1]
        current_index = char_to_index[current_char]
        next_index = torch.multinomial(P[current_index], num_samples=1).item()
        next_char = index_to_char[next_index]
        if next_char == '[S]':
            break
        sequence.append(next_char)
    return ''.join(sequence[1:])

# Generate 10 sequences
generated_sequences = [generate_sequence() for _ in range(10)]
generated_sequences
✓ 0.0s
['iciara', 'm', 'gevere', 'nri', 'ch', 'anan', 'de', 'k', 'al', 'nnn']
```

Evaluation

- We can evaluate a model based on the probabilities it assigns to a dataset
 - E.g., the training set or a held-out test set
- Two widely used metrics in language modeling:
 - Log-likelihood
 - Perplexity

Log-likelihood and perplexity

- (Negative) **Log-likelihood**:

$$\text{NLL} = - \sum_{i=1}^N \sum_{t=1}^{T_i} \log p(x_t^{(i)} | x_{<t}^{(i)}) \quad [0, \infty)$$

- **Per-token average NLL**:

$$\text{NLL}_{avg} = - \frac{1}{T_{\text{total}}} \sum_{i=1}^N \sum_{t=1}^{T_i} \log p(x_t^{(i)} | x_{<t}^{(i)}) \quad [0, \infty)$$

- **Perplexity**

$$\text{PPL}_e = \exp \left(- \frac{1}{T_{\text{total}}} \sum_{i=1}^N \sum_{t=1}^{T_i} \log p(x_t^{(i)} | x_{<t}^{(i)}) \right) \quad [1, \infty)$$

Perplexity

- **Perplexity:**

When a dog sees a squirrel it will usually __

Token: ' bark' - Probability: 0.0352 $\rightarrow e^{-\log p} = 28.4$

Token: ' jump' - Probability: 0.0338 $\rightarrow e^{-\log p} = 29.6$

Token: ' start' - Probability: 0.0289 $\rightarrow e^{-\log p} = 34.6$

Token: ' run' - Probability: 0.0277 $\rightarrow e^{-\log p} = 36.1$

Token: ' try' - Probability: 0.0219 $\rightarrow e^{-\log p} = 45.7$

In Code

```
def log_likelihood(P, dataset):
    n = 0
    ll = 0
    for x in dataset:
        sequence = ['[S]'] + list(x) + ['[S]']
        for x1, x2 in zip(sequence, sequence[1:]):
            i = char_to_index[x1]
            j = char_to_index[x2]
            ll += torch.log(P[i, j])
            n += 1
    return ll, n

ll, n = log_likelihood(P, data)
print(f'Log likelihood: {ll.item():.4f}')
print(f'Average next-token log likelihood {ll.item() / n:.4f}')
✓ 0.5s
```

```
Log likelihood: -559891.7500
Average next-token log likelihood -2.4541
```

In Code

```
def perplexity(model, dataset):  
    ll, n = log_likelihood(model, dataset)  
    return torch.exp(-ll / n).item()  
  
perplexity(P, data)
```

✓ 0.5s

```
11.635889053344727
```

Recap: Bigram models

- A simple language model, but we saw several key concepts:
 - Maximum likelihood estimation
 - Log space
 - Autoregressive generation
 - Evaluating log-likelihood and perplexity
- **Next:** Ngram models

Ngram models

$$P(X) \approx \prod_{t=1}^T p_{\theta} \left(\underbrace{x_t}_{\text{Next Token}} \mid \underbrace{x_{t-1}, x_{t-2}, \dots, x_{t-n+1}}_{\text{n-token context}} \right)$$

- Use an analogous counting procedure to train

Training Ngram Models

- Use an analogous counting procedure to train

$$p(x_t \mid x_{t-n+1:t-1}) = \frac{\text{count}(x_{t-n+1:t-1}, x_t)}{\sum_{x'} \text{count}(x_{t-n+1:t-1}, x')}$$

Training Ngram Models

- Add a ‘fake count’ to each possible ngram to avoid zero probability ngrams

$$p(x_t \mid x_{t-n+1:t-1}) = \frac{1 + \text{count}(x_{t-n+1:t-1}, x_t)}{|V| \sum_{x'} \text{count}(x_{t-n+1:t-1}, x')}$$

- An example of *smoothing*

Problems

- Cannot share strength among **similar words**

she bought a car	she bought a bicycle
she purchased a car	she purchased a bicycle

→ solution: neural networks

- Cannot condition on context with **intervening words**

Dr. Jane Smith	Dr. Gertrude Smith
-----------------------	---------------------------

→ solution: neural networks

- Cannot handle **long-distance dependencies**

for tennis class he wanted to buy his own racquet
for programming class he wanted to buy his own computer

→ solution: neural networks in future lectures

When to use n-gram models?

- Neural language models achieve better performance, but
 - n-gram models are extremely fast to estimate/apply
 - Perfect memorization can be useful
- **Toolkit:** kenlm

<https://github.com/kpu/kenlm>

Feedforward neural language model

$$P(X) \approx \prod_{t=1}^T p_{\theta} \left(\underline{x_t} \mid \underline{x_{t-1}, x_{t-2}, \dots, x_{t-n+1}} \right)$$

Diagram illustrating the components of the feedforward neural language model:

- Neural network parameters :** (green arrow pointing to p_{θ})
- Next Token** (red arrow pointing to x_t)
- n-token context** (blue arrow pointing to $x_{t-1}, x_{t-2}, \dots, x_{t-n+1}$)

[https://github.com/cmu-l3/anlp-spring2026-code/blob/main/
03_lm_fundamentals/lm_basics_neural.ipynb](https://github.com/cmu-l3/anlp-spring2026-code/blob/main/03_lm_fundamentals/lm_basics_neural.ipynb)

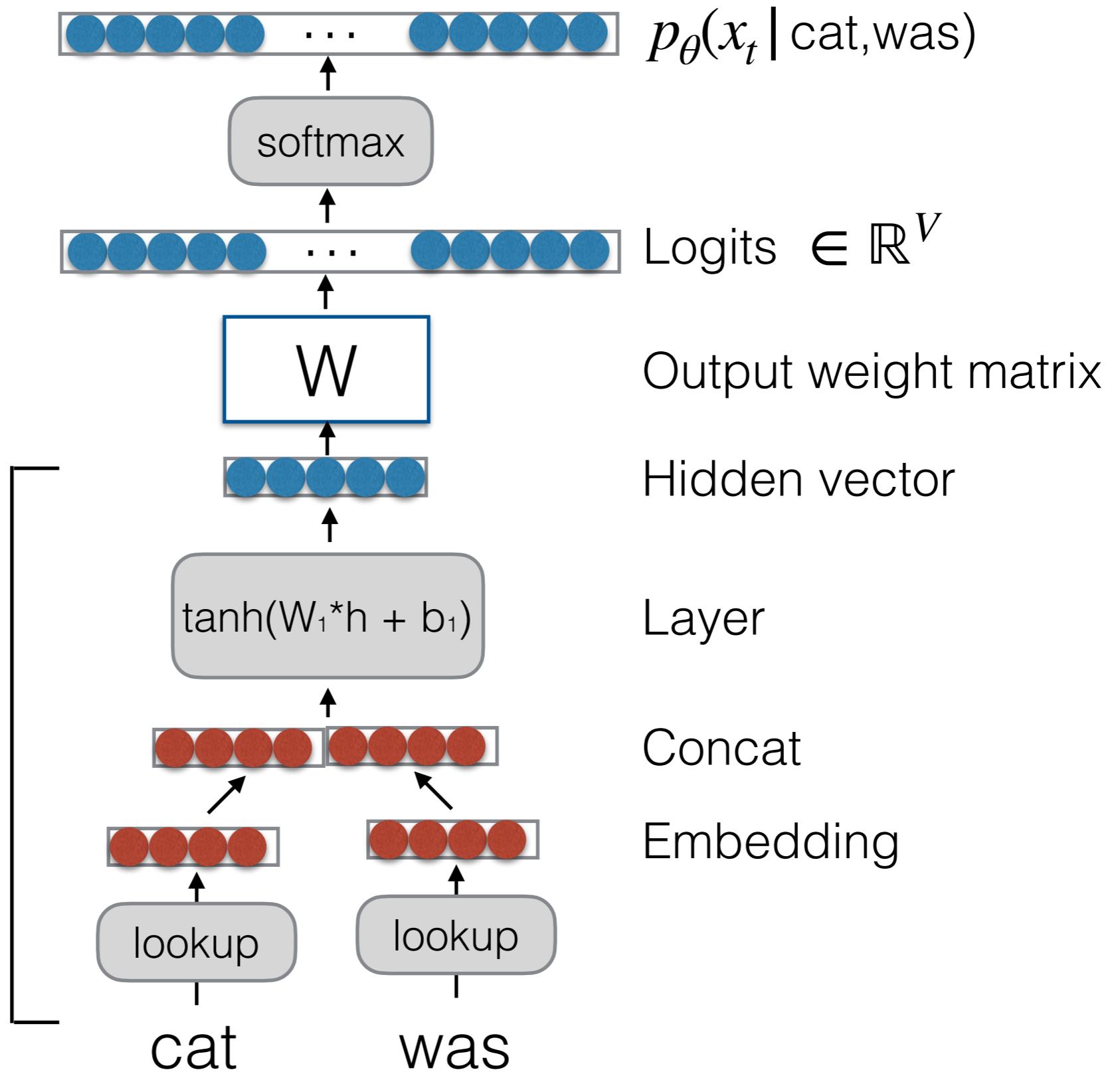
Neural language model

- Ngram language models do not take into account the similarity of words or contexts
 - *The cat was walking in the bedroom*
 - *The dog was running in a room*
- *Solution:* use learned, distributed representations

Feedforward neural language model

$$s = Wh$$

$$h = f_{\theta}(x_{t-2}, x_{t-1})$$



Feedforward neural language model

- Training: maximum likelihood estimation

$$\begin{aligned} \arg \max_{\theta} & \sum_{x \in D_{train}} \log p_{\theta}(x) \\ &= \sum_{x \in D_{train}} \sum_{t=1}^T \log p_{\theta}(x_t \mid x_{1:t-1}) \end{aligned}$$

- Loss: increase probability of target next-token

Loss: $L_t = -\log p_{\theta}(x_t \mid x_{1:t-1})$



Feedforward neural language model

- Cross-entropy loss!
 - Recall from lecture 2:
 - y_i : one-hot next-token
 - p_i : LM probability on that token
 - Classes: possible next-tokens (vocabulary)

$$L = -\log p_{\theta}(x_t | x_{1:t-1})$$

$$L_{CE} = - \sum_{i=1}^{\text{num classes}} y_i \log(p_i)$$

In code

```
class MLPLM(nn.Module):
    def __init__(self, vocab_size, context_size, embedding_size, hidden_size):
        super(MLPLM, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_size)
        self.fc1 = nn.Linear(context_size * embedding_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, vocab_size)

    def forward(self, x):
        x = self.embedding(x)          # (batch_size, context_size, hidden_size)
        x = x.view(x.shape[0], -1)    # (batch_size, context_size * hidden_size)
        x = torch.relu(self.fc1(x))  # (batch_size, hidden_size)
        x = self.fc2(x)              # (batch_size, vocab_size)
        return x
```

In code

```
criterion = nn.CrossEntropyLoss()

# Training loop
for epoch in range(num_epochs):
    # Reshuffle the data
    perm = torch.randperm(len(X_train))
    X_train = X_train[perm]
    Y_train = Y_train[perm]

    model.train()
    total_loss = 0
    for i in range(0, len(X_train), batch_size):
        X_batch = X_train[i:i+batch_size]
        Y_batch = Y_train[i:i+batch_size]

        # Forward pass
        outputs = model(X_batch)
        loss = criterion(outputs, Y_batch)

        # Backward pass and optimization
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
```

Example of Combination Features

- A row in the weight matrix can capture particular *combinations* of token embedding features
 - E.g. the 34th row in the weight matrix:

giving

$$\begin{matrix} \mathbf{a} \\ \hline -0.3 & 2.0 & 0.6 & -0.8 & -0.4 \end{matrix} \quad * \quad \begin{matrix} \mathbf{w}_{34} \\ \hline 1.2 & -0.1 & 0.7 & -2.1 & 0.5 \end{matrix} \quad + \quad b_{34} \quad = \quad \begin{matrix} \mathbf{b} \\ \hline 1.5 & 0 & 0 & 0 & 0 \end{matrix}$$

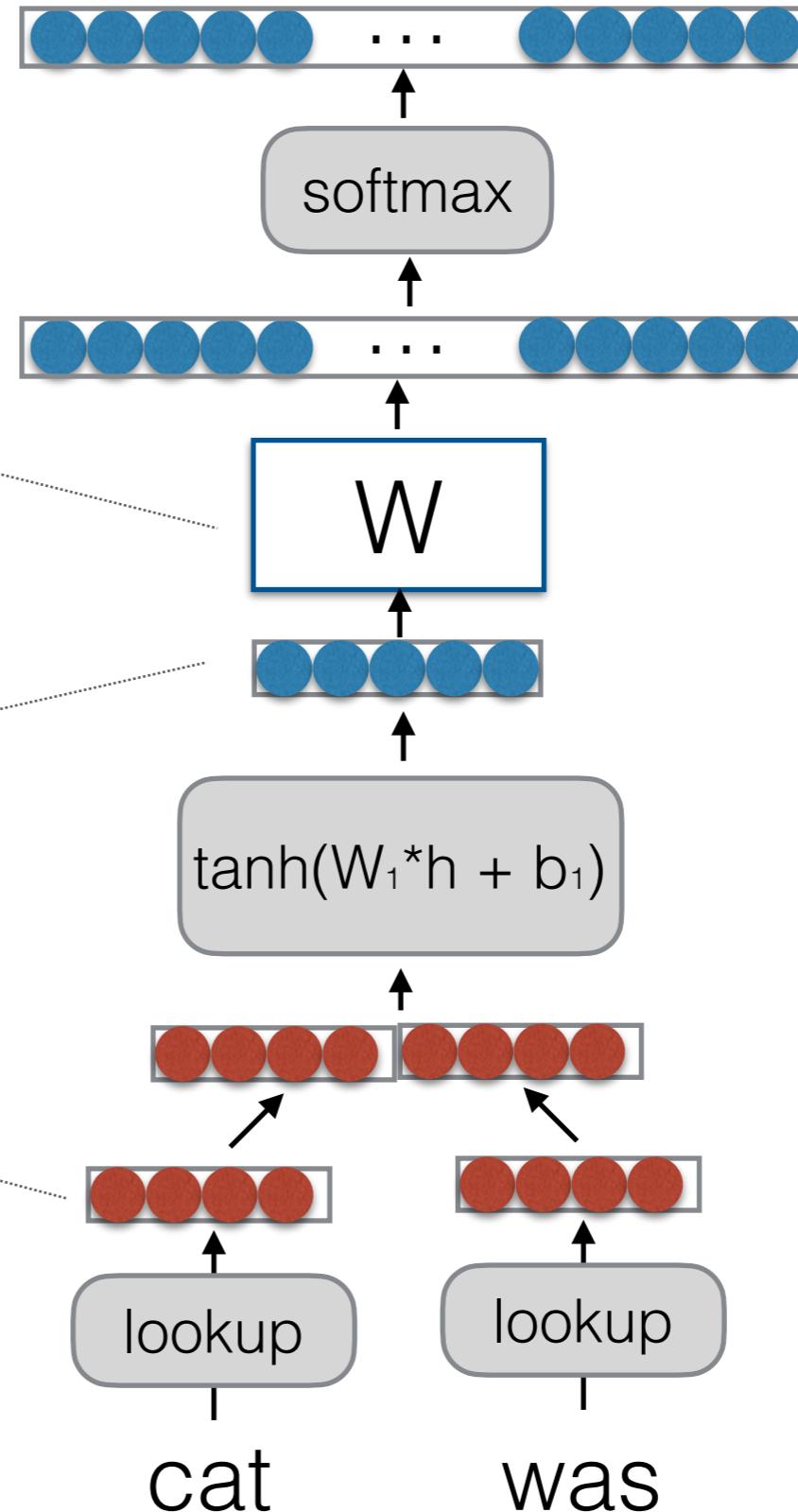
Example possibility:
positive number if
the previous word is a
determiner and
second-to-previous
word is a verb

Where is strength shared?

Similar output words get similar output weights

Similar contexts get similar hidden states

Similar words get similar embeddings



Where is strength shared?

- Consider predicting word w with two similar contexts h_j and h_k

- $$p_j^w = p(w | h_j) = \frac{1}{Z_j} \exp(w^\top h_j)$$

It's a great  movie

- $$p_k^w = p(w | h_k) = \frac{1}{Z_k} \exp(w^\top h_k)$$

It is a wonderful 

- $$\frac{p_j^w}{p_k^w} = \frac{Z_k}{Z_j} \exp(w^\top (h_j - h_k))$$

- The ratio is 1 when $w^\top (h_j - h_k) = 0$

“make hidden vectors h_j and h_k close to each other”

What Problems are Handled?

- Cannot share strength among **similar words**

she bought a car	she bought a bicycle
she purchased a car	she purchased a bicycle

→ solved, and similar contexts as well! 😊

- Cannot condition on context with **intervening words**

Dr. Jane Smith	Dr. Gertrude Smith
-----------------------	---------------------------

→ solved! 😊

- Cannot handle **long-distance dependencies**

for **tennis** class he wanted to buy his own **racquet**

for **programming** class he wanted to buy his own **computer**

→ not solved yet 😞

Recap

- Bigram language models and fundamental concepts
- Ngram language models: count-based
- Neural network language model
- Next: some important practical concepts

Important practical concepts

- A deep learning system has multiple moving parts:
 - The model architecture, the optimizer, the weights, the hyperparameters, ...
 - We want our experiments to give us data that leads to reliable conclusions
 - Here are a few helpful ideas that are often implicit in most deep learning experiments

Splitting into train, valid, and test

- Goal: fit a target distribution p_*
 - **Training data**: samples from p_* , used to fit the model p_θ
 - **Validation data**: hold out samples from p_* to check generalization. We try different configurations and choose one with good generalization.
 - **Test data**: hold out samples from p_* as an unbiased check of the final configuration's generalization

Splitting into train, valid, and test

- In other words:
 - **Training data**: use it to train the model
 - **Validation data**: use it to tune hyperparameters, perform ablations, select a model
 - **Test data**: use it *once at the end* and don't look at it during development

Splitting into train, valid, and test

Model 1

```
iter 0: train loss/sent=0.9047, time=5.91s
iter 0: valid acc=0.6857
iter 1: train loss/sent=0.7726, time=5.78s
iter 1: valid acc=0.7045
iter 2: train loss/sent=0.7378, time=5.77s
iter 2: valid acc=0.7110
iter 3: train loss/sent=0.7223, time=5.78s
iter 3: valid acc=0.7142
iter 4: train loss/sent=0.7142, time=5.83s
iter 4: valid acc=0.7150
```

Model 2

```
iter 0: train loss/sent=0.8373, time=9.63s
iter 0: dev acc=0.7094
iter 1: train loss/sent=0.7401, time=11.23s
iter 1: dev acc=0.7198
iter 2: train loss/sent=0.7160, time=11.52s
iter 2: dev acc=0.7286
iter 3: train loss/sent=0.7048, time=9.75s
iter 3: dev acc=0.7349
iter 4: train loss/sent=0.6967, time=10.02s
iter 4: dev acc=0.7227
```

Model 3

```
epoch 0: train loss/sent=0.8136, time=10.15s
iter 0: dev acc=0.7246
epoch 1: train loss/sent=0.6855, time=11.93s
iter 1: dev acc=0.7493
epoch 2: train loss/sent=0.6229, time=12.35s
iter 2: dev acc=0.7839
epoch 3: train loss/sent=0.5654, time=10.85s
iter 3: dev acc=0.8251
epoch 4: train loss/sent=0.5016, time=10.30s
iter 4: dev acc=0.8507
```

From bow.ipynb: based on this information, which model would you select?

Overfitting

- Goal: fit a target distribution p_*
 - The model may fit the training data (a sample from p_*), but the model may not generalize
- **Symptom**: training loss is decreasing, validation loss is increasing
 - Choose different hyperparameters
 - Add regularization
 - Choose the model with minimum validation loss

Initialization

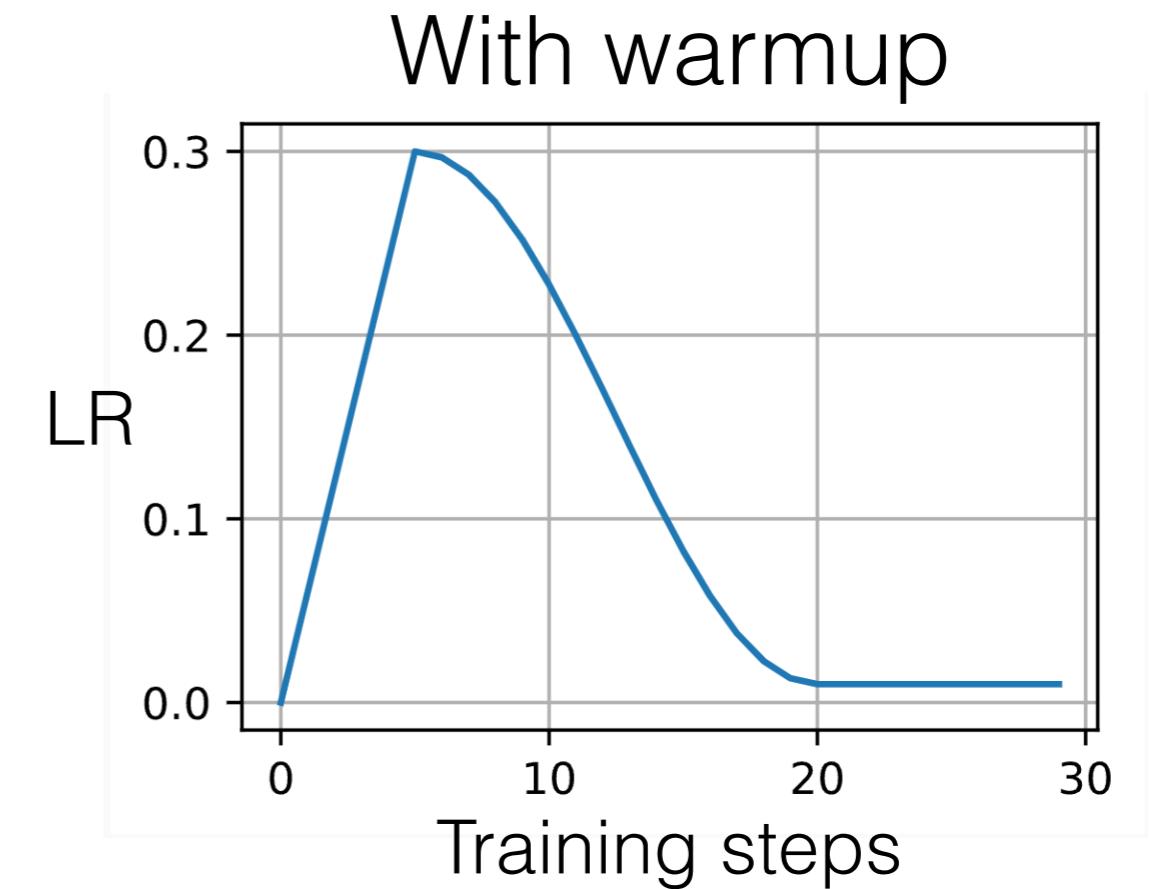
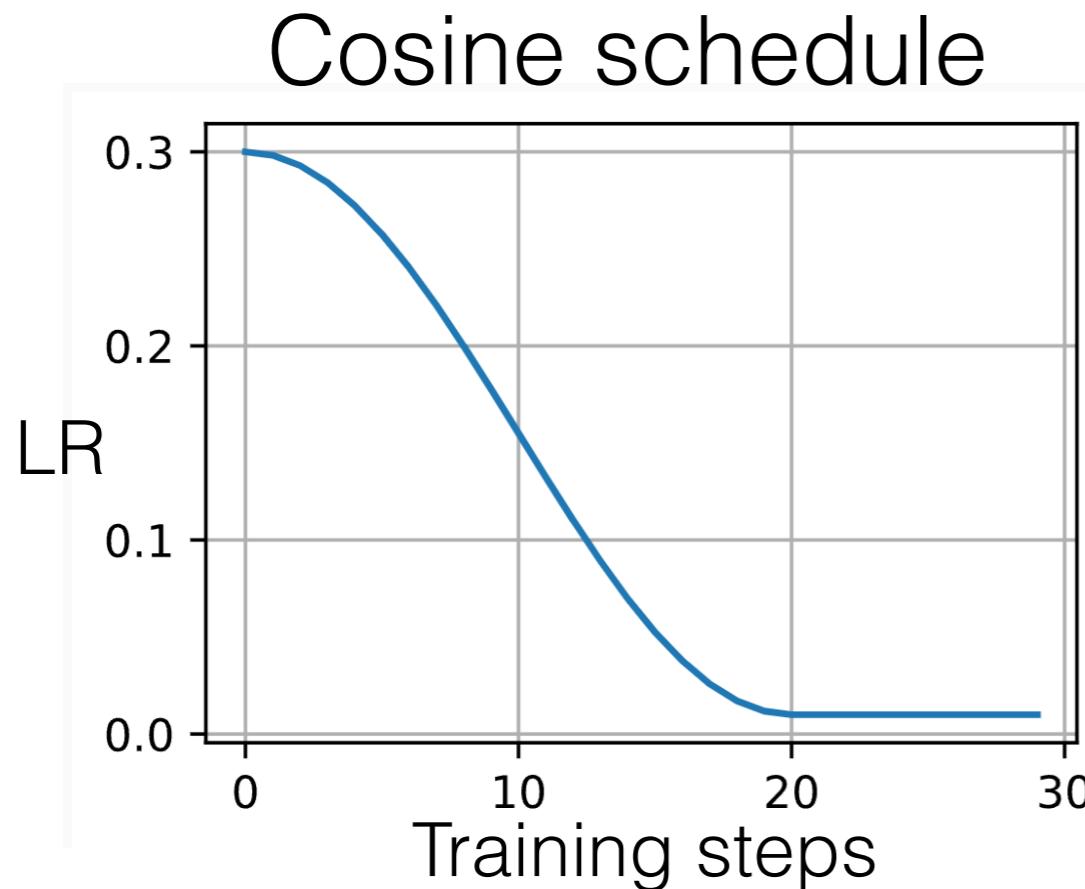
- Weight initialization impacts the optimization trajectory

```
class DeepCBoW(torch.nn.Module):  
    def __init__(self, vocab_size, num_labels, emb_size, hid_size):  
        super(DeepCBoW, self).__init__()  
        self.embedding = nn.Embedding(vocab_size, emb_size)  
        self.linear1 = nn.Linear(emb_size, hid_size)  
        self.output_layer = nn.Linear(hid_size, num_labels)  
  
        nn.init.xavier_uniform_(self.embedding.weight)  
        nn.init.xavier_uniform_(self.linear1.weight)  
        nn.init.xavier_uniform_(self.output_layer.weight)  
  
    def forward(self, tokens):  
        emb = self.embedding(tokens)  
        emb_sum = torch.sum(emb, dim=0)  
        h = emb_sum.view(1, -1)  
        h = torch.tanh(self.linear1(h))  
        out = self.output_layer(h)  
        return out
```

Xavier initialization [Glorot and Bengio 2010]: $W \sim \mathcal{U}\left(-\frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}\right)$

Weights are drawn from a uniform distribution around zero, scaled to balance variance across layers.

Learning rate schedule & warmup



- A *schedule* can help balance between exploration (large updates) and convergence (small updates)
- *Warmup* can help stabilize gradients early in training

Batching

- We typically process multiple examples at once (a batch)
 - Takes advantage of parallel hardware (GPU)
 - Can smooth out noise in individual gradients

example 1
example 2
example 3
...
example B

```
x_batch = X_train[:8]
x_batch
✓ 0.0s
tensor([[26, 26, 26, 26, 26],
       [26, 26, 26, 26, 11],
       [26, 26, 26, 11, 20],
       [26, 26, 11, 20,  0],
       [26, 11, 20,  0, 13],
       [11, 20,  0, 13, 13],
       [26, 26, 26, 26, 26],
       [26, 26, 26, 26, 18]])
```

Batching

- When *inputs* are of variable length, we use a *pad token*

```
tensor([[26, 11, 20, 0, 13, 13, 27, 27, 27, 27],
        [26, 18, 7, 0, 8, 13, 27, 27, 27, 27],
        [26, 17, 20, 15, 4, 17, 19, 27, 27, 27],
        [26, 12, 14, 10, 18, 7, 0, 6, 13, 0]])
['[S]', 'l', 'u', 'a', 'n', 'n', '[PAD]', '[PAD]', '[PAD]', '[PAD]',
 '[S]', 's', 'h', 'a', 'i', 'n', '[PAD]', '[PAD]', '[PAD]', '[PAD]',
 '[S]', 'r', 'u', 'p', 'e', 'r', 't', '[PAD]', '[PAD]', '[PAD]',
 '[S]', 'm', 'o', 'k', 's', 'h', 'a', 'g', 'n', 'a']
```

- We may need to *mask out* operations involving pad tokens

```
def forward(self, words, mask):
    emb = self.embedding(words)
    # Mask out the padding tokens
    emb = emb * mask.unsqueeze(-1)
    h = torch.sum(emb, dim=1)
    for i in range(self.nlayers):
        h = torch.relu(self.linears[i](h))
        h = self.dropout(h)
    out = self.output_layer(h)
    return out
```

Batching

- When *outputs* are of variable length, we mask out the loss for pad tokens

```
# NOTE: We ignore the loss whenever the target token is a padding token
criterion = nn.CrossEntropyLoss(ignore_index=token_to_index['[PAD]'])
```

We'll see a concrete example next class!

Recap: important practical concepts

- Dataset splits
- Overfitting
- Weight initialization
- Optimizer
- Learning rate schedules
- Batching
- (Adam optimizer in the next lecture)

Overall recap

- Language modeling
- Basic methods: bigram/ngram, feedforward neural

Next 2 lectures

- Recurrent architecture
- Transformer architecture

Both of these can be used to parameterize a language model.

Thank you