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

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

2. Learn or approximate a joint probability distribution over sequences

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

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

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

Tokenization and Embedding

1. Convert raw text into sequence data

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

- 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

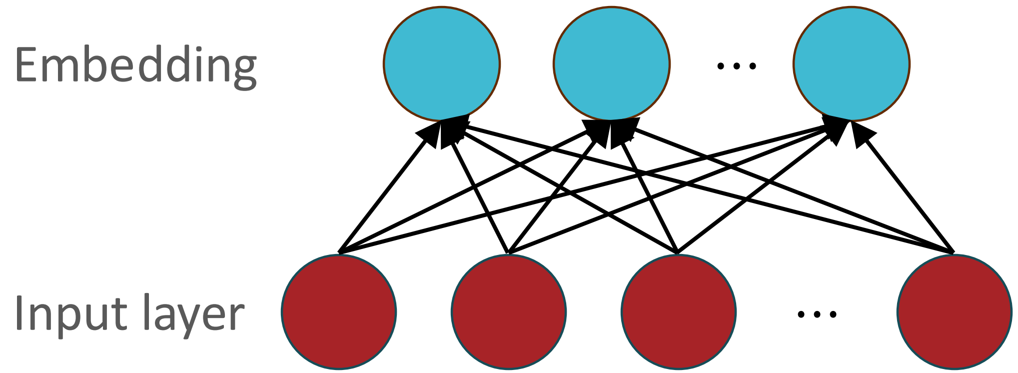
[“henry”, “is”, “giv”, “##in”, “ ‘ ”, “a”, “lect” “##re”, “on”, “language”, “model”, “#s”]

  - 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

$$[0.24, -1.87, \ldots, 0.99] \in \mathbb{R}^D$$

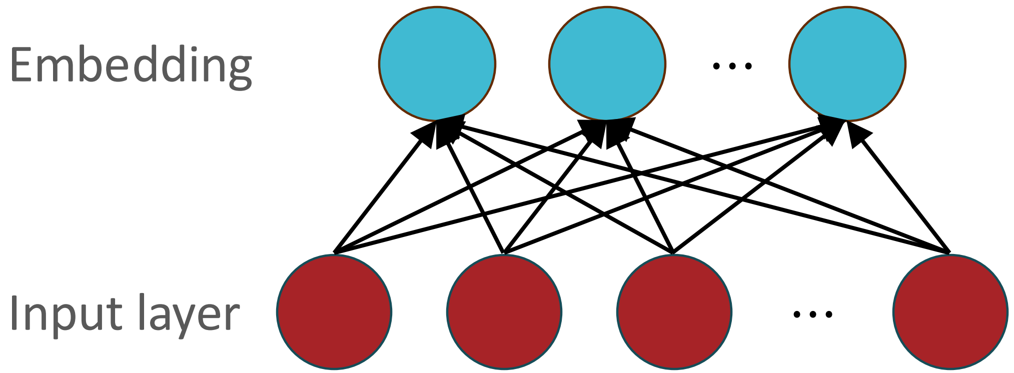


- Example: "is" $\rightarrow$ $[0, \quad 0, \quad 1, \quad \ldots, \quad 0] \in \mathbb{R}^{|V|}$

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

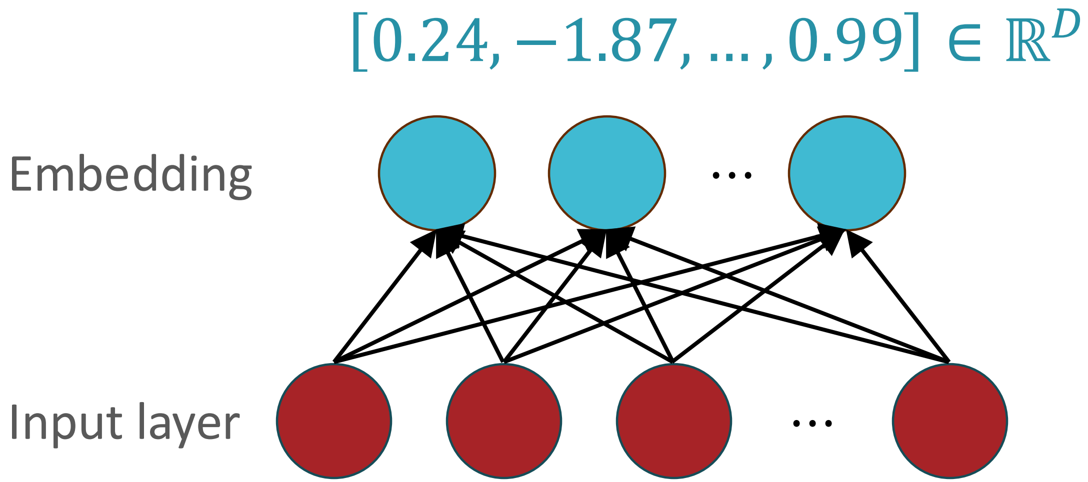
$[0.24, -1.87, \ldots, 0.99] \in \mathbb{R}^D$



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

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

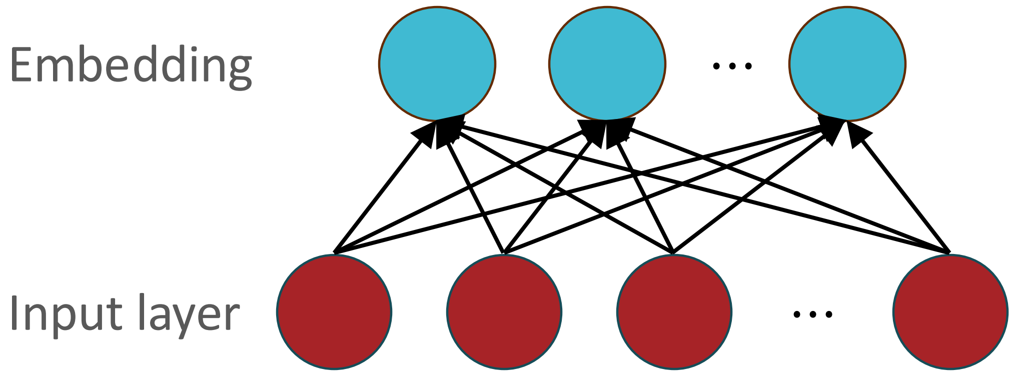


- Example: "is" $\rightarrow$ $[0, \quad 0, \quad 1, \quad \ldots, \quad 0] \in \mathbb{R}^{|V|}$
- Idea: use a pretrained embedding e.g., word2vec or GloVe
  - Requires you to use the same vocabulary/tokenization

Embedding Layer

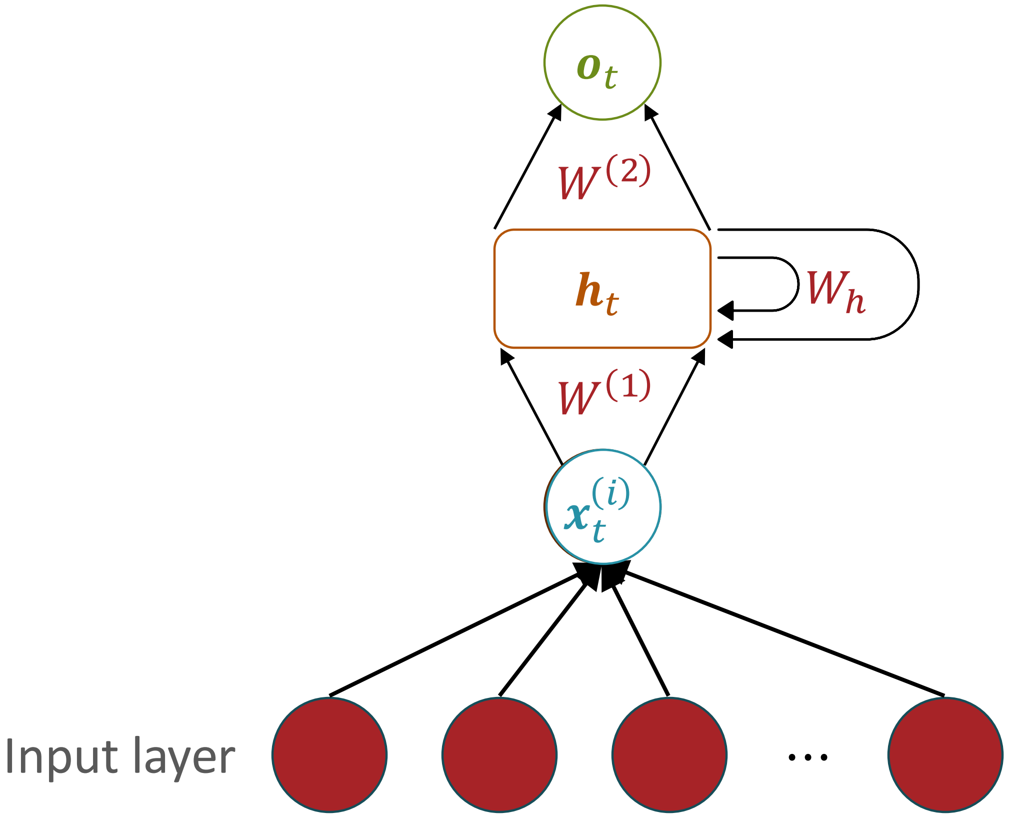
- Given a vocabulary V with $|V|$ tokens, learn an embedding by training a 1-layer, fully-connected feed-forward NN that takes one-hot encoded vectors as input

$$[0.24, -1.87, \dots, 0.99] \in \mathbb{R}^D$$



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

Embedding Layer



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

Language Models

1. Convert raw text into sequence data

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

2. Learn or approximate a joint probability distribution over sequences

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

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

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

Language Models

1. Convert raw text into sequence data

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

2. Learn or approximate a joint probability distribution over sequences

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

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

$$P\left(\boldsymbol{x}^{(i)}\right) = P\left(\boldsymbol{x}_1^{(i)}\right)$$
$$* P\left(\boldsymbol{x}_2^{(i)} \mid \boldsymbol{x}_1^{(i)}\right)$$
$$\vdots$$
$$* P\left(\boldsymbol{x}_{T_i}^{(i)} \mid \boldsymbol{x}_{T_i-1}^{(i)}, \dots, \boldsymbol{x}_1^{(i)}\right)$$

RNN Language Models

1. Convert raw text into sequence data

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

2. Learn or approximate a joint probability distribution over sequences

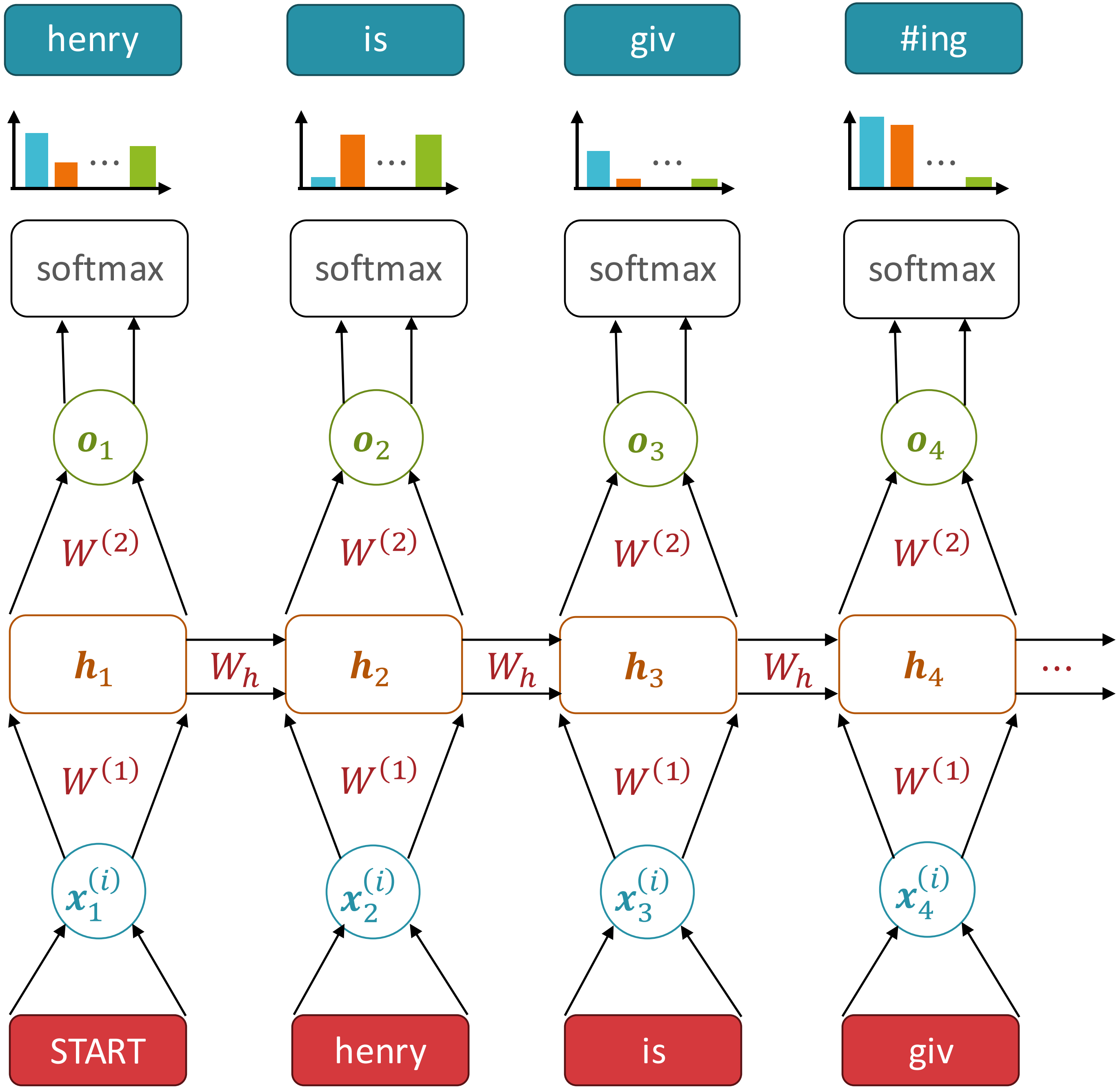
$$P\left(\boldsymbol{x}^{(i)}\right) = P\left(\boldsymbol{x}_1^{(i)}, \dots, \boldsymbol{x}_{T_i}^{(i)}\right)$$

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

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

RNN Language Models: Training

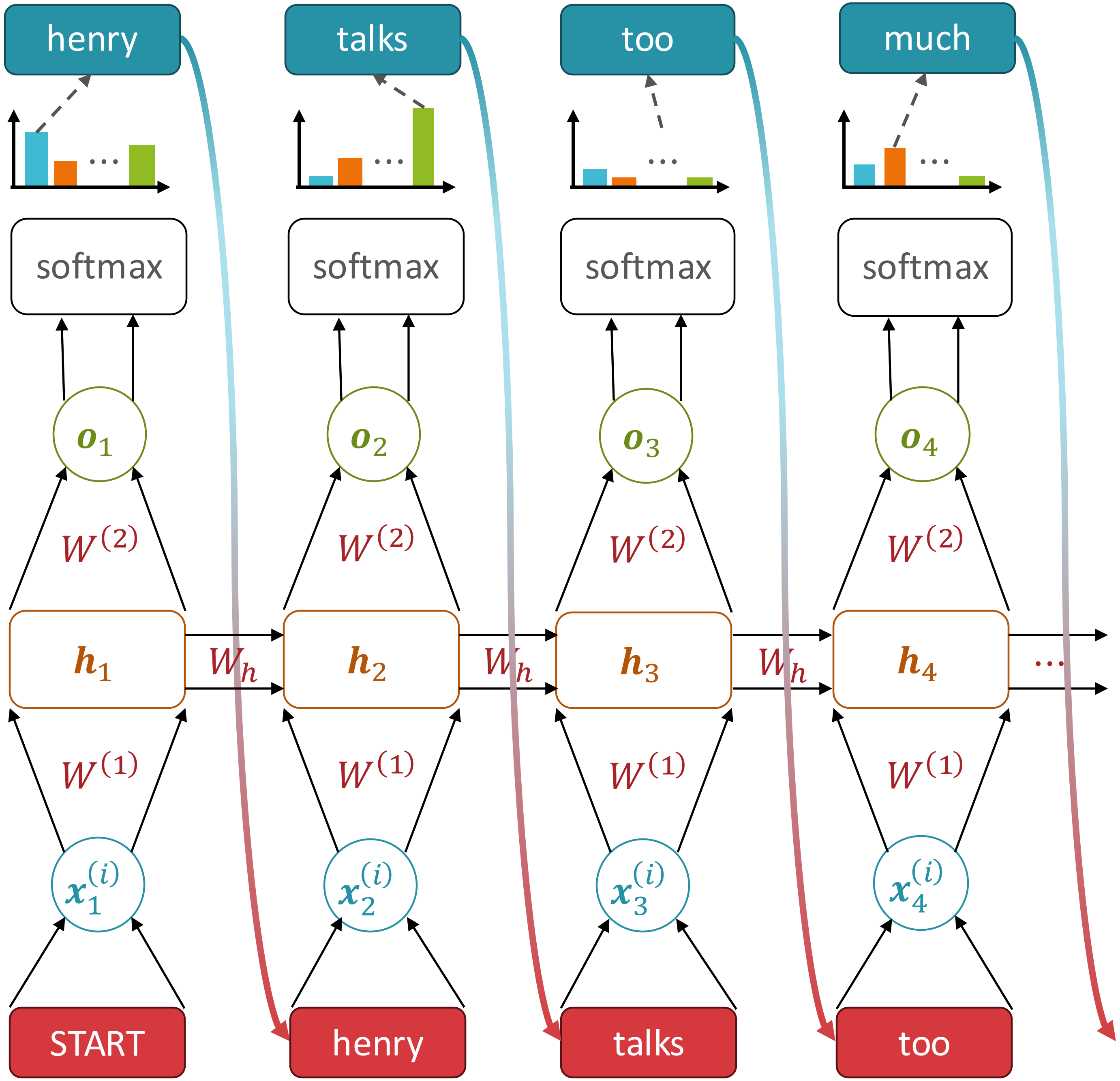
Target sequence (try to predict the next word)



Input sequence

RNN Language Model: Sampling

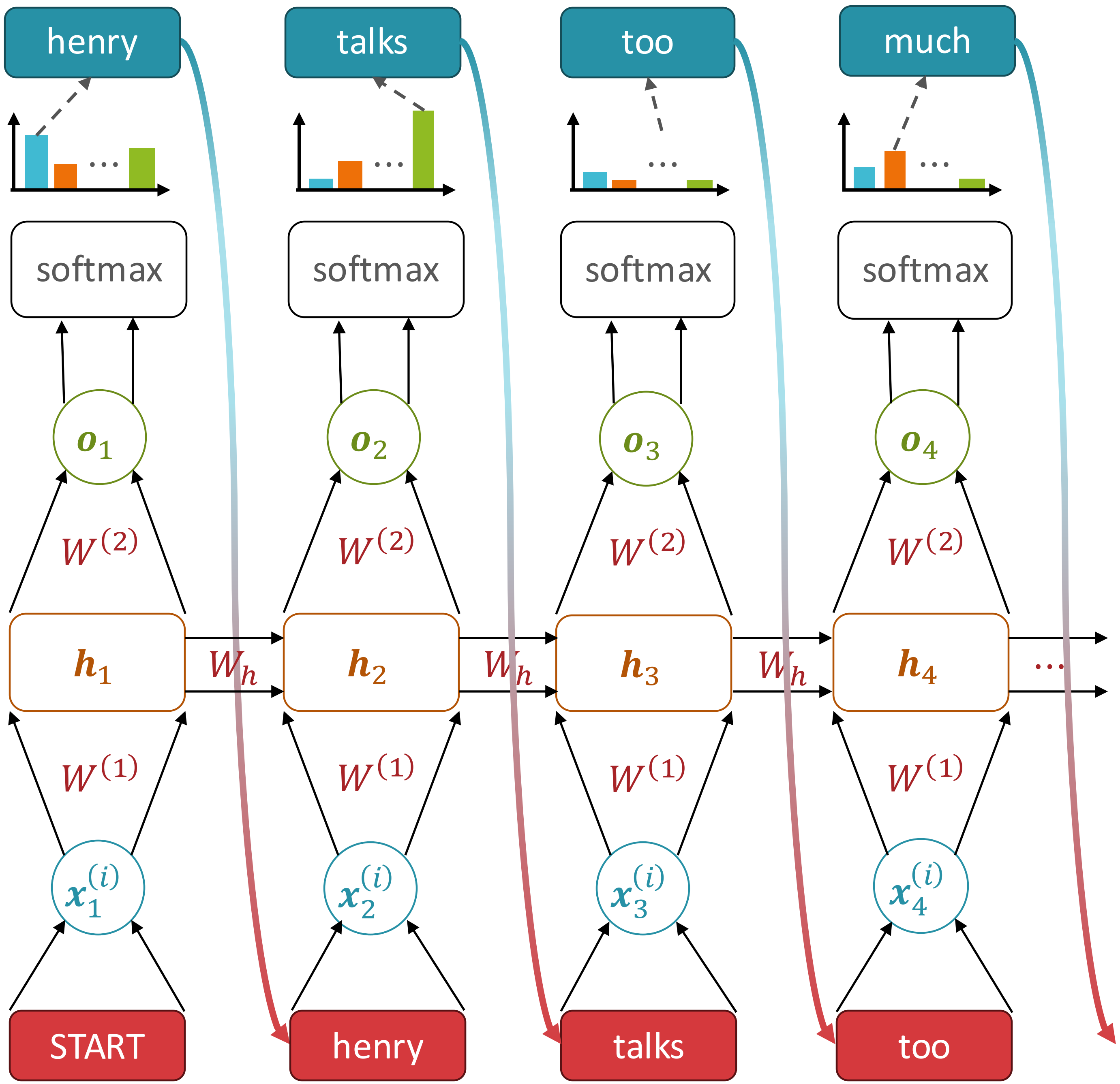
Generated sequence (use each token as the input to the next timestep)



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

procedure NNFORWARD(Training example $(\mathbf{x}, \mathbf{y})$, Params $\boldsymbol{\alpha}, \boldsymbol{\beta}$)
$\mathbf{a} = \boldsymbol{\alpha}\mathbf{x}$
$\mathbf{z} = \sigma(\mathbf{a})$
$\mathbf{b} = \boldsymbol{\beta}\mathbf{z}$
$\hat{\mathbf{y}} = \text{softmax}(\mathbf{b})$
$J = -\mathbf{y}^T \log \hat{\mathbf{y}}$
$\mathbf{o} = \texttt{object}(\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J)$
return intermediate quantities $\mathbf{o}$

Algorithm 2 Backpropagation

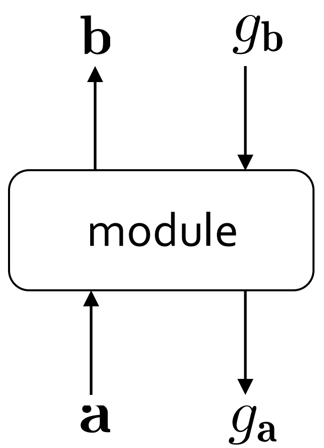
procedure NNBACKWARD(Training example $(\mathbf{x}, \mathbf{y})$, Params $\boldsymbol{\alpha}, \boldsymbol{\beta}$, Intermediates $\mathbf{o}$)
Place intermediate quantities $\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J$ in $\mathbf{o}$ in scope
$\mathbf{g}_{\hat{\mathbf{y}}} = -\mathbf{y} \div \hat{\mathbf{y}}$
$\mathbf{g}_{\mathbf{b}} = \mathbf{g}_{\hat{\mathbf{y}}}^T \left(\text{diag}(\hat{\mathbf{y}}) - \hat{\mathbf{y}}\hat{\mathbf{y}}^T\right)$
$\mathbf{g}_{\boldsymbol{\beta}} = \mathbf{g}_{\mathbf{b}}^T \mathbf{z}^T$
$\mathbf{g}_{\mathbf{z}} = \boldsymbol{\beta}^T \mathbf{g}_{\mathbf{b}}^T$
$\mathbf{g}_{\mathbf{a}} = \mathbf{g}_{\mathbf{z}} \odot \mathbf{z} \odot (1 - \mathbf{z})$
$\mathbf{g}_{\boldsymbol{\alpha}} = \mathbf{g}_{\mathbf{a}}\mathbf{x}^T$
return parameter gradients $\mathbf{g}_{\boldsymbol{\alpha}}, \mathbf{g}_{\boldsymbol{\beta}}$

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.

```
procedure LINEARFORWARD(a, ω)
    b = ωa
    return b
procedure LINEARBACKWARD(a, ω, b, g_b)
    g_ω = g_b a^T
    g_a = ω^T g_b
    return g_ω, g_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 diag$(\mathbf{v})$ returns a $D \times D$ diagonal matrix whose diagonal entries are $v_1, v_2, \ldots, v_D$ and whose non-diagonal entries are zero.

```
procedure SOFTMAXFORWARD(a)
    b = softmax(a)
    return b
procedure SOFTMAXBACKWARD(a, b, g_b)
    g_a = g_b^T (diag(b) − bb^T)
    return g_a
```

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

```
procedure SIGMOIDFORWARD(a)
    b = σ(a)
    return b
procedure SIGMOIDBACKWARD(a, b, g_b)
    g_a = g_b ⊙ b ⊙ (1 − b)
    return g_a
```

Cross-Entropy Module The cross-entropy layer has two inputs: a gold one-hot vector $\mathbf{a}$ and a predicted probability distribution $\hat{\mathbf{a}}$. It's 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.

```
procedure CROSSENTROPYFORWARD(a, â)
    b = −a^T log â
    return b
procedure CROSSENTROPYBACKWARD(a, â, b, g_b)
    g_â = −g_b (a ÷ â)
    return g_a
```

Module-based AutoDiff

Algorithm 1 Forward Computation

procedure NNFORWARD(Training example $(\mathbf{x}, \mathbf{y})$, Parameters $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$)
- $\mathbf{a} = \text{LINEARFORWARD}(\mathbf{x}, \boldsymbol{\alpha})$
- $\mathbf{z} = \text{SIGMOIDFORWARD}(\mathbf{a})$
- $\mathbf{b} = \text{LINEARFORWARD}(\mathbf{z}, \boldsymbol{\beta})$
- $\hat{\mathbf{y}} = \text{SOFTMAXFORWARD}(\mathbf{b})$
- $J = \text{CROSSENTROPYFORWARD}(\mathbf{y}, \hat{\mathbf{y}})$
- $\mathbf{o} = \texttt{object}(\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J)$
- **return** intermediate quantities $\mathbf{o}$

Algorithm 2 Backpropagation

procedure NNBACKWARD(Training example $(\mathbf{x}, \mathbf{y})$, Parameters $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, Intermediates $\mathbf{o}$)
- Place intermediate quantities $\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J$ in $\mathbf{o}$ in scope
- $g_J = \frac{dJ}{dJ} = 1$ ▷ Base case
- $\mathbf{g}_{\hat{\mathbf{y}}} = \text{CROSSENTROPYBACKWARD}(\mathbf{y}, \hat{\mathbf{y}}, J, g_J)$
- $\mathbf{g}_{\mathbf{b}} = \text{SOFTMAXBACKWARD}(\mathbf{b}, \hat{\mathbf{y}}, \mathbf{g}_{\hat{\mathbf{y}}})$
- $\mathbf{g}_{\boldsymbol{\beta}}, \mathbf{g}_{\mathbf{z}} = \text{LINEARBACKWARD}(\mathbf{z}, \mathbf{b}, \mathbf{g}_{\mathbf{b}})$
- $\mathbf{g}_{\mathbf{a}} = \text{SIGMOIDBACKWARD}(\mathbf{a}, \mathbf{z}, \mathbf{g}_{\mathbf{z}})$
- $\mathbf{g}_{\boldsymbol{\alpha}}, \mathbf{g}_{\mathbf{x}} = \text{LINEARBACKWARD}(\mathbf{x}, \mathbf{a}, \mathbf{g}_{\mathbf{a}})$ ▷ We discard $\mathbf{g}_{\mathbf{x}}$
- **return** parameter gradients $\mathbf{g}_{\boldsymbol{\alpha}}, \mathbf{g}_{\boldsymbol{\beta}}$

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(·)**, just follow the computation graph in reverse topological order

```
class Sigmoid(Module)
    method forward(a)
        b = σ(a)
        return b
    method backward(a, b, g_b)
        g_a = g_b ⊙ b ⊙ (1 − b)
        return g_a
```

```
class Softmax(Module)
    method forward(a)
        b = softmax(a)
        return b
    method backward(a, b, g_b)
        g_a = g_b^T (diag(b) − bb^T)
        return g_a
```

```
class Linear(Module)
    method forward(a, ω)
        b = ωa
        return b
    method backward(a, ω, b, g_b)
        g_ω = g_b a^T
        g_a = ω^T g_b
        return g_ω, g_a
```

```
class CrossEntropy(Module)
    method forward(a, â)
        b = −a^T log â
        return b
    method backward(a, â, b, g_b)
        g_â = −g_b(a ÷ â)
        return g_a
```

Module-based AutoDiff (OOP Version)

```
class NeuralNetwork(Module):

    method init()
        lin1_layer = Linear()
        sig_layer = Sigmoid()
        lin2_layer = Linear()
        soft_layer = Softmax()
        ce_layer = CrossEntropy()

    method forward(Tensor x, Tensor y, Tensor α, Tensor β)
        a = lin1_layer.apply_fwd(x, α)
        z = sig_layer.apply_fwd(a)
        b = lin2_layer.apply_fwd(z, β)
        ŷ = soft_layer.apply_fwd(b)
        J = ce_layer.apply_fwd(y, ŷ)
        return J.out_tensor

    method backward(Tensor x, Tensor y, Tensor α, Tensor β)
        tape_bwd()
        return lin1_layer.in_gradients[1], lin2_layer.in_gradients[1]
```

Module-based AutoDiff (OOP Version)

```
global tape = stack()

class Module:

    method init()
        out_tensor = null
        out_gradient = 1

    method apply_fwd(List in_modules)
        in_tensors = [x.out_tensor for x in in_modules]
        out_tensor = forward(in_tensors)
        tape.push(self)
        return self

    method apply_bwd():
        in_gradients = backward(in_tensors, out_tensor, out_gradient)
        for i in 1,..., len(in_modules):
            in_modules[i].out_gradient += in_gradients[i]
        return self

function tape_bwd():
    while len(tape) > 0
        m = tape.pop()
        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