UNIVERSITY OF GEORGIA

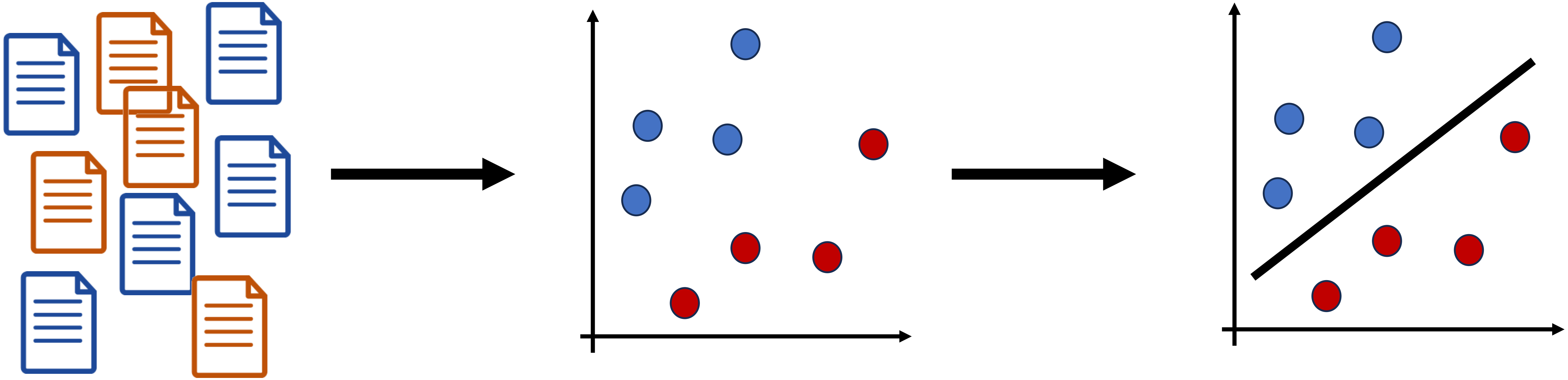
CSCI 4360/6360 Data Science II

Recurrent Neural Networks

Ninghao Liu
Assistant Professor
School of Computing
University of Georgia

In our previous lectures…

- The machine learning pipeline for text data.



- It is challenging to design feature vectors for text documents.
  - Documents can have arbitrary lengths.
  - How to encode semantic meanings?

In our previous lectures…

- We know how to obtain **vector** representations for words.
  - A word is represented by its **embedding**.
  - A word embedding encodes the word’s semantic meaning.

  - A document contains multiple words.
  - So <u>how to represent the document</u> with an embedding vector?

Outline

- **Motivation**
  - High-level idea behind the new model architecture.

- **Major components of RNNs.**

- **RNN training.**
  - Supervised training
  - Unsupervised training

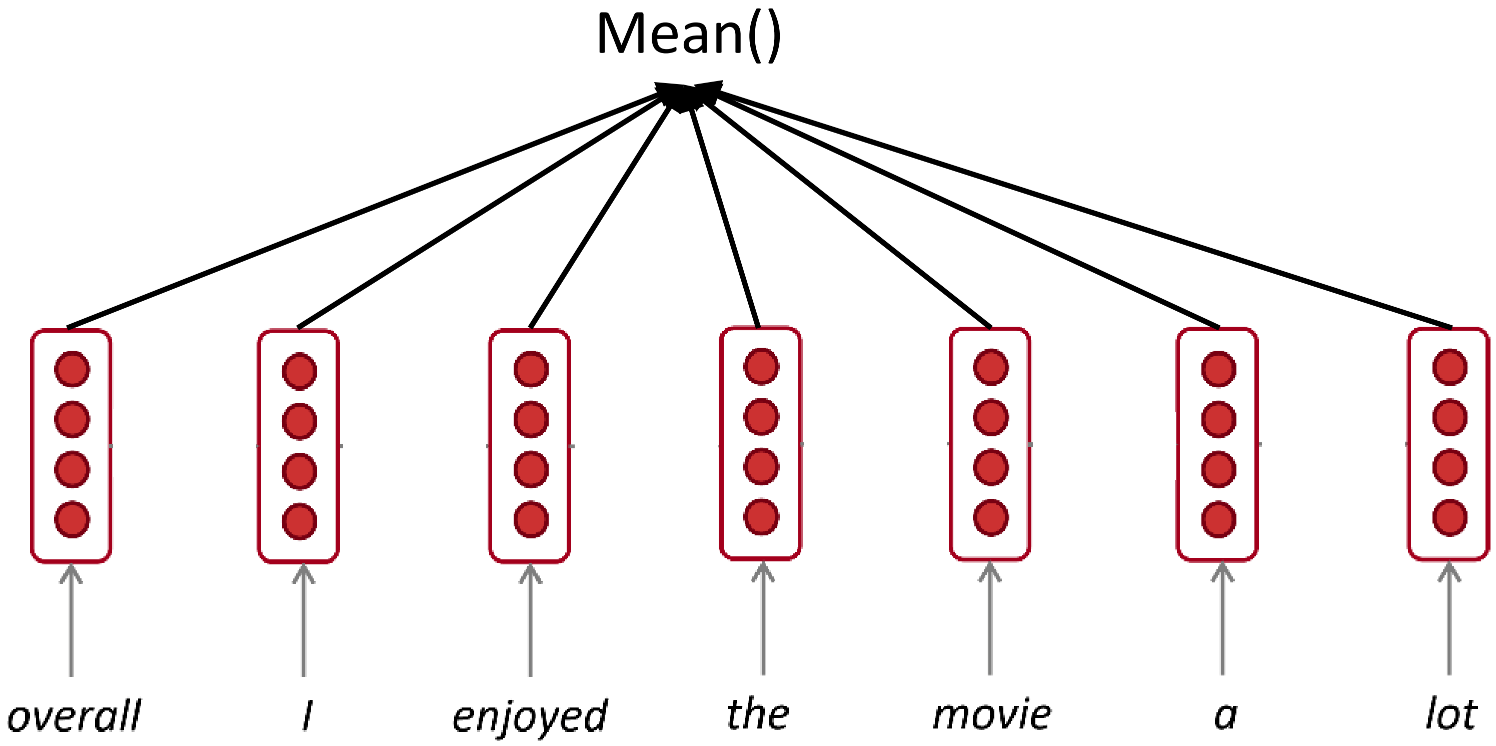
Outline

- **Motivation**
  - High-level idea behind the new model architecture.
- **Major components of RNNs.**
- **RNN training.**
  - Supervised training
  - Unsupervised training

Motivation

A native solution:

- Compute the average of word embeddings to represent the whole document.



Motivation

Challenge.

- Multiple words could be combined to express new ideas.
  - E.g., “I like the movie”, “I don’t like the movie”.
    - **Adding** the embeddings of “don’t” and “like” is not a good idea, because “don’t” contains no sentiment.
  - Many other examples
    - “check” + “in” ≠ ”check in”.
  - To understand the logics, we treat sentences as **sequences**, where the order of words matters.

Motivation

Language Model

- The obvious question: how we should model a sequence?

What does it mean?

Example 1: Consider the following continuations of the phrase “It is raining - ”.

1. “It is raining outside”
2. “It is raining banana tree”
3. “It is raining piouw;kcj pwepoiut”

Example 2: See blackboard.

Motivation

Language Model

- The obvious question: how we should model a sequence?

- Let $(x_1, x_2, \ldots, x_T)$ denote a sequence of words.
- By applying basic probability rules:

$$P(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, \ldots, x_{t-1})$$

- For example,

$$\begin{aligned}&P(\text{deep}, \text{learning}, \text{is}, \text{fun})\\ =&P(\text{deep})P(\text{learning} \mid \text{deep})P(\text{is} \mid \text{deep}, \text{learning})P(\text{fun} \mid \text{deep}, \text{learning}, \text{is})\end{aligned}$$

Motivation

Language Model

- The probability is difficult to compute for long sequences:

$$P(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, \ldots, x_{t-1})$$

- Thus, according to the Markov property, the impact of earlier words can be ignored.

$$P(x_1, x_2, x_3, x_4) = P(x_1)P(x_2 \mid x_1)P(x_3 \mid x_2)P(x_4 \mid x_3),$$
$$P(x_1, x_2, x_3, x_4) = P(x_1)P(x_2 \mid x_1)P(x_3 \mid x_1, x_2)P(x_4 \mid x_2, x_3).$$

Motivation

We need a novel model architecture that can

- Process sequential data.
- Store the information of **previous words** at each position.

Recurrent neural networks (RNNs) can handle this.

- But, instead of directly predicting $P(x_t \mid x_1, \ldots, x_{t-1})$, RNN maintain a hidden state h_{t-1}, so that

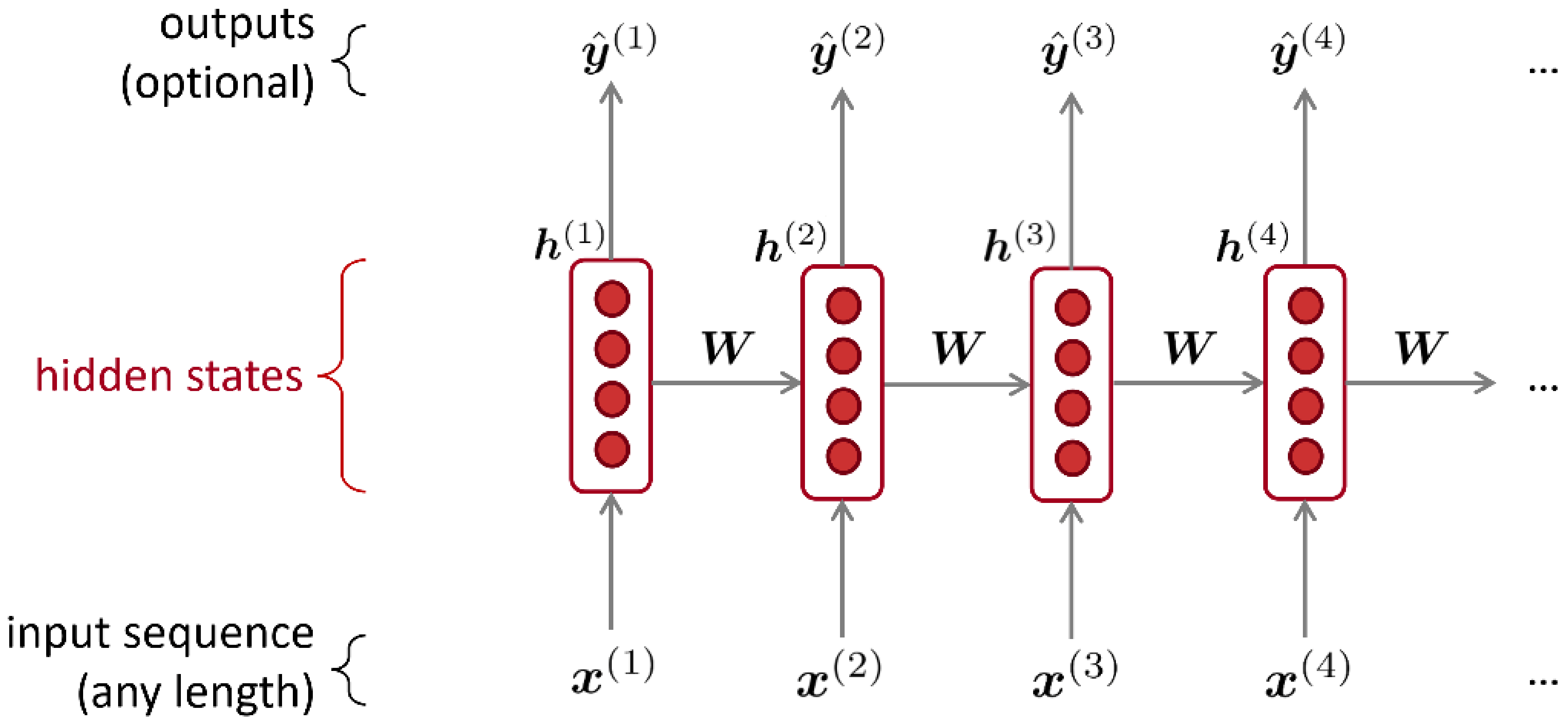
$$P(x_t \mid x_{t-1}, \ldots, x_1) \approx P(x_t \mid h_{t-1})$$

Outline

- **Motivation**
  - High-level idea behind the new model architecture.
- **Major components of RNNs.**
- **RNN training.**
  - Supervised training
  - Unsupervised training

RNN: Architecture (Simplified)

Core idea: Apply the same weights $\boldsymbol{W}$ *repeatedly*



RNN: Architecture

output distribution

$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}\left(\boldsymbol{U}\boldsymbol{h}^{(t)} + \boldsymbol{b}_2\right) \in \mathbb{R}^{|V|}$

hidden states

$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e\boldsymbol{e}^{(t)} + \boldsymbol{b}_1\right)$

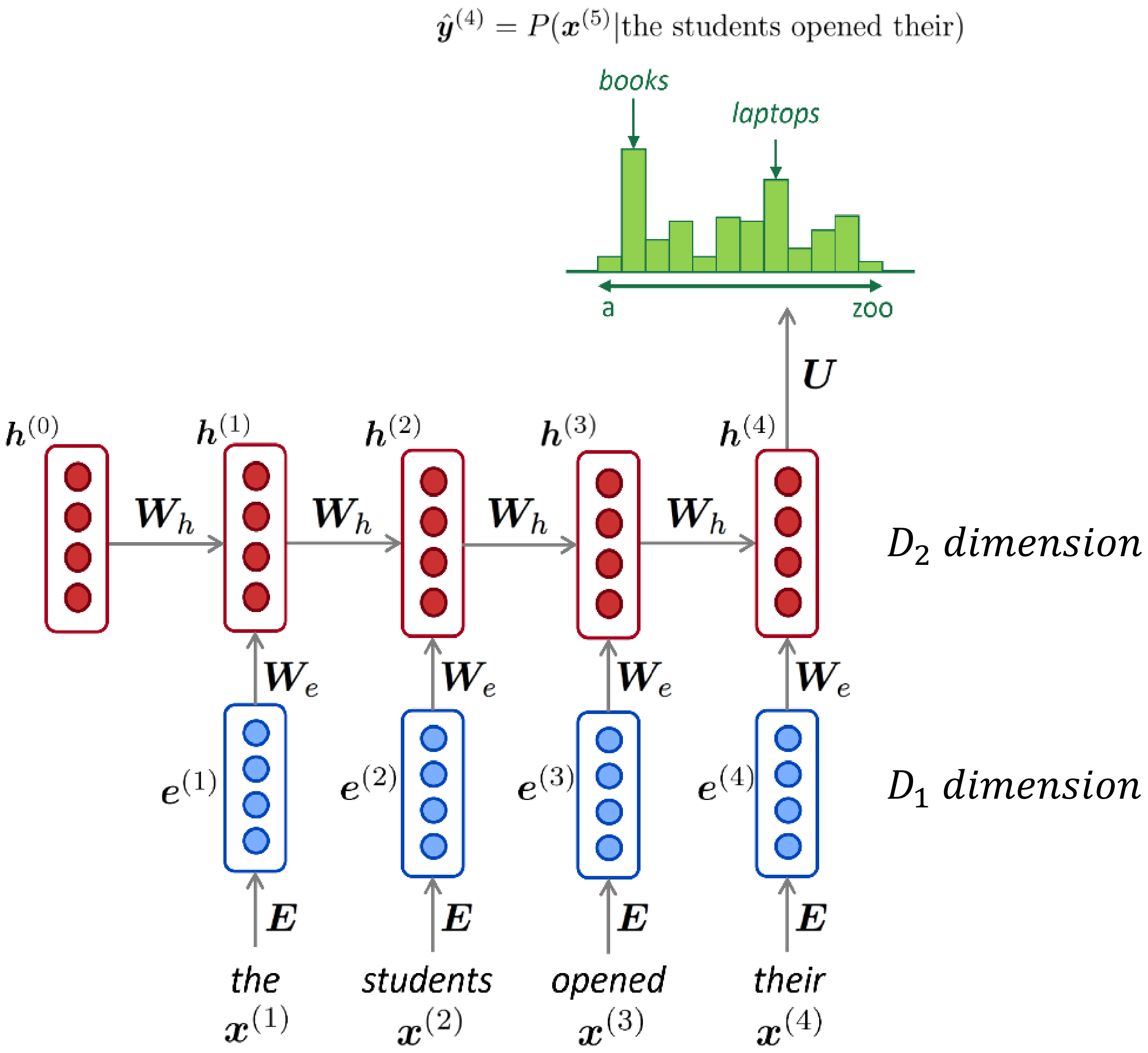
$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

$\boldsymbol{e}^{(t)} = \boldsymbol{E}\boldsymbol{x}^{(t)}$

words / one-hot vectors

$\boldsymbol{x}^{(t)} \in \mathbb{R}^{|V|}$

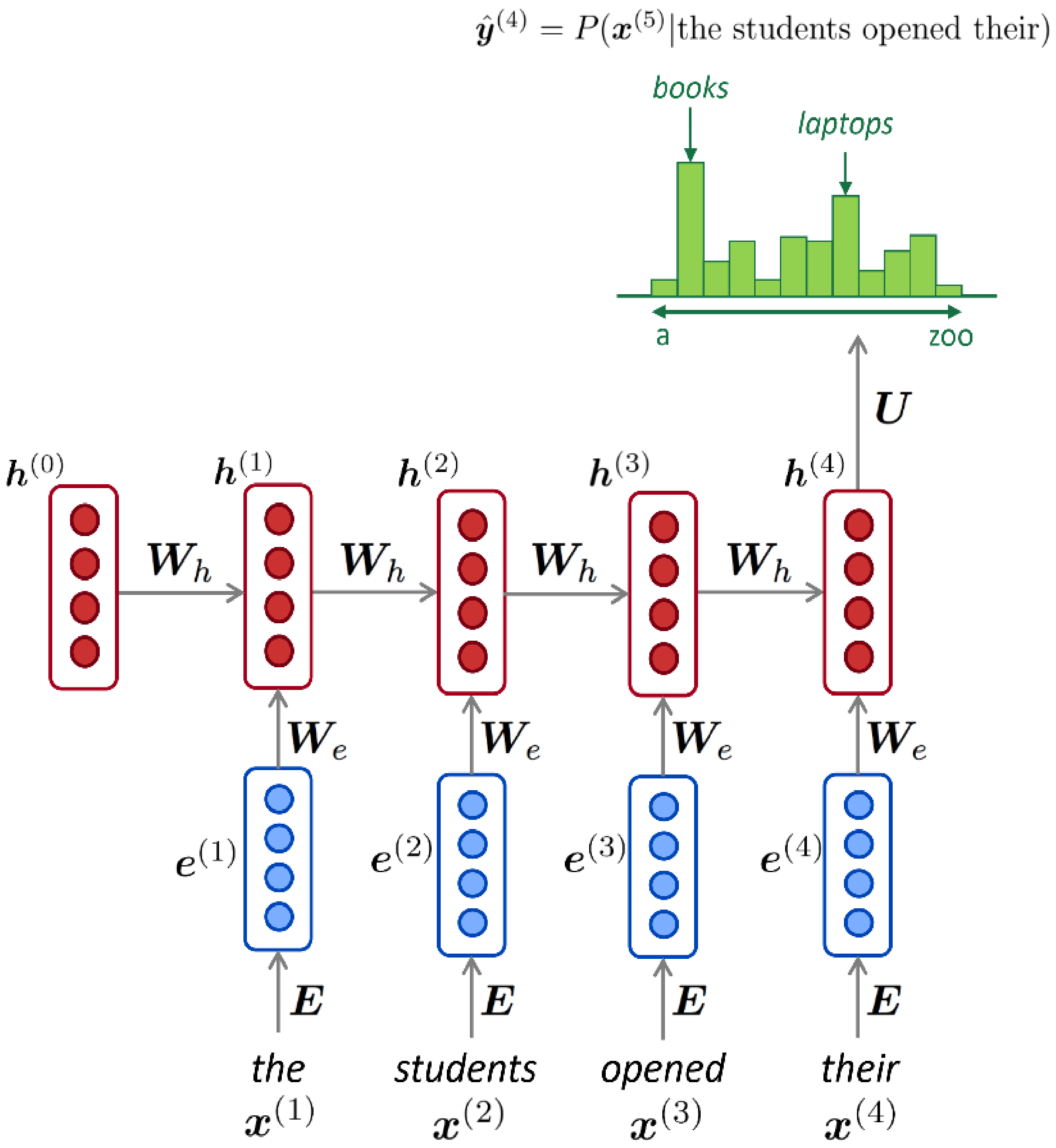


RNN: Architecture

RNN **Advantages**:

- Can process any length input
- Computation for step *t* can (in theory) use information from many steps back

- Model size doesn't increase for longer input
- Same weights applied on every timestep, so there is symmetry in how inputs are processed.

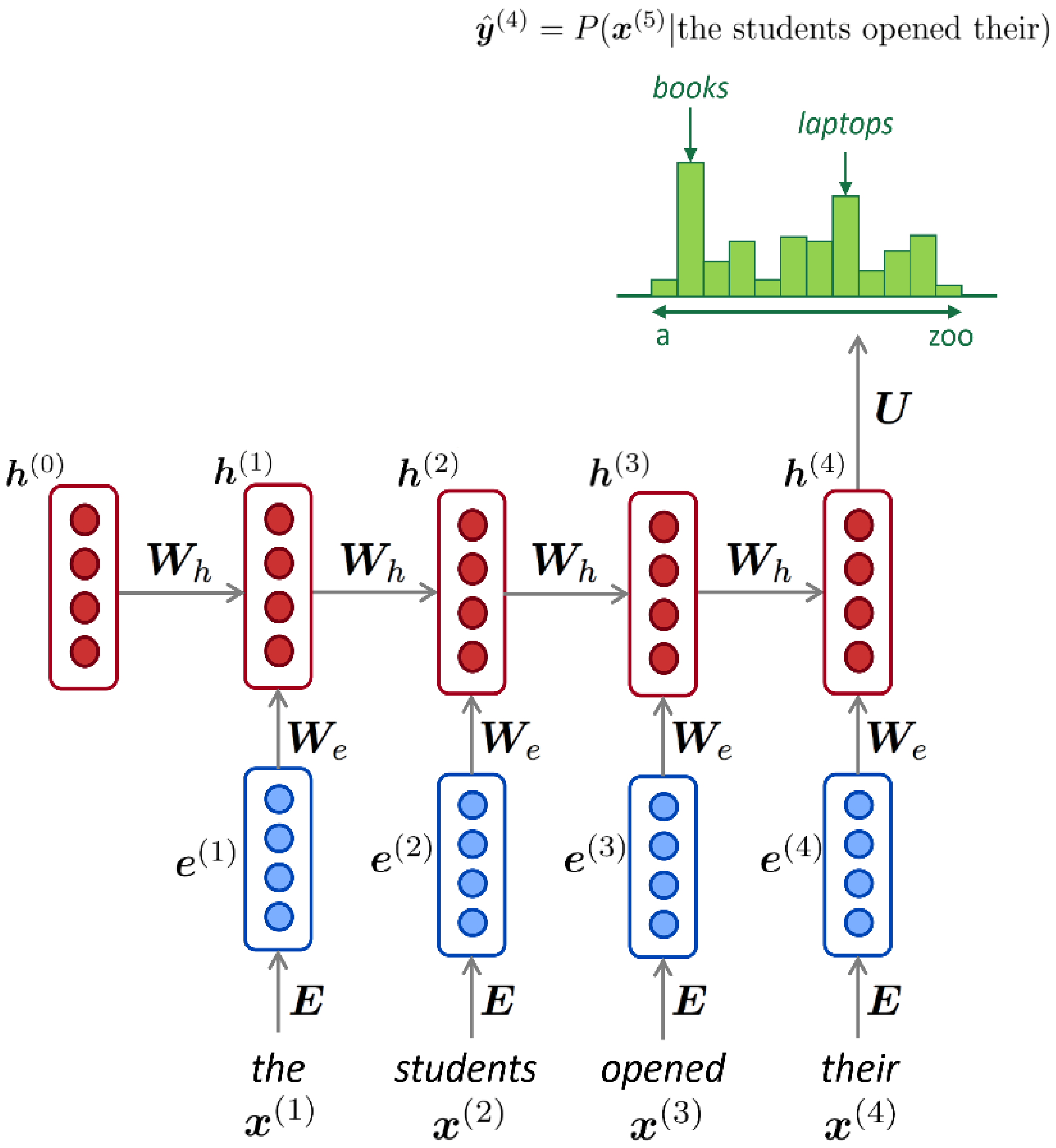


RNN: Architecture

RNN **Disadvantages**:
- Recurrent computation is slow
- In practice, difficult to access information from many steps back

More on these later in the course

RNN is designed to process sequential data, but it cannot handle very-long sequences. 😂



Outline

- **Motivation**
  - High-level idea behind the new model architecture.

- **Major components of RNNs.**

- **RNN training.**
  - Supervised training
  - Unsupervised training

RNN: Training

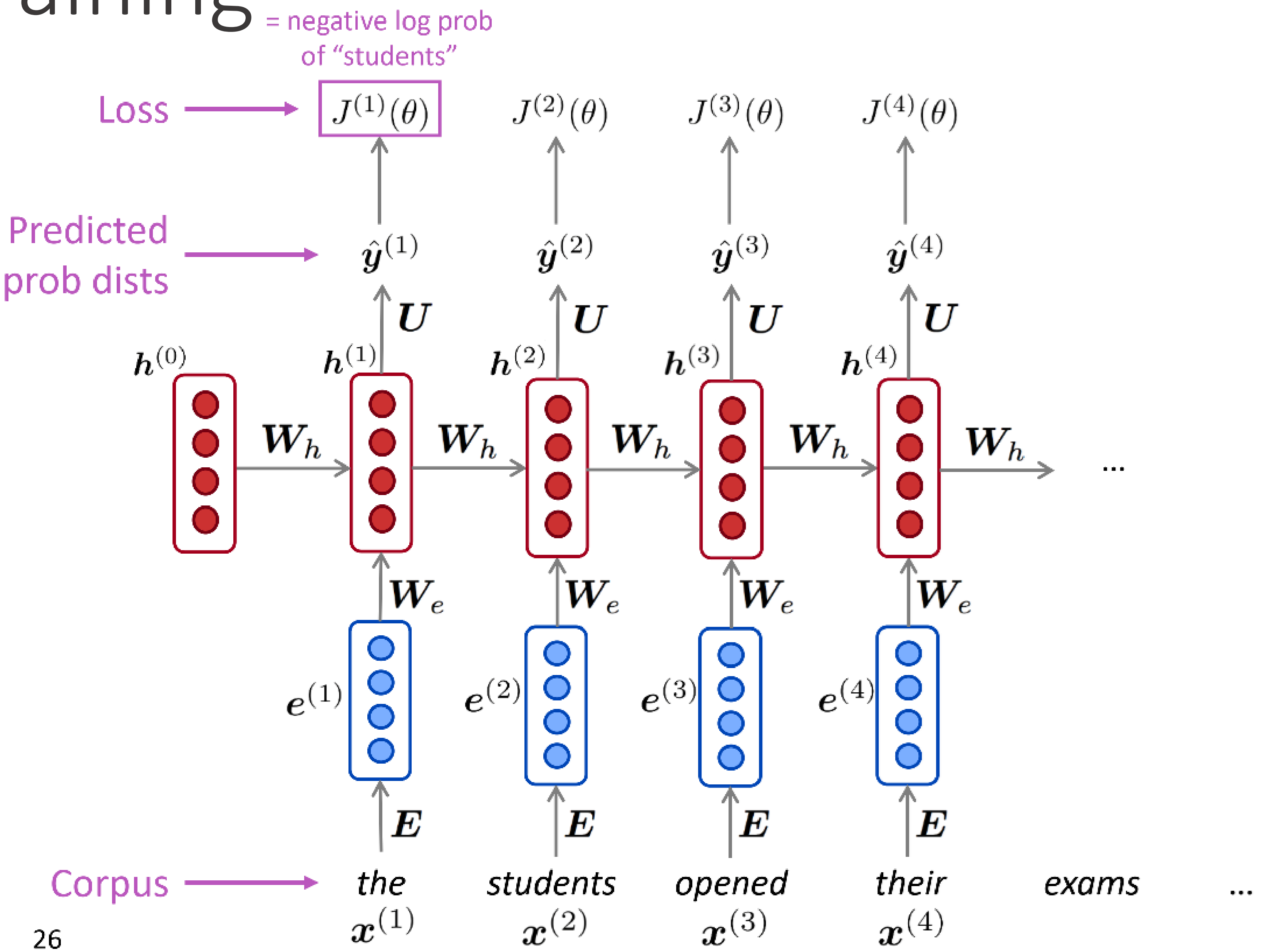
- **Supervised learning**
  - The data is labeled
  - $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \dots, (\boldsymbol{x}_N, y_N)\}$

  - Procedure:
    - Let RNNs read the whole text, and produce an output
    - Apply a loss function to measure the degree of match between model output and the label.
    - Similar to other deep models, such as MLPs.

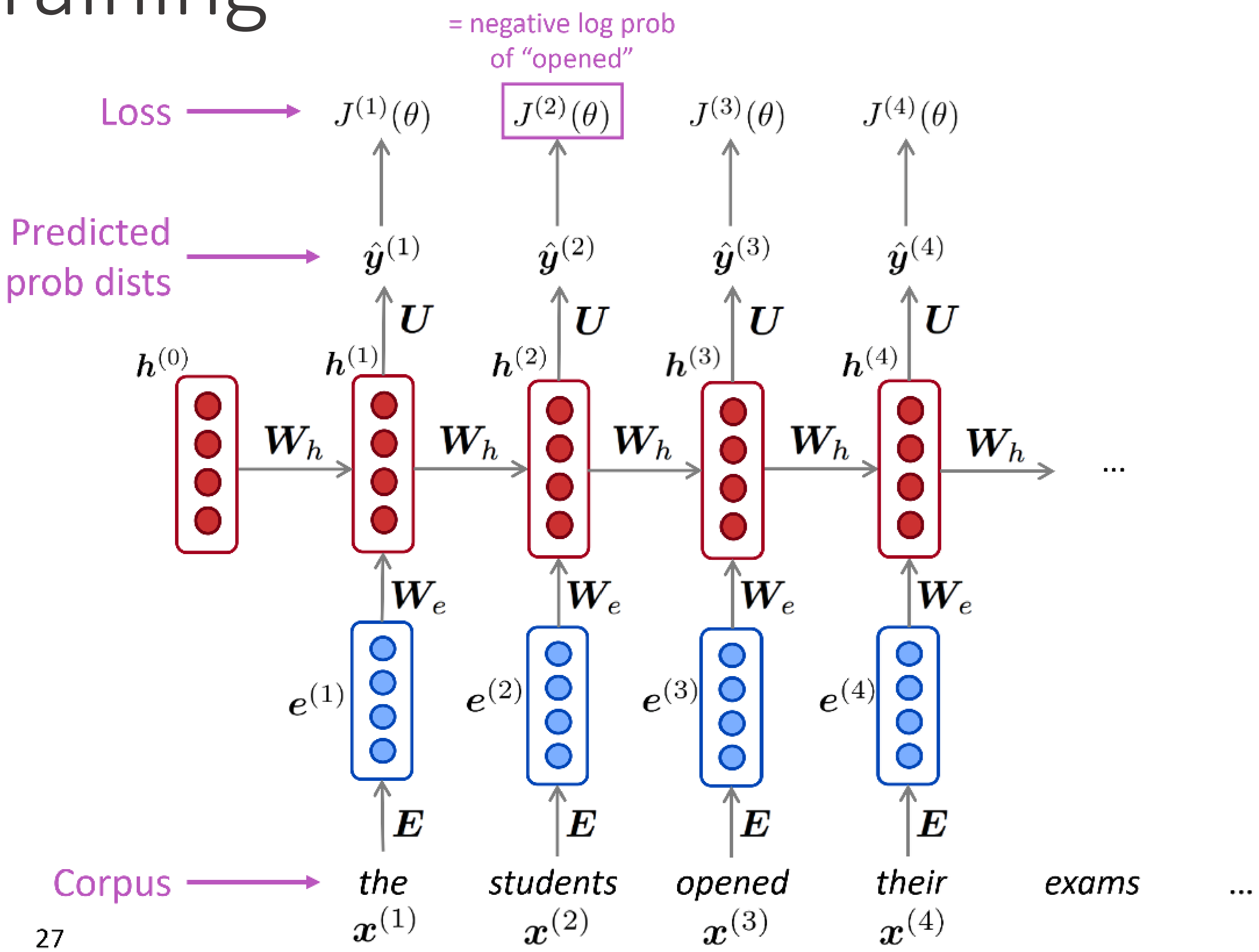
RNN: Training

- RNNs (and many other NLP models) can be trained with **unsupervised learning**.
  - $D = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_N\}$

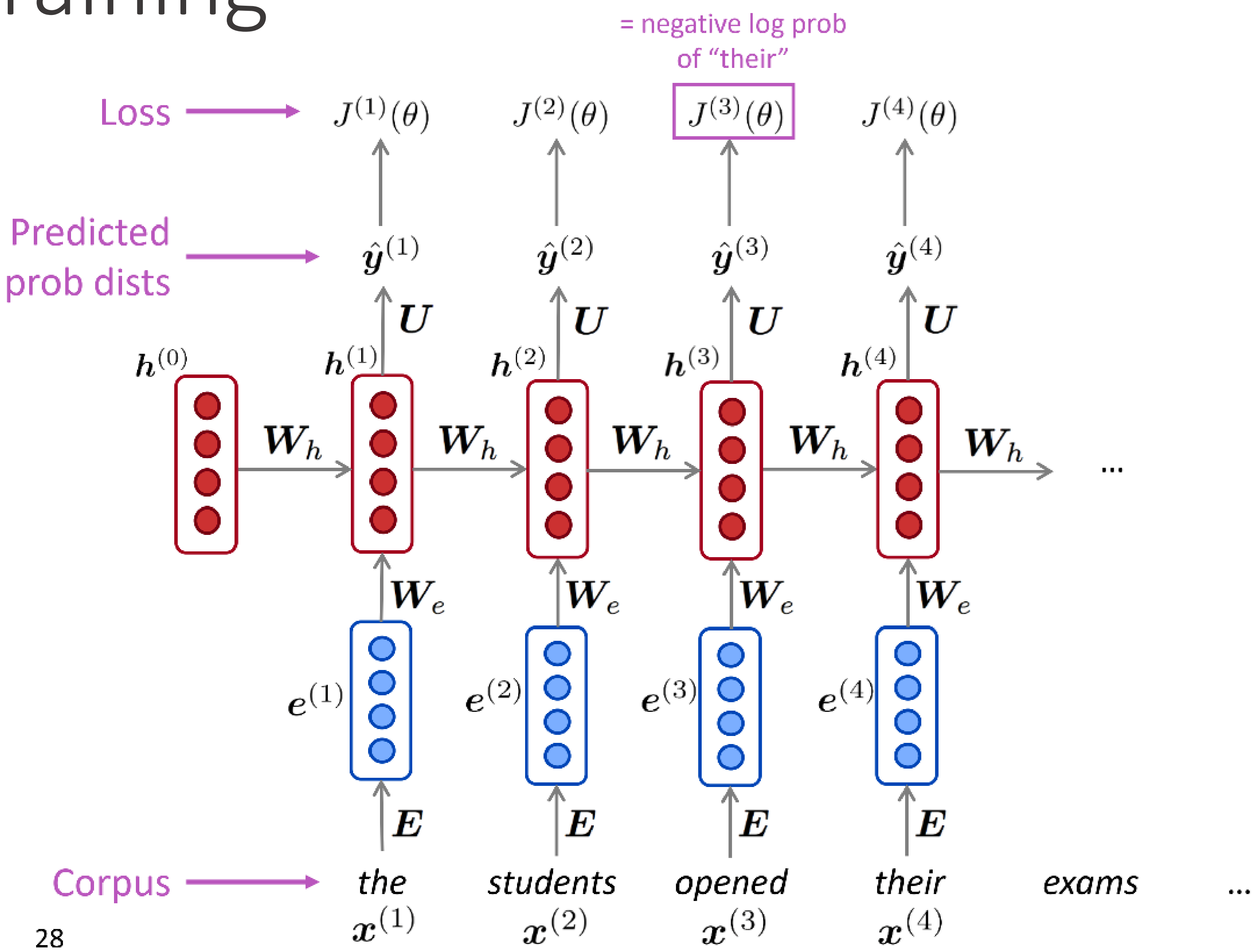
RNN: Training



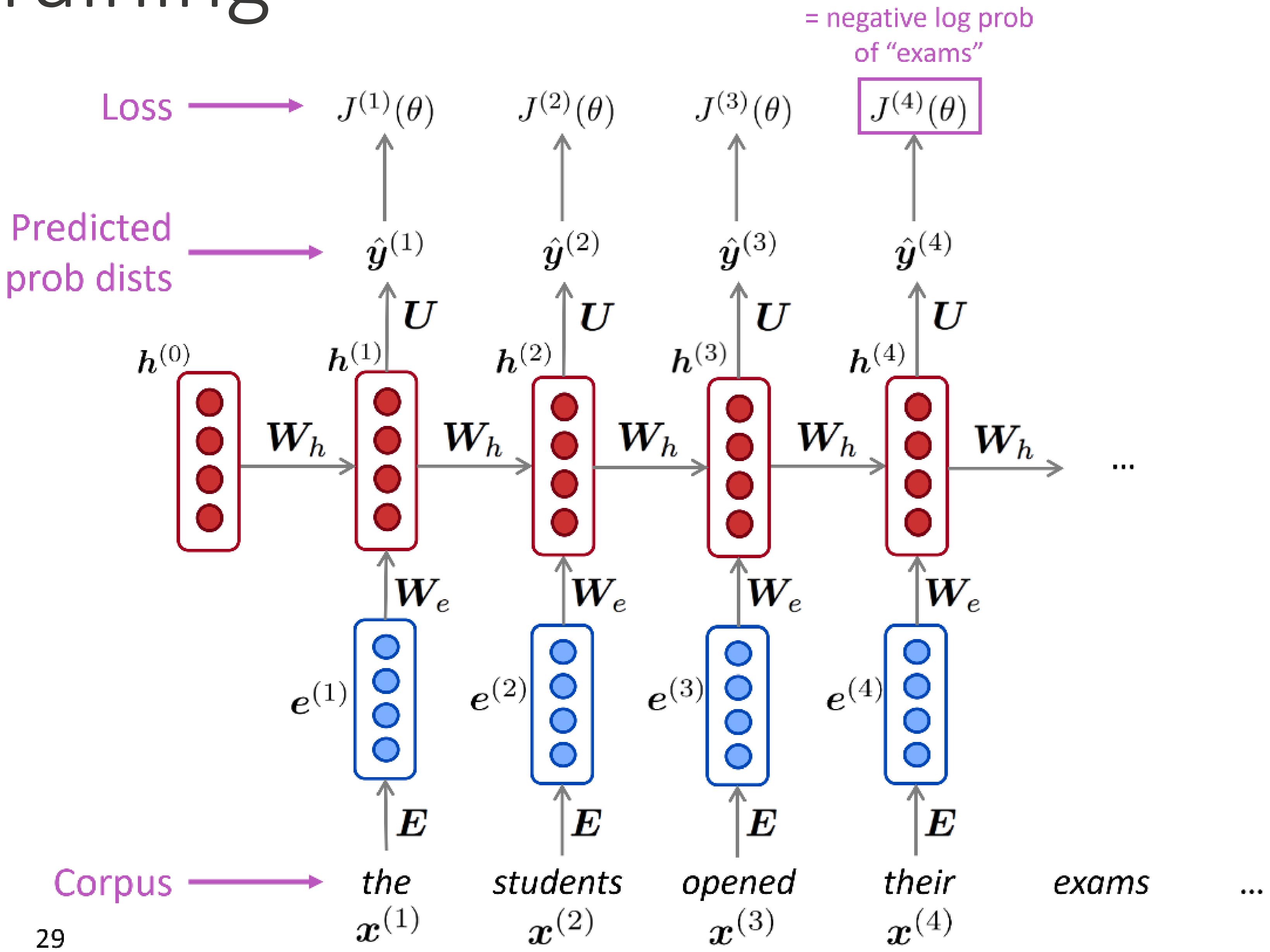
RNN: Training



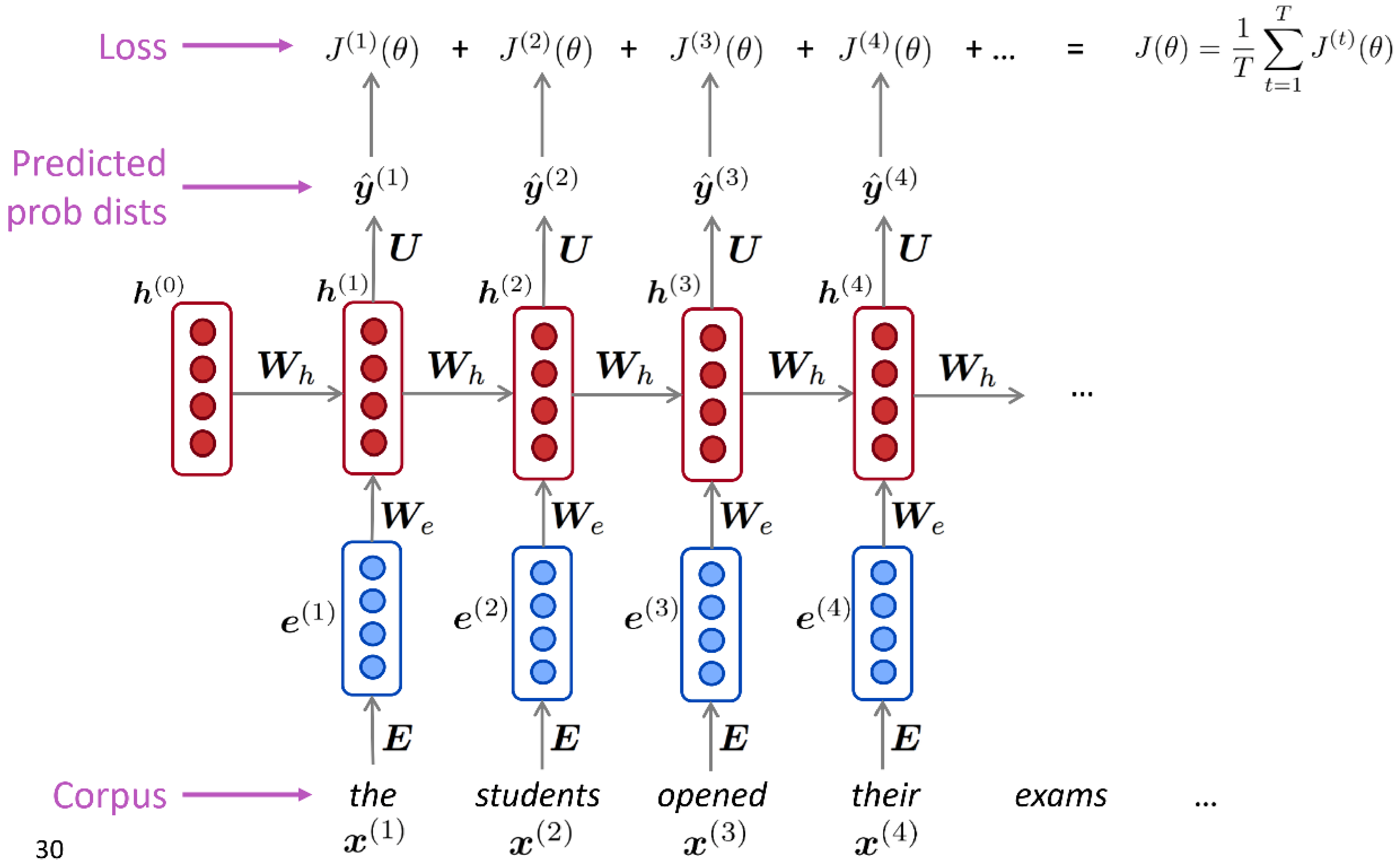
RNN: Training



RNN: Training



RNN: Training



RNN: Training

- However: Computing loss and gradients across entire corpus $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}$ is too expensive!

$$J(\theta) = \frac{1}{T} \sum_{t=1}^{T} J^{(t)}(\theta)$$

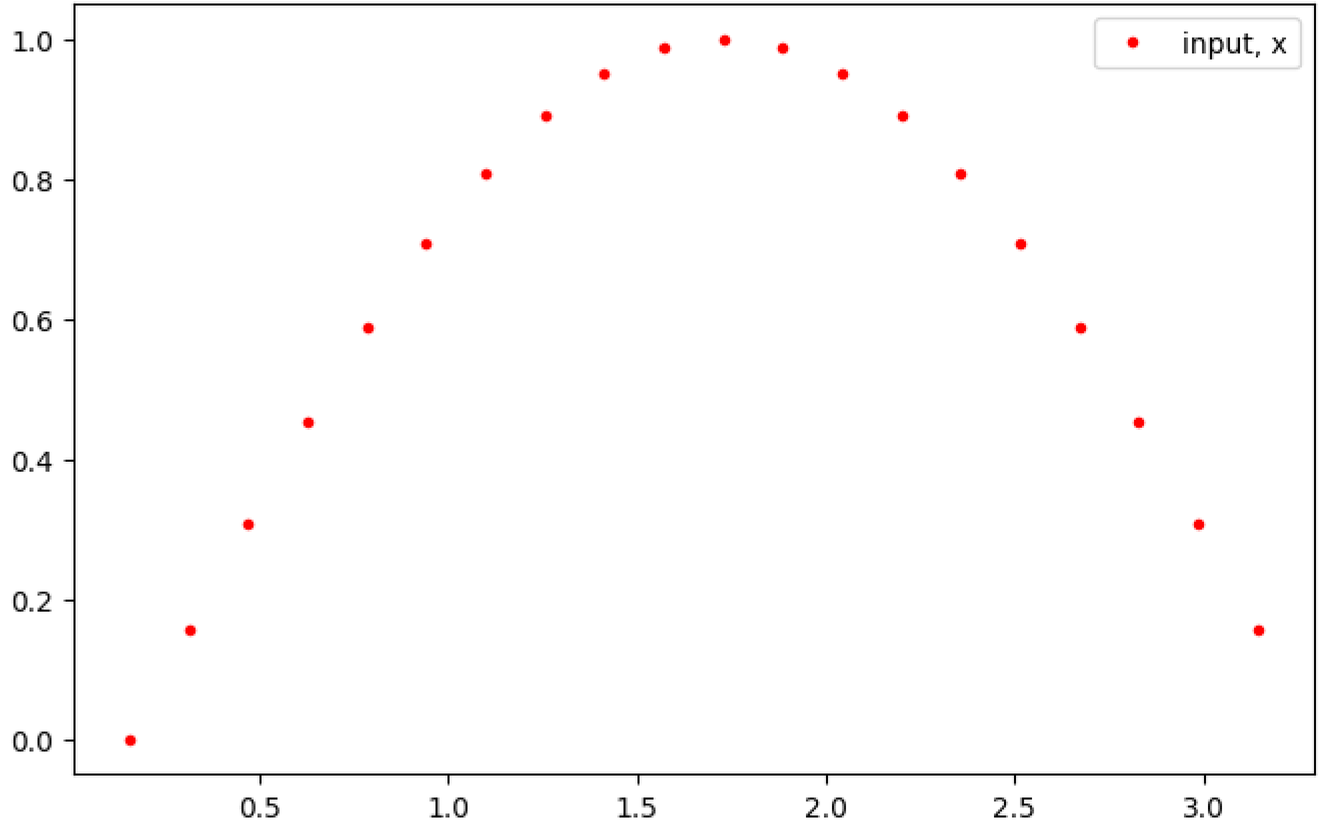
- In practice, consider $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}$ as a sentence (or a document)

- Recall: Stochastic Gradient Descent allows us to compute loss and gradients for small chunk of data, and update.

- Compute loss $J(\theta)$ for a sentence (actually a batch of sentences), compute gradients and update weights. Repeat.

RNN: Training - Example

- **Goal**: Train an RNN model to do time-series prediction. (code available on eLC.)



RNN: Training - Example

- **Step 1:** Import *numpy* and *pytorch* modules

```
import torch
from torch import nn
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

RNN: Training

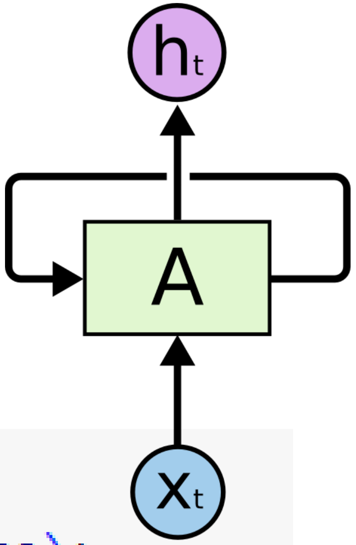
- **Step 2:** Define RNN.

Define model components.

Define data flow.

```python
class RNN(nn.Module):
    def __init__(self, input_size, output_size, hidden_dim, n_layers):
        super(RNN, self).__init__()

        self.hidden_dim=hidden_dim

        # define an RNN with specified parameters
        # "batch_first": the first dim of input and output is the batch_size
        self.rnn = nn.RNN(input_size, hidden_dim,
                          n_layers, batch_first=True)

        # last, fully-connected layer
        self.fc = nn.Linear(hidden_dim, output_size)

    def forward(self, x, hidden):
        # x (batch_size, seq_length, input_size)
        # hidden (n_layers, batch_size, hidden_dim)
        # r_out (batch_size, time_step, hidden_size)
        batch_size = x.size(0)

        # get RNN outputs
        r_out, hidden = self.rnn(x, hidden)
        # shape output to be (batch_size*seq_length, hidden_dim)
        r_out = r_out.view(-1, self.hidden_dim)

        # get final output
        output = self.fc(r_out)

        return output, hidden
```

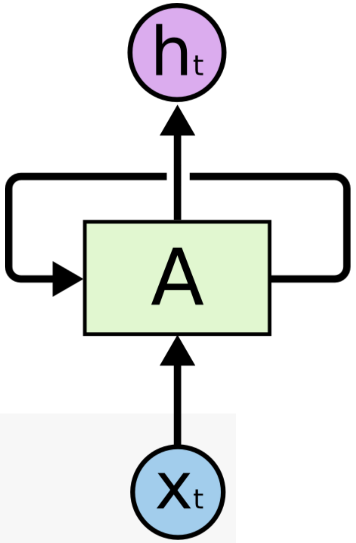
RNN: Training



- **Step 2.1: RNN components**

```
class RNN(nn.Module):
    def __init__(self, input_size, output_size, hidden_dim, n_layers):
        super(RNN, self).__init__()

        self.hidden_dim=hidden_dim

        # define an RNN with specified parameters
        # "batch_first": the first dim of input and output is the batch_size
        self.rnn = nn.RNN(input_size, hidden_dim,
                          n_layers, batch_first=True)

        # last, fully-connected layer
        self.fc = nn.Linear(hidden_dim, output_size)
```

RNN: Training



- **Step 2.2: RNN flow**

```python
def forward(self, x, hidden):
    # x (batch_size, seq_length, input_size)
    # hidden (n_layers, batch_size, hidden_dim)
    # r_out (batch_size, time_step, hidden_size)
    batch_size = x.size(0)

    # get RNN outputs
    r_out, hidden = self.rnn(x, hidden)
    # shape output to be (batch_size*seq_length, hidden_dim)
    r_out = r_out.view(-1, self.hidden_dim)

    # get final output
    output = self.fc(r_out)

    return output, hidden
```

RNN: Training

- **Step 3: Training Loss and Optimizer**

```
# MSE loss and Adam optimizer with a learning rate of 0.01
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(rnn.parameters(), lr=0.01)
```

RNN: Training

- **Step 4: Training Process**

Prepare data for this batch

Feedforward flow

Gradient descent

```
# train the RNN
def train(rnn, n_steps, print_every):

    # initialize the hidden state
    hidden = None

    for batch_i, step in enumerate(range(n_steps)):
        # defining the training data
        time_steps = np.linspace(step * np.pi, (step+1)*np.pi, seq_length + 1)
        data = np.sin(time_steps)
        data.resize((seq_length + 1, 1)) # input_size=1

        x = data[:-1]
        y = data[1:]

        # convert data into Tensors
        x_tensor = torch.Tensor(x).unsqueeze(0) # unsqueeze gives a 1, batch_size dimension
        y_tensor = torch.Tensor(y)

        # outputs from the rnn
        prediction, hidden = rnn(x_tensor, hidden)

        ## Representing Memory ##
        # make a new variable for hidden and detach the hidden state from its history
        # this way, we don't backpropagate through the entire history
        hidden = hidden.data

        # calculate the loss
        loss = criterion(prediction, y_tensor)
        # zero gradients
        optimizer.zero_grad()
        # perform backprop and update weights
        loss.backward()
        optimizer.step()

        # display loss and predictions
        if batch_i%print_every == 0:
            print('Loss: ', loss.item())
            plt.plot(time_steps[1:], x, 'r.') # input
            plt.plot(time_steps[1:], prediction.data.numpy().flatten(), 'b.') # predictions
            plt.show()

    return rnn
```

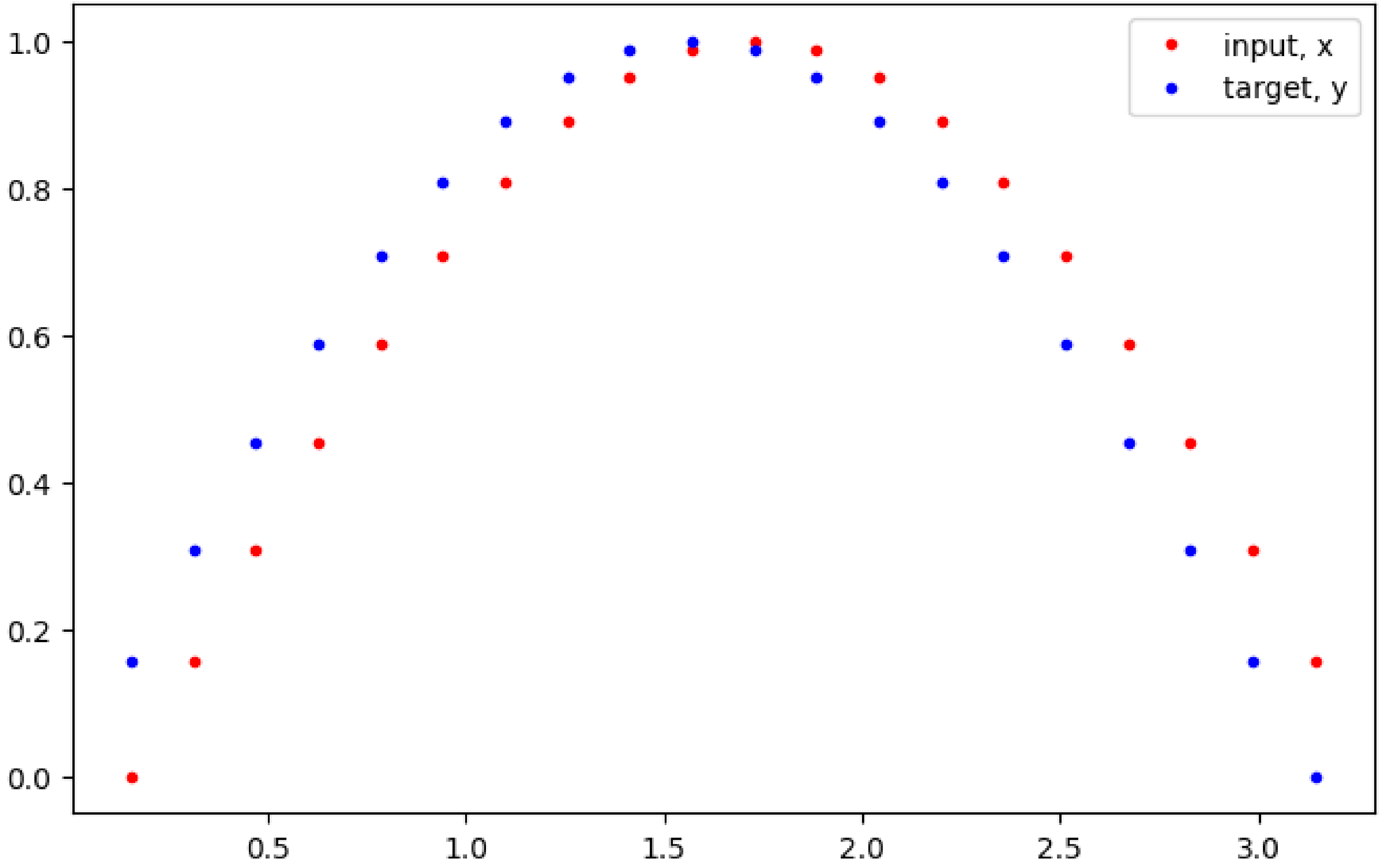
RNN: Training

- **Step 4.1: Get training data for this batch**

```
# defining the training data
time_steps = np.linspace(step * np.pi, (step+1)*np.pi, seq_length + 1)
data = np.sin(time_steps)
data.resize((seq_length + 1, 1)) # input_size=1

x = data[:-1]
y = data[1:]

# convert data into Tensors
x_tensor = torch.Tensor(x).unsqueeze(0) # unsqueeze gives a 1, batch_size dimension
y_tensor = torch.Tensor(y)
```

RNN: Training

- **Step 4.1: Get training data for this batch (if visualized)**



RNN: Training

- **Step 4.2: Feedforward flow**

Call **forward()**

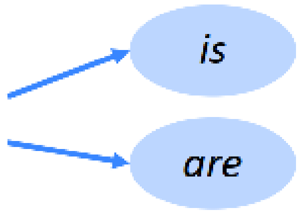
```python
# outputs from the rnn
prediction, hidden = rnn(x_tensor, hidden)
```

```python
def forward(self, x, hidden):
    # x (batch_size, seq_length, input_size)
    # hidden (n_layers, batch_size, hidden_dim)
    # r_out (batch_size, time_step, hidden_size)
    batch_size = x.size(0)

    # get RNN outputs
    r_out, hidden = self.rnn(x, hidden)
    # shape output to be (batch_size*seq_length, hidden_dim)
    r_out = r_out.view(-1, self.hidden_dim)

    # get final output
    output = self.fc(r_out)

    return output, hidden
```

RNN: Training - Vanishing Gradient

Effect of vanishing gradient on RNN-LM

- **LM task:** *The writer of the books ___*



- **Correct answer**: *The writer of the books is planning a sequel*
- **Syntactic recency:** *The writer of the books is* (correct)
- **Sequential recency:** *The writer of the books are* (incorrect)
- Due to vanishing gradient, RNN-LMs are better at learning from sequential recency than syntactic recency, so they make this type of error more often than we'd like [Linzen et al 2016]

RNN: Training - Vanishing Gradient

- Problems with RNNs!
  - Vanishing gradients

motivates

- Fancy RNN variants!
  - LSTM
  - GRU
  - multi-layer
  - bidirectional

People are not satisfied!

Transformer