

# Recurrent Neural Networks

---

CMSC 473/673 - NATURAL LANGUAGE PROCESSING

*Slides modified from Dr. Frank Ferraro*

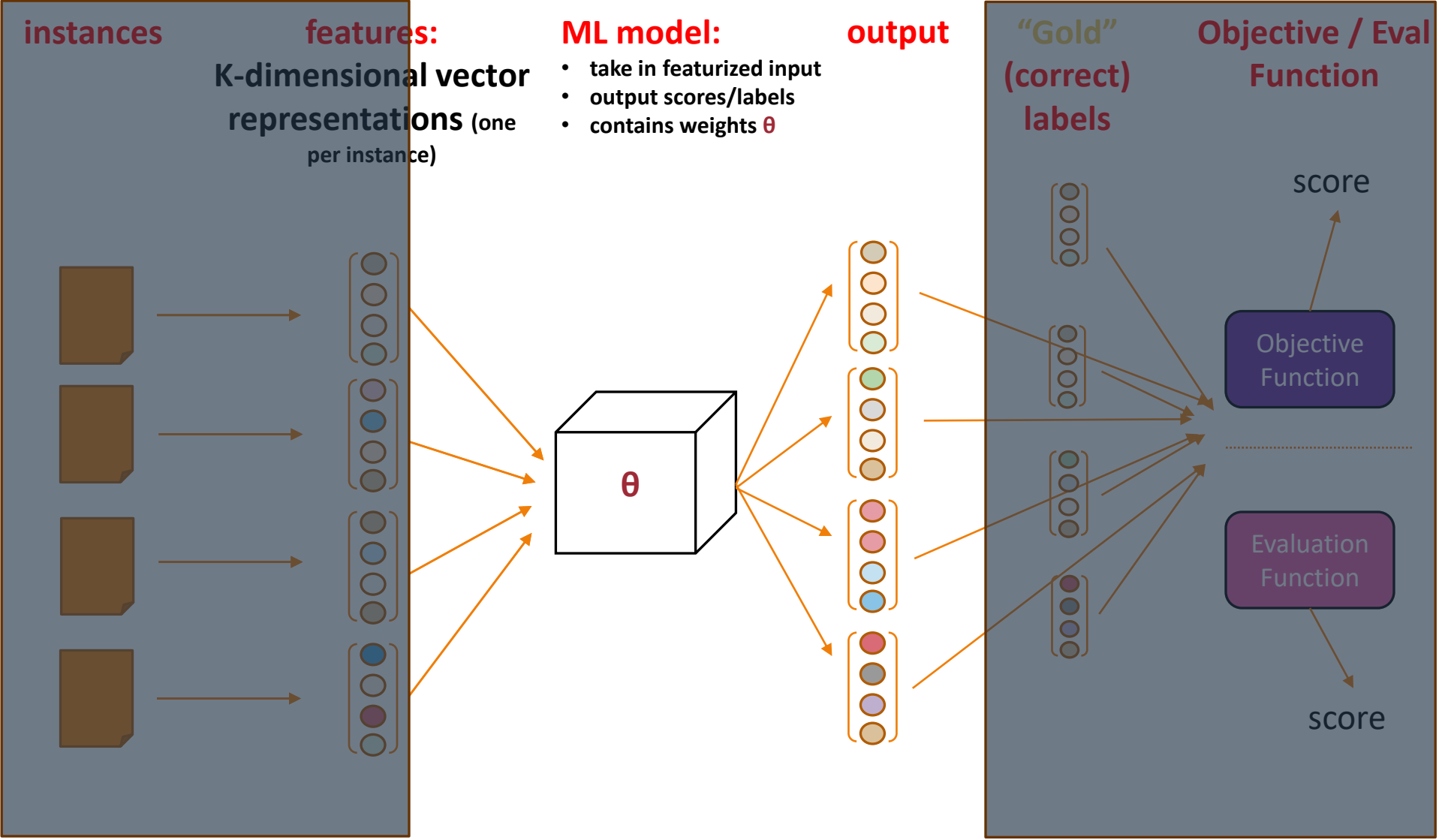
# Learning Objectives

---

Define the basic cell architecture of an RNN

Backpropagate loss through an example RNN

Create a simple RNN with PyTorch



# Review: Maxent Language Models

given some context...



compute beliefs about what is likely...

$$p(w_i | w_{i-3}, w_{i-2}, w_{i-1}) = \text{softmax}(\theta_{w_i} \cdot f(w_{i-3}, w_{i-2}, w_{i-1}))$$

predict the next word

can we learn word-specific weights (by type)?



# Review: Neural Language Models

given some context...



can we learn the feature function(s) for just the context?

compute beliefs about what is likely...



$$p(w_i | w_{i-3}, w_{i-2}, w_{i-1}) = \text{softmax}(\theta_{w_i} \cdot f(w_{i-3}, w_{i-2}, w_{i-1}))$$

predict the next word

can we learn word-specific weights (by type)?



# Review: Neural Language Models

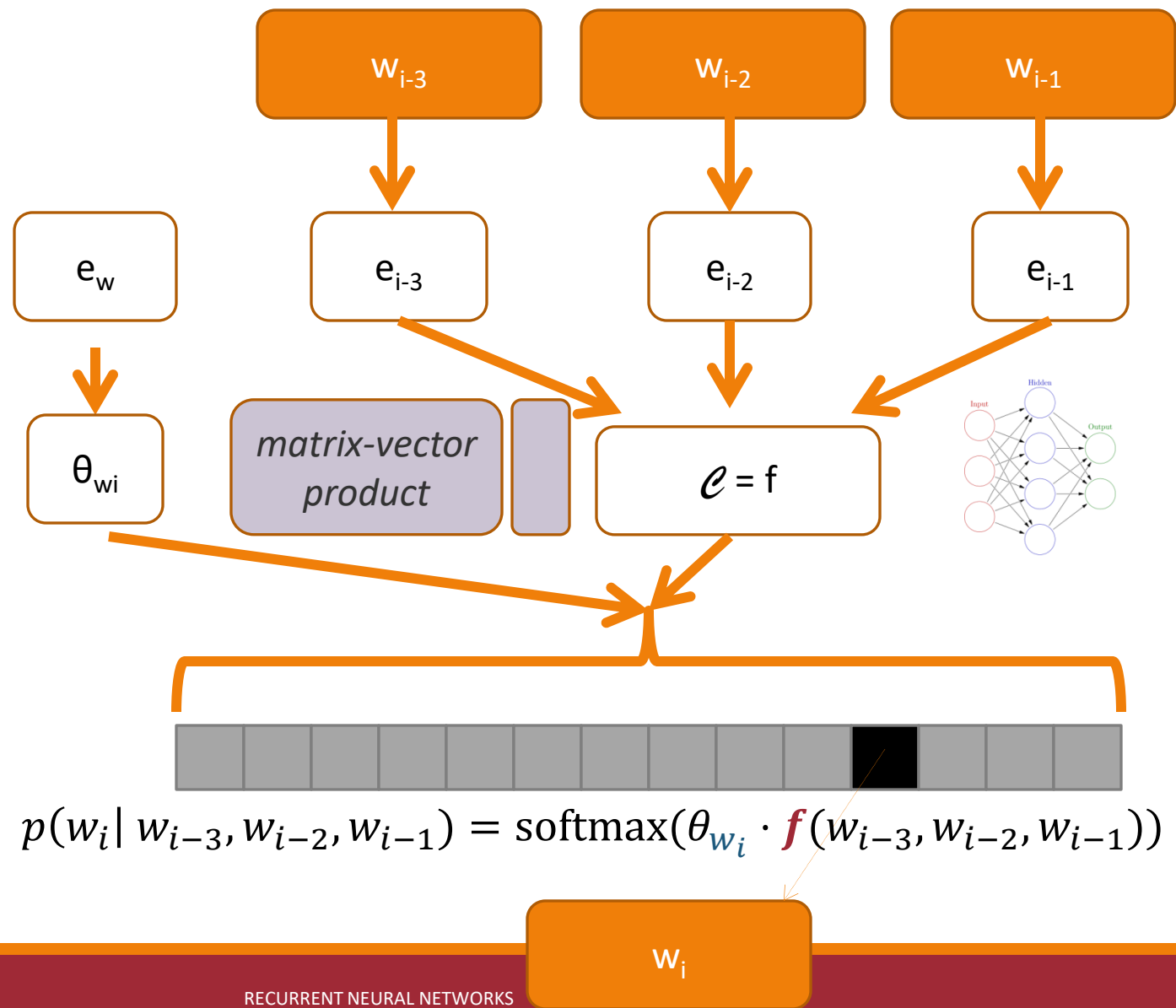
given some context...

create/use  
“distributed  
representations” ...

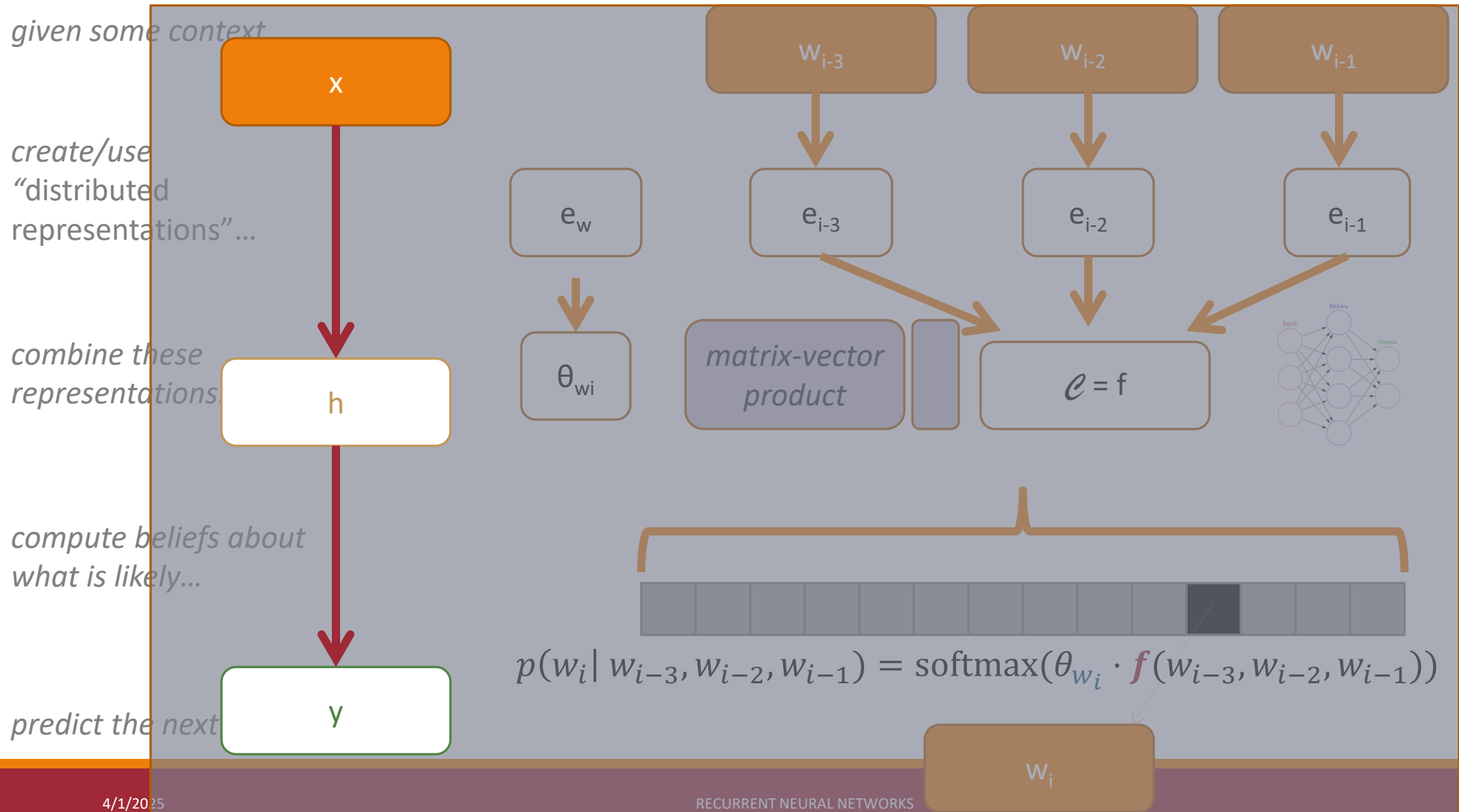
combine these  
representations...

compute beliefs about  
what is likely...

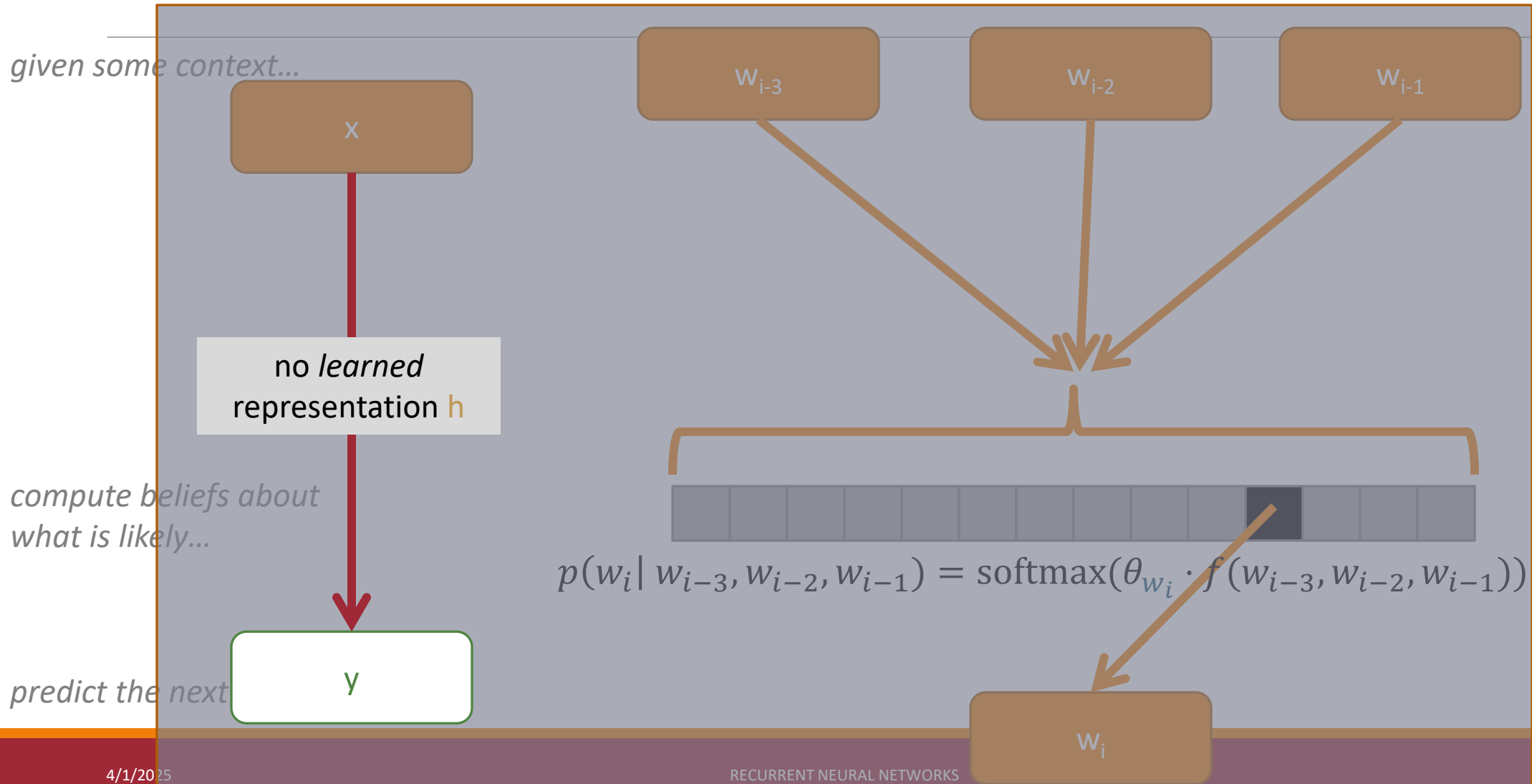
predict the next word



# Review: Neural Language Models



# Review: Maxent Language Models





# Review: LM Comparison

---

## COUNT-BASED

Class-specific

## MAXENT

Class-based

Uses features

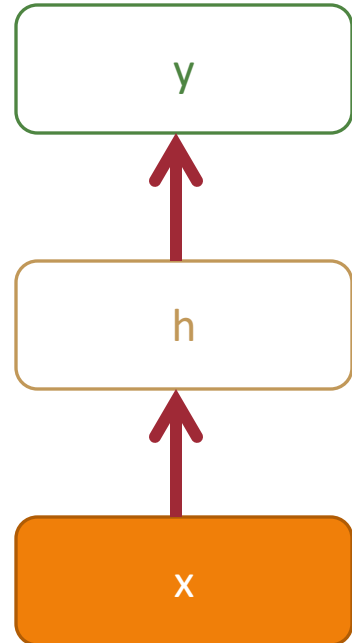
## NEURAL

Class-based

Uses *embedded* features

# Review: Network Types: Flat **Input**, Flat **Output**

---



## 1. Feed forward

Linearizable feature input  
Bag-of-items classification/regression  
Basic non-linear model

# Review: A Neural N-Gram Model

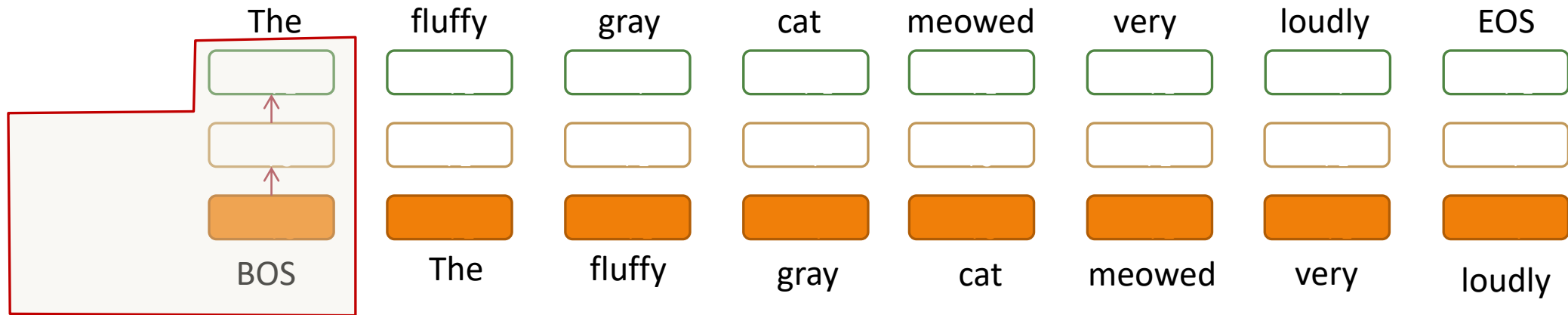
---

The fluffy gray cat meowed very loudly



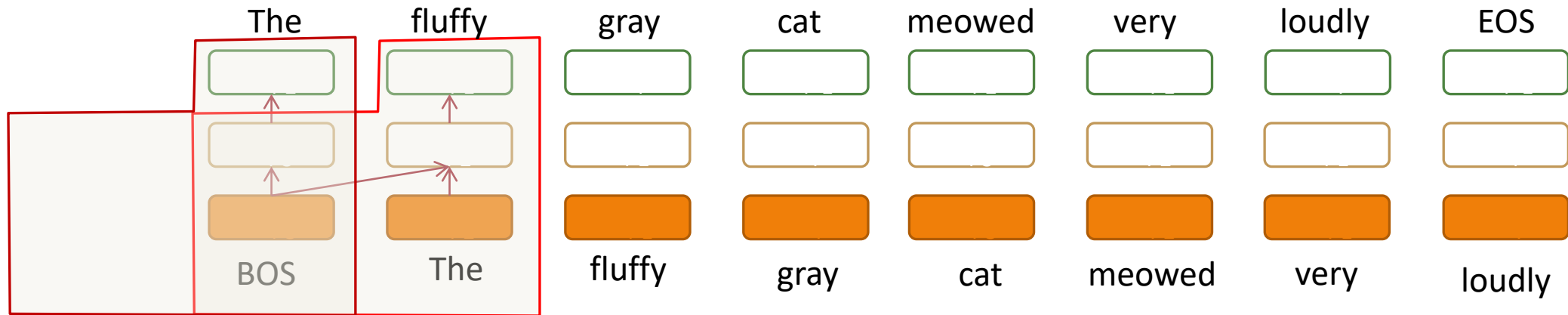
# Review: A Neural N-Gram Model (N=3)

The fluffy gray cat meowed very loudly



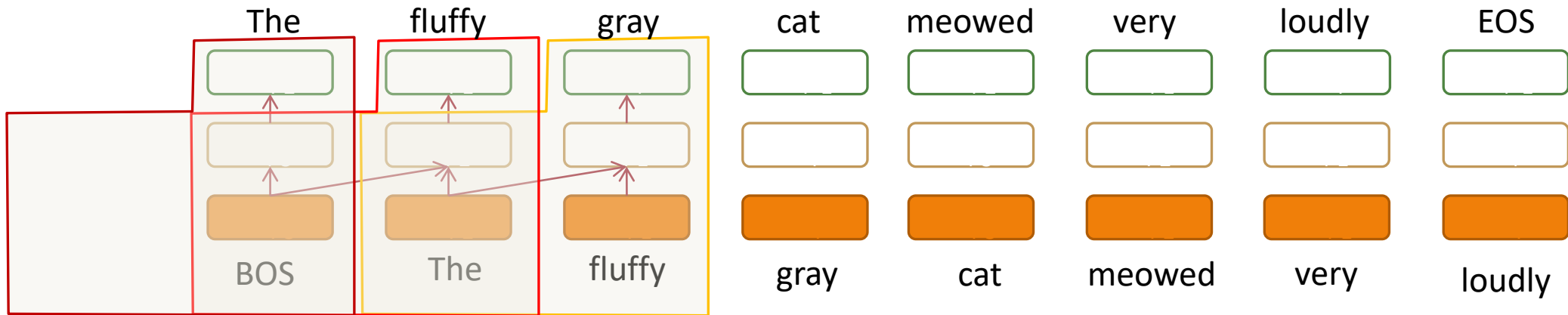
# Review: A Neural N-Gram Model (N=3)

The fluffy gray cat meowed very loudly



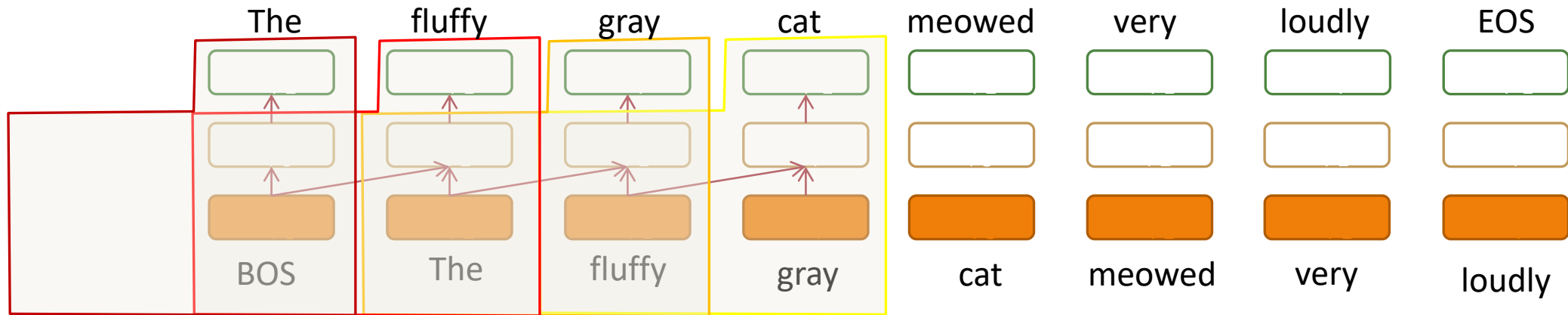
# Review: A Neural N-Gram Model (N=3)

The fluffy gray cat meowed very loudly



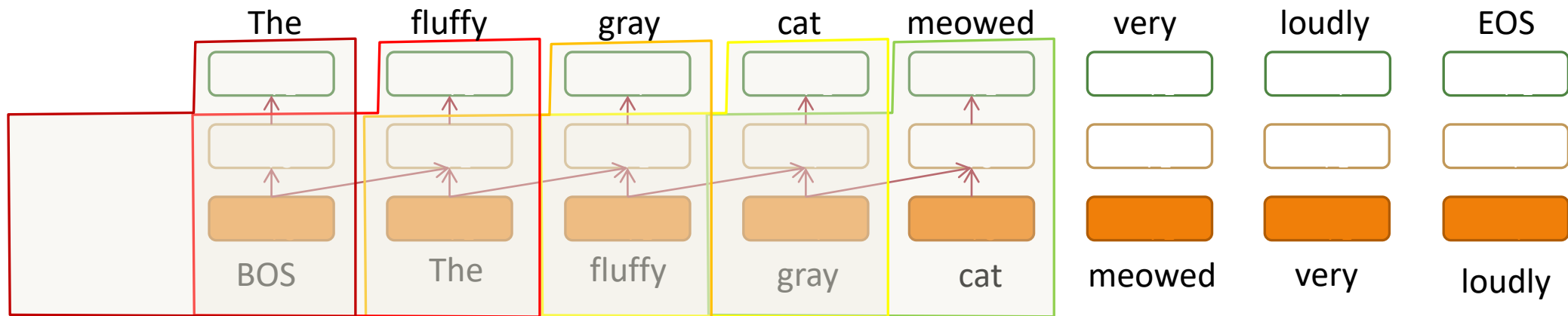
# Review: A Neural N-Gram Model (N=3)

The fluffy gray cat meowed very loudly



# Review: A Neural N-Gram Model (N=3)

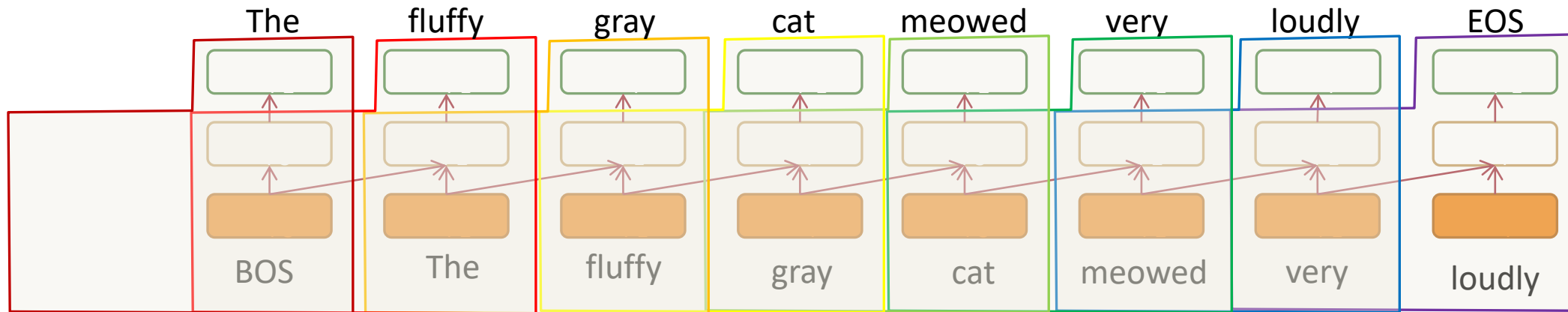
The fluffy gray cat meowed very loudly





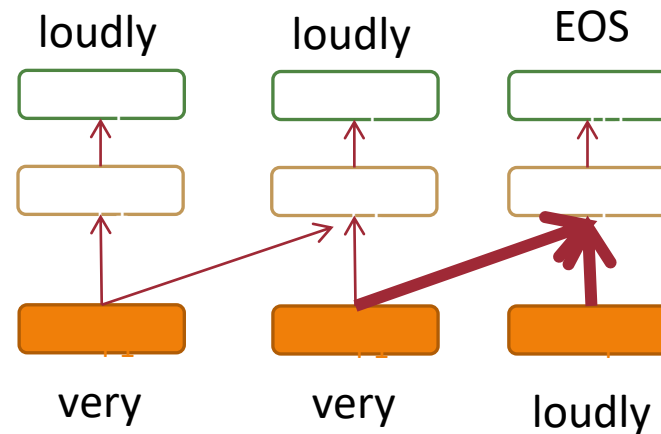
# Review: A Neural N-Gram Model (N=3)

The fluffy gray cat meowed very loudly



# A Neural N-Gram Model (N=3)

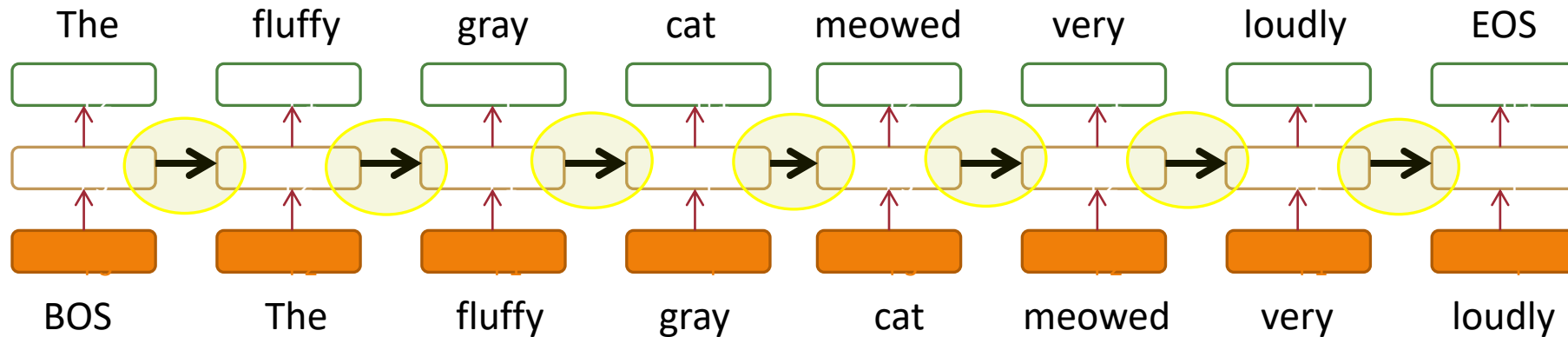
The fluffy gray cat meowed very loudly



Critical issue: the amount of information flow is fundamentally restricted!!!

# A Recurrent Neural Language Model

The fluffy gray cat meowed very loudly

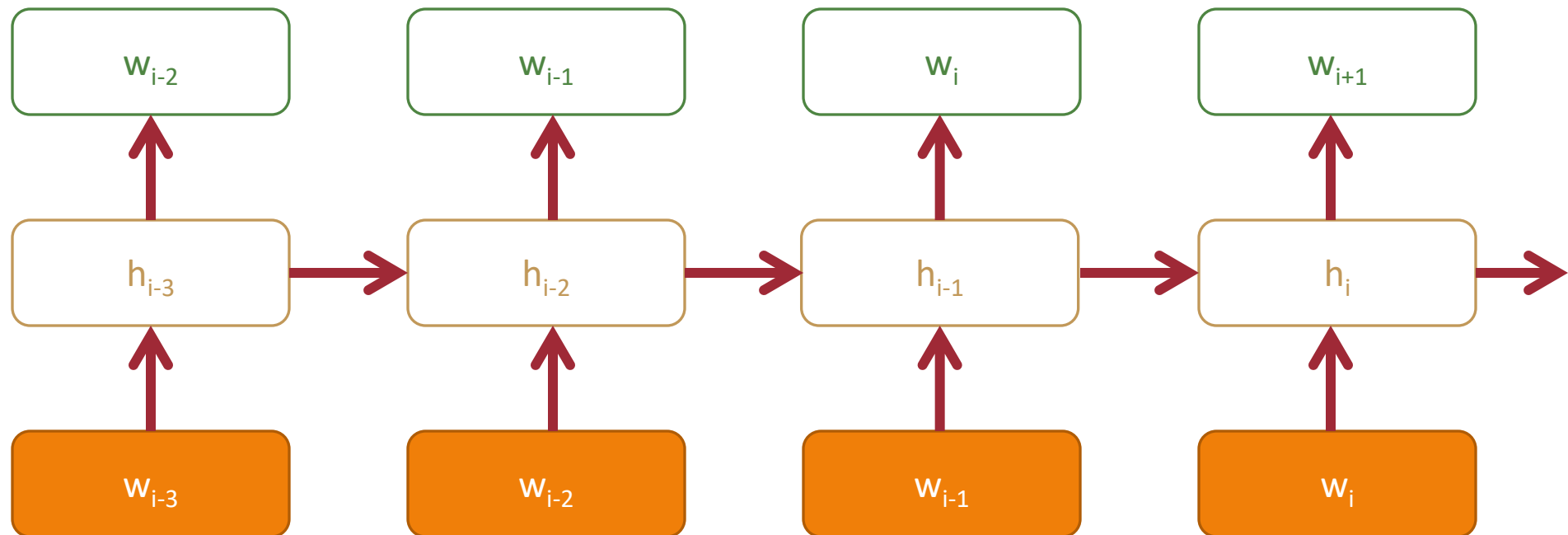


Critical issue: the amount of information flow is fundamentally restricted!!!

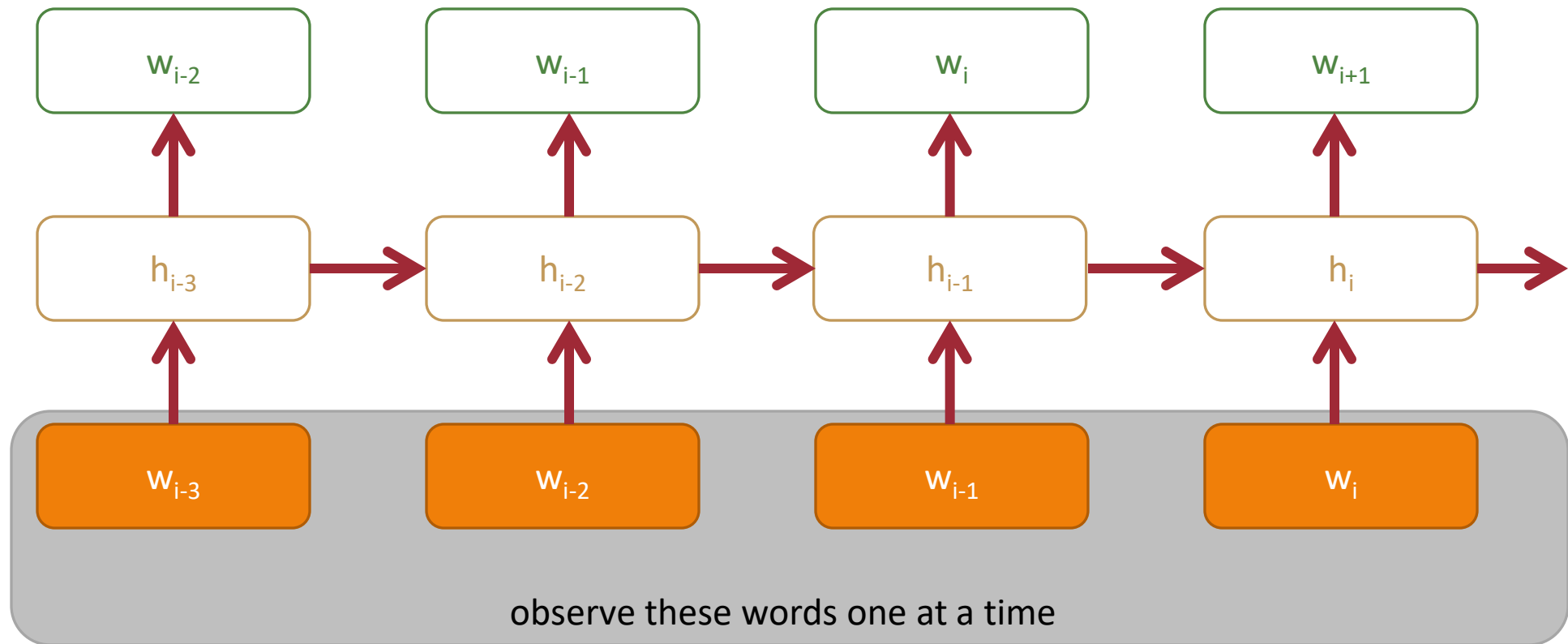
Allowing signal to flow from one hidden state to another could help solve this!

# A Classic View of Recurrent Neural Language Modeling

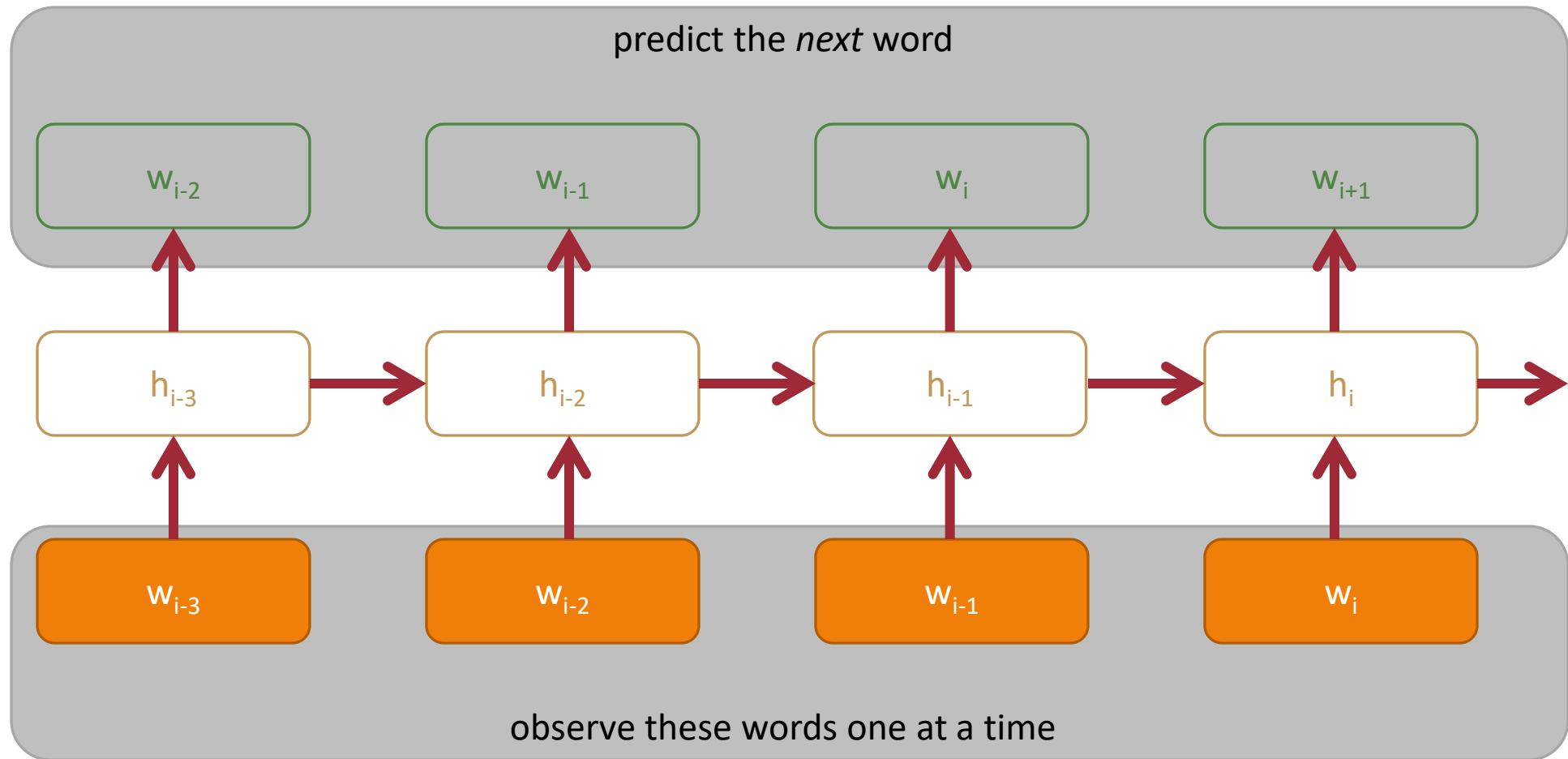
---



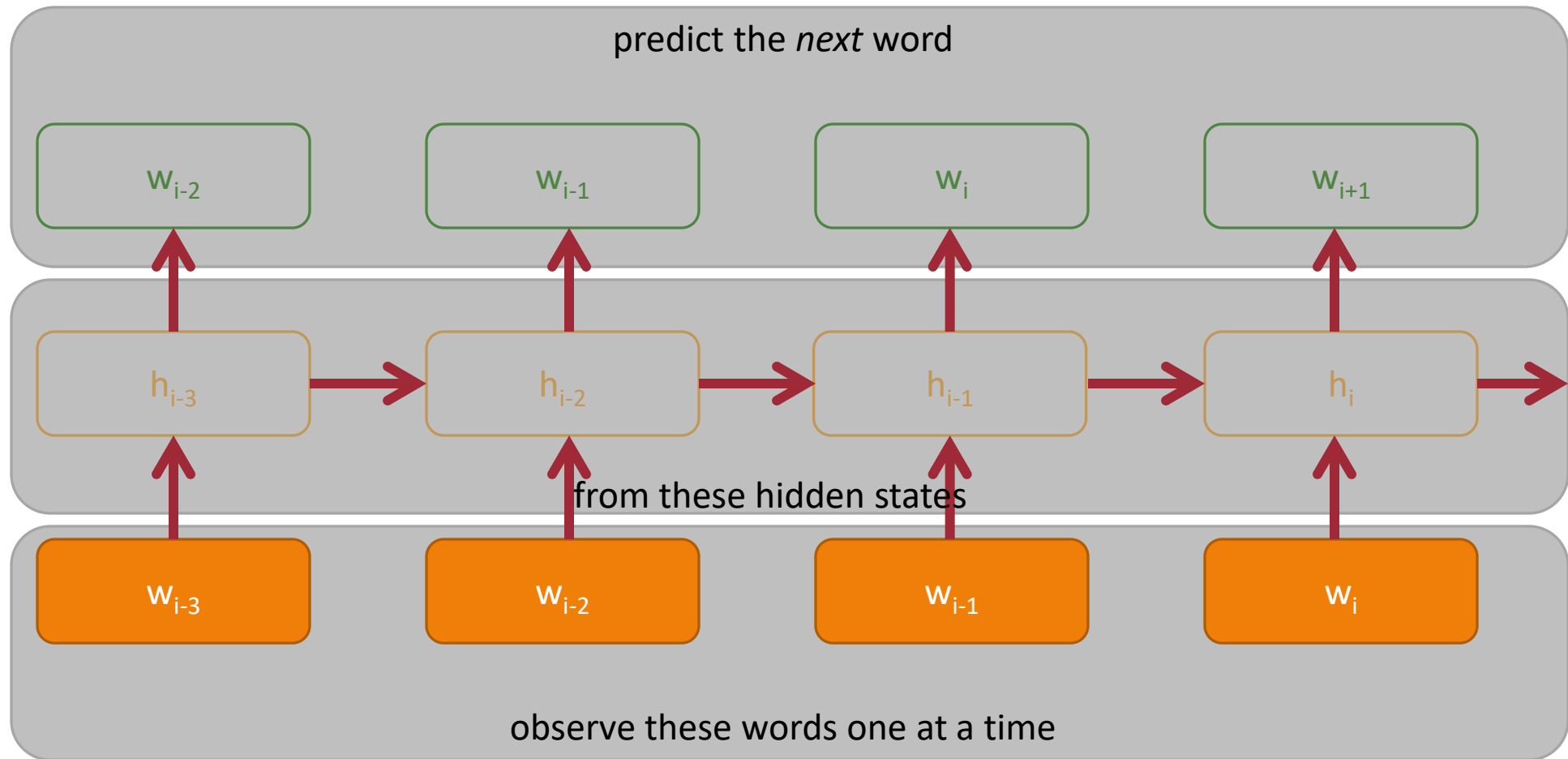
# A Classic View of Recurrent Neural Language Modeling



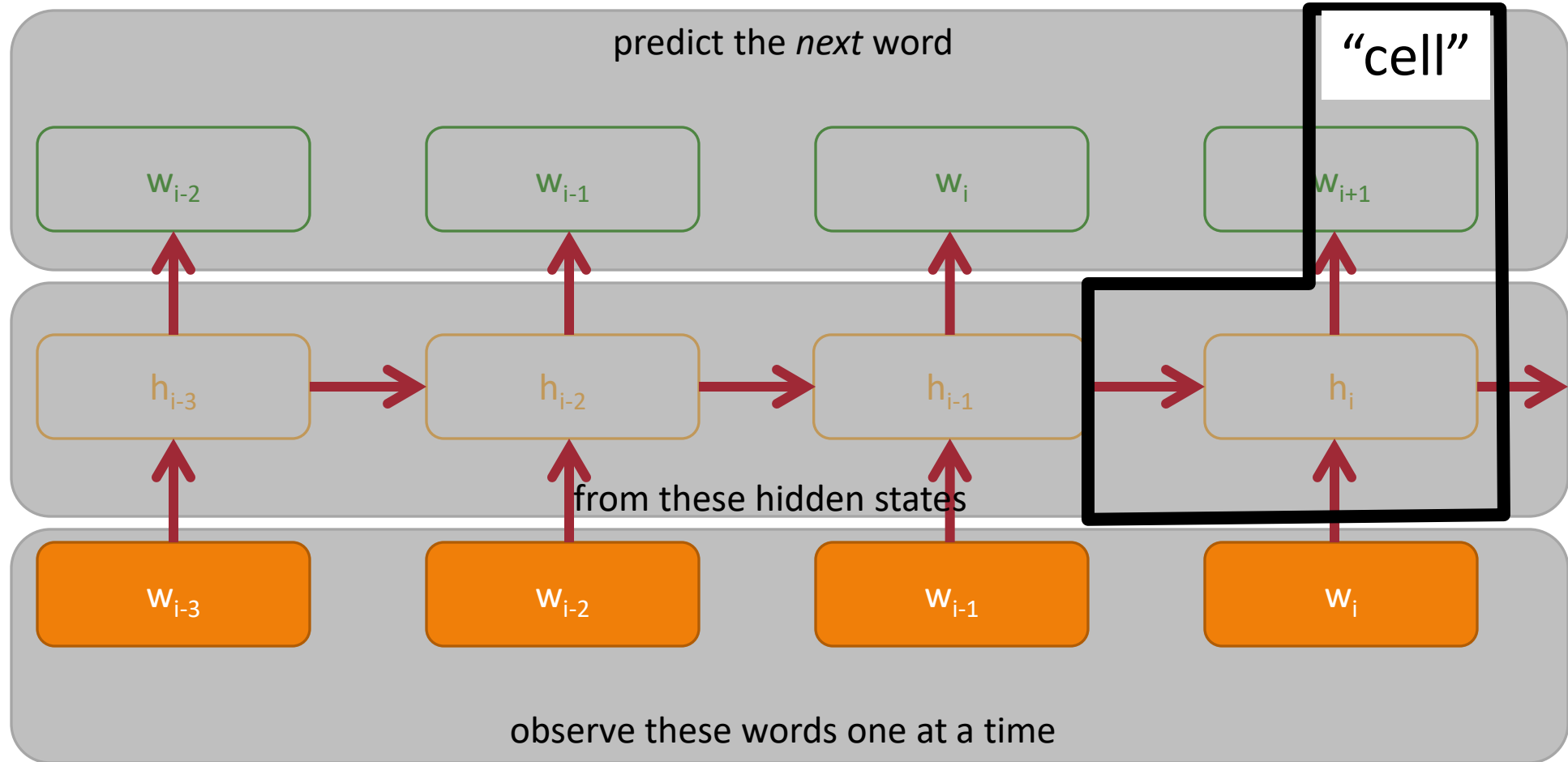
# A Classic View of Recurrent Neural Language Modeling



# A Classic View of Recurrent Neural Language Modeling



# A Classic View of Recurrent Neural Language Modeling





# Review: Forward Propagation Example

Calculate outputs to the hidden layer (units h1 and h2):

How do we do this?

Use our activation function!

$$g(x) = \frac{1}{1 + e^{-x}}$$

What will be our  $x$ ?

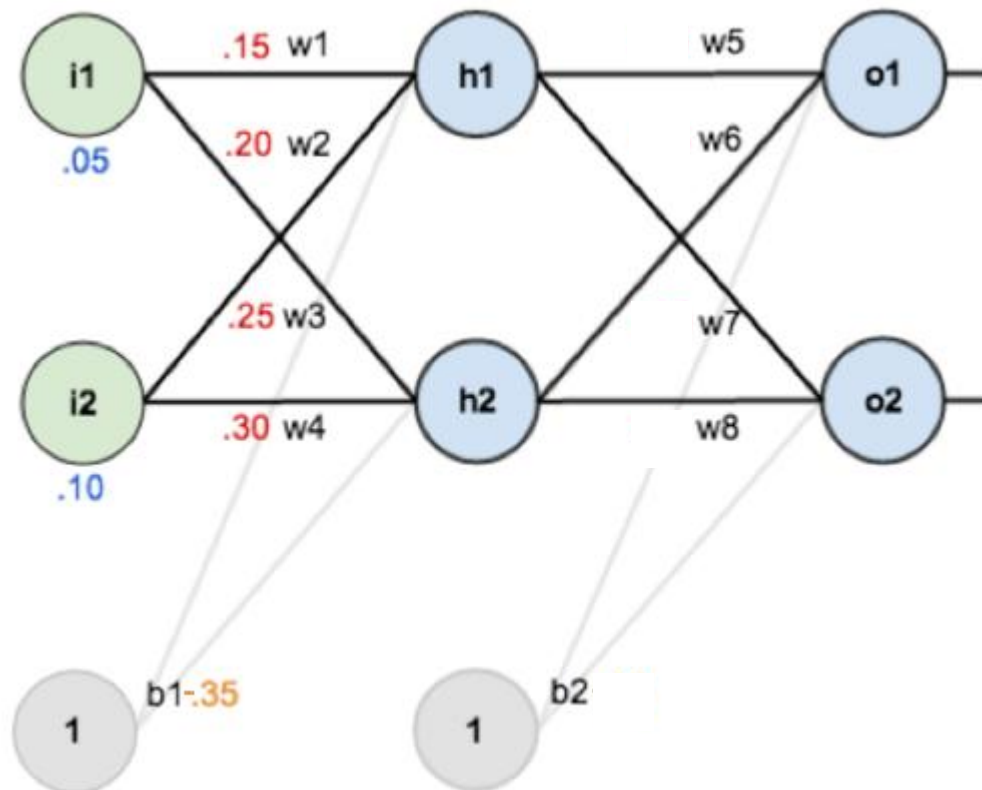
$$in_{h1} = -.3225$$

$$in_{h2} = -.3075$$

For each layer:

1. Calculate the weighted sum of inputs to each neuron unit
2. Evaluate the activation function to determine the output of each neuron unit
3. Use outputs as inputs for the next layer

4/1/2025

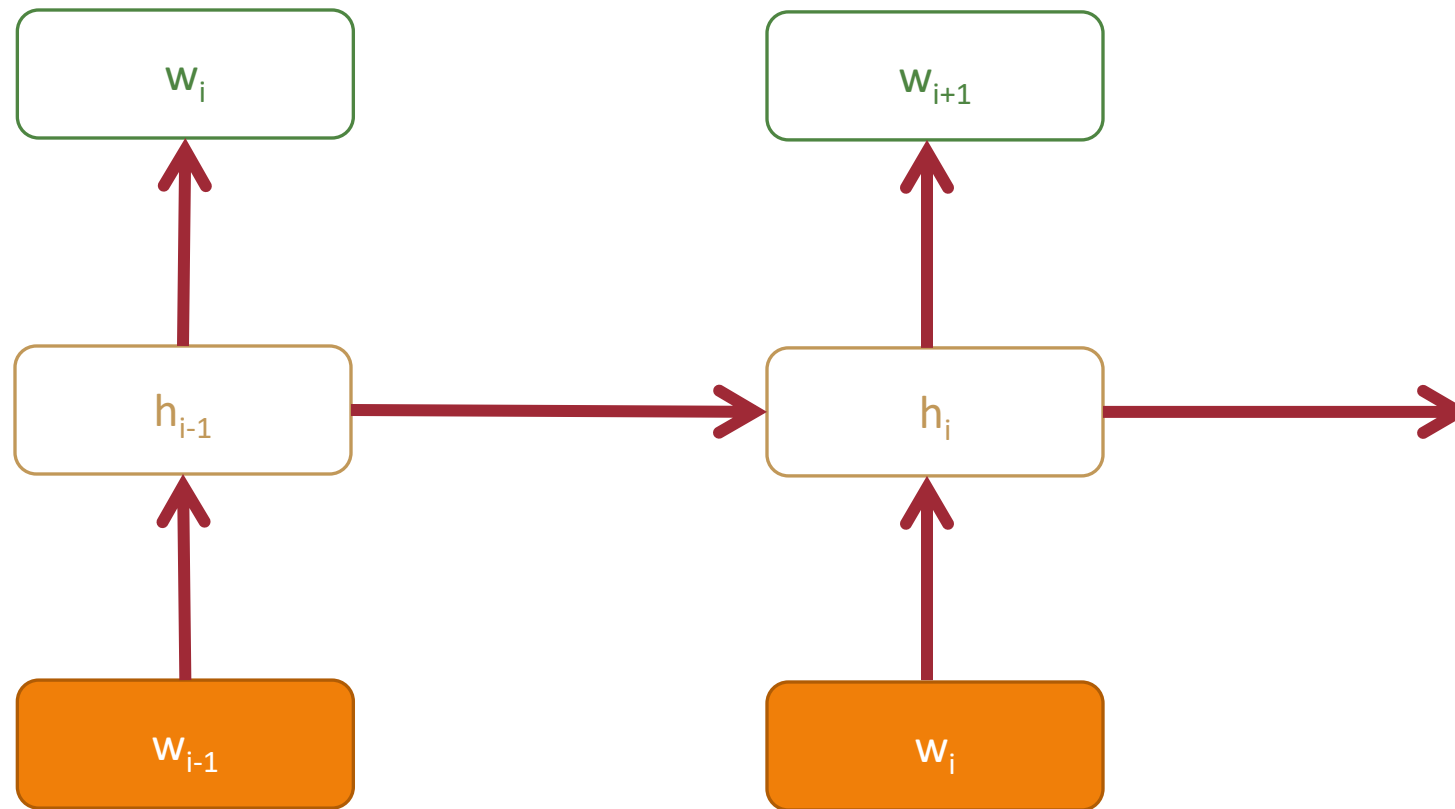


$$\begin{aligned} out_{h1} &= g(in_{h1}) \\ &= \frac{1}{1 + e^{-in_{h1}}} \\ &= \frac{1}{1 + e^{-(-.3225)}} \\ &= .4188 \end{aligned}$$

$$\begin{aligned} out_{h2} &= g(in_{h2}) \\ &= \frac{1}{1 + e^{-in_{h2}}} \\ &= \frac{1}{1 + e^{-(-.3075)}} \\ &= .4237 \end{aligned}$$

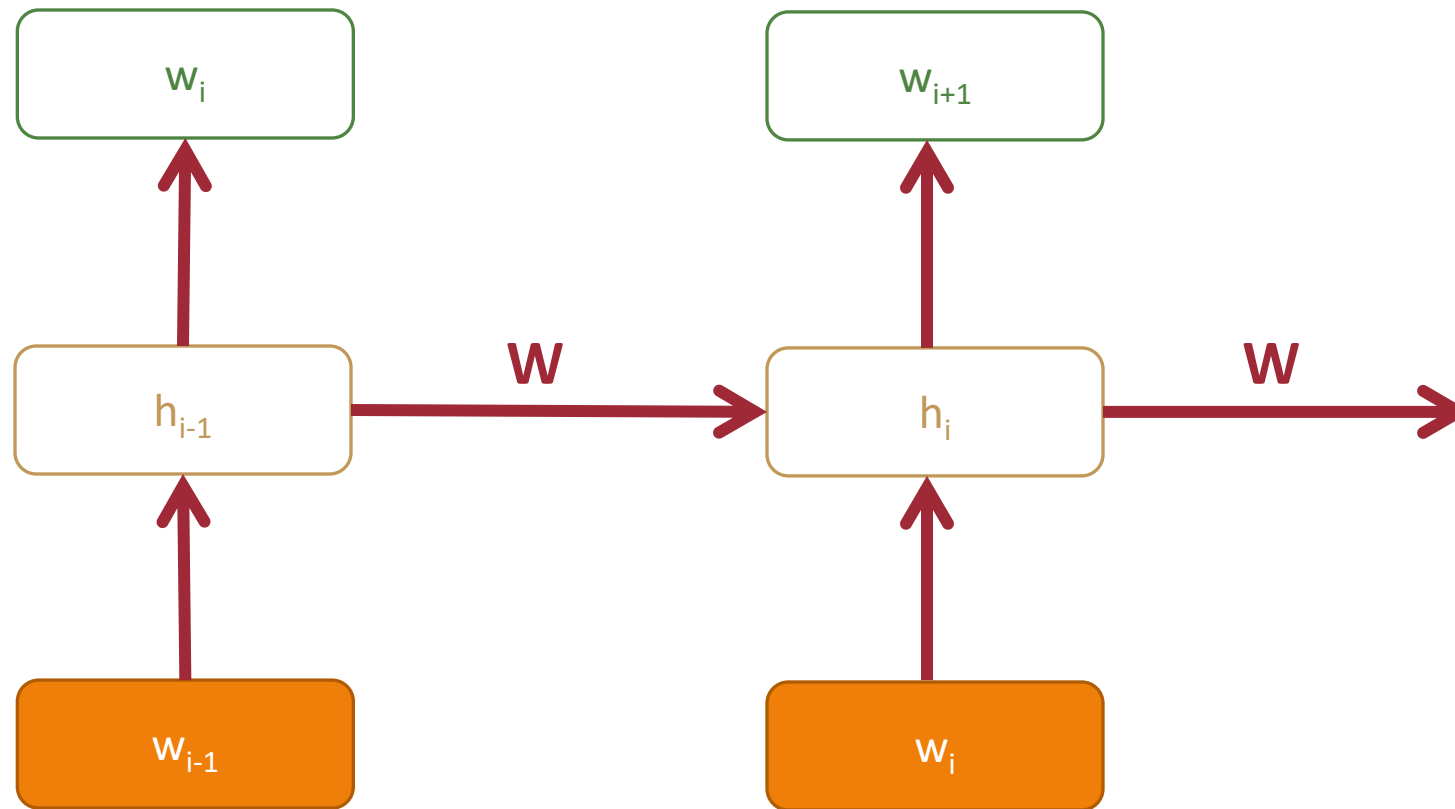
# A Recurrent Neural Network Cell

---



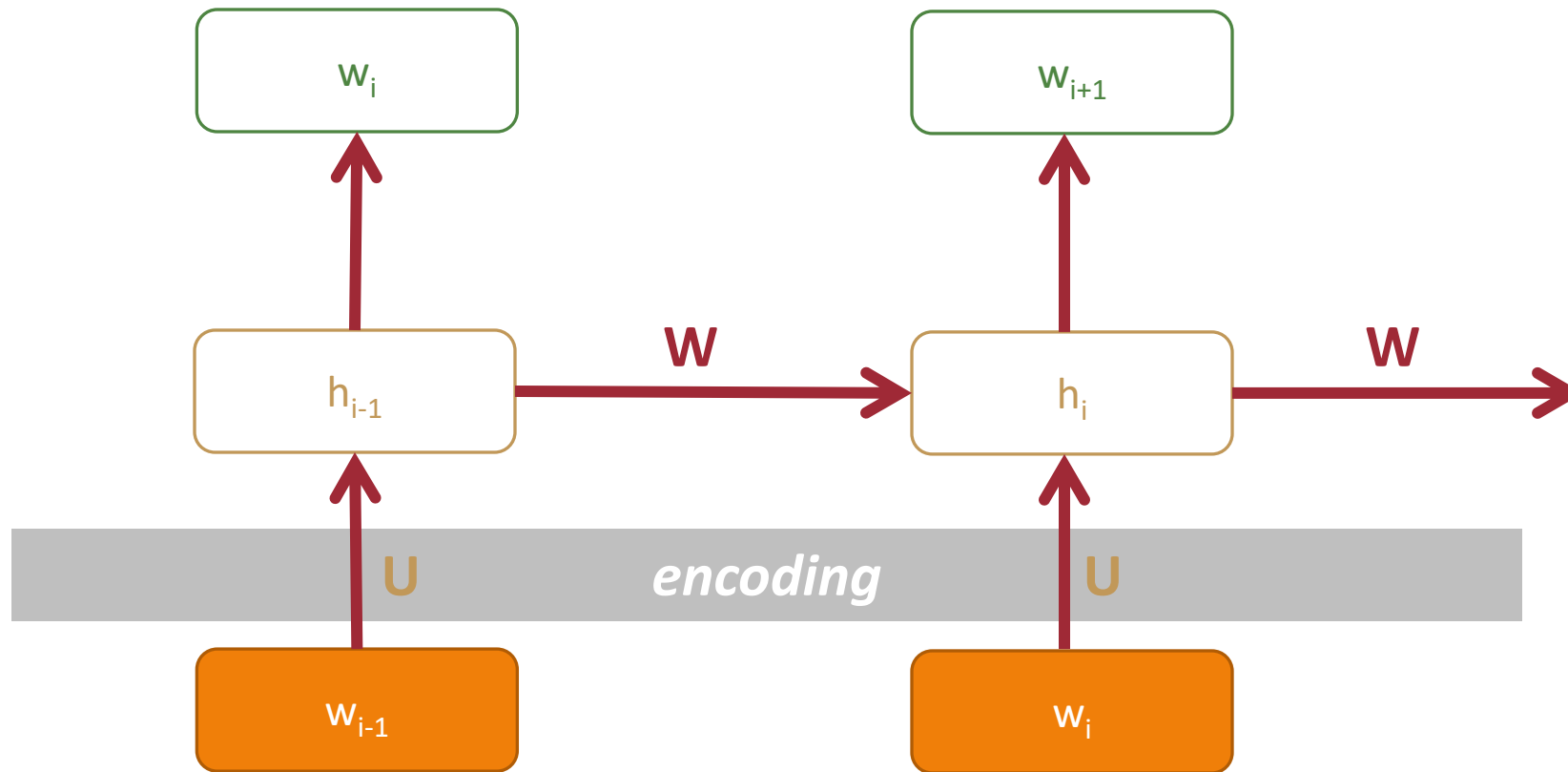
# A Recurrent Neural Network Cell

---

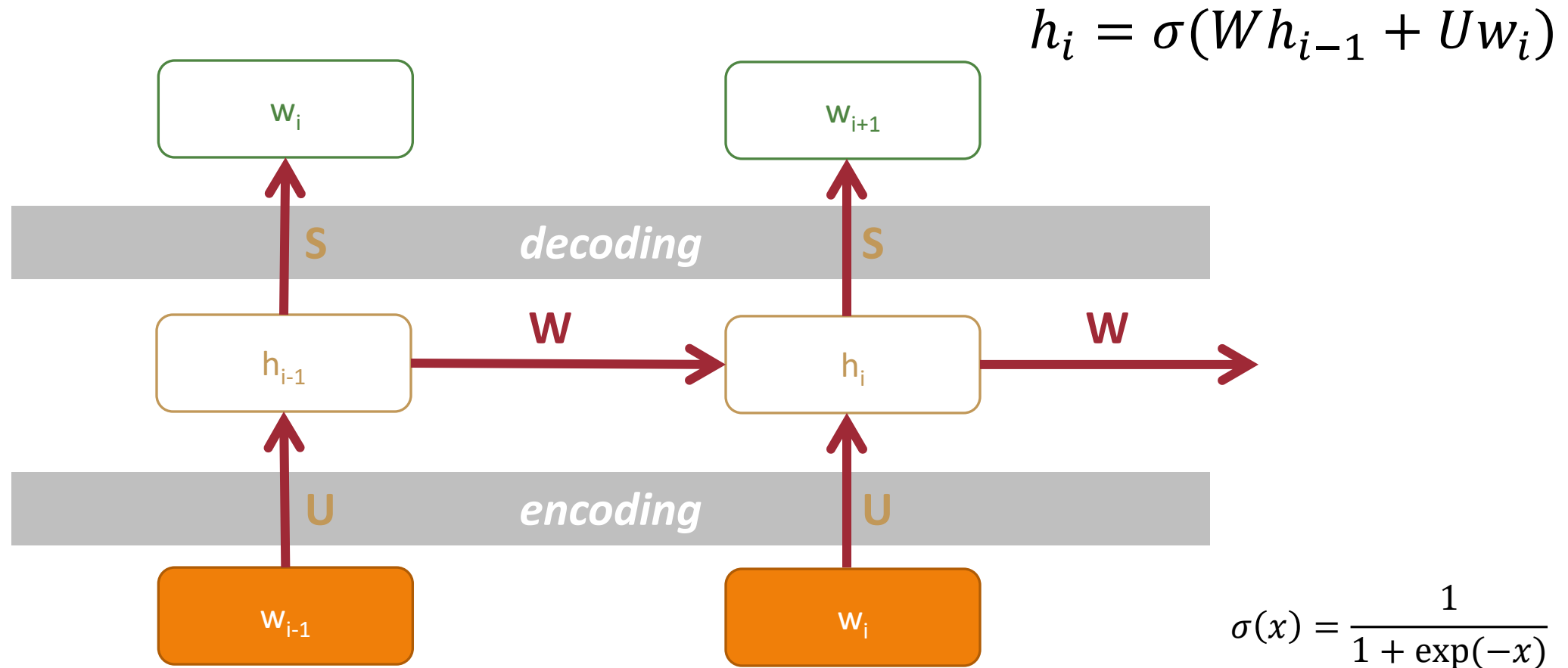


# A Recurrent Neural Network Cell

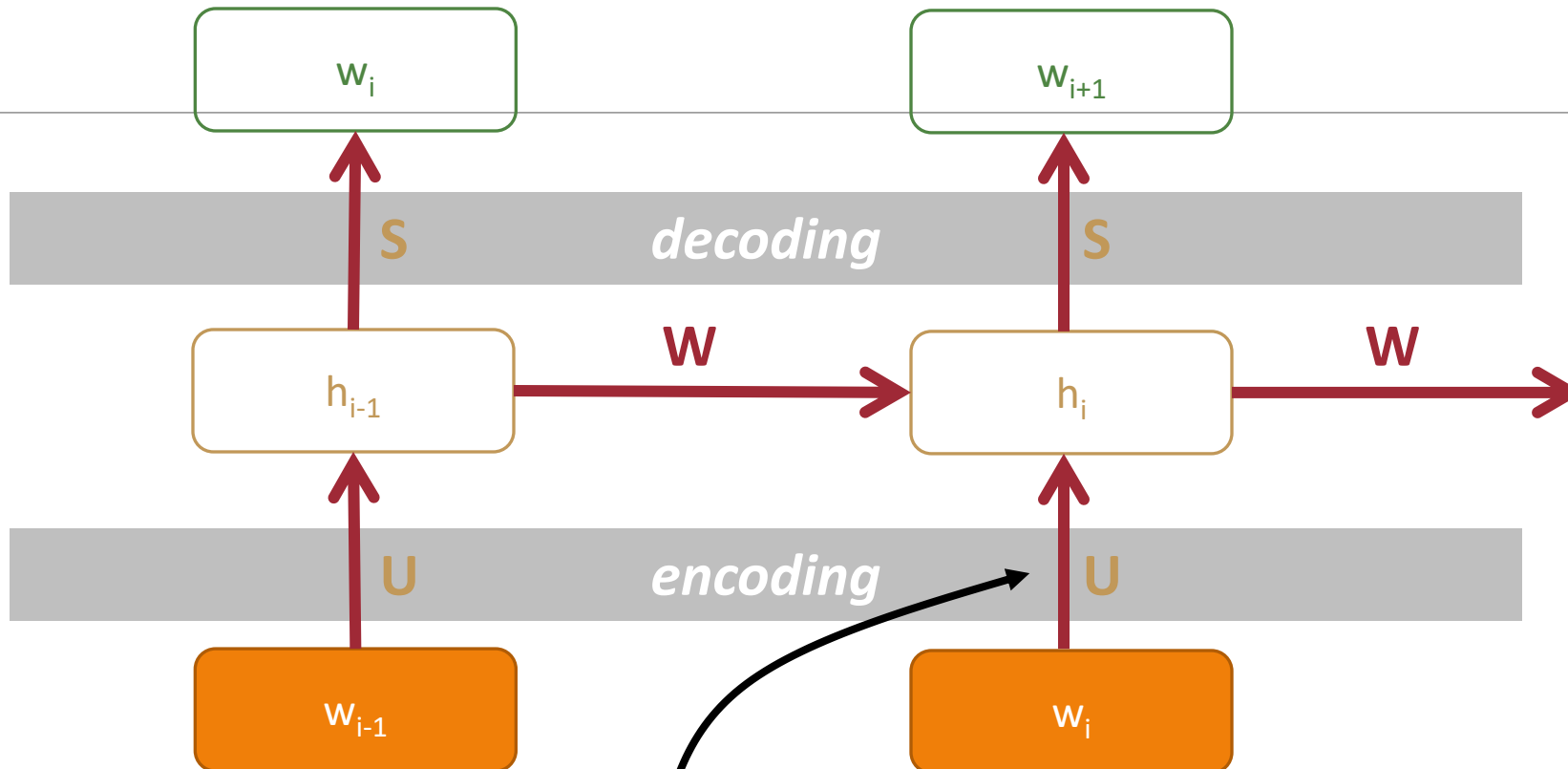
---



# A Recurrent Neural Network Cell



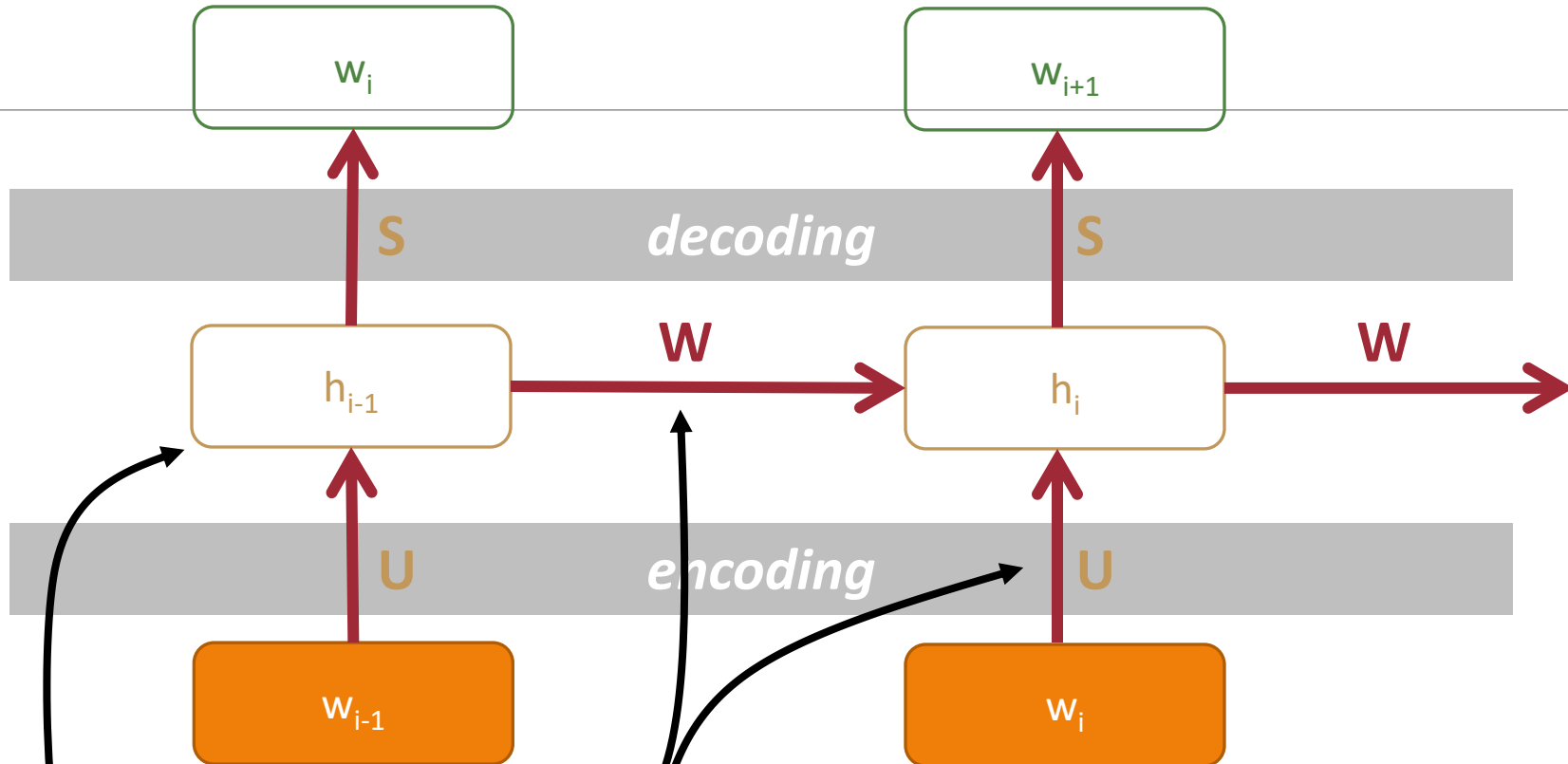
# A Simple Recurrent Neural Network Cell



$$h_i = \sigma(W h_{i-1} + U w_i)$$

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

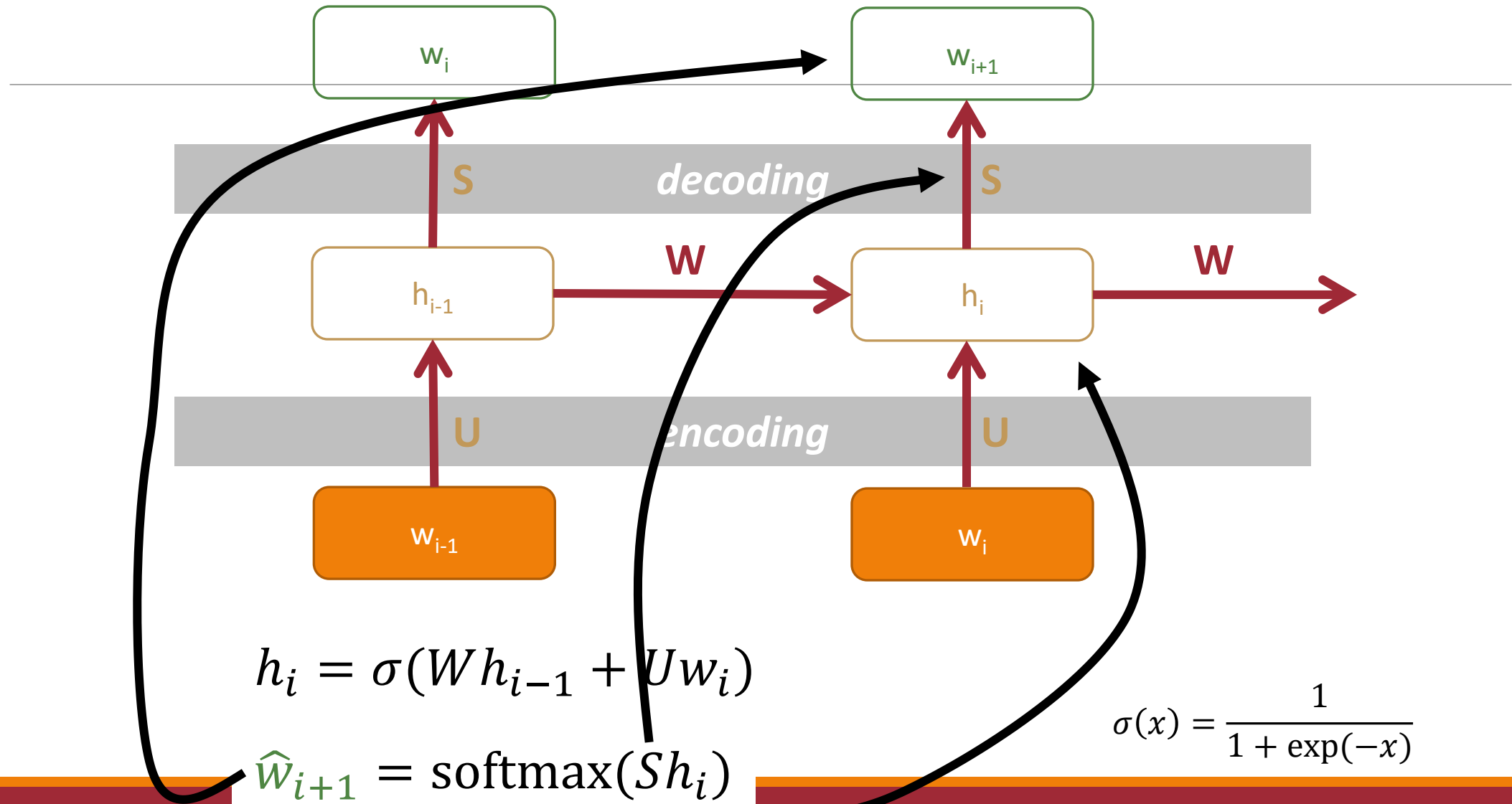
# A Simple Recurrent Neural Network Cell



$$h_i = \sigma(W h_{i-1} + U w_i)$$

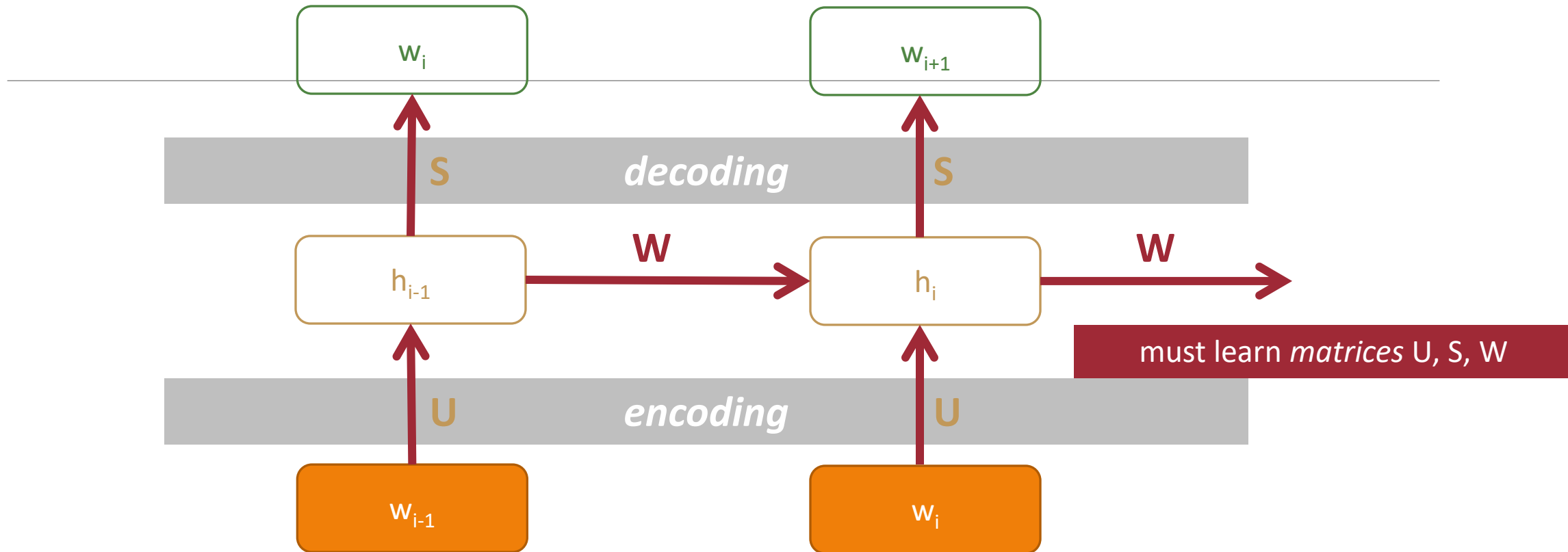
$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

# A Simple Recurrent Neural Network Cell





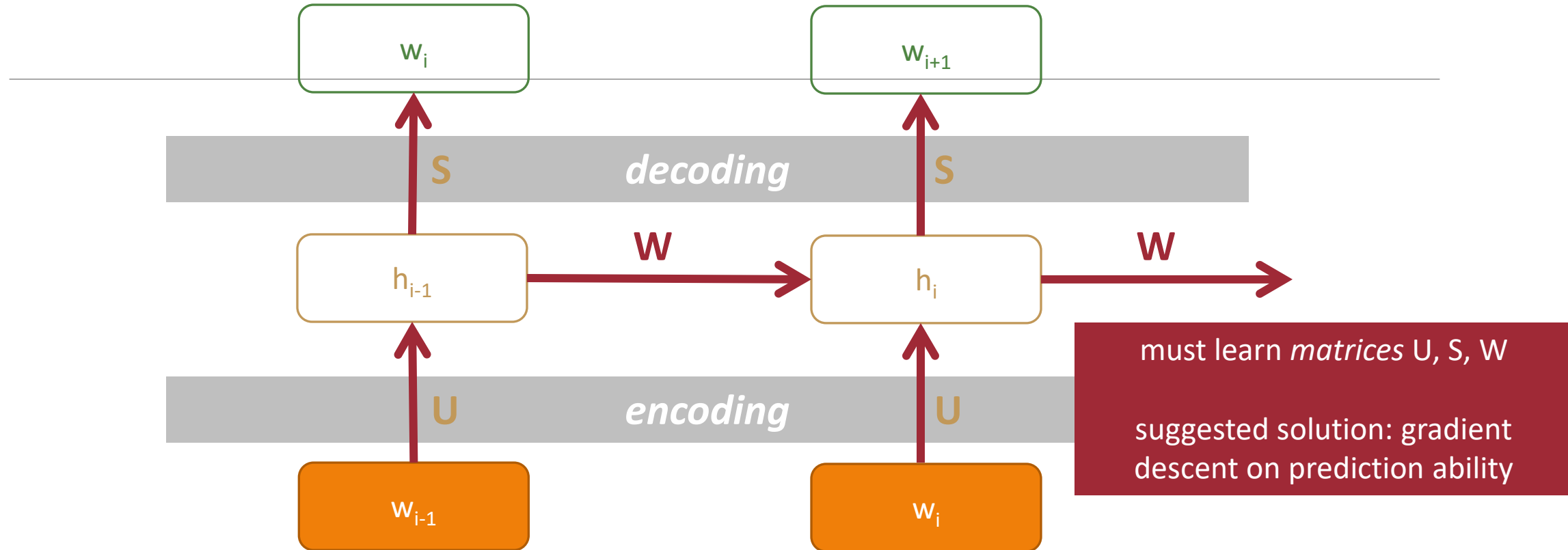
# A Simple Recurrent Neural Network Cell



$$h_i = \sigma(W h_{i-1} + U w_i)$$

$$\hat{w}_{i+1} = \text{softmax}(S h_i)$$

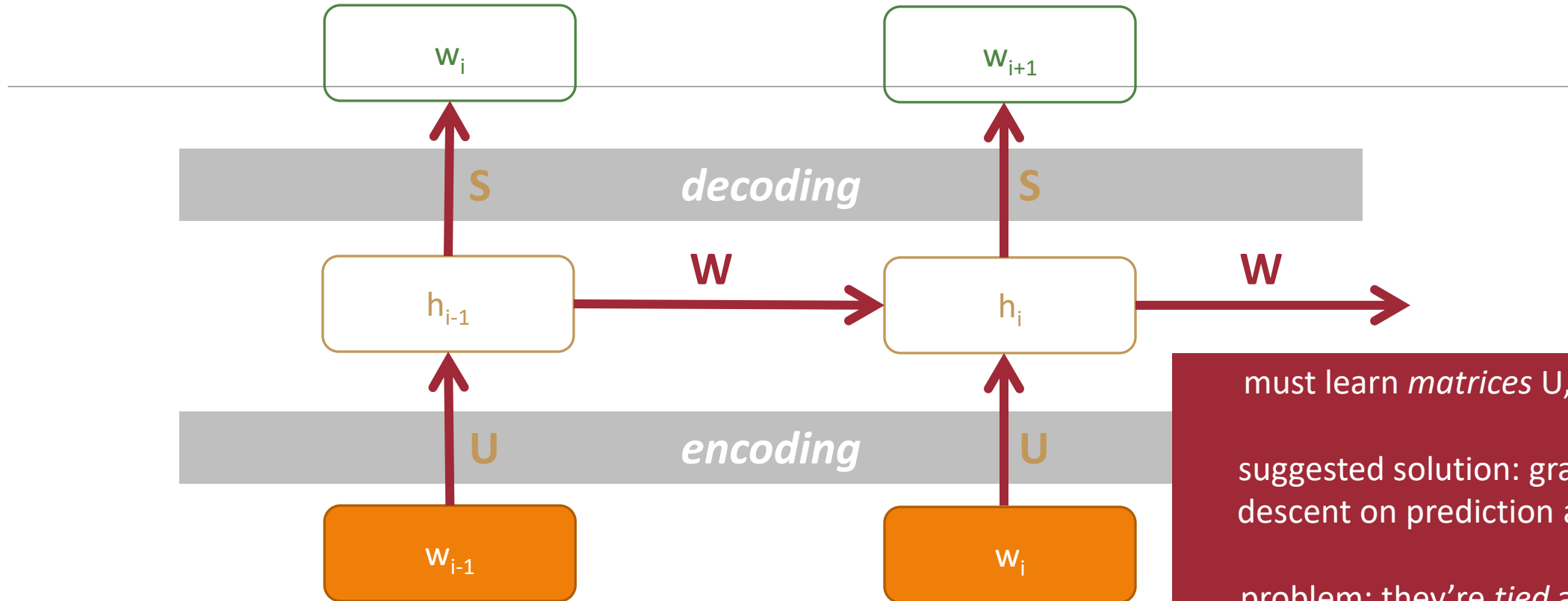
# A Simple Recurrent Neural Network Cell



$$h_i = \sigma(W h_{i-1} + U w_i)$$

$$\hat{w}_{i+1} = \text{softmax}(S h_i)$$

# A Simple Recurrent Neural Network Cell



must learn *matrices*  $U, S, W$

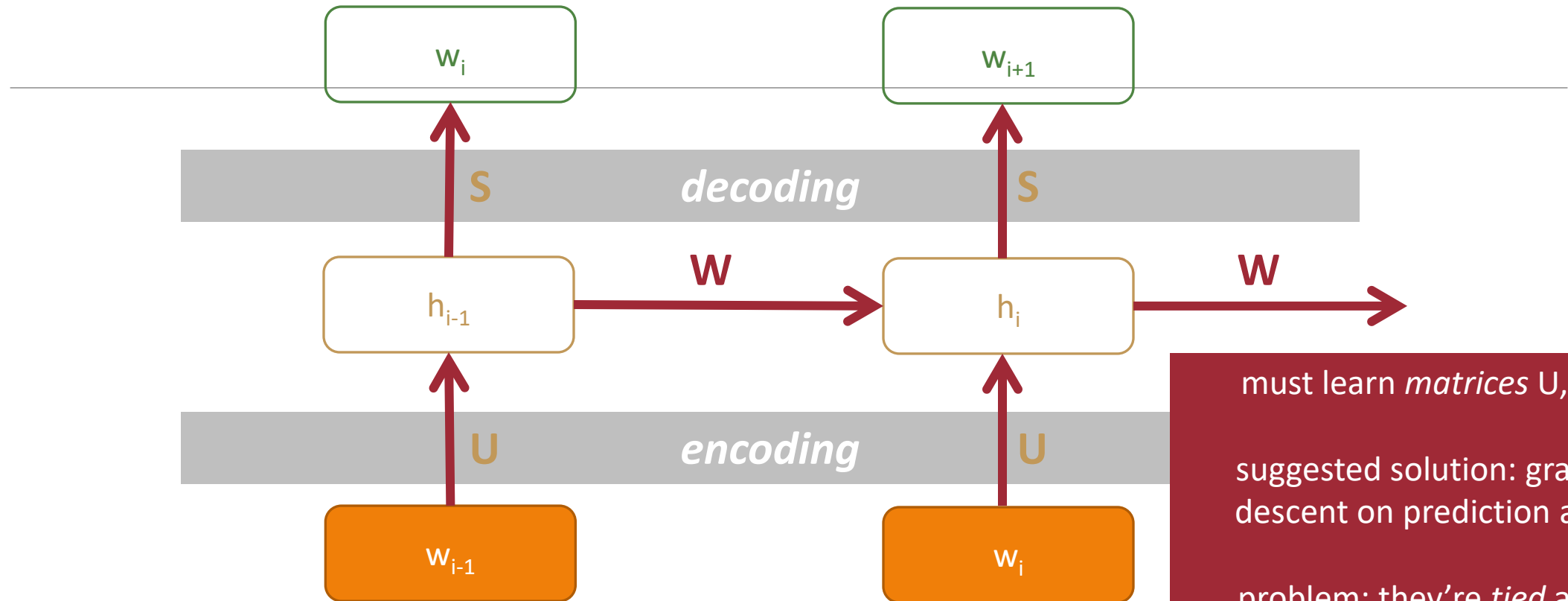
suggested solution: gradient descent on prediction ability

problem: they're *tied* across inputs/timesteps

$$h_i = \sigma(W h_{i-1} + U w_i)$$

$$\hat{w}_{i+1} = \text{softmax}(S h_i)$$

# A Simple Recurrent Neural Network Cell



$$h_i = \sigma(W h_{i-1} + U w_i)$$

$$\hat{w}_{i+1} = \text{softmax}(S h_i)$$

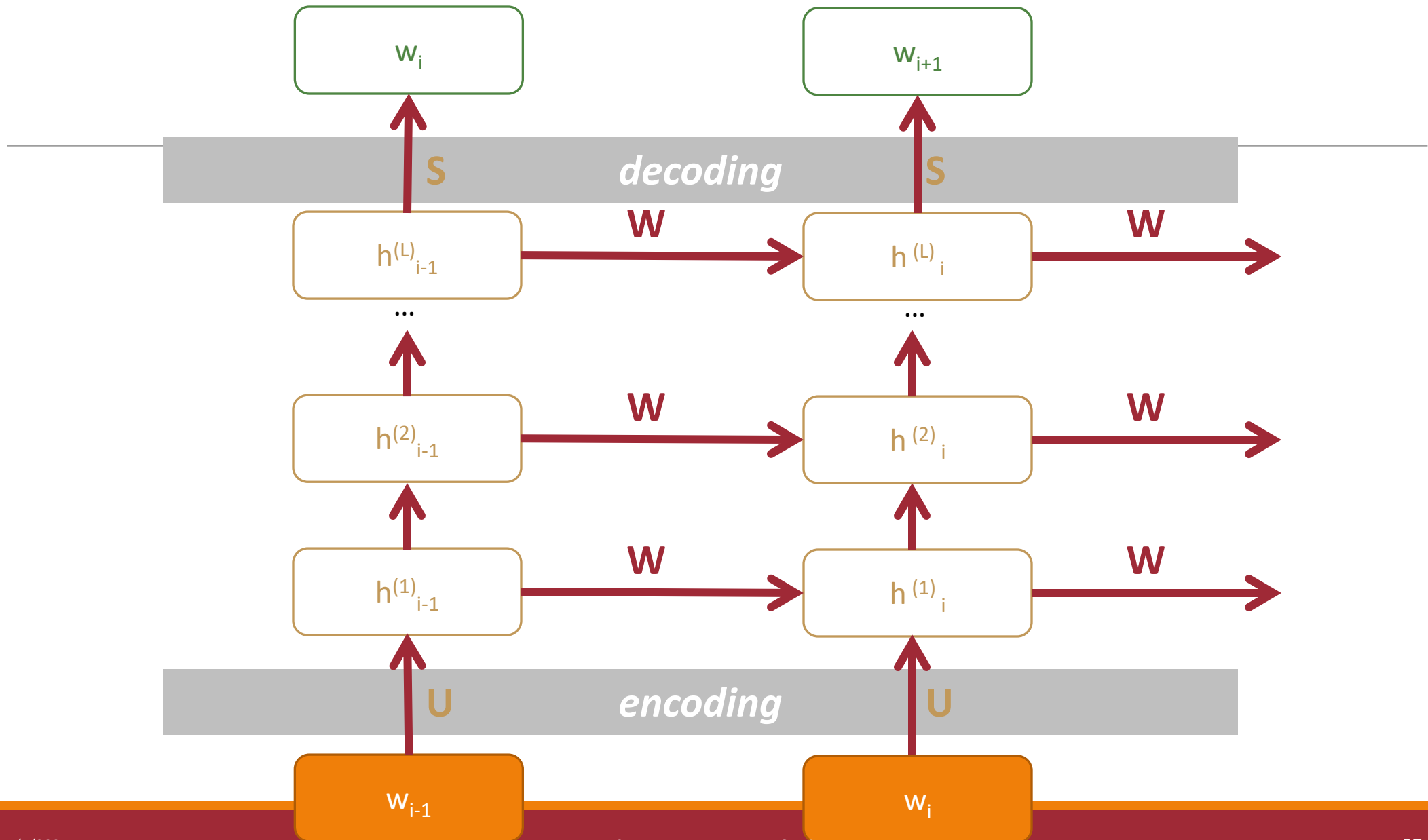
must learn *matrices*  $U, S, W$

suggested solution: gradient descent on prediction ability

problem: they're *tied* across inputs/timesteps

good news for you: many toolkits do this automatically

# A Multi-Layer Simple Recurrent Neural Network Cell



# How do you learn an RNN?

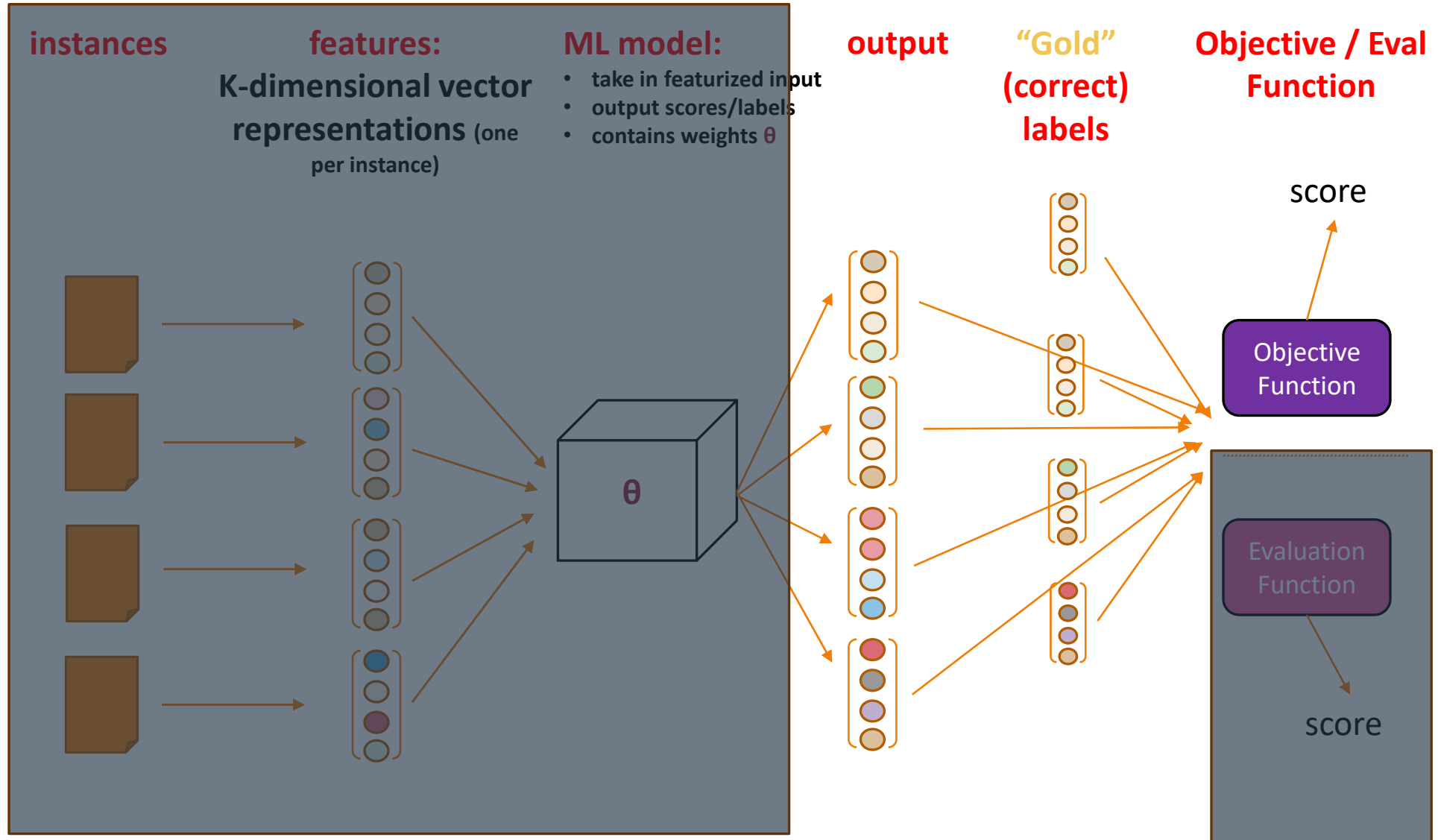
---

As with other approaches: Compute the loss and perform gradient descent

Loss: Cross-entropy, computed per output word

- Just as with prior LM approaches!

# Defining the Objective



# Review:

## Minimize Cross Entropy Loss

Classifier output

True probability (i.e., correct output)

$$L^{\text{xent}}(\hat{y}, y) = - \sum_{\text{label } k} \hat{y}[k] \log p(y = k|x)$$


index of "1" indicates correct value

one-hot vector

The diagram illustrates the components of the cross entropy loss function. The classifier output  $\hat{y}$  is shown as a vector of values. The true probability  $y$  is shown as a one-hot vector, where the index of the '1' indicates the correct value. The loss function is defined as  $L^{\text{xent}}(\hat{y}, y) = - \sum_{\text{label } k} \hat{y}[k] \log p(y = k|x)$ . Arrows point from the labels 'Classifier output' and 'True probability (i.e., correct output)' to the  $\hat{y}$  and  $y$  terms in the equation, respectively. Another arrow points from the label 'index of "1" indicates correct value' to the '1' in the one-hot vector, which is also labeled 'one-hot vector'.

**Cross entropy:**  
How much  $\hat{y}$  differs from the true  $y$

objective is convex  
(when  $f(x)$  is not learned)





# Gradient Descent: Backpropagate the Error

---

Initialize model

Set  $t = 0$

Pick a starting value  $\theta_t$

Until converged:

for example(s) sentence  $i$ :

1. Compute loss  $l$  on  $x_i$   
 $l = \text{model}(x_i)$
2. Get gradient  $g_t = l'(x_i)$
3. Get scaling factor  $\rho_t$
4. Set  $\theta_{t+1} = \theta_t - \rho_t * g_t$
5. Set  $t += 1$

**Core idea:** Train the model to predict what the next word is via maximum likelihood (equivalently, minimizing cross-entropy loss).

# Gradient Descent: Backpropagate the Error

---

Initialize model

Set  $t = 0$

Pick a starting value  $\theta_t$

Until converged:

for example(s) sentence  $i$ :

1. Compute loss  $l$  on  $x_i$

$$l = \text{model}(x_i)$$

2. Get gradient  $g_t = l'(x_i)$

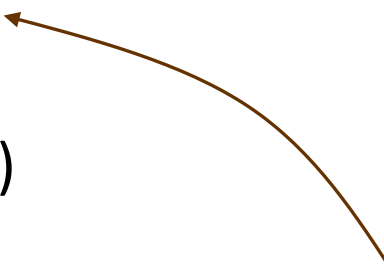
3. Get scaling factor  $\rho_t$

4. Set  $\theta_{t+1} = \theta_t - \rho_t * g_t$

5. Set  $t += 1$

**Core idea:** Train the model to predict what the next word is via maximum likelihood (equivalently, minimizing cross-entropy loss).

This **loss** is the sum of the per-token cross-entropy loss



# Recurrent NN Loss

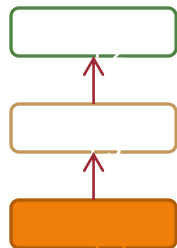
$\log .2$

word	prob.
The	.2
gray	.01
blue	.001
fluffy	.0005
wet	.0005
...	...

Remember: These probabilities are *computed* as a function of the model parameters!

The

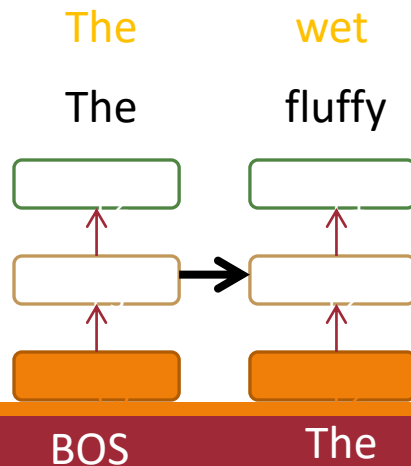
The



# Recurrent NN Loss

$\log.2 + \log.12$

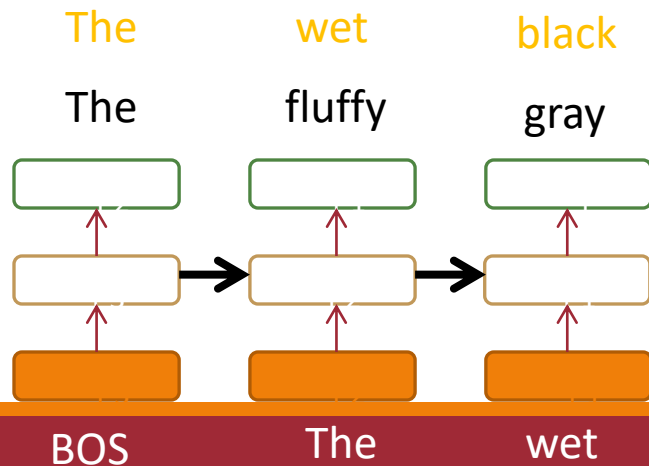
word	prob.	word	prob.
The	.2	black	.2
gray	.01	wet	.12
blue	.001	blue	.001
fluffy	.0005	fluffy	.0005
wet	.0005	gray	.0005
...	...	...	...



# Recurrent NN Loss

$$\log.2 + \log.12 + \log.2$$

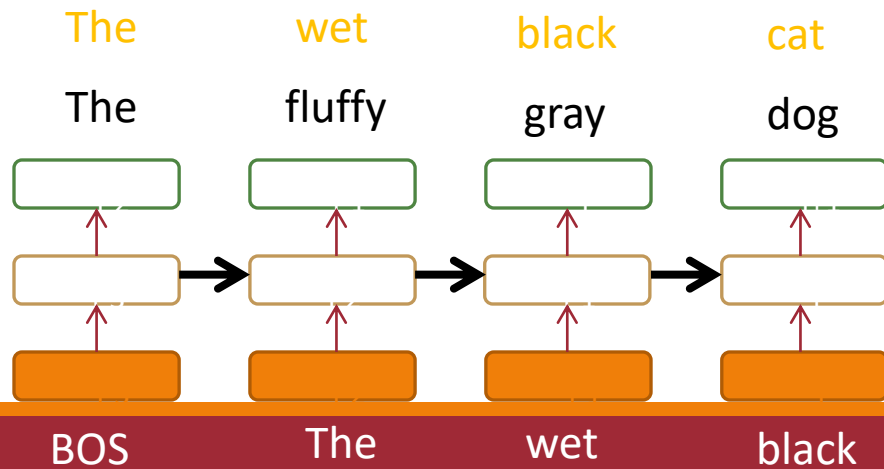
word	prob.	word	prob.	word	prob.
The	.2	black	.2	black	.2
gray	.01	wet	.12	gray	.01
blue	.001	blue	.001	blue	.001
fluffy	.0005	fluffy	.0005	bald	.0005
wet	.0005	gray	.0005	wet	.0005
...	...	...	...	...	...



# Recurrent NN Loss

$$\log.2 + \log.12 + \log.2 + \log.19$$

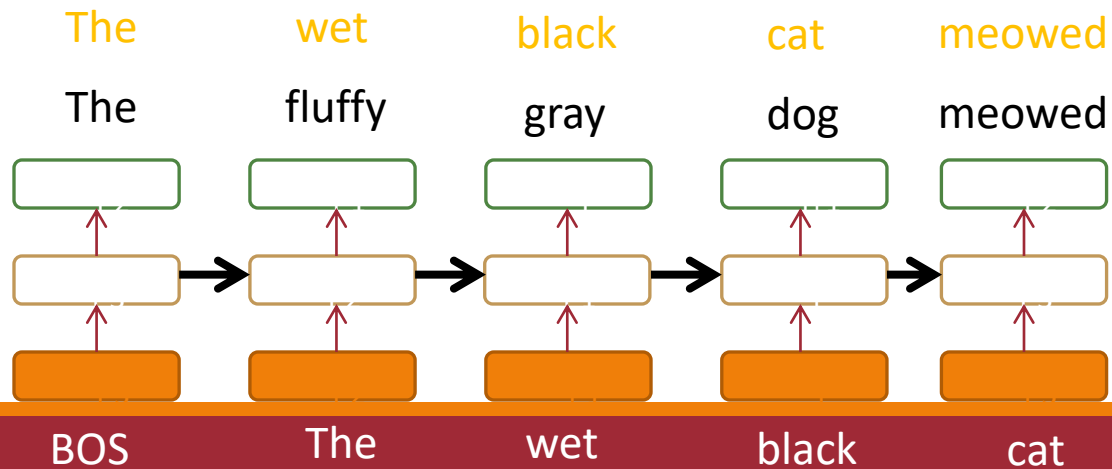
word	prob.	word	prob.	word	prob.	word	prob.
The	.2	black	.2	black	.2	dog	.2
gray	.01	wet	.12	gray	.01	cat	.19
blue	.001	blue	.001	blue	.001	blue	.001
fluffy	.0005	fluffy	.0005	bald	.0005	fluffy	.0005
wet	.0005	gray	.0005	wet	.0005	wet	.0005
...	...	...	...	...	...	...	...



# Recurrent NN Loss

$$\log.2 + \log.12 + \log.2 + \log.19 + \log.3$$

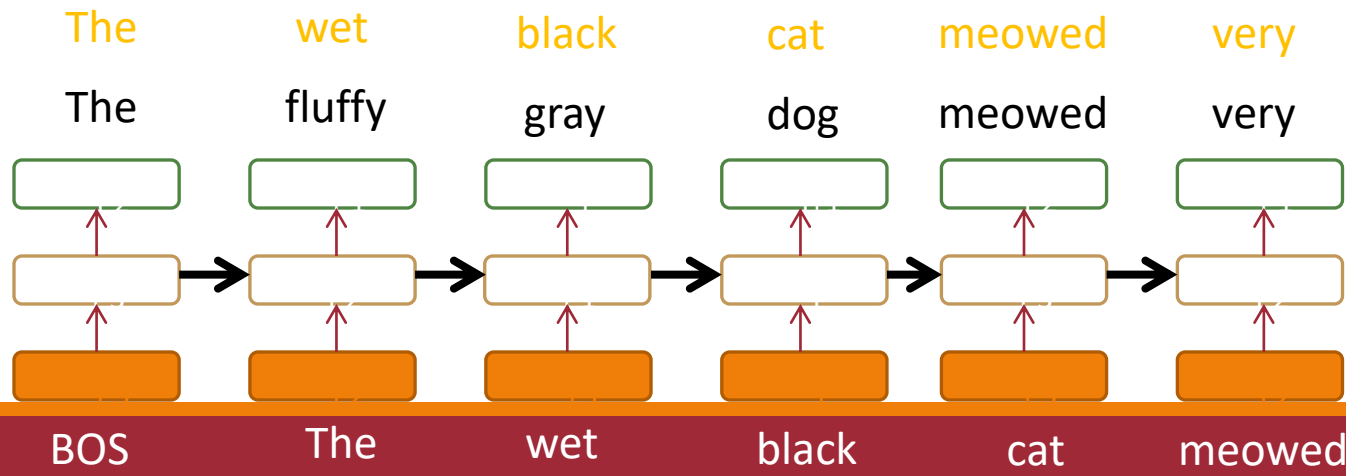
word	prob.	word	prob.	word	prob.	word	prob.	word	prob.
The	.2	black	.2	black	.2	dog	.2	meowed	.3
gray	.01	wet	.12	gray	.01	cat	.19	purred	.2
blue	.001	blue	.001	blue	.001	blue	.001	hissed	.1
fluffy	.0005	fluffy	.0005	bald	.0005	fluffy	.0005	fluffy	.001
wet	.0005	gray	.0005	wet	.0005	wet	.0005	wet	.001
...	...	...	...	...	...	...	...	...	...



# Recurrent NN Loss

$\log.2 + \log.12 + \log.2 + \log.19 + \log.3 + \log.2$

word	prob.	word	prob.	word	prob.	word	prob.	word	prob.	word	prob.
The	.2	black	.2	black	.2	dog	.2	meowed	.3	very	.2
gray	.01	wet	.12	gray	.01	cat	.19	purred	.2	lots	.1
blue	.001	blue	.001	blue	.001	blue	.001	hissed	.1	softly	.1
fluffy	.0005	fluffy	.0005	bald	.0005	fluffy	.0005	fluffy	.001	fluffy	.0005
wet	.0005	gray	.0005	wet	.0005	wet	.0005	wet	.001	wet	.0005
...	...	...	...	...	...	...	...	...	...	...	...



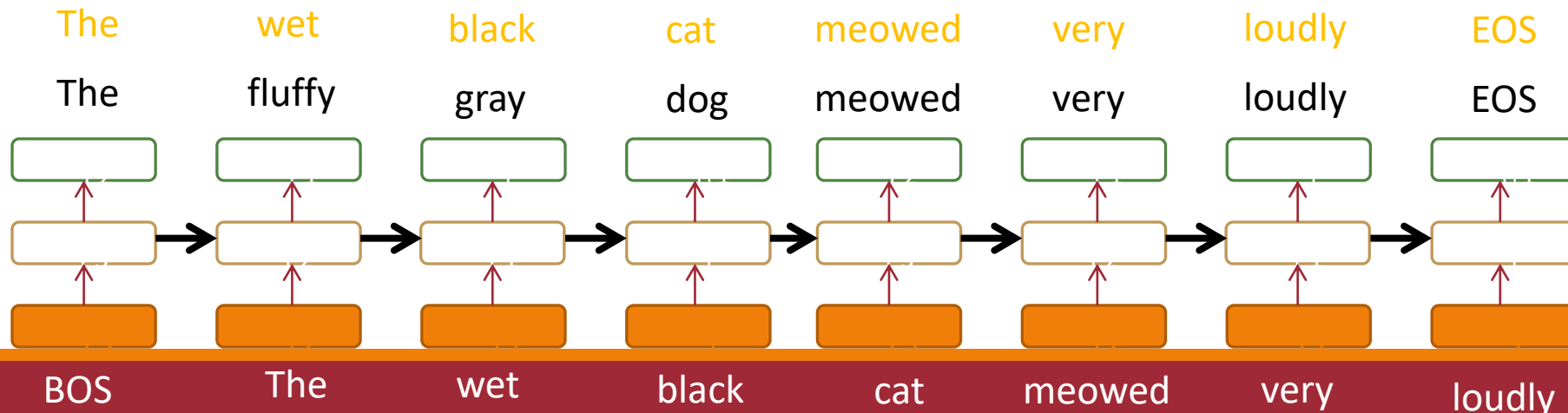


# Recurrent NN Loss

(then negate, average)

log.2 + log.12 + log.2 + log.19 + log.3 + log.2 + log.2 + log.2

word	prob.	word	prob.	word	prob.	word	prob.	word	prob.	word	prob.	word	prob.	word	prob.
The	.2	black	.2	black	.2	dog	.2	meowed	.3	very	.2	loudly	.2	EOS	.3
gray	.01	wet	.12	gray	.01	cat	.19	purred	.2	lots	.1	softly	.01	and	.1
blue	.001	blue	.001	blue	.001	blue	.001	hissed	.1	softly	.1	quiet	.001	blue	.001
fluffy	.0005	fluffy	.0005	bald	.0005	fluffy	.0005	fluffy	.001	fluffy	.0005	fluffy	.001	fluffy	.0005
wet	.0005	gray	.0005	wet	.0005	wet	.0005	wet	.001	wet	.0005	wet	.001	wet	.0005
...	...	...	...	...	...	...	...	...	...	...	...	...	...		



# Gradient Descent: Backpropagate the Error

Initialize model

Set  $t = 0$

Pick a starting value  $\theta_t$

Until converged:

for example(s) sentence  $i$ :

1. Compute loss  $l$  on  $x_i$   
 $l = \text{model}(x_i)$
2. Get gradient  $g_t = l'(x_i)$
3. Get scaling factor  $\rho_t$
4. Set  $\theta_{t+1} = \theta_t - \rho_t * g_t$
5. Set  $t += 1$

**Core idea:** Train the model to predict what the next word is via maximum likelihood (equivalently, minimizing cross-entropy loss).

This **loss** is the sum of the per-token cross-entropy loss

(then negate, average)

word	prob.	word	prob.	word	prob.	word	prob.	word	prob.	word	prob.	word	prob.
The	.2	black	.2	black	.2	dog	.2	meowed	.3	very	.2	loudly	.2
gray	.01	wet	.12	gray	.01	cat	.19	purred	.2	lots	.1	softly	.01
blue	.001	blue	.001	blue	.001	blue	.001	hissed	.1	softly	.1	quiet	.001
fluffy	.0005	fluffy	.0005	bald	.0005	fluffy	.0005	fluffy	.001	fluffy	.0005	fluffy	.001
wet	.0005	gray	.0005	wet	.0005	wet	.0005	wet	.001	wet	.0005	wet	.001
...	...	...	...	...	...	...	...	...	...	...	...	...	...

# Gradient Descent: Backpropagate the Error

---

Set  $t = 0$

Pick a starting value  $\theta_t$

Until converged: .....

for example(s) sentence  $i$ :

1. Compute loss  $l$  on  $x_i$
2. Get gradient  $g_t = l'(x_i)$
3. Get scaling factor  $\rho_t$
4. Set  $\theta_{t+1} = \theta_t - \rho_t * g_t$
5. Set  $t += 1$

(mini)batch

epoch

Think-pair-share: When would you want to use batches?

**epoch:** a single run over all training data

**(mini-)batch:** a run over a subset of the data

# Flavors of Gradient Descent

## “Online”

Set  $t = 0$   
Pick a starting value  $\theta_t$   
Until converged:

for example  $i$  in full data:

1. Compute loss  $l$  on  $x_i$
2. **Get** gradient  
 $g_t = l'(x_i)$
3. Get scaling factor  $\rho_t$
4. Set  $\theta_{t+1} = \theta_t - \rho_t * g_t$
5. Set  $t += 1$

*done*

## “Minibatch”

Set  $t = 0$   
Pick a starting value  $\theta_t$   
Until converged:

get batch  $B \subset$  full data  
set  $g_t = 0$   
for example(s)  $i$  in  $B$ :

1. Compute loss  $l$  on  $x_i$
2. **Accumulate** gradient  
 $g_t += l'(x_i)$

*done*  
Get scaling factor  $\rho_t$   
Set  $\theta_{t+1} = \theta_t - \rho_t * g_t$   
Set  $t += 1$

## “Batch”

Set  $t = 0$   
Pick a starting value  $\theta_t$   
Until converged:

set  $g_t = 0$   
for example(s)  $i$  in **full data**:

1. Compute loss  $l$  on  $x_i$
2. **Accumulate** gradient  
 $g_t += l'(x_i)$

*done*  
Get scaling factor  $\rho_t$   
Set  $\theta_{t+1} = \theta_t - \rho_t * g_t$   
Set  $t += 1$

# Why Is Training RNNs Hard?

---

Conceptually, it can get strange

But really getting the gradient just requires many applications of the chain rule for derivatives

# Why Is Training RNNs Hard?

---

Conceptually, it can get strange

But really getting the gradient just requires many applications of the chain rule for derivatives

Vanishing gradients

- Multiply the *same* matrices at *each* timestep → multiply *many* matrices in the gradients

# Why Is Training RNNs Hard?

---

Conceptually, it can get strange

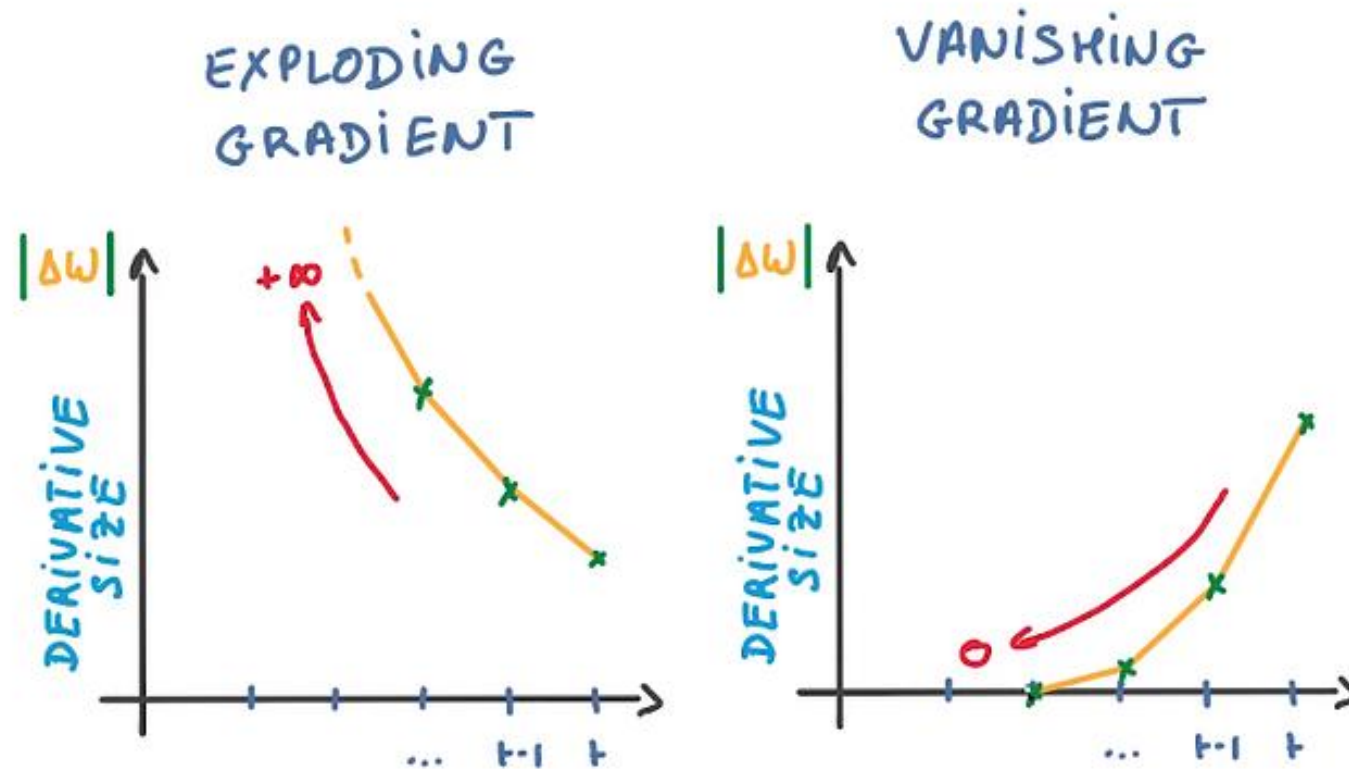
But really getting the gradient just requires many applications of the chain rule for derivatives

Vanishing gradients

- Multiply the *same* matrices at *each* timestep → multiply *many* matrices in the gradients
- Causes the network to forget information from many timesteps back

One solution: clip the gradients to a max value

# Vanishing Gradients



[https://miro.medium.com/v2/resize:fit:700/0\\*pp5wIxZW4zvD9UH](https://miro.medium.com/v2/resize:fit:700/0*pp5wIxZW4zvD9UH)



# PyTorch RNN LMs

---

# Pick Your Toolkit

---

<b>PyTorch</b>	MXNet
Deeplearning4j	<b>Torch</b>
<b>TensorFlow</b>	...
Caffe	
<b>Keras</b>	

Comparisons:

[https://en.wikipedia.org/wiki/Comparison\\_of\\_deep\\_learning\\_software](https://en.wikipedia.org/wiki/Comparison_of_deep_learning_software)

# Defining A Simple RNN in Python

[http://pytorch.org/tutorials/intermediate/char\\_rnn\\_classification\\_tutorial.html](http://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html)

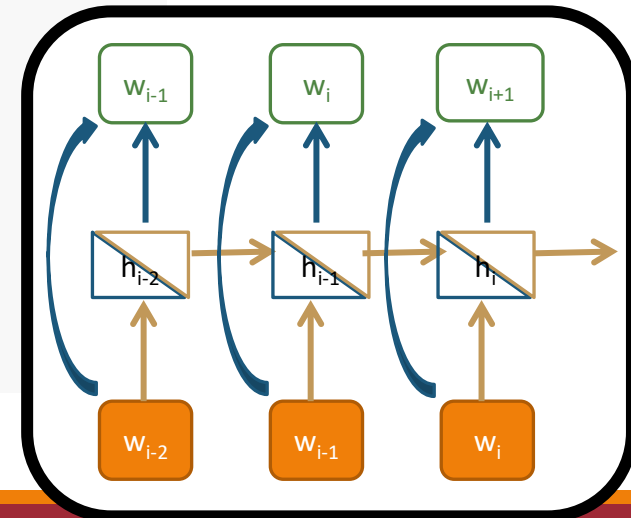
```
import torch.nn as nn
import torch.nn.functional as F

class CharRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(CharRNN, self).__init__()

        self.rnn = nn.RNN(input_size, hidden_size)
        self.h2o = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, line_tensor):
        rnn_out, hidden = self.rnn(line_tensor)
        output = self.h2o(hidden[0])
        output = self.softmax(output)

        return output
```



# Defining A Simple RNN in Python

[http://pytorch.org/tutorials/intermediate/char\\_rnn\\_classification\\_tutorial.html](http://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html)

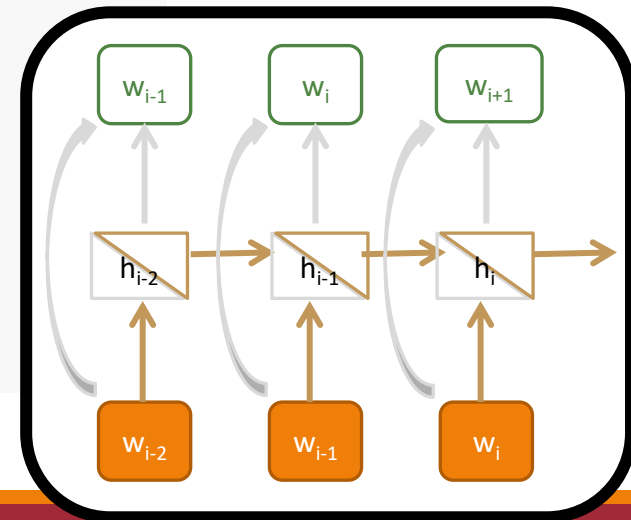
```
import torch.nn as nn
import torch.nn.functional as F

class CharRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(CharRNN, self).__init__()

        self.rnn = nn.RNN(input_size, hidden_size)
        self.h2o = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, line_tensor):
        rnn_out, hidden = self.rnn(line_tensor)
        output = self.h2o(hidden[0])
        output = self.softmax(output)

        return output
```



# Defining A Simple RNN in Python

[http://pytorch.org/tutorials/intermediate/char\\_rnn\\_classification\\_tutorial.html](http://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html)

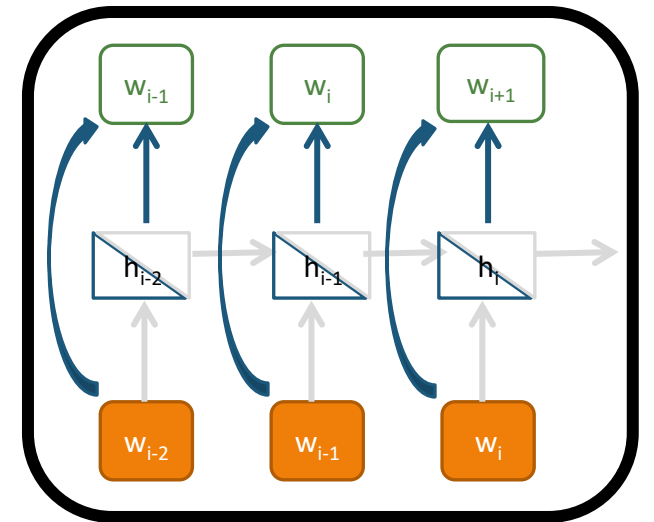
```
import torch.nn as nn
import torch.nn.functional as F

class CharRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(CharRNN, self).__init__()

        self.rnn = nn.RNN(input_size, hidden_size)
        self.h2o = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, line_tensor):
        rnn_out, hidden = self.rnn(line_tensor)
        output = self.h2o(hidden[0])
        output = self.softmax(output)

        return output
```



# Defining A Simple RNN in Python

[http://pytorch.org/tutorials/intermediate/char\\_rnn\\_classification\\_tutorial.html](http://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html)

```
import torch.nn as nn
import torch.nn.functional as F

class CharRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(CharRNN, self).__init__()

        self.rnn = nn.RNN(input_size, hidden_size)
        self.h2o = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)
```

```
def forward(self,
            rnn_out, hidden)
    output = self
    output = self
    return output
```

## SOFTMAX

CLASS torch.nn.Softmax(*dim=None*) [SOURCE]

Applies the Softmax function to an n-dimensional input Tensor rescaling them so that the elements of the n-dimensional output Tensor lie in the range [0,1] and sum to 1.

Softmax is defined as:

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

When the input Tensor is a sparse tensor then the unspecified values are treated as `-inf`.

# Defining A Simple RNN in Python

[http://pytorch.org/tutorials/intermediate/char\\_rnn\\_classification\\_tutorial.html](http://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html)

```
import torch.nn as nn
import torch.nn.functional as F

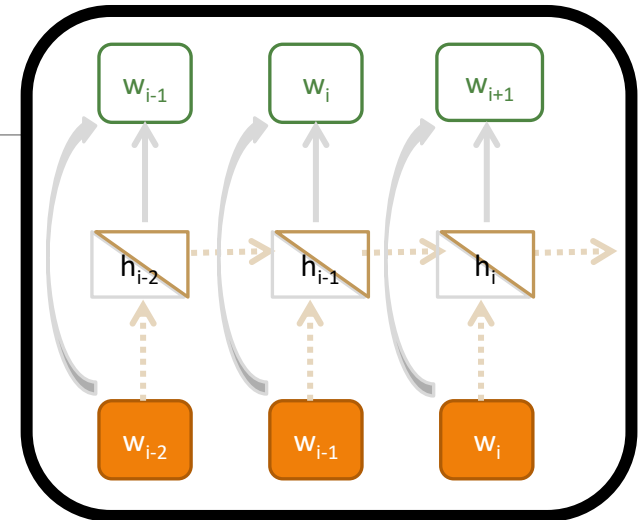
class CharRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(CharRNN, self).__init__()

        self.rnn = nn.RNN(input_size, hidden_size)
        self.h2o = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, line_tensor):
        rnn_out, hidden = self.rnn(line_tensor)
        output = self.h2o(hidden[0])
        output = self.softmax(output)

        return output
```

encode



# Defining A Simple RNN in Python

[http://pytorch.org/tutorials/intermediate/char\\_rnn\\_classification\\_tutorial.html](http://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html)

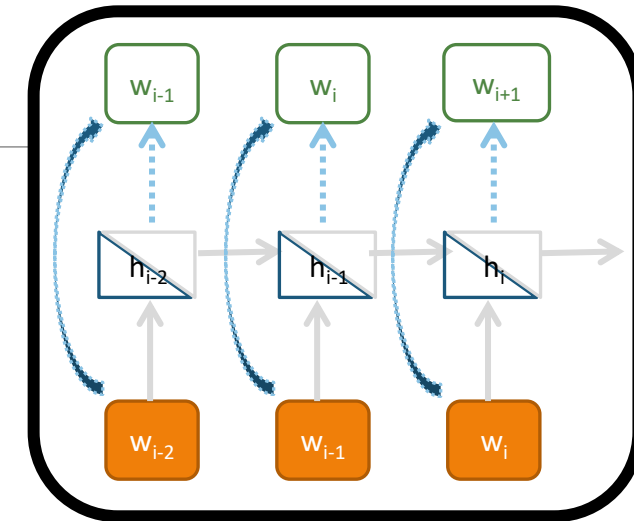
```
import torch.nn as nn
import torch.nn.functional as F

class CharRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(CharRNN, self).__init__()

        self.rnn = nn.RNN(input_size, hidden_size)
        self.h2o = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, line_tensor):
        rnn_out, hidden = self.rnn(line_tensor)
        output = self.h2o(hidden[0])
        output = self.softmax(output)

        return output
```



decode



# Training A Simple RNN in Python

Negative log-likelihood

(we'll talk about this)

```
def train(rnn, training_data, n_epoch = 10, n_batch_size = 64, report_every = 50, learning_rate = 0.2, criterion = nn.NLLLoss()):  
    """  
    Learn on a batch of training_data for a specified number of iterations and reporting thresholds  
    """  
    # Keep track of losses for plotting  
    current_loss = 0  
    all_losses = []  
    rnn.train()  
    optimizer = torch.optim.SGD(rnn.parameters(), lr=learning_rate)  
  
    start = time.time()  
    print(f"training on data set with n = {len(training_data)}")
```

Set learning rate and type of optimizer

[http://pytorch.org/tutorials/intermediate/char\\_rnn\\_classification\\_tutorial.html](http://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html)

# Training A Simple RNN in Python

[http://pytorch.org/tutorials/intermediate/char\\_rnn\\_classification\\_tutorial.html](http://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html)

```
for iter in range(1, n_epoch + 1):
    rnn.zero_grad() # clear the gradients

    # create some minibatches
    # we cannot use dataloaders because each of our names is a different length
    batches = list(range(len(training_data)))
    random.shuffle(batches)
    batches = np.array_split(batches, len(batches) // n_batch_size )

    for idx, batch in enumerate(batches):
        batch_loss = 0
        for i in batch: #for each example in this batch
            (label_tensor, text_tensor, label, text) = training_data[i]
            output = rnn.forward(text_tensor)
            loss = criterion(output, label_tensor)
            batch_loss += loss

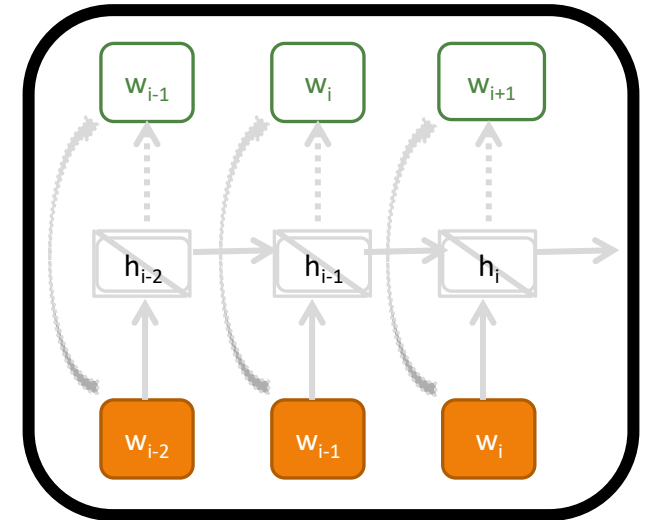
            # optimize parameters
            batch_loss.backward()
            nn.utils.clip_grad_norm_(rnn.parameters(), 3)
            optimizer.step()
            optimizer.zero_grad()

            current_loss += batch_loss.item() / len(batch)

        all_losses.append(current_loss / len(batches) )
        if iter % report_every == 0:
            print(f"{iter} ( {iter / n_epoch:.0%} ): \t average batch loss = {all_losses[-1]}")
        current_loss = 0

    return all_losses
```

get predictions



# Training A Simple RNN in Python

[http://pytorch.org/tutorials/intermediate/char\\_rnn\\_classification\\_tutorial.html](http://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html)

```
for iter in range(1, n_epoch + 1):
    rnn.zero_grad() # clear the gradients

    # create some minibatches
    # we cannot use dataloaders because each of our names is a different length
    batches = list(range(len(training_data)))
    random.shuffle(batches)
    batches = np.array_split(batches, len(batches) // n_batch_size )

    for idx, batch in enumerate(batches):
        batch_loss = 0
        for i in batch: #for each example in this batch
            (label_tensor, text_tensor, label, text) = training_data[i]
            output = rnn.forward(text_tensor)
            loss = criterion(output, label_tensor)
            batch_loss += loss

            # optimize parameters
            batch_loss.backward()
            nn.utils.clip_grad_norm_(rnn.parameters(), 3)
            optimizer.step()
            optimizer.zero_grad()

            current_loss += batch_loss.item() / len(batch)

    all_losses.append(current_loss / len(batches) )
    if iter % report_every == 0:
        print(f"{iter} ({iter} / n_epoch:.0%): \t average batch loss = {all_losses[-1]}")
    current_loss = 0

return all_losses
```

get predictions

eval predictions

$$L^{\text{xent}}(\hat{y}, y) = - \sum_{\text{label } k} \hat{y}[k] \log p(y = k|x)$$

Set  $t = 0$

Pick a starting value  $\theta_t$

Until converged:

for example(s) sentence  $i$ :

1. Compute loss  $l$  on  $x_i$
2. Get gradient  $g_t = l'(x_i)$
3. Get scaling factor  $\rho_t$
4. Set  $\theta_{t+1} = \theta_t - \rho_t * g_t$
5. Set  $t += 1$

# Training A Simple RNN in Python

[http://pytorch.org/tutorials/intermediate/char\\_rnn\\_classification\\_tutorial.html](http://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html)

```
for iter in range(1, n_epoch + 1):
    rnn.zero_grad() # clear the gradients

    # create some minibatches
    # we cannot use dataloaders because each of our names is a different length
    batches = list(range(len(training_data)))
    random.shuffle(batches)
    batches = np.array_split(batches, len(batches) // n_batch_size )

    for idx, batch in enumerate(batches):
        batch_loss = 0
        for i in batch: #for each example in this batch
            (label_tensor, text_tensor, label, text) = training_data[i]
            output = rnn.forward(text_tensor)
            loss = criterion(output, label_tensor)
            batch_loss += loss

            # optimize parameters
            batch_loss.backward()
            nn.utils.clip_grad_norm_(rnn.parameters(), 3)
            optimizer.step()
            optimizer.zero_grad()

            current_loss += batch_loss.item() / len(batch)

    all_losses.append(current_loss / len(batches) )
    if iter % report_every == 0:
        print(f"{iter} ({iter / n_epoch:.0%}): \t average batch loss = {all_losses[-1]}")
    current_loss = 0

return all_losses
```

get predictions

eval predictions

compute gradient

- Set  $t = 0$   
Pick a starting value  $\theta_t$   
Until converged:  
for example(s) sentence  $i$ :
1. Compute loss  $l$  on  $x_i$
  2. Get gradient  $g_t = l'(x_i)$
  3. Get scaling factor  $\rho_t$
  4. Set  $\theta_{t+1} = \theta_t - \rho_t * g_t$
  5. Set  $t += 1$

# Training A Simple RNN in Python

[http://pytorch.org/tutorials/intermediate/char\\_rnn\\_classification\\_tutorial.html](http://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html)

```
for iter in range(1, n_epoch + 1):
    rnn.zero_grad() # clear the gradients

    # create some minibatches
    # we cannot use dataloaders because each of our names is a different length
    batches = list(range(len(training_data)))
    random.shuffle(batches)
    batches = np.array_split(batches, len(batches) // n_batch_size )

    for idx, batch in enumerate(batches):
        batch_loss = 0
        for i in batch: #for each example in this batch
            (label_tensor, text_tensor, label, text) = training_data[i]
            output = rnn.forward(text_tensor)
            loss = criterion(output, label_tensor)
            batch_loss += loss

            # optimize parameters
            batch_loss.backward()
            nn.utils.clip_grad_norm_(rnn.parameters(), 3)
            optimizer.step()
            optimizer.zero_grad()

        current_loss += batch_loss.item() / len(batch)

    all_losses.append(current_loss / len(batches) )
    if iter % report_every == 0:
        print(f"{iter} ({iter} / n_epoch:.0%): \t average batch loss = {all_losses[-1]}")
    current_loss = 0

return all_losses
```

get predictions

eval predictions

compute gradient

perform SGD

Set  $t = 0$   
Pick a starting value  $\theta_t$   
Until converged:  
for example(s) sentence  $i$ :

1. Compute loss  $l$  on  $x_i$
2. Get gradient  $g_t = l'(x_i)$
3. Get scaling factor  $\rho_t$
4. Set  $\theta_{t+1} = \theta_t - \rho_t * g_t$
5. Set  $t += 1$

# Suggested Implementation Changes

```
import torch.nn as nn
import torch.nn.functional as F
```

```
class CharRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(CharRNN, self).__init__()

        self.rnn = nn.RNN(input_size, hidden_size)
        self.h2o = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, line_tensor):
        rnn_out, hidden = self.rnn(line_tensor)
        output = self.h2o(hidden[0])
output = self.softmax(output)

        return output
```

current Pytorch refers  
to this a "cell"

PyTorch's  
CrossEntropyLoss  
does a softmax  
and then takes  
the log

```
def train(rnn, training_data, = 50, learning_rate =
0.2, criterion = nn.NLLLoss() nn.CrossEntropyLoss())
    """
    Learn on a batch of training_data for a specified number of iterations and reporting thresholds
    """
    # Keep track of losses for plotting
    current_loss = 0
    all_losses = []
    rnn.train()
    optimizer = torch.optim.SGD(rnn.parameters(), lr=learning_rate)

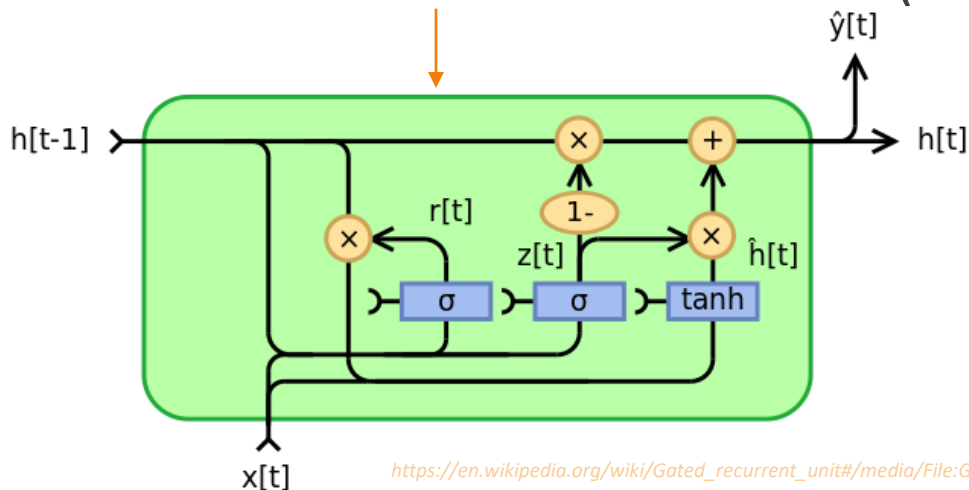
    start = time.time()
    print(f"training on data set with n = {len(training_data)}")
```

# Another Solution: LSTMs/GRUs

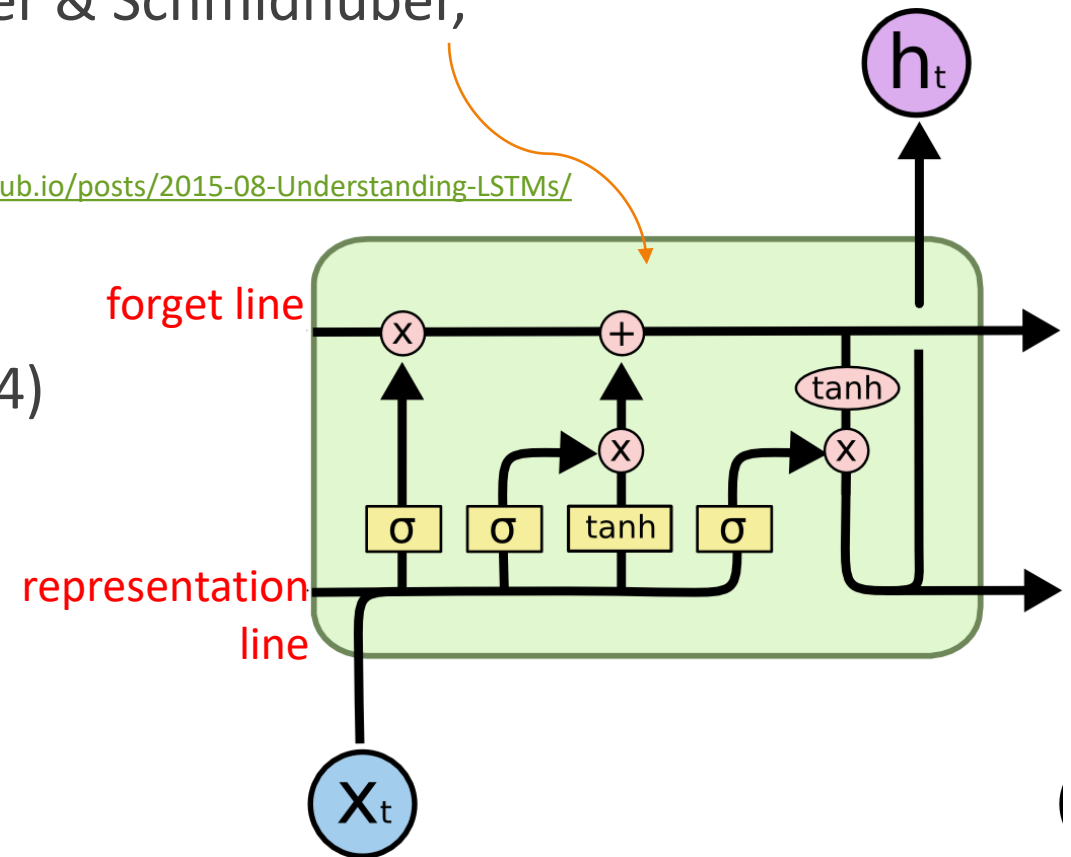
LSTM: Long Short-Term Memory (Hochreiter & Schmidhuber, 1997)

Basic Ideas: *learn to forget* <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

GRU: Gated Recurrent Unit (Cho et al., 2014)



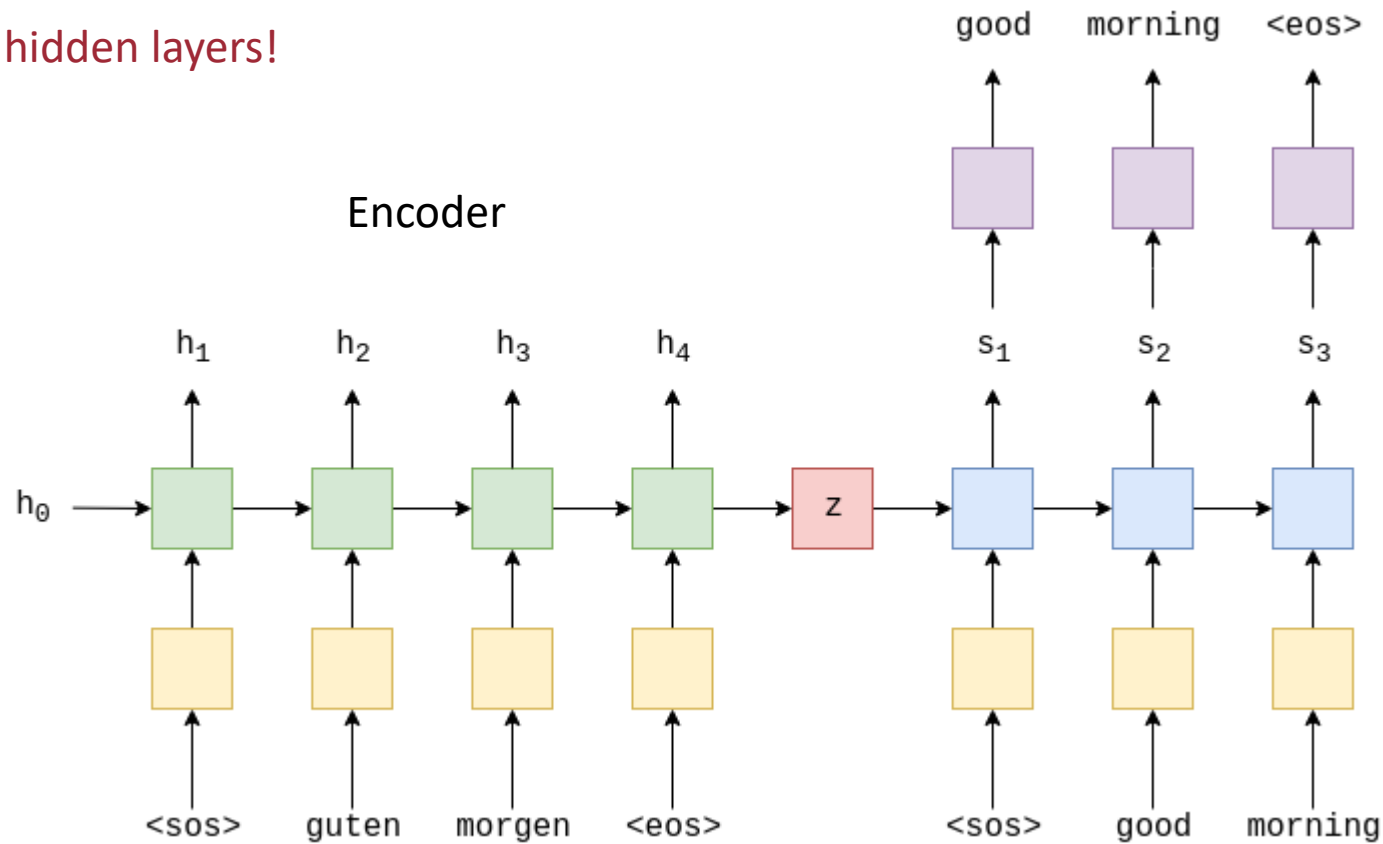
[https://en.wikipedia.org/wiki/Gated\\_recurrent\\_unit#/media/File:Gated\\_Recurrent\\_Unit\\_base\\_type.svg](https://en.wikipedia.org/wiki/Gated_recurrent_unit#/media/File:Gated_Recurrent_Unit_base_type.svg)



# Sequence-to-Sequence

Decoder

Note that this still has hidden layers!

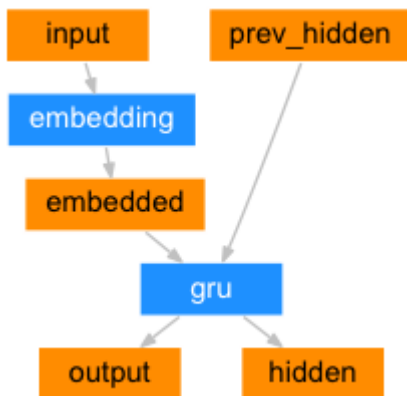


<https://colab.research.google.com/github/bentrevett/pytorch-seq2seq/blob/main/1%20-%20Sequence%20to%20Sequence%20Learning%20with%20Neural%20Networks.ipynb#scrollTo=k6sRrL4wKsmi>



# Encoder

---



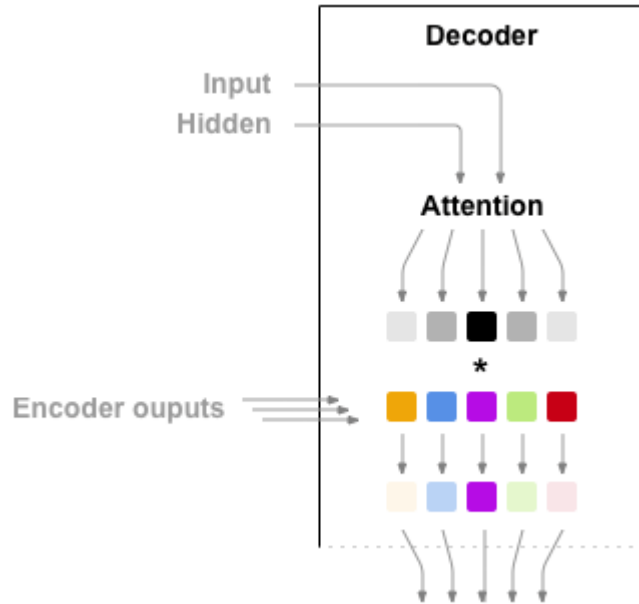
```
class EncoderRNN(nn.Module):
    def __init__(self, input_size, hidden_size, dropout_p=0.1):
        super(EncoderRNN, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(input_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size, batch_first=True)
        self.dropout = nn.Dropout(dropout_p)

    def forward(self, input):
        embedded = self.dropout(self.embedding(input))
        output, hidden = self.gru(embedded)
        return output, hidden
```

[https://pytorch.org/tutorials/intermediate/seq2seq\\_translation\\_tutorial.html](https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html)

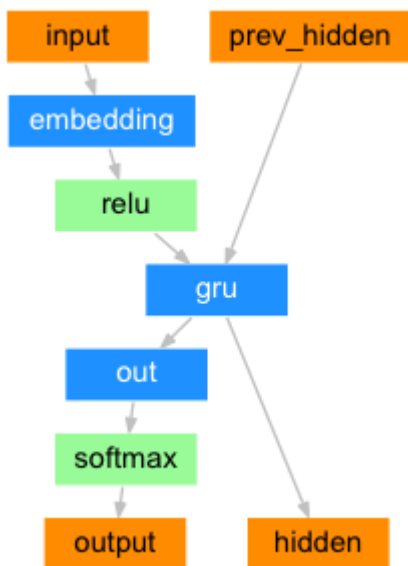
# Attention



```
class BahdanauAttention(nn.Module):  
    def __init__(self, hidden_size):  
        super(BahdanauAttention, self).__init__()  
        self.Wa = nn.Linear(hidden_size, hidden_size)  
        self.Ua = nn.Linear(hidden_size, hidden_size)  
        self.Va = nn.Linear(hidden_size, 1)  
  
    def forward(self, query, keys):  
        scores = self.Va(torch.tanh(self.Wa(query) + self.Ua(keys)))  
        scores = scores.squeeze(2).unsqueeze(1)  
  
        weights = F.softmax(scores, dim=-1)  
        context = torch.bmm(weights, keys)  
  
        return context, weights
```

[https://pytorch.org/tutorials/intermediate/seq2seq\\_translation\\_tutorial.html](https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html)

# Decoder



```
class DecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size):
        super(DecoderRNN, self).__init__()
        self.embedding = nn.Embedding(output_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size, batch_first=True)
        self.out = nn.Linear(hidden_size, output_size)

    def forward(self, encoder_outputs, encoder_hidden, target_tensor=None):
        batch_size = encoder_outputs.size(0)
        decoder_input = torch.empty(batch_size, 1, dtype=torch.long,
device=device).fill_(SOS_token)
        decoder_hidden = encoder_hidden
        decoder_outputs = []

        for i in range(MAX_LENGTH):
            decoder_output, decoder_hidden = self.forward_step(decoder_input, decoder_hidden)
            decoder_outputs.append(decoder_output)

            if target_tensor is not None:
                # Teacher forcing: Feed the target as the next input
                decoder_input = target_tensor[:, i].unsqueeze(1) # Teacher forcing
            else:
                # Without teacher forcing: use its own predictions as the next input
                _, topi = decoder_output.topk(1)
                decoder_input = topi.squeeze(-1).detach() # detach from history as input

        decoder_outputs = torch.cat(decoder_outputs, dim=1)
        decoder_outputs = F.log_softmax(decoder_outputs, dim=-1)
        return decoder_outputs, decoder_hidden, None # We return 'None' for consistency in the
training loop


    def forward_step(self, input, hidden):
        output = self.embedding(input)
        output = F.relu(output)
        output, hidden = self.gru(output, hidden)
        output = self.out(output)
        return output, hidden
```


[https://pytorch.org/tutorials/intermediate/seq2seq\\_translation\\_tutorial.html](https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html)


# Seq2Seq Tutorial

You will need to download the data. You can run  
`!wget https://download.pytorch.org/tutorial/data.zip`  
`!unzip data.zip`

[https://pytorch.org/tutorials/intermediate/seq2seq\\_translation\\_tutorial.html](https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html)

 Run in Google Colab

 Download Notebook

 View on GitHub



## NLP From Scratch: Translation with a Sequence to Sequence Network and Attention

Created On: Mar 24, 2017 | Last Updated: Oct 21, 2024 | Last Verified: Nov 05, 2024

**Author:** [Sean Robertson](#)

This tutorial is part of a three-part series:

- [NLP From Scratch: Classifying Names with a Character-Level RNN](#)
- [NLP From Scratch: Generating Names with a Character-Level RNN](#)
- [NLP From Scratch: Translation with a Sequence to Sequence Network and Attention](#)

This is the third and final tutorial on doing **NLP From Scratch**, where we write our own classes and functions to preprocess the data to do our NLP modeling tasks.

Alternative tutorial that shows loss plotted in real time but uses a different NN library:

[https://d2l.ai/chapter\\_recurrent-modern/seq2seq.html](https://d2l.ai/chapter_recurrent-modern/seq2seq.html)