

# 10-301/601: Introduction to Machine Learning

## Lecture 21 – Language Modeling

Henry Chai

6/3/25

# Front Matter

- Announcements
  - HW5 to be released on 6/3 (today!), due 6/6 at 11:59 PM
  - Schedule change: two recitations this week
    - **Recitation on 6/4 will be a PyTorch tutorial**
    - Recitation on 6/5 will be Quiz 3 preparation

# Language Generation

- Goal: generate realistic sentences in some human language and engage in conversation
- Idea: condition on the previous words in the sentence to predict the next word
- Better idea: condition on the previous words in the sentence to predict a *distribution* over the next word
- A **language model** defines a probability distribution over sequences of words
  - We can use a language model to compute conditional probabilities i.e., the probability of the next word *conditioned* on all the previous words.

# Language Models

1. Convert raw text into sequence data

$$\mathbf{x}^{(i)} = [\mathbf{x}_1^{(i)}, \dots, \mathbf{x}_{T_i}^{(i)}]$$

2. Learn or approximate a joint probability distribution over sequences

$$P(\mathbf{x}^{(i)}) = P(\mathbf{x}_1^{(i)}, \dots, \mathbf{x}_{T_i}^{(i)})$$

3. Sample from the implied conditional distribution to generate new sequences

$$P(\mathbf{x}_{T_i+1} \mid \mathbf{x}_1^{(i)}, \dots, \mathbf{x}_{T_i}^{(i)}) = \frac{P(\mathbf{x}_1^{(i)}, \dots, \mathbf{x}_{T_i}^{(i)}, \mathbf{x}_{T_i+1})}{P(\mathbf{x}_1^{(i)}, \dots, \mathbf{x}_{T_i}^{(i)})}$$

# Tokenization and Embedding

1. Convert raw text into sequence data

$$\mathbf{x}^{(i)} = [\mathbf{x}_1^{(i)}, \dots, \mathbf{x}_{T_i}^{(i)}]$$

- High-level approach: split raw text into smaller units (“tokens”), then learn a dense, numerical vector representation (“embedding”) for each token

# Tokenization

- Example: “Henry is giving a lecture on language models”
- Idea: word-based tokenization  
[“henry”, “is”, “giving”, “a”, “lecture”, “on”, “language”,  
“models”]

# Tokenization

- Example: “Henry is givin’ a lectrue on LMs”
- Idea: word-based tokenization?

[“henry”, “is”, “?”, “a”, “?”, “on”, “????”]

- Can have difficulty trading off between vocabulary size and computational tractability
- Similar words e.g., “model” and “models” can get mapped to completely disparate representations
- Typos or acronyms will likely be out-of-vocabulary (OOV)

# Tokenization

- Example: “Henry is givin’ a lectrue on LMs”

- Idea: character-based tokenization:

[“h”, “e”, “n”, “r”, “y”, “i”, “s”, “g”, “i”, “v”, “i”, “n”, “ ’ ”, ... ]

- Much smaller vocabularies but a lot of semantic meaning is lost...
- Sequences will be much longer than word-based tokenization, potentially causing computational issues
- Can do well on logographic languages e.g., Kanji 漢字



# Tokenization

- Example: “Henry is givin’ a lectrue on LMs”

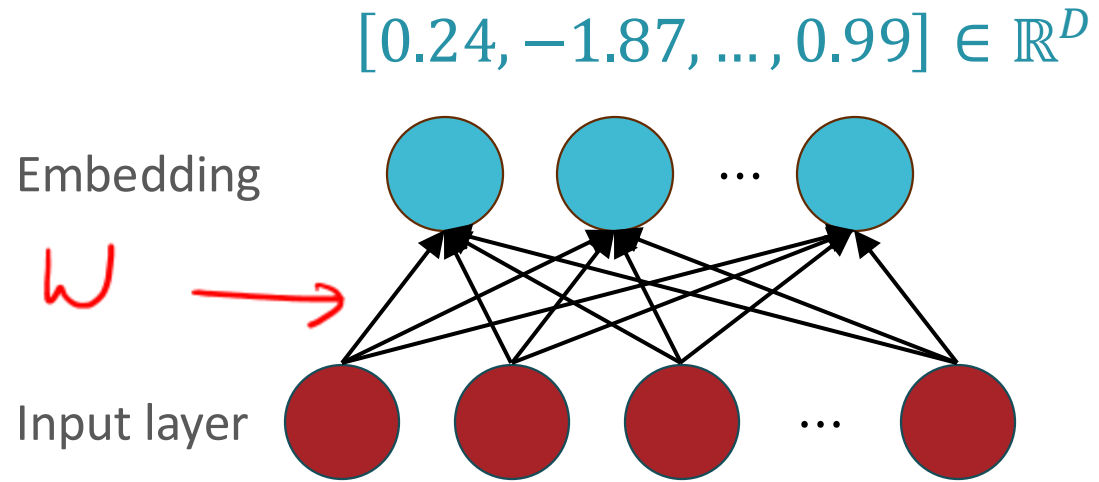
- Common practice: subword tokenization

[“henry”, “is”, “giv”, “##in”, “ ’”, “a”, “lect” “##re”, “on”, “language”, “model”, “#s”]

- Split long or rare words into smaller, semantically meaningful components or *subwords*
- Common algorithms for computing subwords consider the most frequently occurring substrings

# Embedding

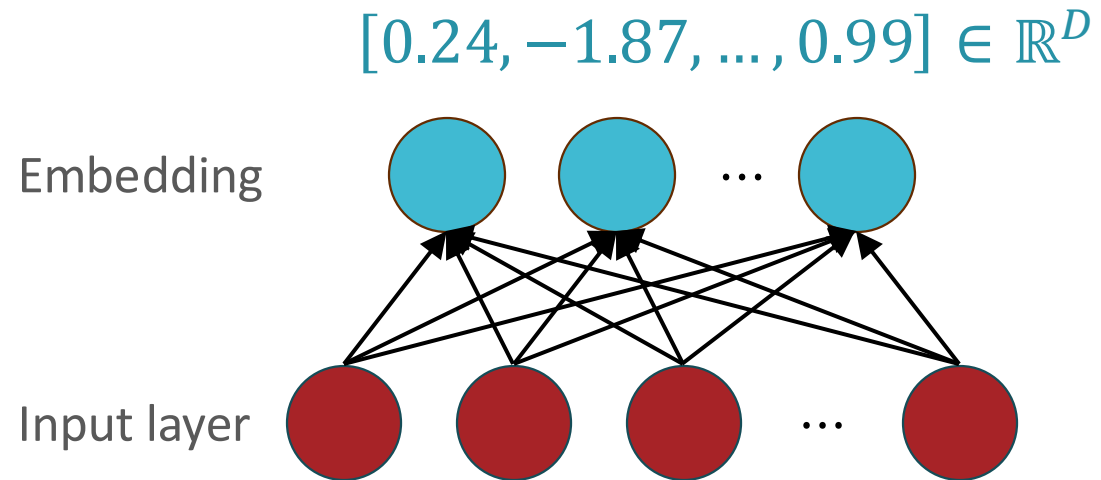
- Given a vocabulary  $V$  with  $|V|$  tokens, learn an embedding by training a 1-layer, fully-connected feed-forward NN that takes one-hot encoded vectors as input



- Example: "is"  $\rightarrow [0, 0, 1, \dots, 0] \in \mathbb{R}^{|V|}$

Okay but how  
do I go about  
training this  
neural  
network?

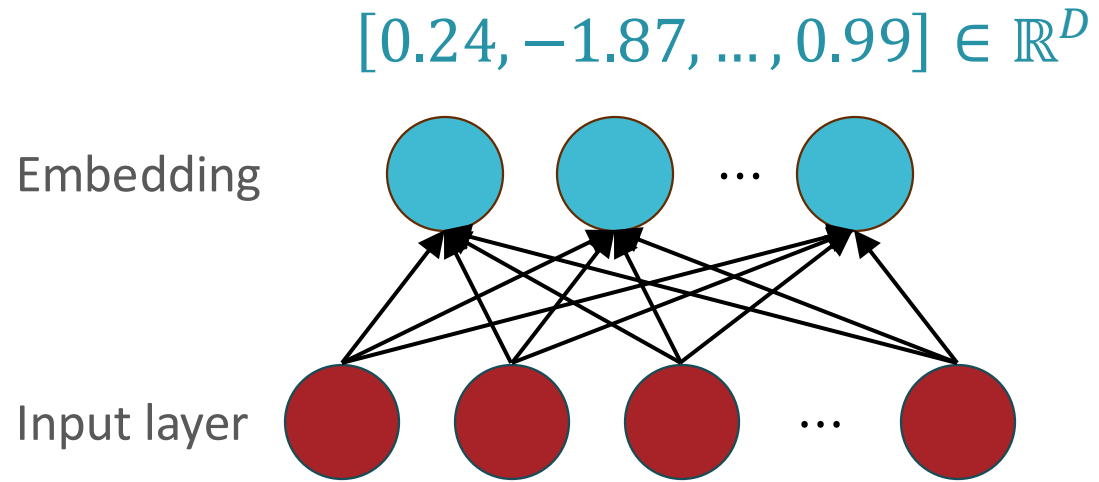
- Given a vocabulary  $V$  with  $|V|$  tokens, learn an embedding by training a 1-layer, fully-connected feed-forward NN that takes one-hot encoded vectors as input



- Example: "is"  $\rightarrow [0, 0, 1, \dots, 0] \in \mathbb{R}^{|V|}$

# Embedding

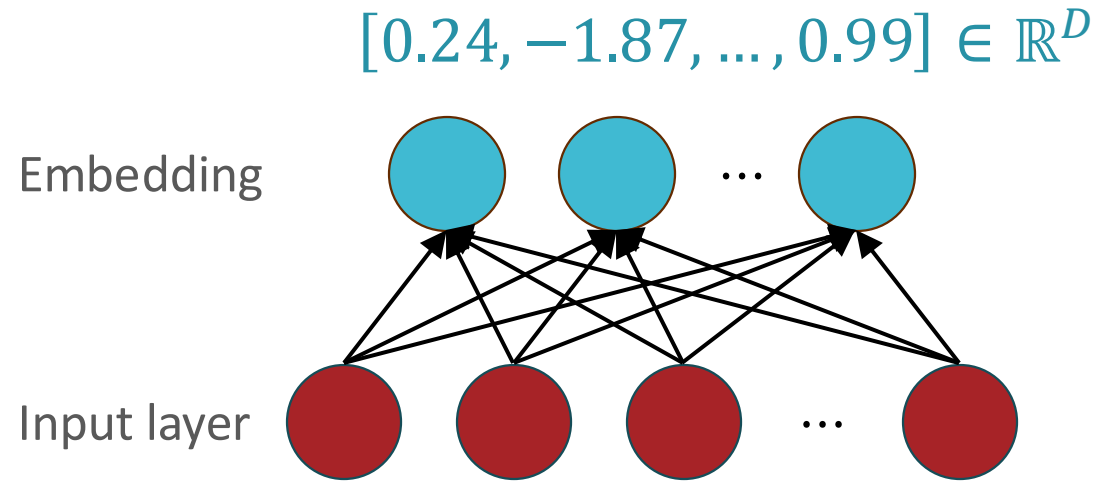
- Given a vocabulary  $V$  with  $|V|$  tokens, learn an embedding by training a 1-layer, fully-connected feed-forward NN that takes one-hot encoded vectors as input



- Example: “is”  $\rightarrow [0, 0, 1, \dots, 0] \in \mathbb{R}^{|V|}$
- Idea: use a pretrained embedding e.g., [word2vec](#) or [GloVe](#)
  - Requires you to use the same vocabulary/tokenization

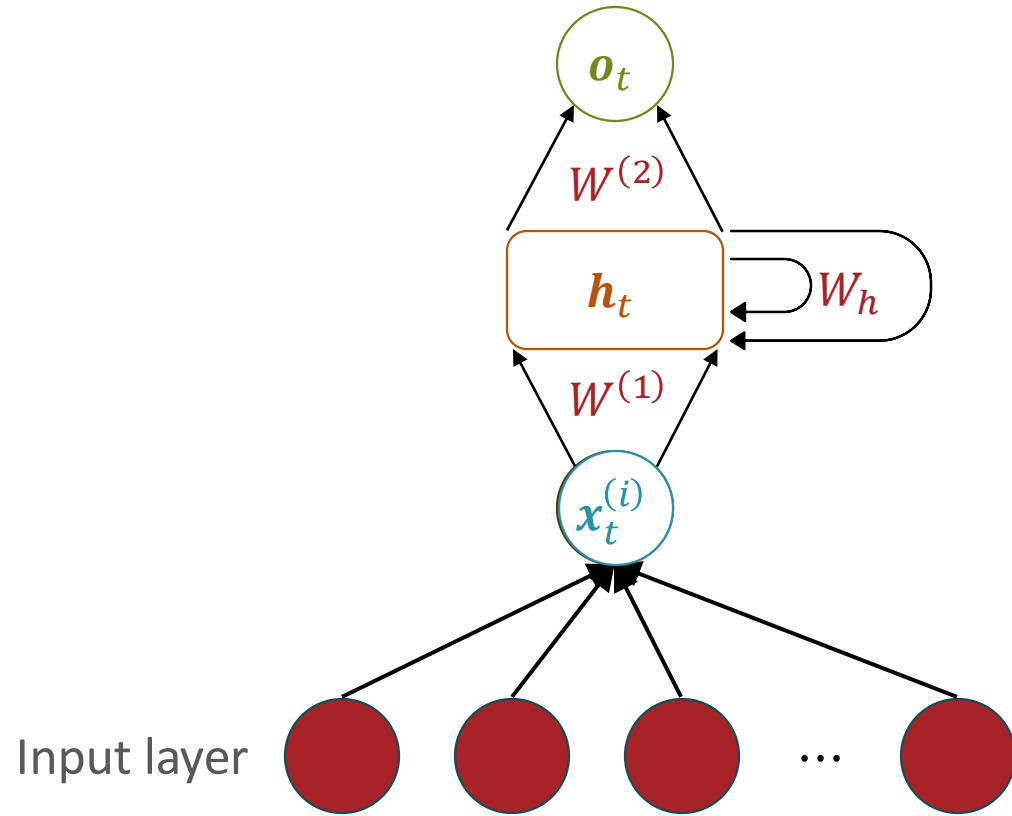
# Embedding Layer

- Given a vocabulary  $V$  with  $|V|$  tokens, learn an embedding by training a 1-layer, fully-connected feed-forward NN that takes one-hot encoded vectors as input



- Example: “is”  $\rightarrow [0, 0, 1, \dots, 0] \in \mathbb{R}^{|V|}$
- Common practice: add this 1-layer NN to whatever architecture you’re using and fit it to the task

# Embedding Layer



- Example: “is”  $\rightarrow [0, 0, 1, \dots, 0] \in \mathbb{R}^{|V|}$
- Common practice: add this 1-layer NN to whatever architecture you’re using and fit it to the task

$$P(A \cap B) \\ = P(A) P(B|A)$$

## Language Models

1. Convert raw text into sequence data

$$\mathbf{x}^{(i)} = [\mathbf{x}_1^{(i)}, \dots, \mathbf{x}_{T_i}^{(i)}]$$

2. Learn or approximate a joint probability distribution over sequences

$$P(\mathbf{x}^{(i)}) = P(\mathbf{x}_1^{(i)}, \dots, \mathbf{x}_{T_i}^{(i)})$$

- Use the chain rule of probability: predict the next word based on the previous words in the sequence

$$\begin{aligned} P(\mathbf{x}^{(i)}) &= P(\mathbf{x}_1^{(i)}) \\ &\quad * P(\mathbf{x}_2^{(i)} \mid \mathbf{x}_1^{(i)}) \\ &\quad \vdots \\ &\quad * P(\mathbf{x}_{T_i}^{(i)} \mid \mathbf{x}_{T_i-1}^{(i)}, \dots, \mathbf{x}_1^{(i)}) \end{aligned}$$

# Language Models

1. Convert raw text into sequence data

$$\mathbf{x}^{(i)} = [\mathbf{x}_1^{(i)}, \dots, \mathbf{x}_{T_i}^{(i)}]$$

2. Learn or approximate a joint probability distribution over sequences

$$P(\mathbf{x}^{(i)}) = P(\mathbf{x}_1^{(i)}, \dots, \mathbf{x}_{T_i}^{(i)})$$

- ~~Use the chain rule of probability~~ Just throw an RNN at it!

$$\begin{aligned} P(\mathbf{x}^{(i)}) &= P(\mathbf{x}_1^{(i)}) \\ &\quad * P(\mathbf{x}_2^{(i)} \mid \mathbf{x}_1^{(i)}) \\ &\quad \vdots \\ &\quad * P(\mathbf{x}_{T_i}^{(i)} \mid \mathbf{x}_{T_i-1}^{(i)}, \dots, \mathbf{x}_1^{(i)}) \end{aligned}$$



# RNN Language Models

1. Convert raw text into sequence data

$$\mathbf{x}^{(i)} = [\mathbf{x}_1^{(i)}, \dots, \mathbf{x}_{T_i}^{(i)}]$$

2. Learn or approximate a joint probability distribution over sequences

$$P(\mathbf{x}^{(i)}) = P(\mathbf{x}_1^{(i)}, \dots, \mathbf{x}_{T_i}^{(i)})$$

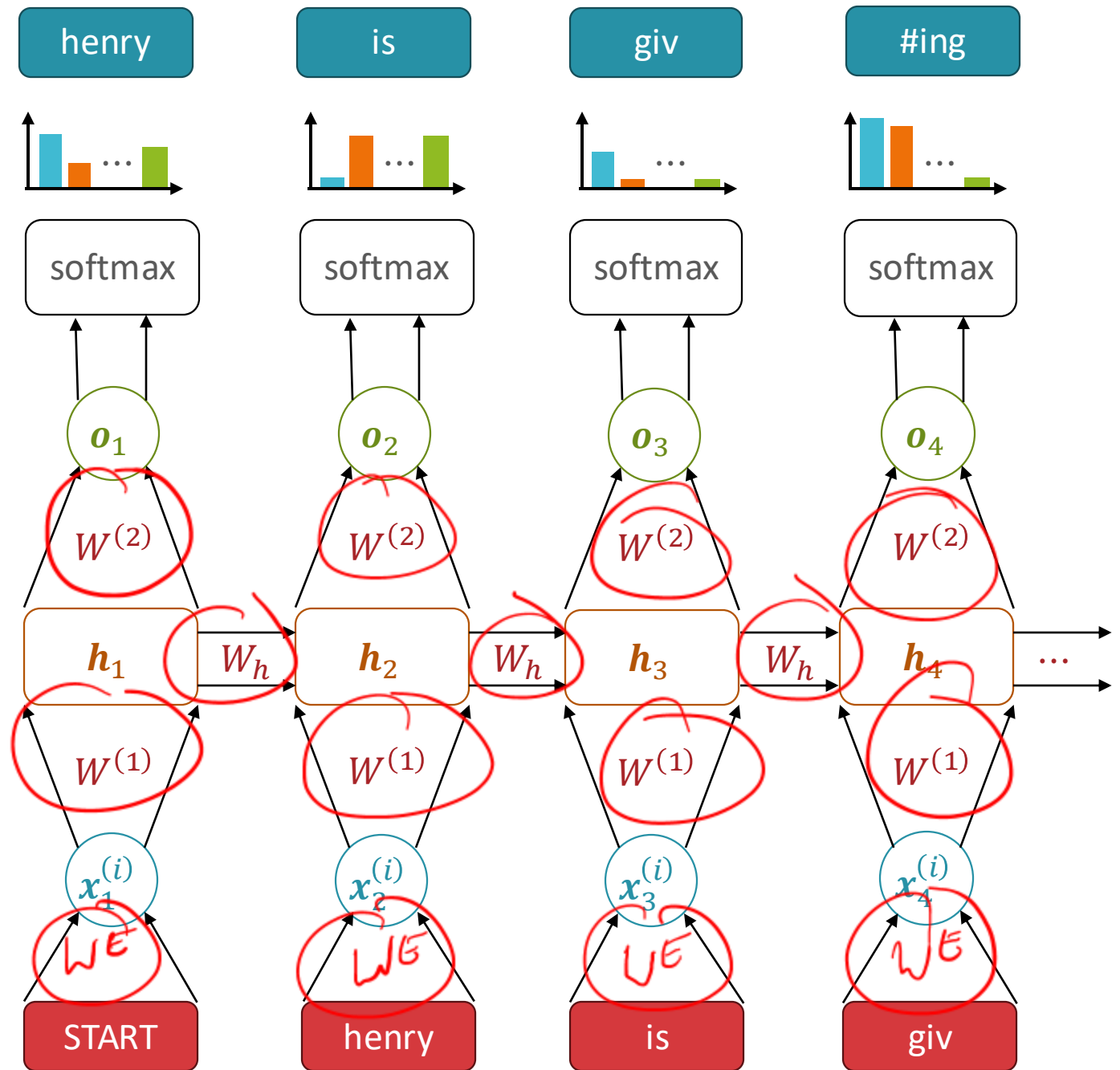
- ~~Use the chain rule of probability~~ Just throw an RNN at it!

$$\begin{aligned} P(\mathbf{x}^{(i)}) &\approx \mathbf{o}_1(\mathbf{x}_1^{(i)}) \\ &\quad * \mathbf{o}_2(\mathbf{x}_2^{(i)}, \mathbf{h}_1(\mathbf{x}_1^{(i)})) \\ &\quad \vdots \\ &\quad * \mathbf{o}_{T_i}(\mathbf{x}_{T_i}^{(i)}, \mathbf{h}_{T_i-1}(\mathbf{x}_{T_i-1}^{(i)}, \dots, \mathbf{x}_1^{(i)})) \end{aligned}$$

# RNN Language Models: Training

Target sequence (try to  
predict the next word)

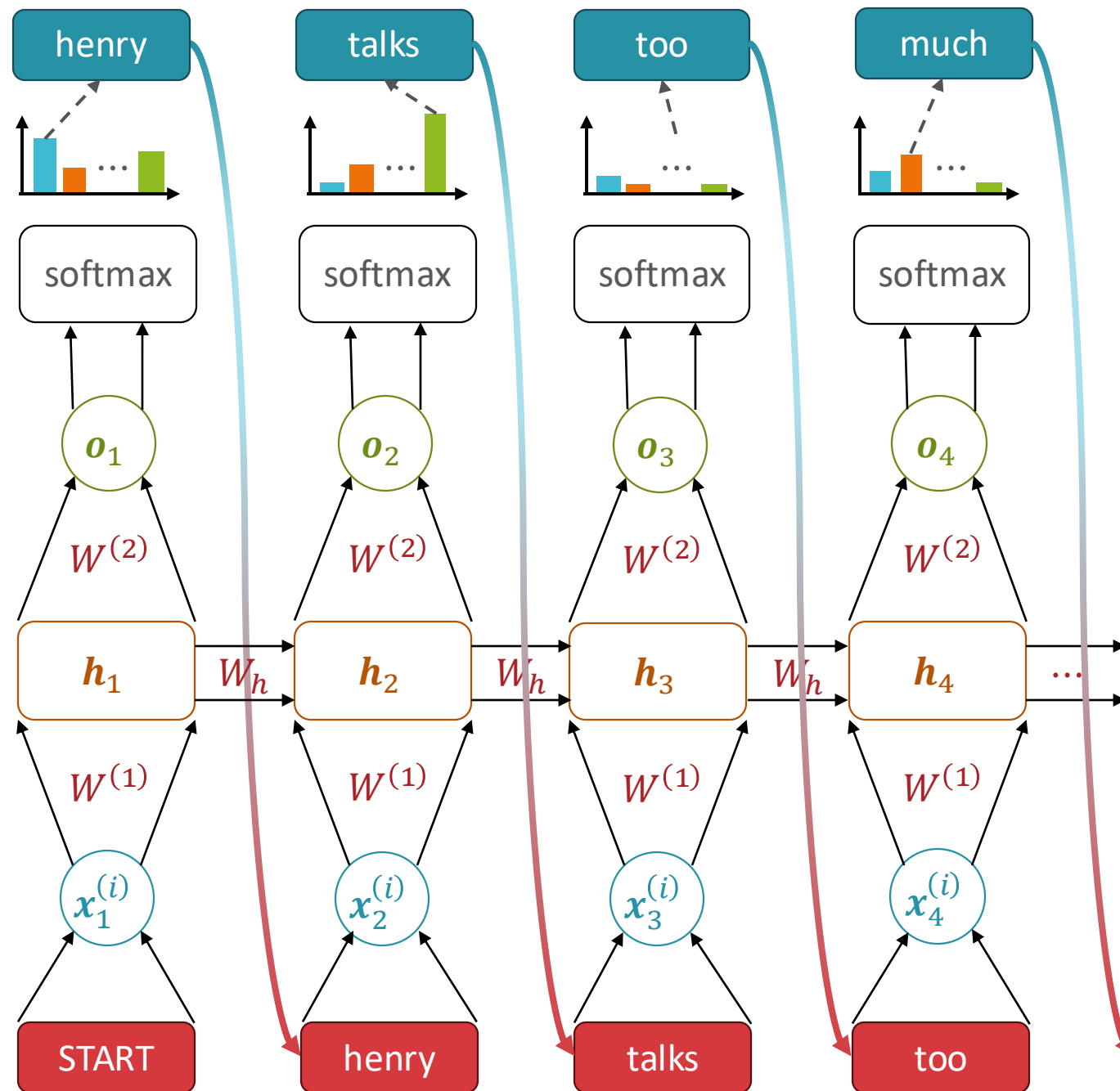
Input sequence



Generated sequence (use each token as the input to the next timestep)

# RNN Language Model: Sampling

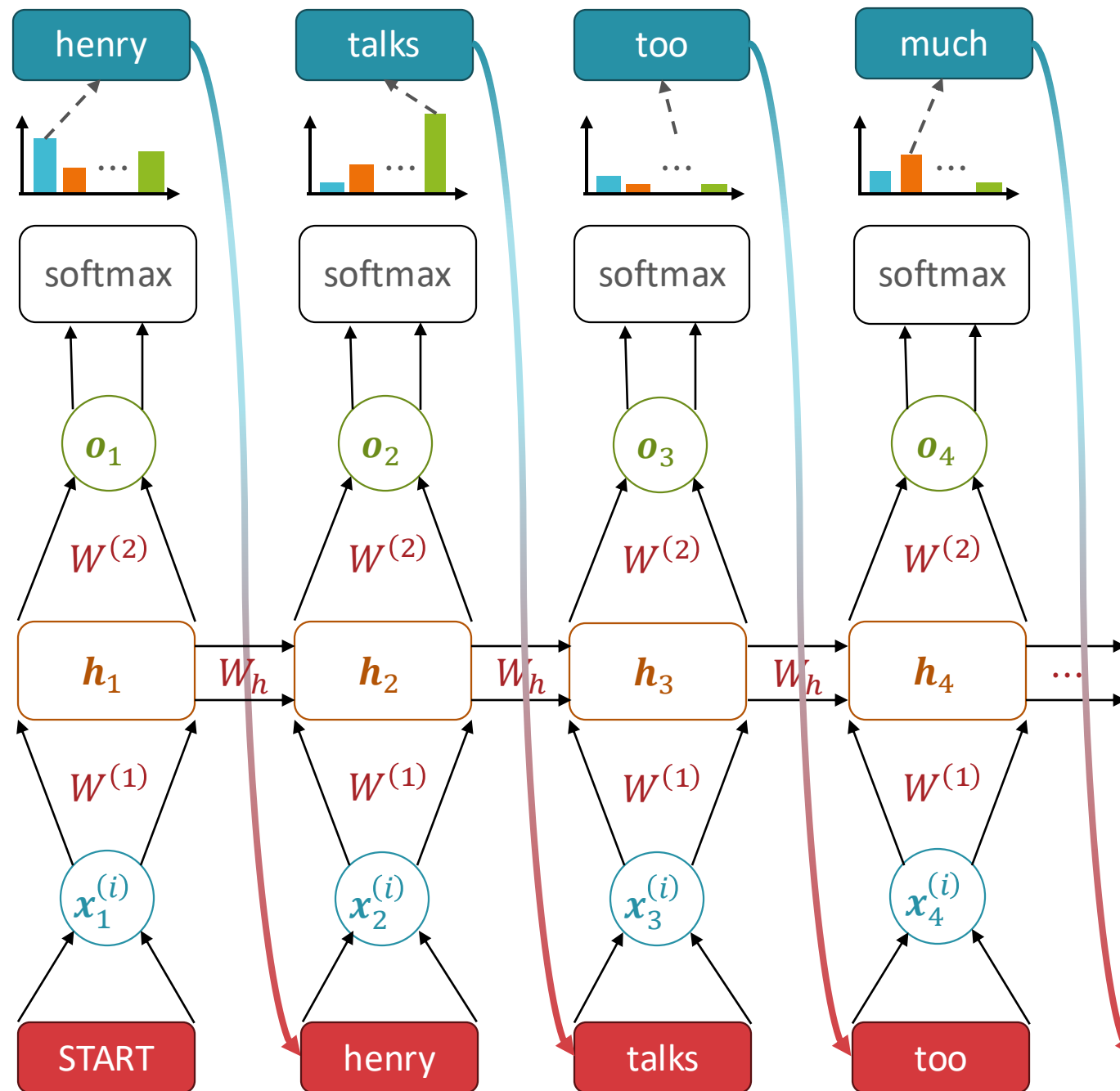
Input sequence



Generated sequence (use each token as the input to the next timestep)

Aside:  
Sampling from these distributions to get the next word is not always the best thing to do

Input sequence



# RNN Language Models: Pros & Cons

- Pros:
  - Can handle arbitrary sequence lengths without having to increase model size (i.e., # of learnable parameters)
  - Trainable via backpropagation
- Cons

# Backpropagation: Procedural Method

## Algorithm 1 Forward Computation

```
1: procedure NNFORWARD(Training example  $(\mathbf{x}, \mathbf{y})$ , Params  $\alpha, \beta$ )
2:    $\mathbf{a} = \alpha \mathbf{x}$ 
3:    $\mathbf{z} = \sigma(\mathbf{a})$ 
4:    $\mathbf{b} = \beta \mathbf{z}$ 
5:    $\hat{\mathbf{y}} = \text{softmax}(\mathbf{b})$ 
6:    $J = -\mathbf{y}^T \log \hat{\mathbf{y}}$ 
7:    $\mathbf{o} = \text{object}(\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J)$ 
8:   return intermediate quantities  $\mathbf{o}$ 
```

## Algorithm 2 Backpropagation

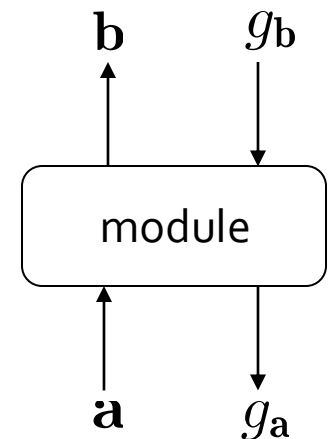
```
1: procedure NNBACKWARD(Training example  $(\mathbf{x}, \mathbf{y})$ , Params  $\alpha, \beta$ ,  
  Intermediates  $\mathbf{o}$ )
2:   Place intermediate quantities  $\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J$  in  $\mathbf{o}$  in scope
3:    $\mathbf{g}_{\hat{\mathbf{y}}} = -\mathbf{y} \div \hat{\mathbf{y}}$ 
4:    $\mathbf{g}_{\mathbf{b}} = \mathbf{g}_{\hat{\mathbf{y}}}^T (\text{diag}(\hat{\mathbf{y}}) - \hat{\mathbf{y}} \hat{\mathbf{y}}^T)$ 
5:    $\mathbf{g}_{\beta} = \mathbf{g}_{\mathbf{b}}^T \mathbf{z}^T$ 
6:    $\mathbf{g}_{\mathbf{z}} = \beta^T \mathbf{g}_{\mathbf{b}}$ 
7:    $\mathbf{g}_{\mathbf{a}} = \mathbf{g}_{\mathbf{z}} \odot \mathbf{z} \odot (1 - \mathbf{z})$ 
8:    $\mathbf{g}_{\alpha} = \mathbf{g}_{\mathbf{a}} \mathbf{x}^T$ 
9:   return parameter gradients  $\mathbf{g}_{\alpha}, \mathbf{g}_{\beta}$ 
```

Issues:

1. Hard to reuse / adapt for other models
2. Hard to optimize individual steps
3. Hard to debug using the finite-difference check

# Module-based AutoDiff

- Key Idea:
  - componentize the computation of the neural-network into layers
  - each layer consolidates multiple **real-valued nodes** in the computation graph (a subset of them) into one **vector-valued node** (aka. a **module**)
- Each **module** is capable of two actions:
  - **Forward computation of the output given some input**
  - **Backward computation of the gradient with respect to the input given the gradient with respect to the output**



# Module-based AutoDiff

**Linear Module** The linear layer has two inputs: a vector  $\mathbf{a}$  and parameters  $\omega \in \mathbb{R}^{B \times A}$ . The output  $\mathbf{b}$  is not used by LINEARBACKWARD, but we pass it in for consistency of form.

```
1: procedure LINEARFORWARD( $\mathbf{a}, \omega$ )
2:    $\mathbf{b} = \omega \mathbf{a}$ 
3:   return  $\mathbf{b}$ 
4: procedure LINEARBACKWARD( $\mathbf{a}, \omega, \mathbf{b}, \mathbf{g}_\mathbf{b}$ )
5:    $\mathbf{g}_\omega = \mathbf{g}_\mathbf{b} \mathbf{a}^T$ 
6:    $\mathbf{g}_\mathbf{a} = \omega^T \mathbf{g}_\mathbf{b}$ 
7:   return  $\mathbf{g}_\omega, \mathbf{g}_\mathbf{a}$ 
```

**Softmax Module** The softmax layer has only one input vector  $\mathbf{a}$ . For any vector  $\mathbf{v} \in \mathbb{R}^D$ , we have that  $\text{diag}(\mathbf{v})$  returns a  $D \times D$  diagonal matrix whose diagonal entries are  $v_1, v_2, \dots, v_D$  and whose non-diagonal entries are zero.

```
1: procedure SOFTMAXFORWARD( $\mathbf{a}$ )
2:    $\mathbf{b} = \text{softmax}(\mathbf{a})$ 
3:   return  $\mathbf{b}$ 
4: procedure SOFTMAXBACKWARD( $\mathbf{a}, \mathbf{b}, \mathbf{g}_\mathbf{b}$ )
5:    $\mathbf{g}_\mathbf{a} = \mathbf{g}_\mathbf{b}^T (\text{diag}(\mathbf{b}) - \mathbf{b} \mathbf{b}^T)$ 
6:   return  $\mathbf{g}_\mathbf{a}$ 
```

**Sigmoid Module** The sigmoid layer has only one input vector  $\mathbf{a}$ . Below  $\sigma$  is the sigmoid applied element-wise, and  $\odot$  is element-wise multiplication s.t.  $\mathbf{u} \odot \mathbf{v} = [u_1 v_1, \dots, u_M v_M]$ .

```
1: procedure SIGMOIDFORWARD( $\mathbf{a}$ )
2:    $\mathbf{b} = \sigma(\mathbf{a})$ 
3:   return  $\mathbf{b}$ 
4: procedure SIGMOIDBACKWARD( $\mathbf{a}, \mathbf{b}, \mathbf{g}_\mathbf{b}$ )
5:    $\mathbf{g}_\mathbf{a} = \mathbf{g}_\mathbf{b} \odot \mathbf{b} \odot (1 - \mathbf{b})$ 
6:   return  $\mathbf{g}_\mathbf{a}$ 
```

**Cross-Entropy Module** The cross-entropy layer has two inputs: a gold one-hot vector  $\mathbf{a}$  and a predicted probability distribution  $\hat{\mathbf{a}}$ . Its output  $b \in \mathbb{R}$  is a scalar. Below  $\div$  is element-wise division. The output  $b$  is not used by CROSSENTROPYBACKWARD, but we pass it in for consistency of form.

```
1: procedure CROSSENTROPYFORWARD( $\mathbf{a}, \hat{\mathbf{a}}$ )
2:    $b = -\mathbf{a}^T \log \hat{\mathbf{a}}$ 
3:   return  $b$ 
4: procedure CROSSENTROPYBACKWARD( $\mathbf{a}, \hat{\mathbf{a}}, b, \mathbf{g}_b$ )
5:    $\mathbf{g}_\hat{\mathbf{a}} = -\mathbf{g}_b (\mathbf{a} \div \hat{\mathbf{a}})$ 
6:   return  $\mathbf{g}_\hat{\mathbf{a}}$ 
```



# Module-based AutoDiff

## Algorithm 1 Forward Computation

```
1: procedure NNFORWARD(Training example  $(\mathbf{x}, \mathbf{y})$ , Parameters  $\alpha, \beta$ )
2:    $\mathbf{a} = \text{LINEARFORWARD}(\mathbf{x}, \alpha)$ 
3:    $\mathbf{z} = \text{SIGMOIDFORWARD}(\mathbf{a})$ 
4:    $\mathbf{b} = \text{LINEARFORWARD}(\mathbf{z}, \beta)$ 
5:    $\hat{\mathbf{y}} = \text{SOFTMAXFORWARD}(\mathbf{b})$ 
6:    $J = \text{CROSSENTROPYFORWARD}(\mathbf{y}, \hat{\mathbf{y}})$ 
7:    $\mathbf{o} = \text{object}(\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J)$ 
8:   return intermediate quantities  $\mathbf{o}$ 
```

## Algorithm 2 Backpropagation

```
1: procedure NNBACKWARD(Training example  $(\mathbf{x}, \mathbf{y})$ , Parameters  $\alpha, \beta$ , Intermediates  $\mathbf{o}$ )
2:   Place intermediate quantities  $\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J$  in  $\mathbf{o}$  in scope
3:    $g_J = \frac{dJ}{dJ} = 1$  ▷ Base case
4:    $\mathbf{g}_{\hat{\mathbf{y}}} = \text{CROSSENTROPYBACKWARD}(\mathbf{y}, \hat{\mathbf{y}}, J, g_J)$ 
5:    $\mathbf{g}_{\mathbf{b}} = \text{SOFTMAXBACKWARD}(\mathbf{b}, \hat{\mathbf{y}}, \mathbf{g}_{\hat{\mathbf{y}}})$ 
6:    $\mathbf{g}_{\beta}, \mathbf{g}_{\mathbf{z}} = \text{LINEARBACKWARD}(\mathbf{z}, \mathbf{b}, \mathbf{g}_{\mathbf{b}})$ 
7:    $\mathbf{g}_{\mathbf{a}} = \text{SIGMOIDBACKWARD}(\mathbf{a}, \mathbf{z}, \mathbf{g}_{\mathbf{z}})$ 
8:    $\mathbf{g}_{\alpha}, \mathbf{g}_{\mathbf{x}} = \text{LINEARBACKWARD}(\mathbf{x}, \mathbf{a}, \mathbf{g}_{\mathbf{a}})$  ▷ We discard  $\mathbf{g}_{\mathbf{x}}$ 
9:   return parameter gradients  $\mathbf{g}_{\alpha}, \mathbf{g}_{\beta}$ 
```

1. Easy to reuse / adapt for other models
2. Individual layers are easier to optimize
3. Simple to debug: just run a finite-difference check on each layer separately

# Module-based AutoDiff (OOP Version)

## Object-Oriented Implementation:

- Let each module be an object and allow the **control flow** of the program to define the computation graph
- **No longer need to implement NNBackward( $\cdot$ )**, just follow the computation graph in reverse topological order

```
1 class Sigmoid(Module)
2     method forward(a)
3          $b = \sigma(a)$ 
4         return b
5     method backward(a, b, gb)
6          $g_a = g_b \odot b \odot (1 - b)$ 
7         return ga
```

```
1 class Softmax(Module)
2     method forward(a)
3          $b = \text{softmax}(a)$ 
4         return b
5     method backward(a, b, gb)
6          $g_a = g_b^T (\text{diag}(b) - bb^T)$ 
7         return ga
```

```
1 class Linear(Module)
2     method forward(a,  $\omega$ )
3          $b = \omega a$ 
4         return b
5     method backward(a,  $\omega$ , b, gb)
6          $g_\omega = g_b a^T$ 
7          $g_a = \omega^T g_b$ 
8         return g $_\omega$ , ga
```

```
1 class CrossEntropy(Module)
2     method forward(a,  $\hat{a}$ )
3          $b = -a^T \log \hat{a}$ 
4         return b
5     method backward(a,  $\hat{a}$ , b, gb)
6          $g_{\hat{a}} = -g_b (a \div \hat{a})$ 
7         return ga
```

# Module-based AutoDiff (OOP Version)

```
1  class NeuralNetwork(Module):
2
3      method init()
4          lin1_layer = Linear()
5          sig_layer = Sigmoid()
6          lin2_layer = Linear()
7          soft_layer = Softmax()
8          ce_layer = CrossEntropy()
9
10     method forward(Tensor x , Tensor y , Tensor  $\alpha$  , Tensor  $\beta$ )
11         a = lin1_layer.apply_fwd(x,  $\alpha$ )
12         z = sig_layer.apply_fwd(a)
13         b = lin2_layer.apply_fwd(z,  $\beta$ )
14          $\hat{y}$  = soft_layer.apply_fwd(b)
15         J = ce_layer.apply_fwd(y,  $\hat{y}$ )
16         return J.out_tensor
17
18     method backward(Tensor x , Tensor y , Tensor  $\alpha$  , Tensor  $\beta$ )
19         tape_bwd()
20         return lin1_layer.in_gradients[1] , lin2_layer.in_gradients[1]
```

# Module-based AutoDiff (OOP Version)

```
1 global tape = stack()
2
3 class Module:
4
5     method init()
6         out_tensor = null
7         out_gradient = 1
8
9     method apply_fwd(List in_modules)
10         in_tensors = [x.out_tensor for x in in_modules]
11         out_tensor = forward(in_tensors)
12         tape.push(self)
13         return self
14
15     method apply_bwd():
16         in_gradients = backward(in_tensors, out_tensor, out_gradient)
17         for i in 1, ..., len(in_modules):
18             in_modules[i].out_gradient += in_gradients[i]
19         return self
20
21 function tape_bwd():
22     while len(tape) > 0
23         m = tape.pop()
24         m.apply_bwd()
```

# Key Takeaways

- Language models fit joint probability distributions to sequences of tokens
  - Tokenization and embedding to generate dense vector representations of texts
  - Can be sampled from to generate text