CS11-711 Advanced NLP

Language Modeling

Sean Welleck

https://cmu-l3.github.io/anlp-fall2025/

https://github.com/cmu-l3/anlp-fall2025-code

Slides adapted from Graham Neubig's Fall 2024 course

Types of Prediction: Binary, Multi-class, Structured

- Two classes (**binary classification**)

I hate this movie → positive / negative

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

I hate this movie → very good / good / neutral / bad / very bad

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

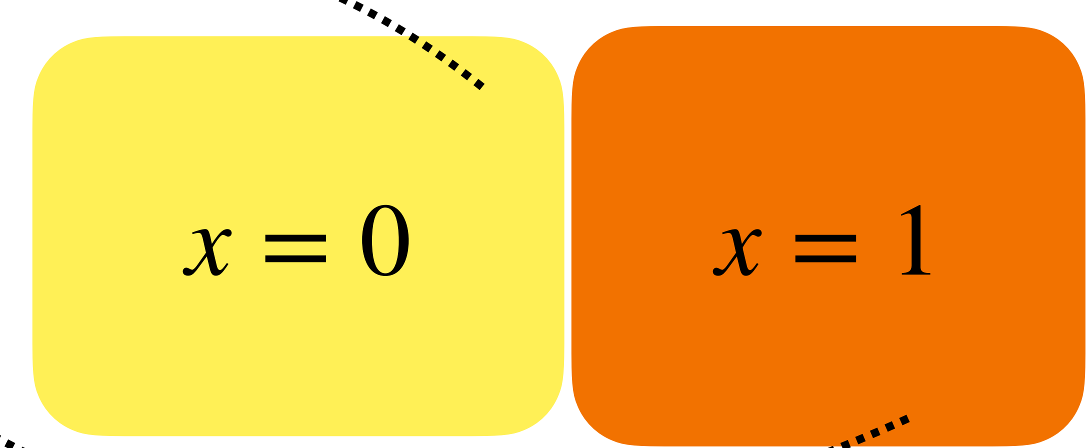
I hate this movie → PRP VBP DT NN

I hate this movie → *kono eiga ga kirai*

Language Modeling

What is a language model?

- A language model is a probability distribution over all sequences
  - $P(X)$
- Example probability distribution: **biased coin**
  - $P(X) = \begin{cases} 0.4 & x \textbf{ is } 0 \\ 0.6 & x \textbf{ is } 1 \end{cases}$



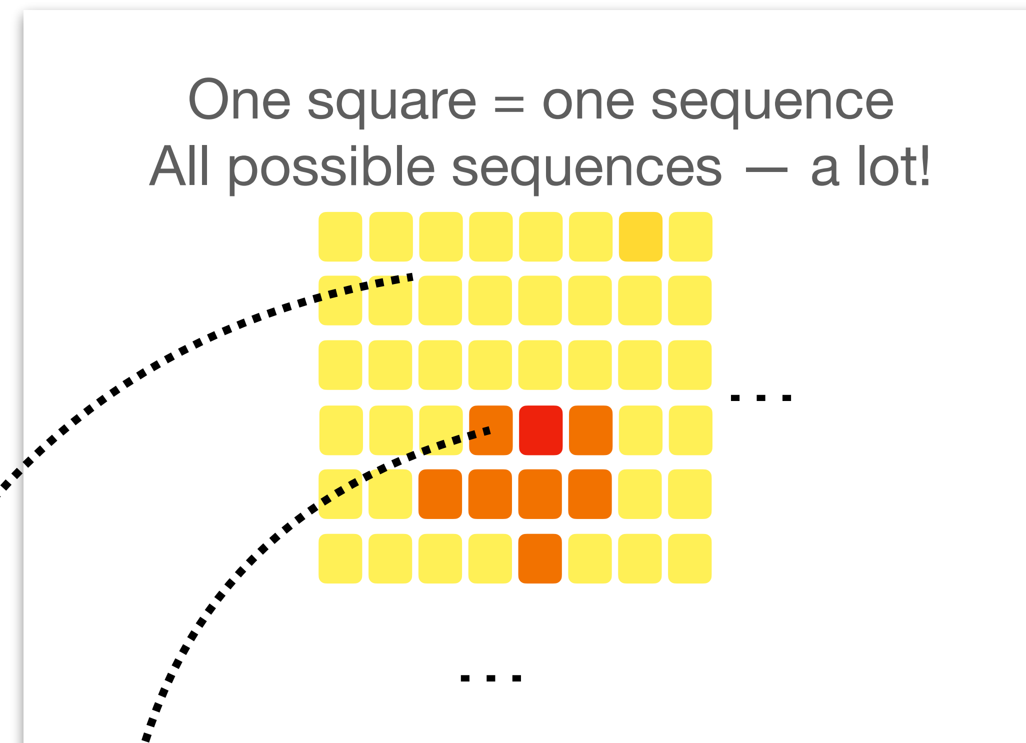
What is a language model?

- 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 `a a .`
    ...
    0.019100 if x is `a cat sat.`
    ...



What can we do with language models?

- **Score** sequences:

P(Jane went to the store .) → high
P(store to Jane went the .) → 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

$$P(X) = \prod_{t=1}^{T} P\left(\underbrace{x_t}_{\text{Next Token}} \mid \underbrace{x_1, \ldots, x_{t-1}}_{\text{Context}}\right)$$

Decomposes sequence modeling into
next-token modeling

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

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

Auto-regressive Language Models

$$P(X) = \prod_{t=1}^{T} P\left(\underbrace{x_t}_{\text{Next Token}} \mid \underbrace{x_1, \ldots, x_{t-1}}_{\text{Context}}\right)$$

Key question: modeling

$$P\left(x_t \mid x_1, \ldots, x_{t-1}\right)$$

Roadmap

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

Bigram models

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

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.

```python
data = open('names.txt').read().splitlines()
data[:10]
```

✓ 0.0s

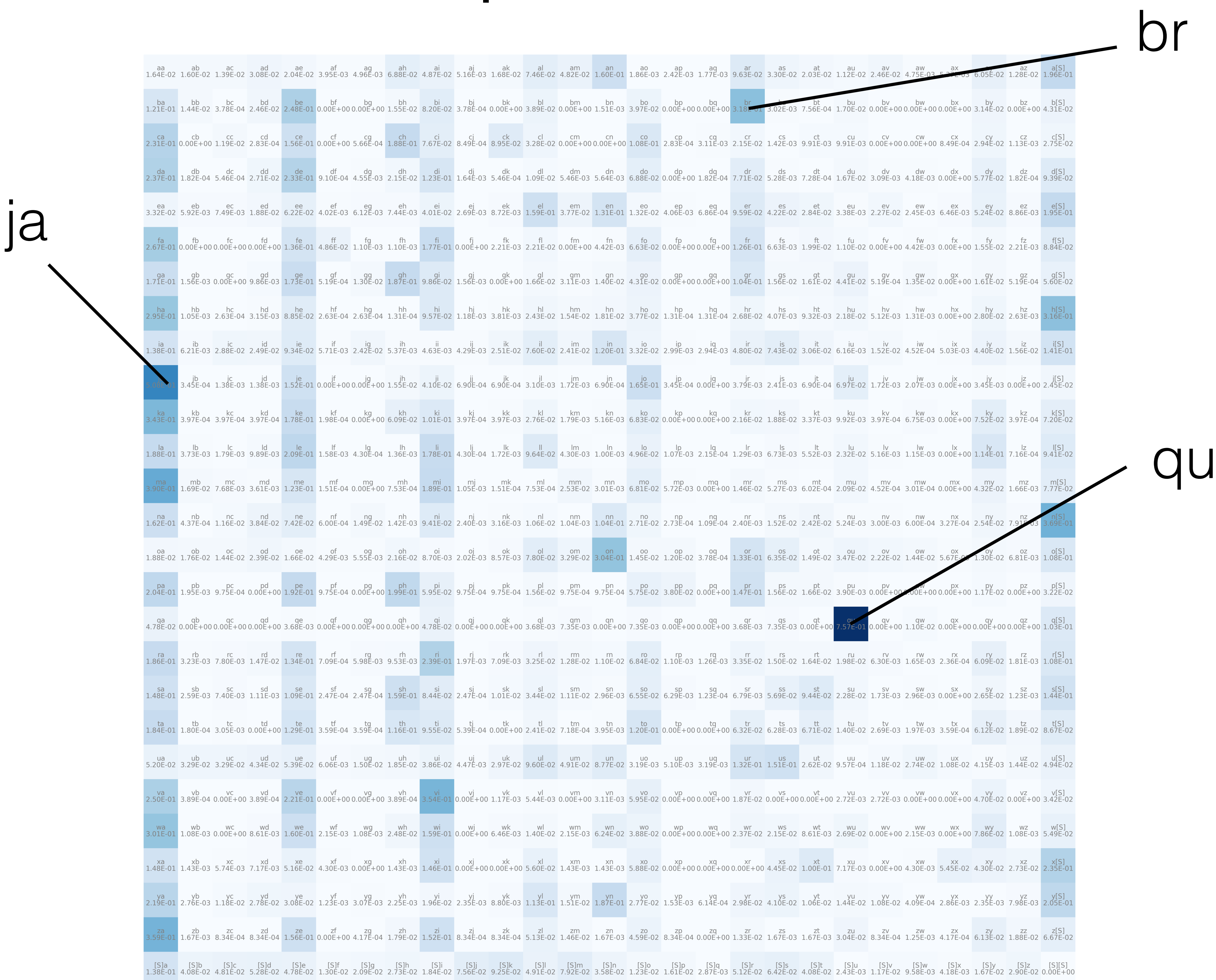
```
['emma',
 'olivia',
 'ava',
 'isabella',
 'sophia',
 'charlotte',
 'mia',
 'amelia',
 'harper',
 'evelyn']
```

```python
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)]
```

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

Model probabilities



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

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)

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

Dataset:
samples from p_*

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:

$$\hat{x}_t \sim p_\theta(x_t \mid \hat{x}_{t-1})$$

- 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

- **Log-likelihood:**

$$LL(\mathcal{X}_{\text{test}}) = \sum_{X \in \mathcal{X}_{\text{test}}} \log P(X))$$

- **Per-word Log Likelihood:**

$$WLL(\mathcal{X}_{\text{test}}) = \frac{1}{\sum_{X \in \mathcal{X}_{\text{test}}} |X|} \sum_{X \in \mathcal{X}_{\text{test}}} \log P(X))$$

Papers often also report negative log likelihood (lower better), as that is used in loss.

Perplexity

- **Perplexity:**

$$PPL(\mathcal{X}_{\text{test}}) = 2^{H(\mathcal{X}_{\text{test}})} = e^{-WLL(\mathcal{X}_{\text{test}})}$$

When a dog sees a squirrel it will usually ___

Token: ' be' - Probability: 0.0352 → PPL= 28.4
Token: ' jump' - Probability: 0.0338 → PPL= 29.6
Token: ' start' - Probability: 0.0289 → PPL= 34.6
Token: ' run' - Probability: 0.0277 → PPL= 36.1
Token: ' try' - Probability: 0.0219 → PPL= 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
  - Limited context size
- **Next**: Ngram models

Ngram models

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

- 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

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

Neural network parameters :)

Bengio et al 2003, *A Neural Probabilistic Language Model*

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

Neural language model

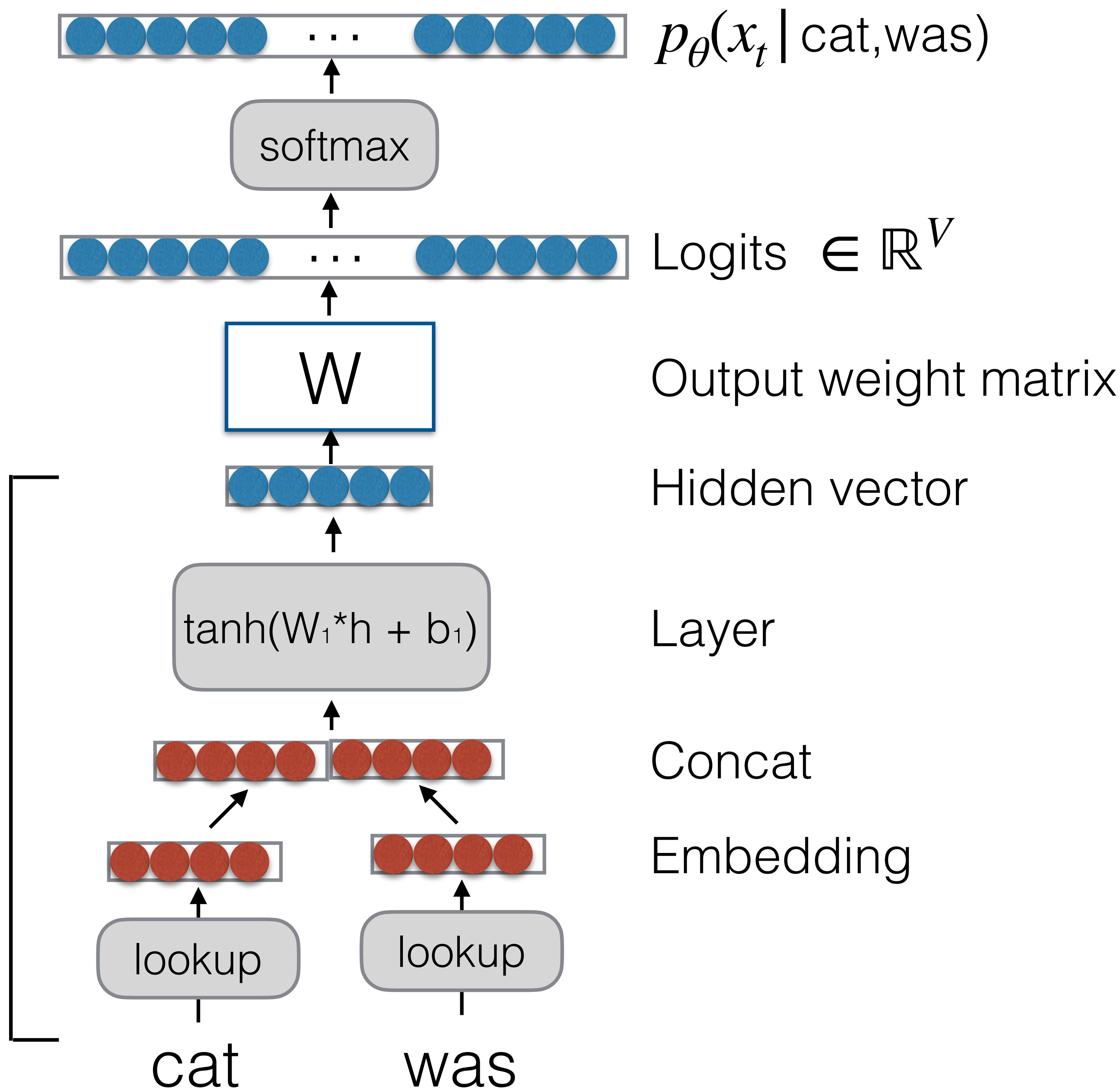
- Ngram language models do not take into account the similarity of words or contexts
    - *The cat was walking in the bedroom*
    - *The dog was running in a room*
- *Solution*: use learned, distributed representations

Bengio et al 2003, *A Neural Probabilistic Language Model*

Feedforward neural language model

$$s = Wh$$

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



Feedforward neural language model

- Training: maximum likelihood estimation

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

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

- 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)

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

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

Bengio et al 2003, *A Neural Probabilistic Language Model*

In code

```python
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:

$$
\begin{array}{rl}
\text{giving} & \begin{bmatrix} 1.2 \\ -0.1 \\ 0.7 \\ -2.1 \\ 0.5 \end{bmatrix} \\
\text{a} & \begin{bmatrix} -0.3 \\ 2.0 \\ 0.6 \\ -0.8 \\ -0.4 \end{bmatrix}
\end{array}
\;*\;
\begin{array}{c}
\boldsymbol{w}_{34} \\
\begin{bmatrix} 1.5 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1.3 \\ 0 \\ 0 \\ 0 \end{bmatrix}
\end{array}
\;+\;
\begin{array}{c}
b_{34} \\
-2
\end{array}
\;=
$$

Example possibility:

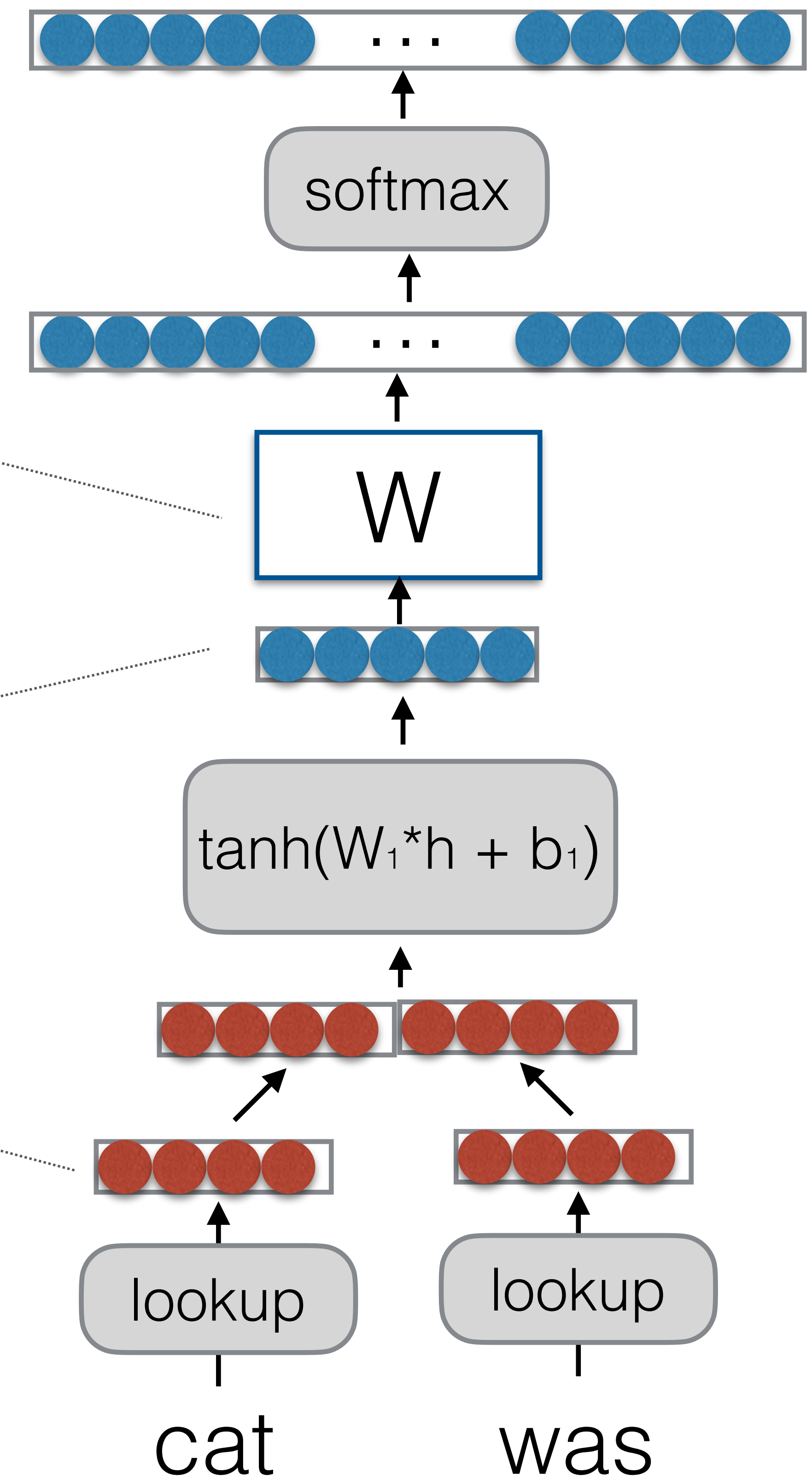
positive number if the previous word is a determiner and second-to-previous word is a verb

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

- $p_j^w = p(w \mid h_j) = \dfrac{1}{Z_j} \exp\left(w^\top h_j\right)$

- $p_k^w = p(w \mid h_k) = \dfrac{1}{Z_k} \exp\left(w^\top h_k\right)$

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

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

It's a great → movie

It is a wonderful → movie

"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 purchased a car	she bought a bicycle 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

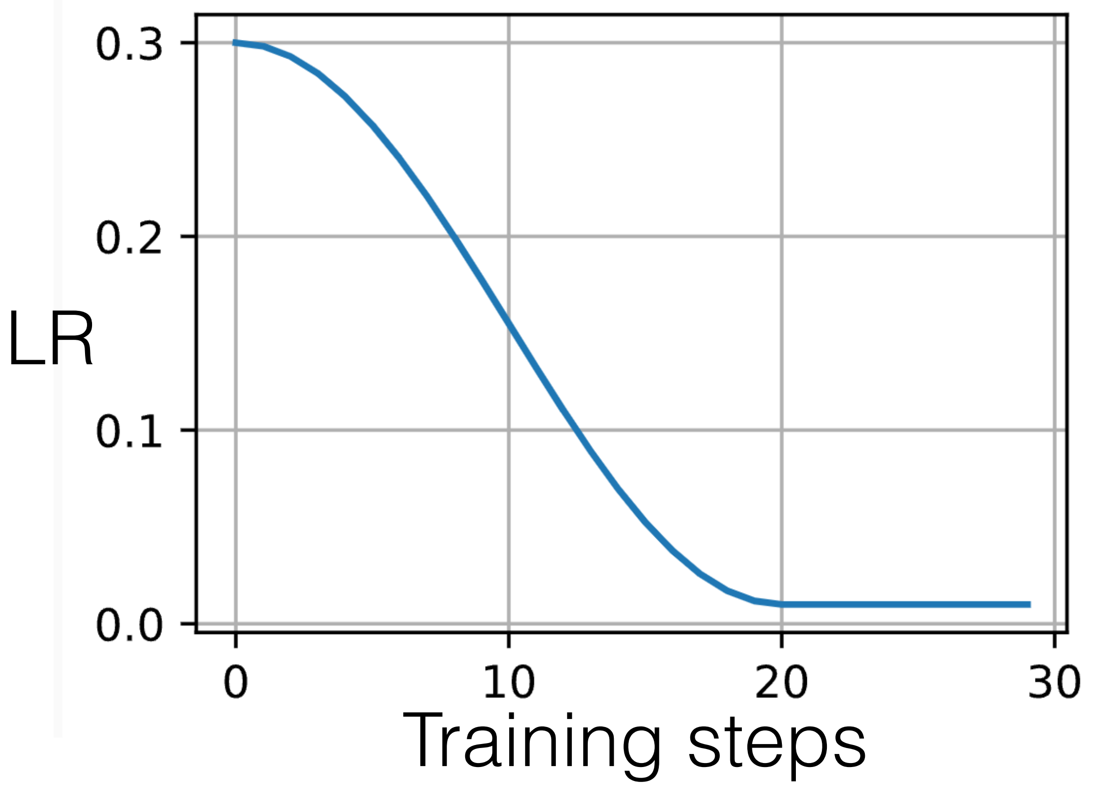
```
class DeepCBoW(torch.nn.Module):
    def __init__(self, vocab_size, num_labels, emb_size, hid_size):
        super(DeepCBoW, self).__init__()
        self.embedding = nn.Embedding(vocab_size, emb_size)
        self.linear1 = nn.Linear(emb_size, hid_size)
        self.output_layer = nn.Linear(hid_size, 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]: $W \sim \mathcal{U}\left(-\frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}\right)$

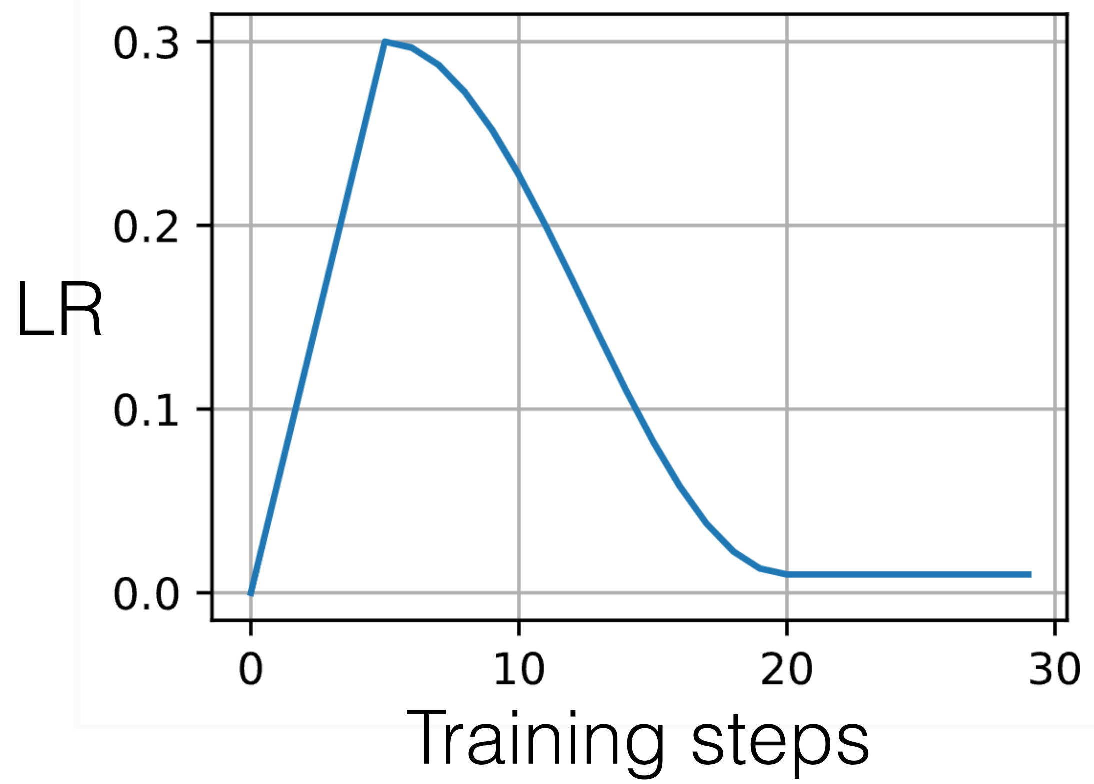
Weights are drawn from a uniform distribution around zero, scaled to balance variance across layers.

Learning rate schedule & warmup

Cosine schedule



With 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