

Recurrent Neural Networks

CMSC 473/673 - NATURAL LANGUAGE PROCESSING

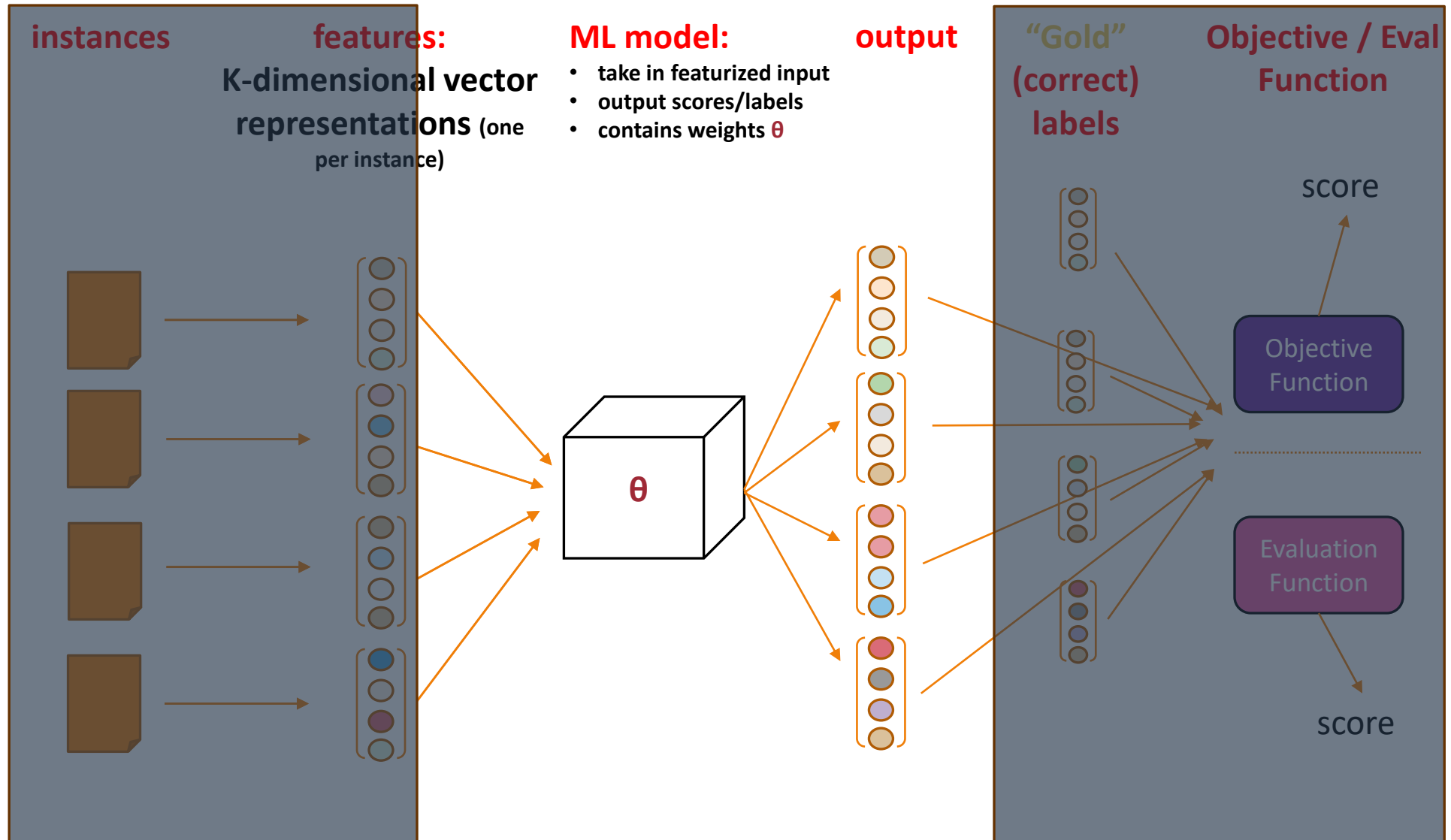
Slides modified from Dr. Frank Ferraro

Learning Objectives

Define the basic cell architecture of an RNN

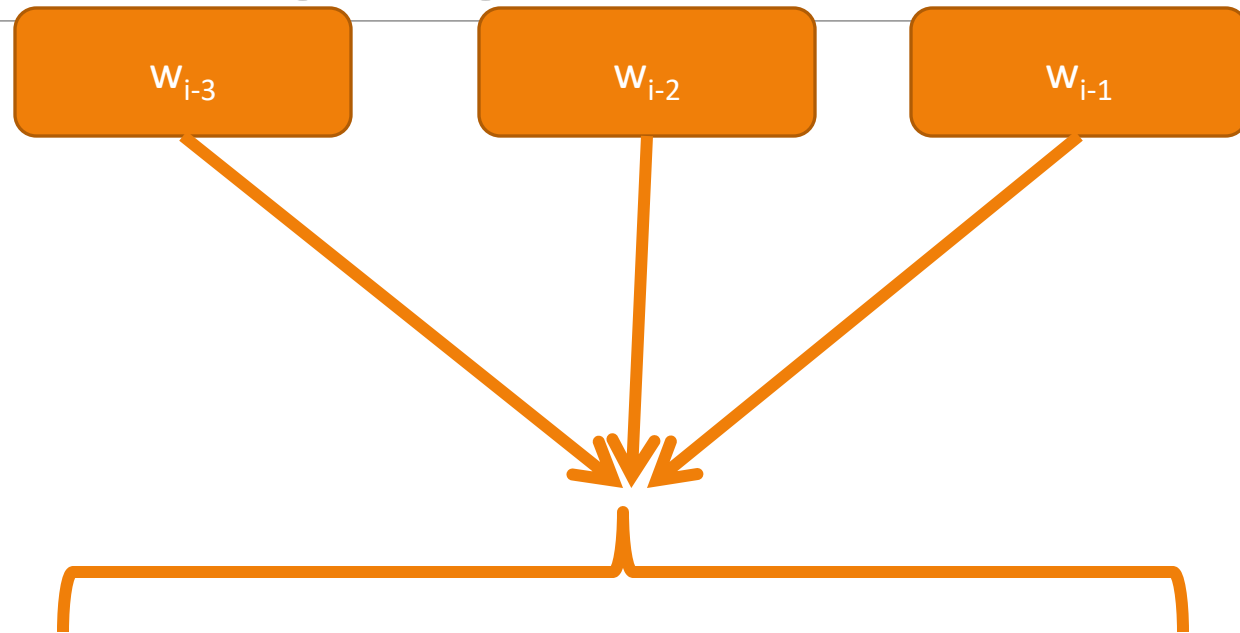
Backpropagate loss through an example RNN

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

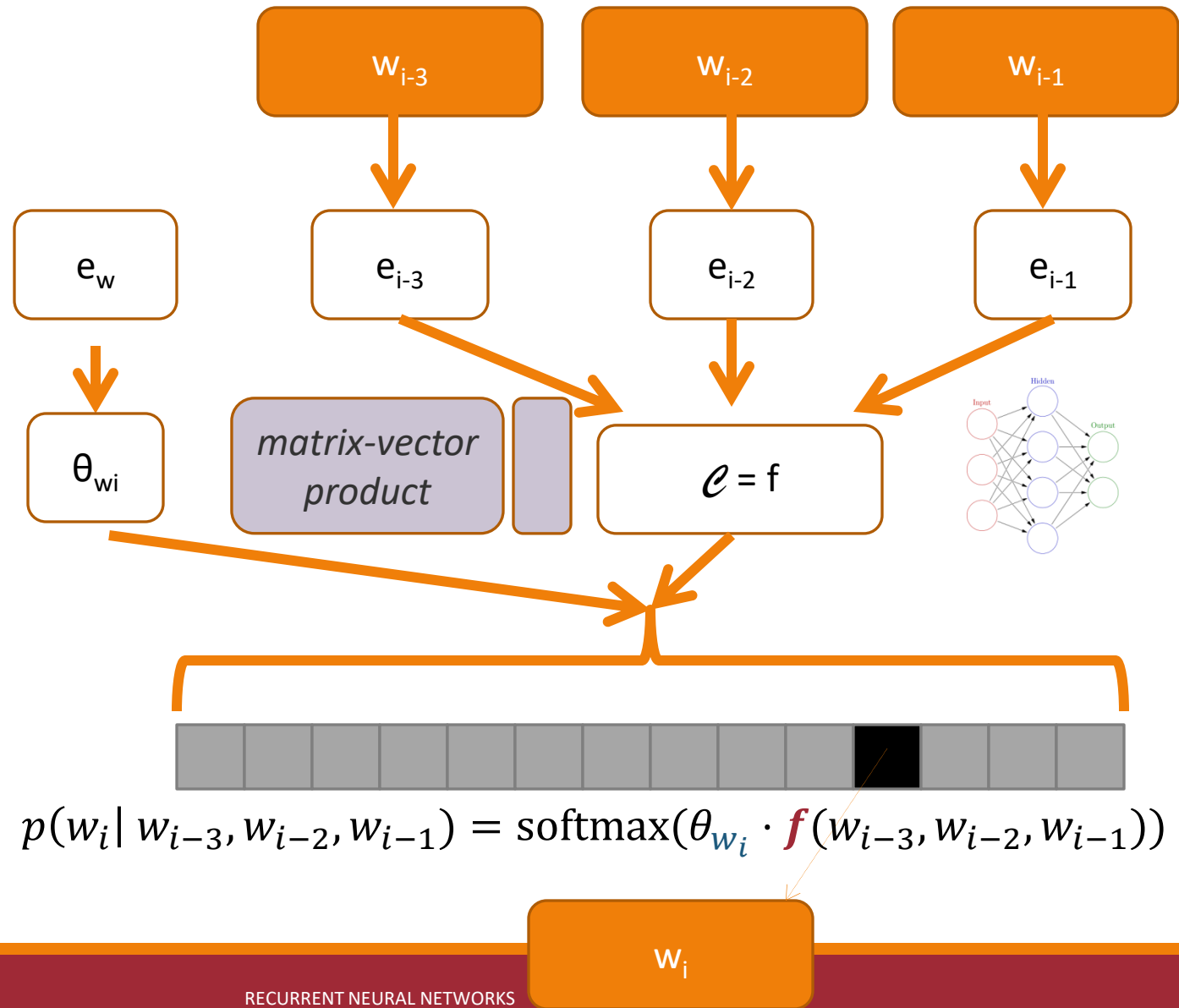
given some context...

*create/use
“distributed
representations”...*

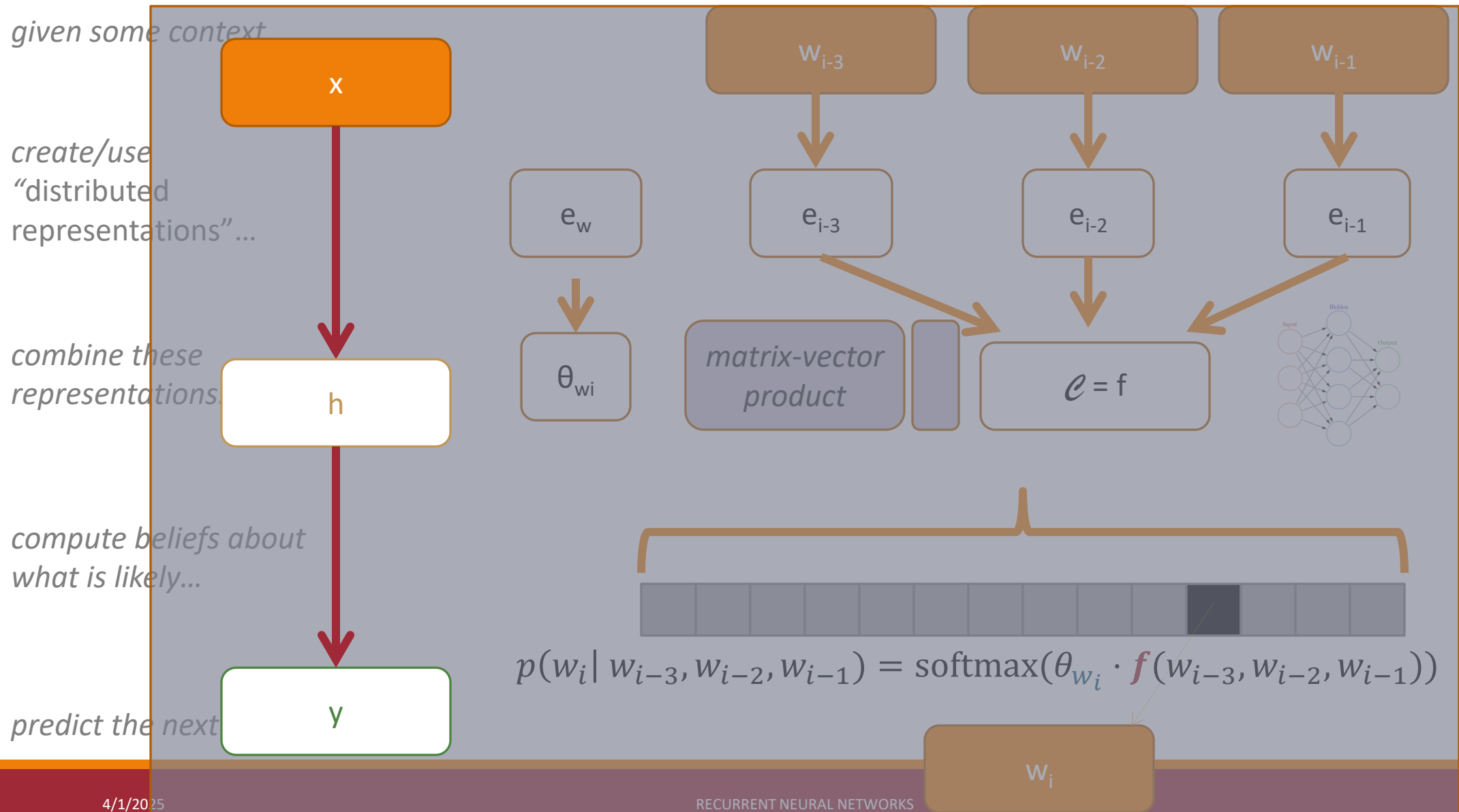
*combine these
representations...*

*compute beliefs about
what is likely...*

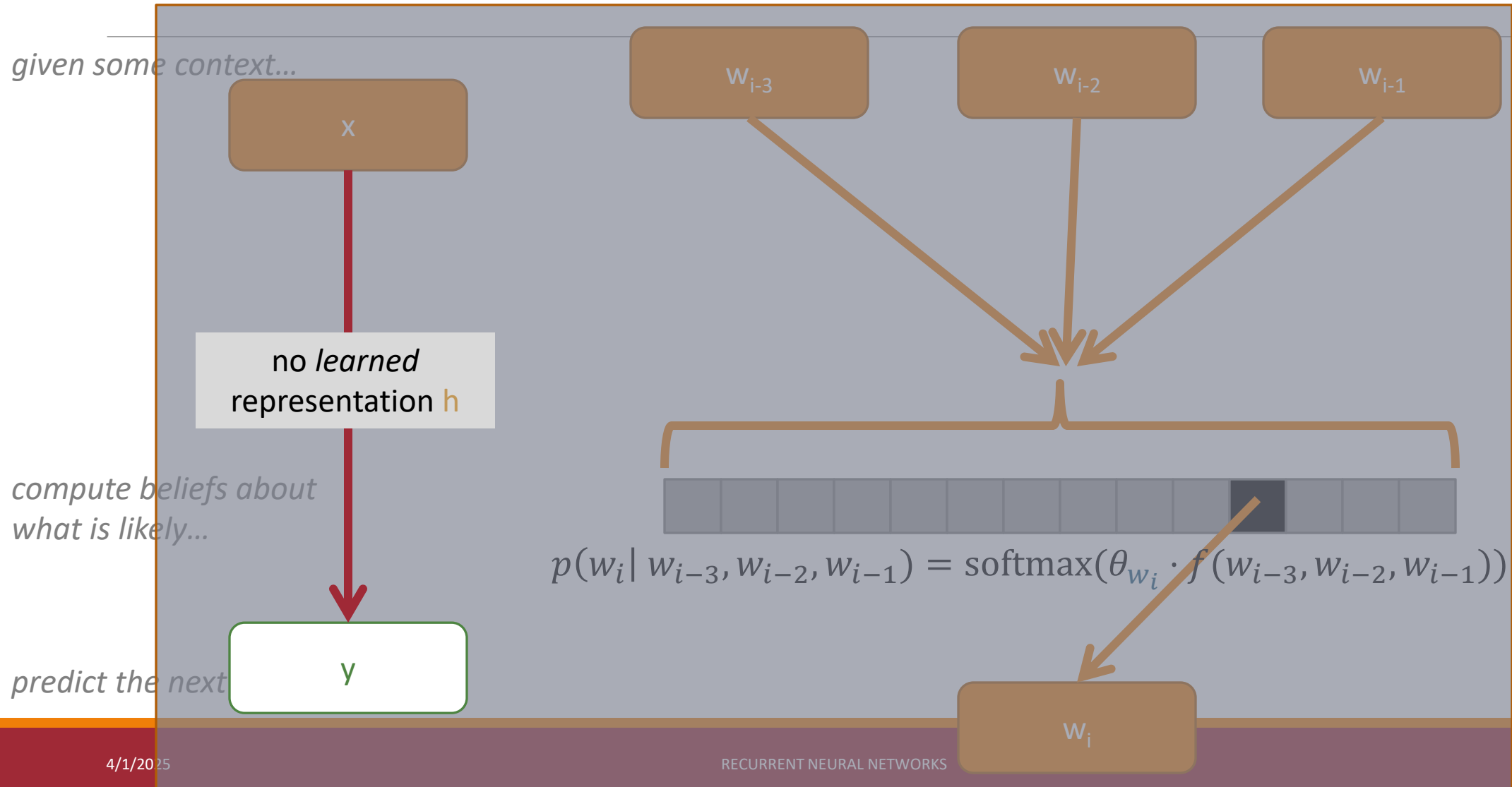
predict the next word



Review: Neural Language Models



Review: Maxent Language Models



Review: LM Comparison

COUNT-BASED

Class-specific

MAXENT

Class-based

Uses features

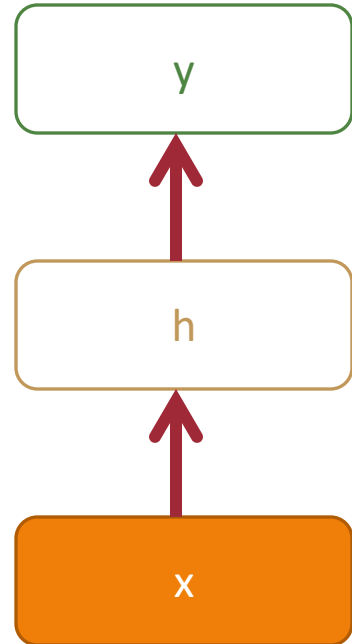
NEURAL

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

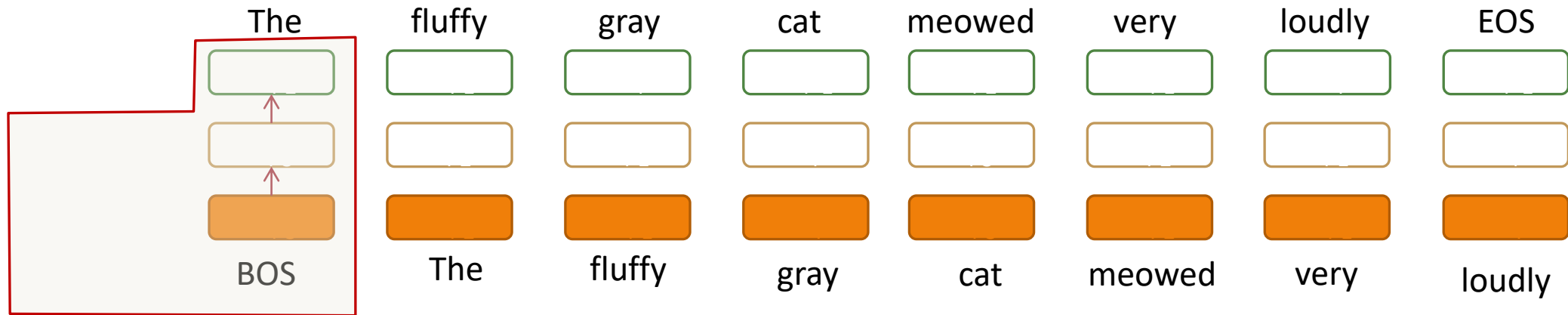
Review: A Neural N-Gram Model

The fluffy gray cat meowed very loudly



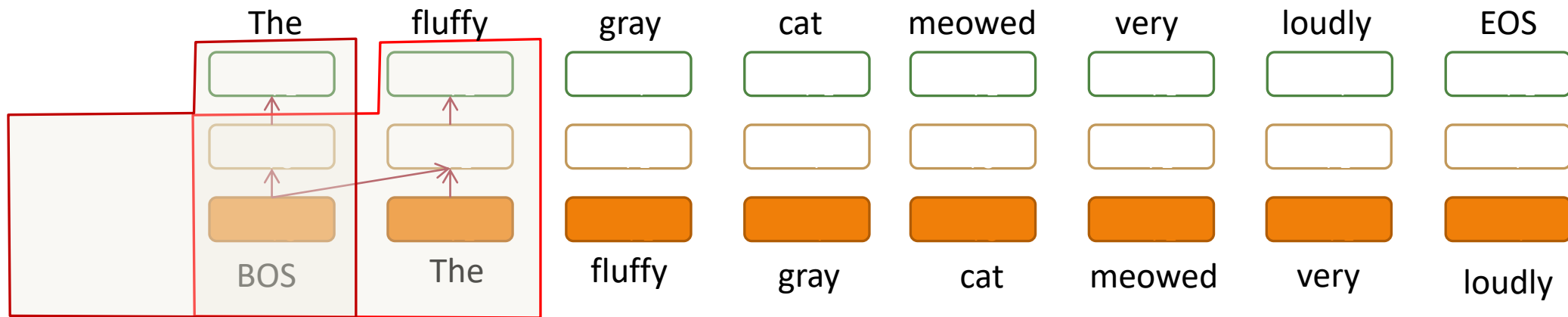
Review: A Neural N-Gram Model (N=3)

The fluffy gray cat meowed very loudly



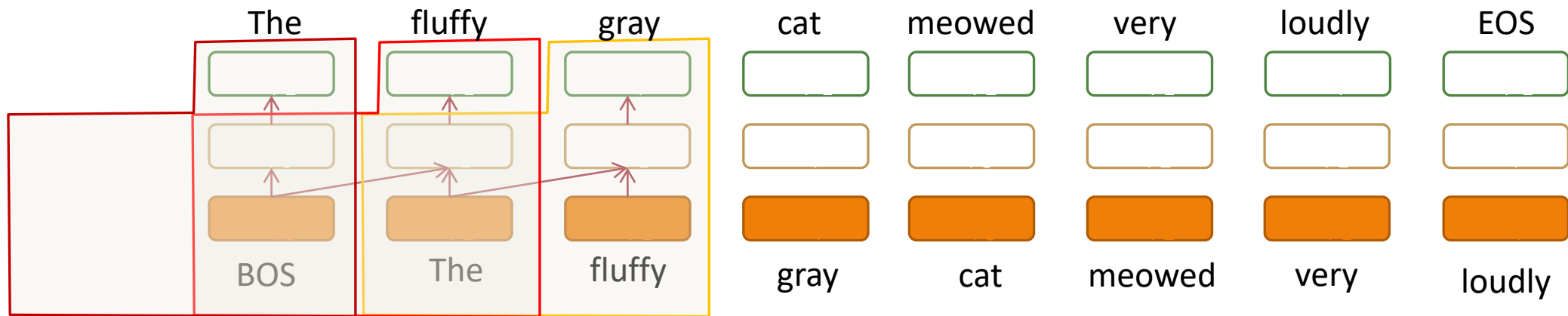
Review: A Neural N-Gram Model (N=3)

The fluffy gray cat meowed very loudly



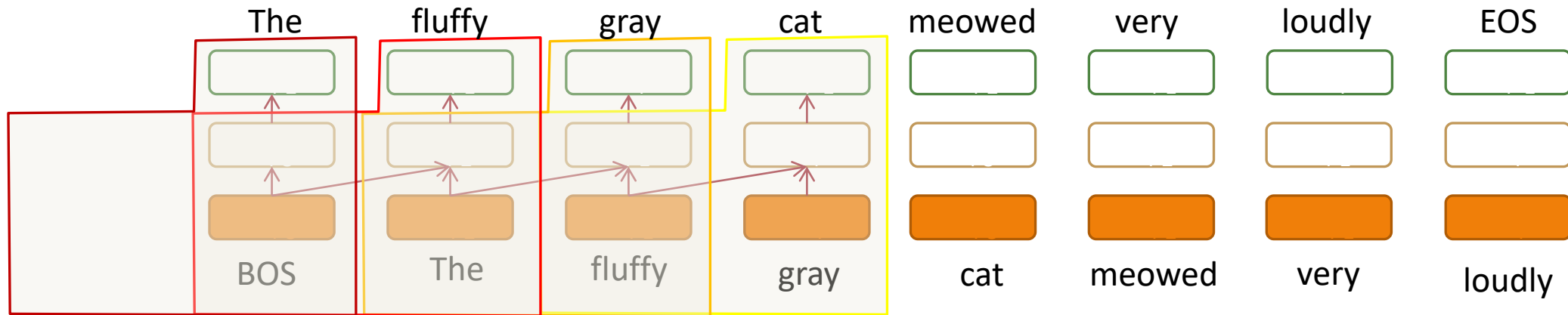
Review: A Neural N-Gram Model (N=3)

The fluffy gray cat meowed very loudly



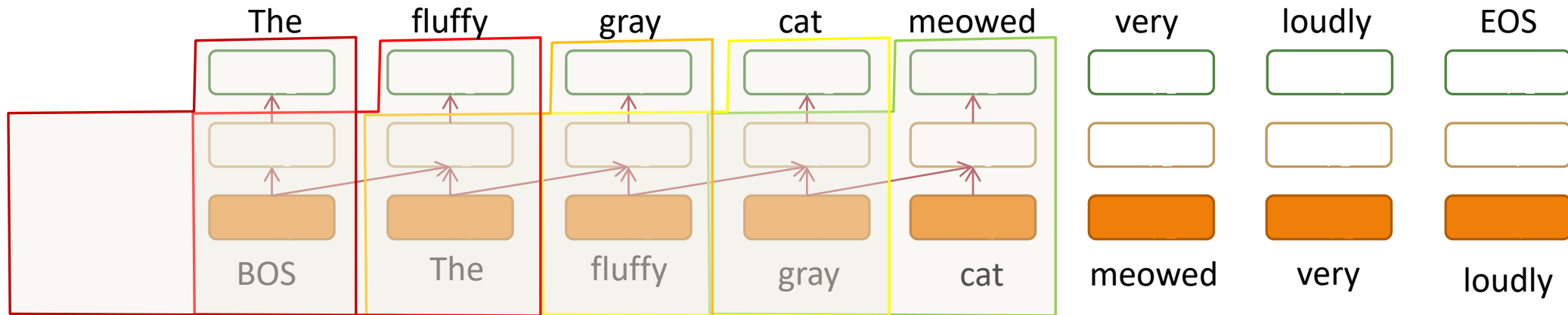
Review: A Neural N-Gram Model (N=3)

The fluffy gray cat meowed very loudly



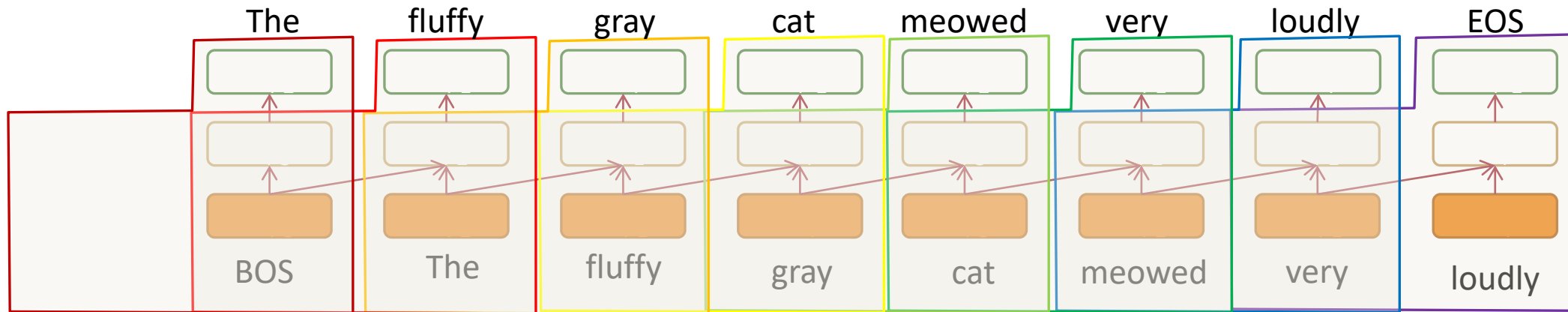
Review: A Neural N-Gram Model (N=3)

The fluffy gray cat meowed very loudly



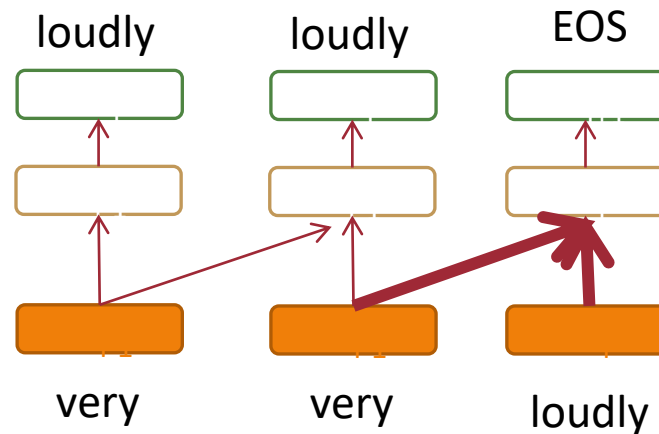
Review: A Neural N-Gram Model (N=3)

The fluffy gray cat meowed very loudly



A Neural N-Gram Model (N=3)

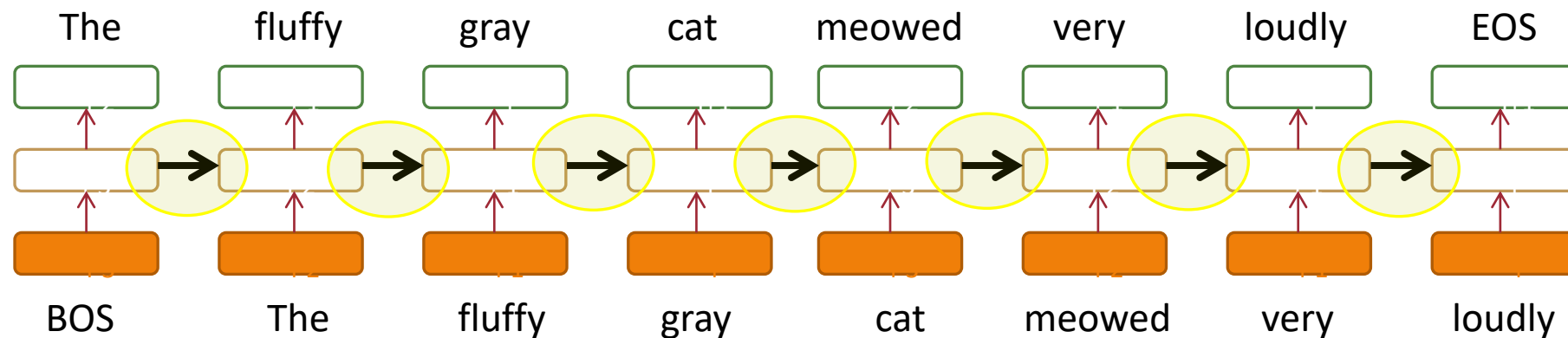
The fluffy gray cat meowed very loudly



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

A Recurrent Neural Language Model

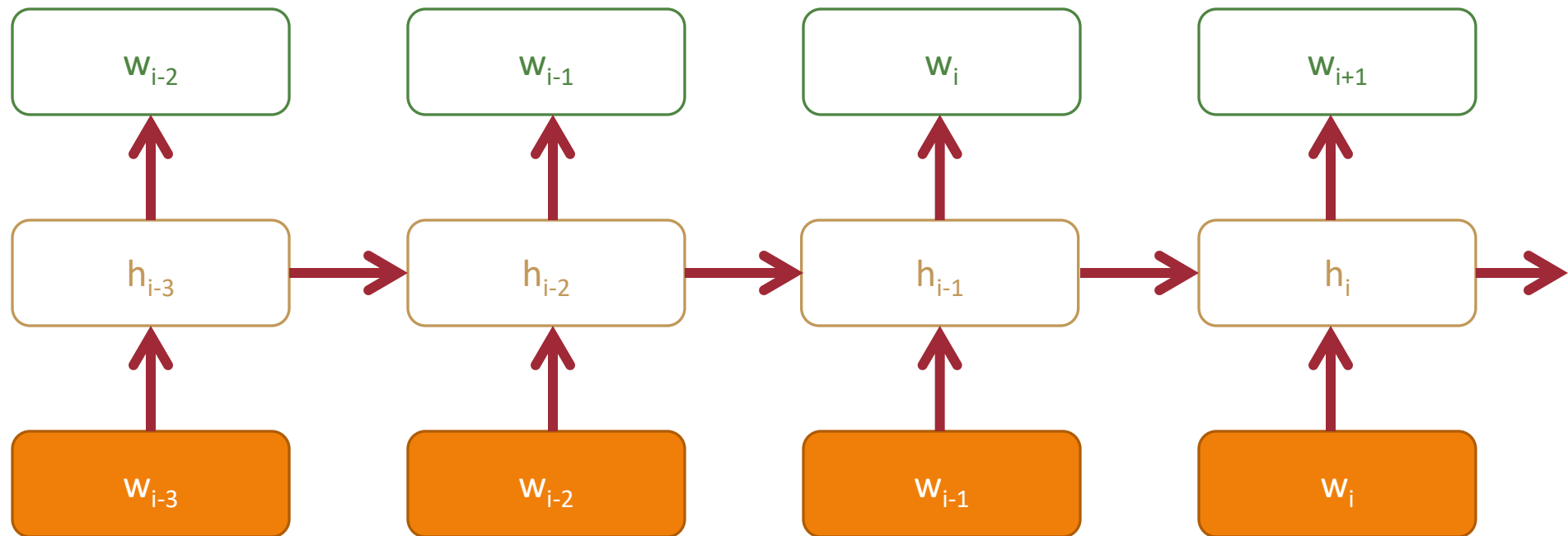
The fluffy gray cat meowed very loudly



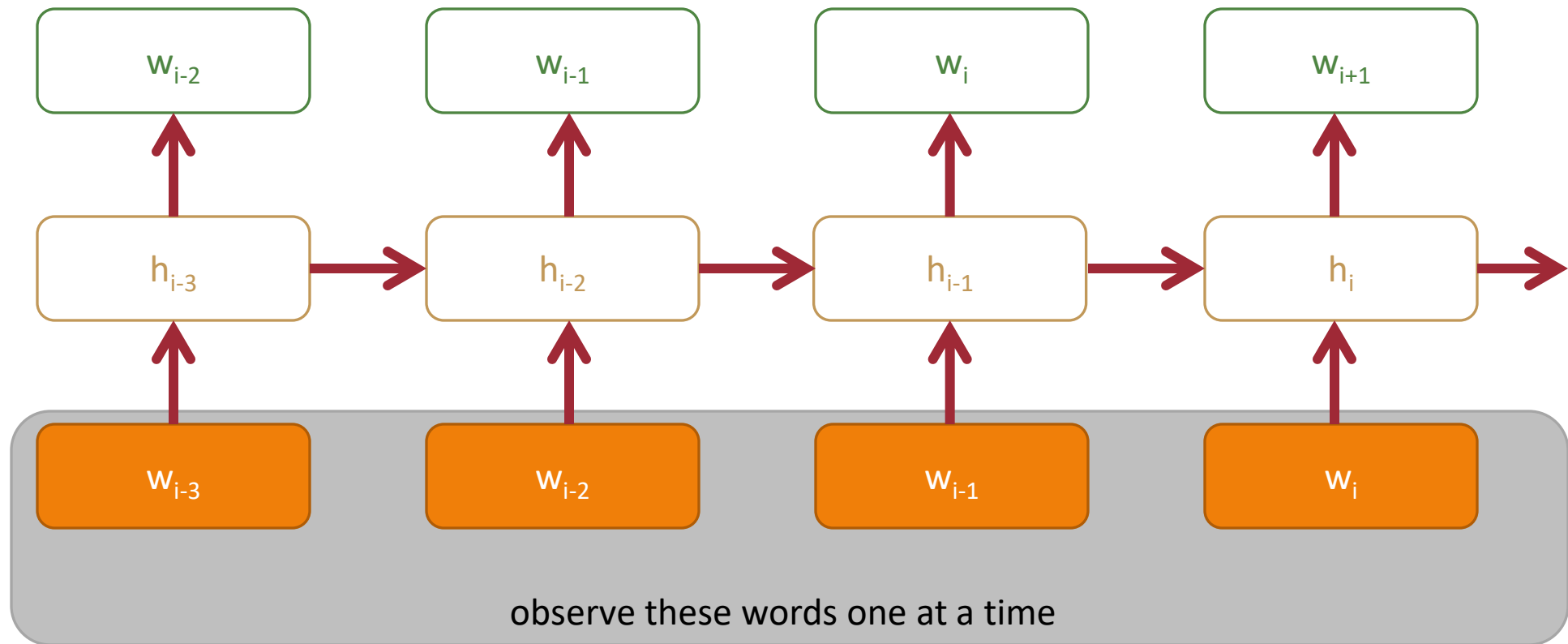
Critical issue: the amount of information flow is fundamentally restricted!!!

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

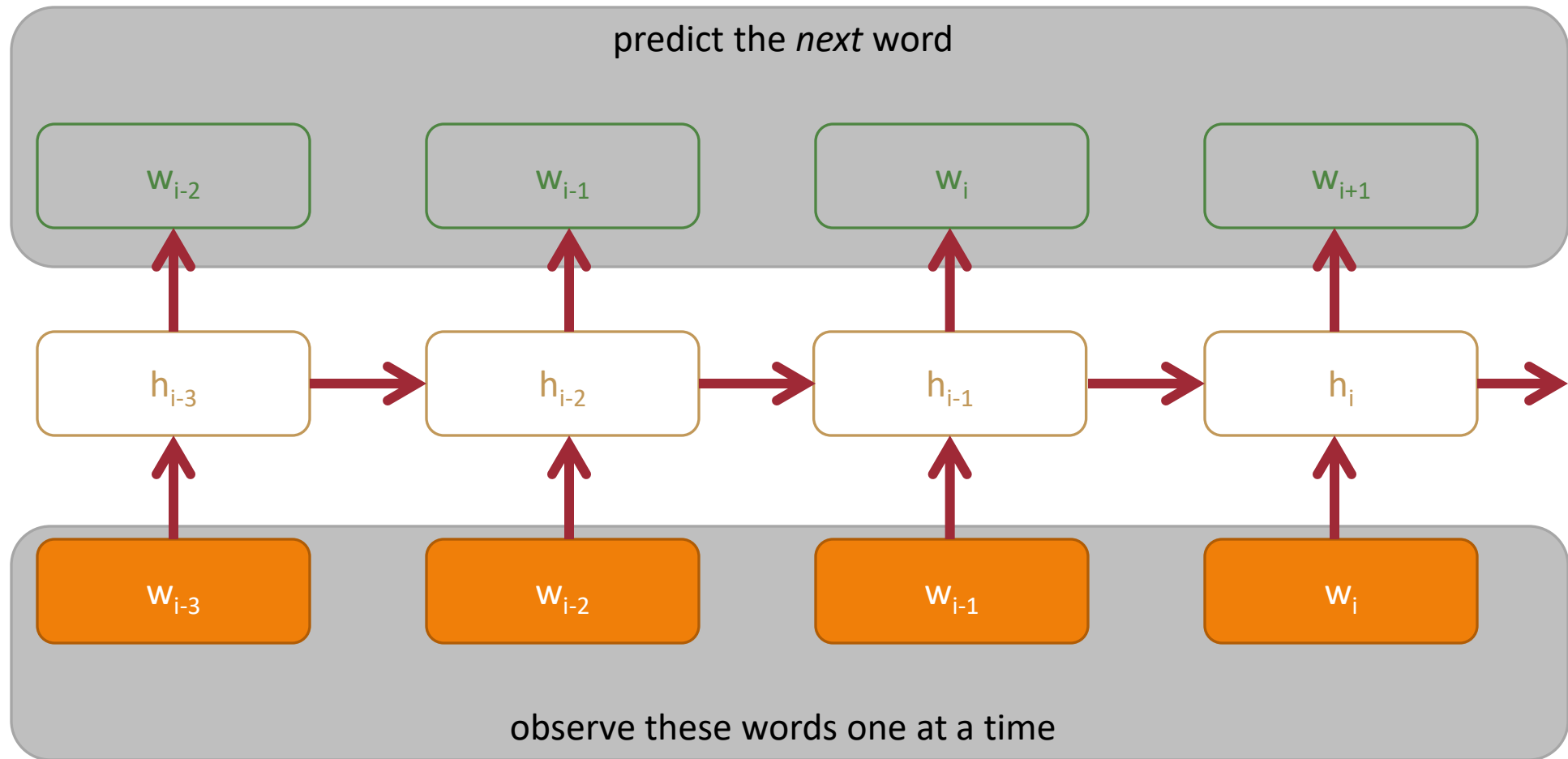
A Classic View of Recurrent Neural Language Modeling



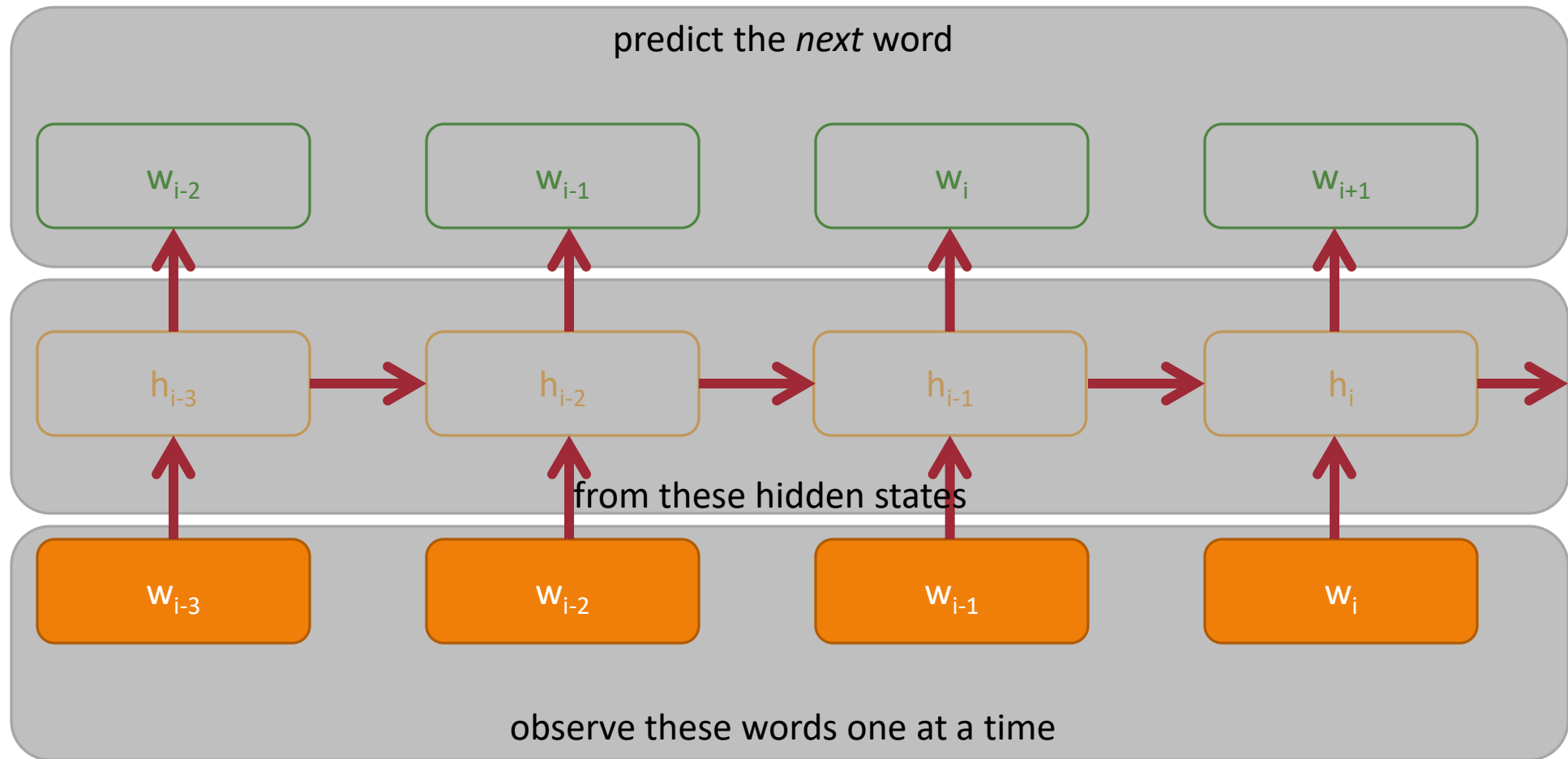
A Classic View of Recurrent Neural Language Modeling



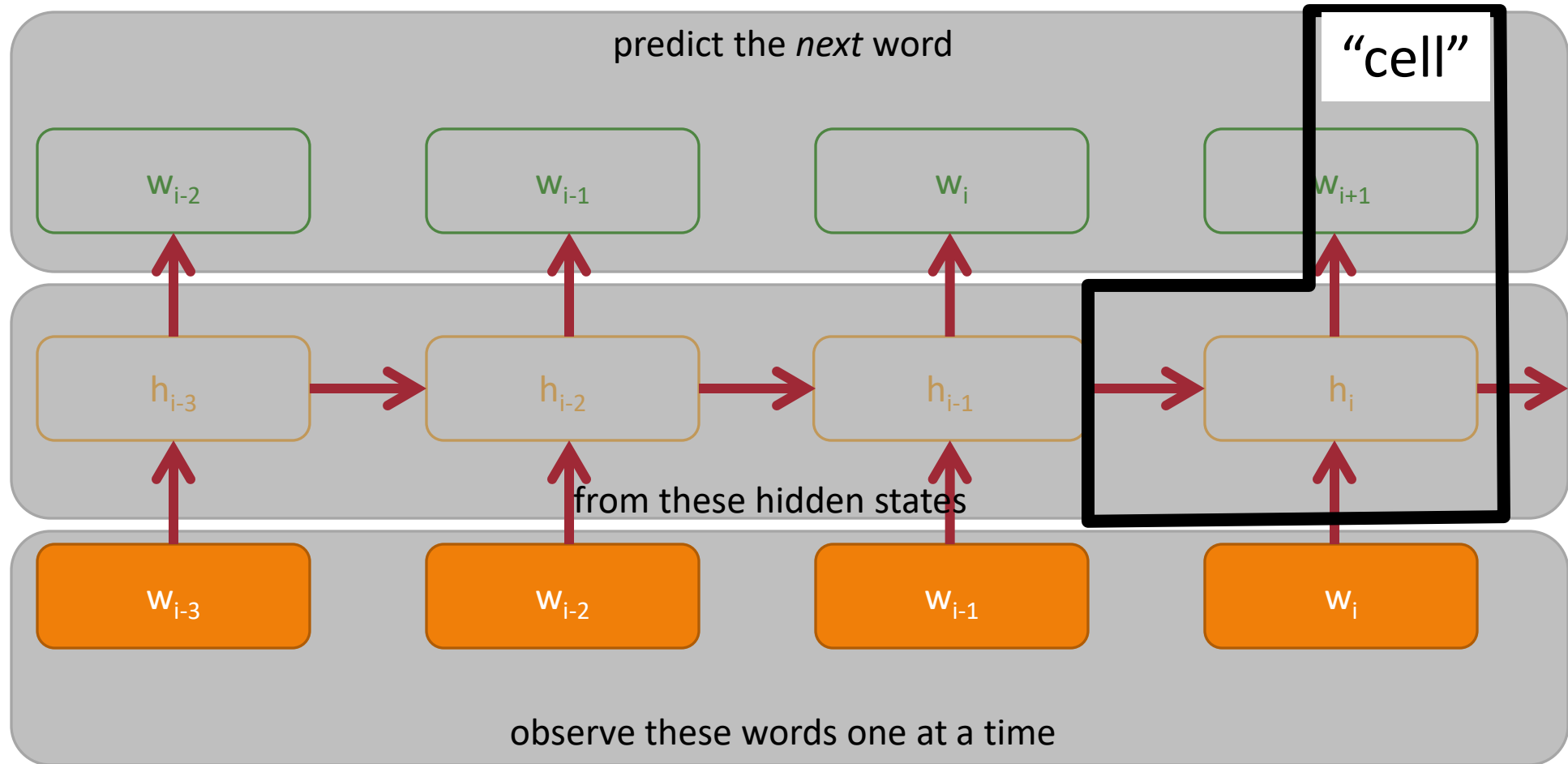
A Classic View of Recurrent Neural Language Modeling



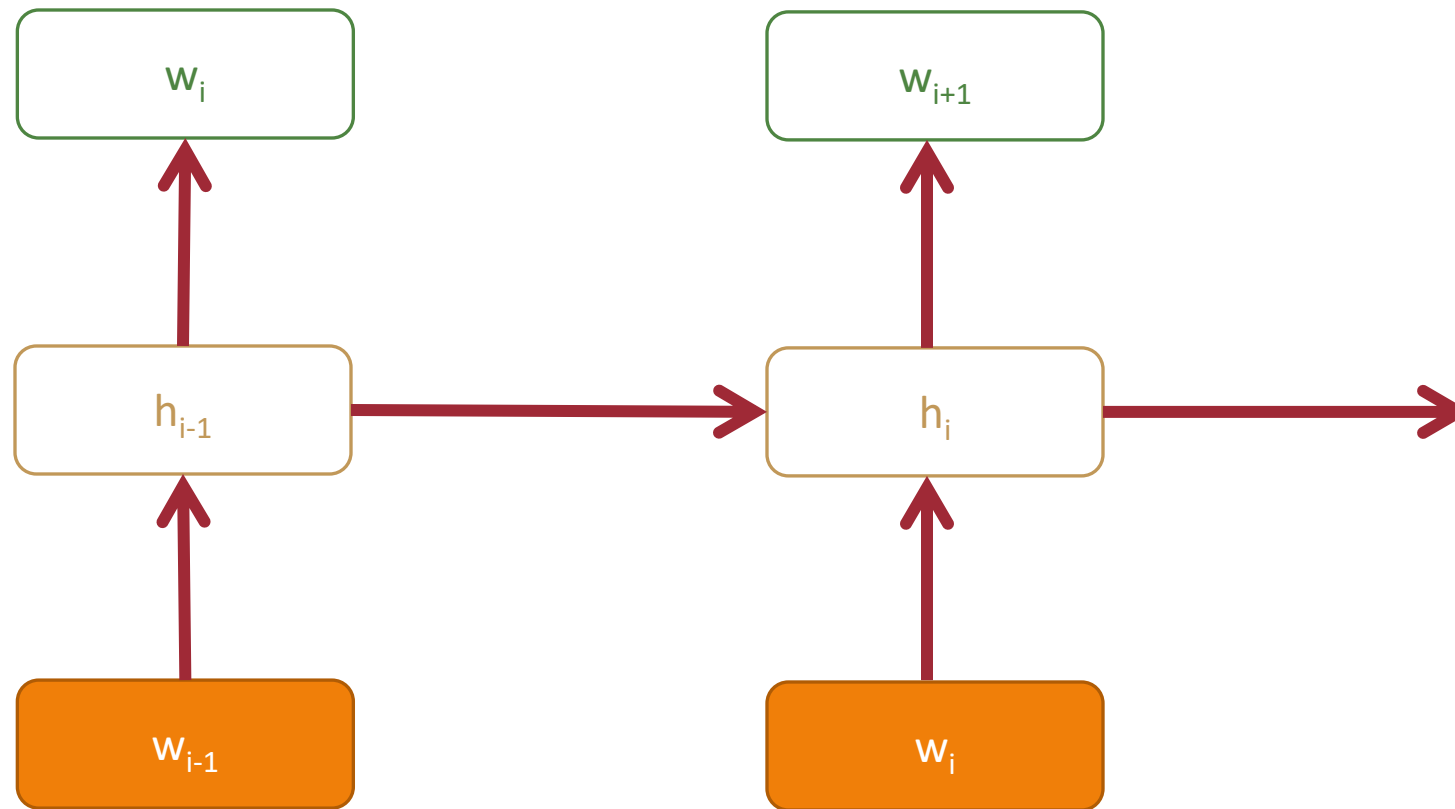
A Classic View of Recurrent Neural Language Modeling



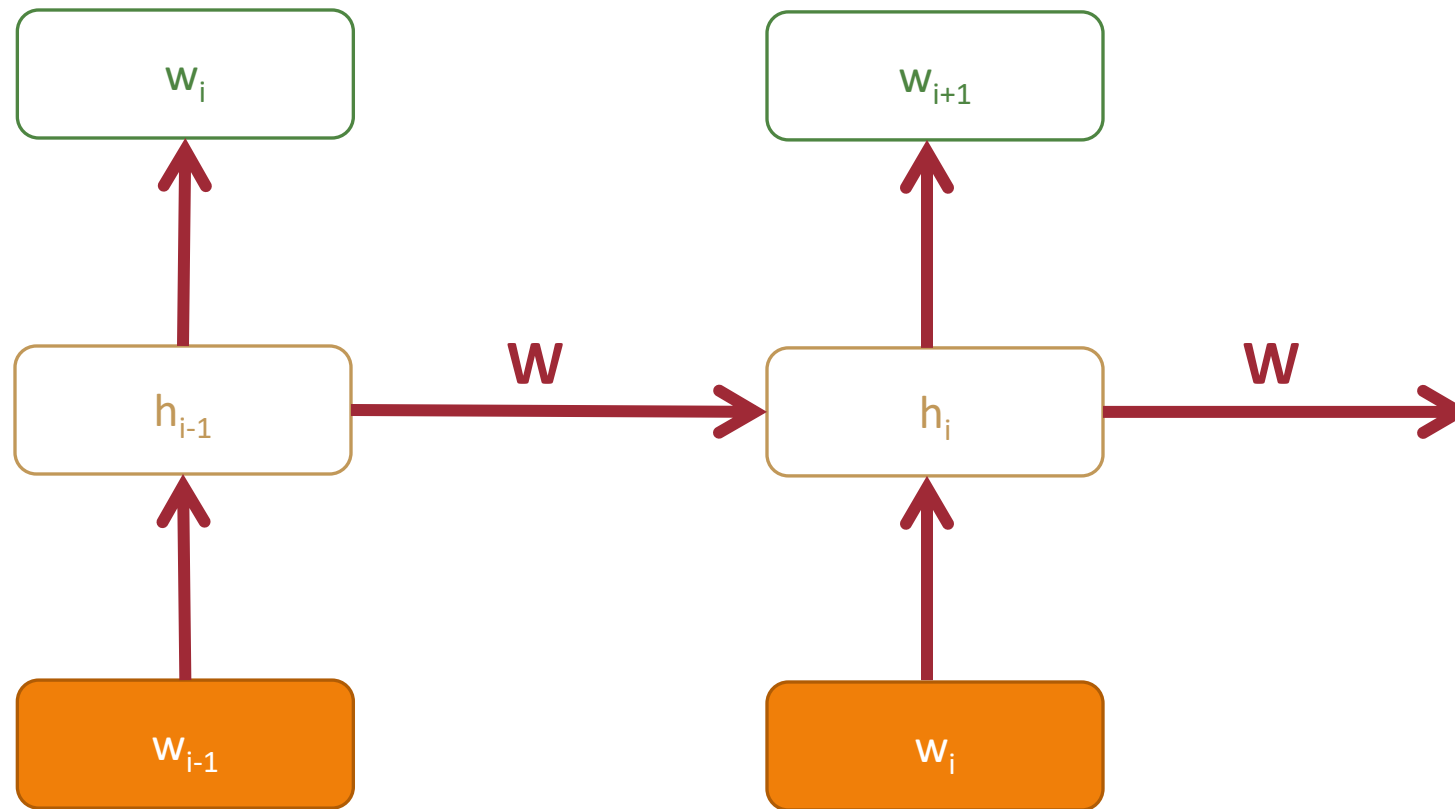
A Classic View of Recurrent Neural Language Modeling



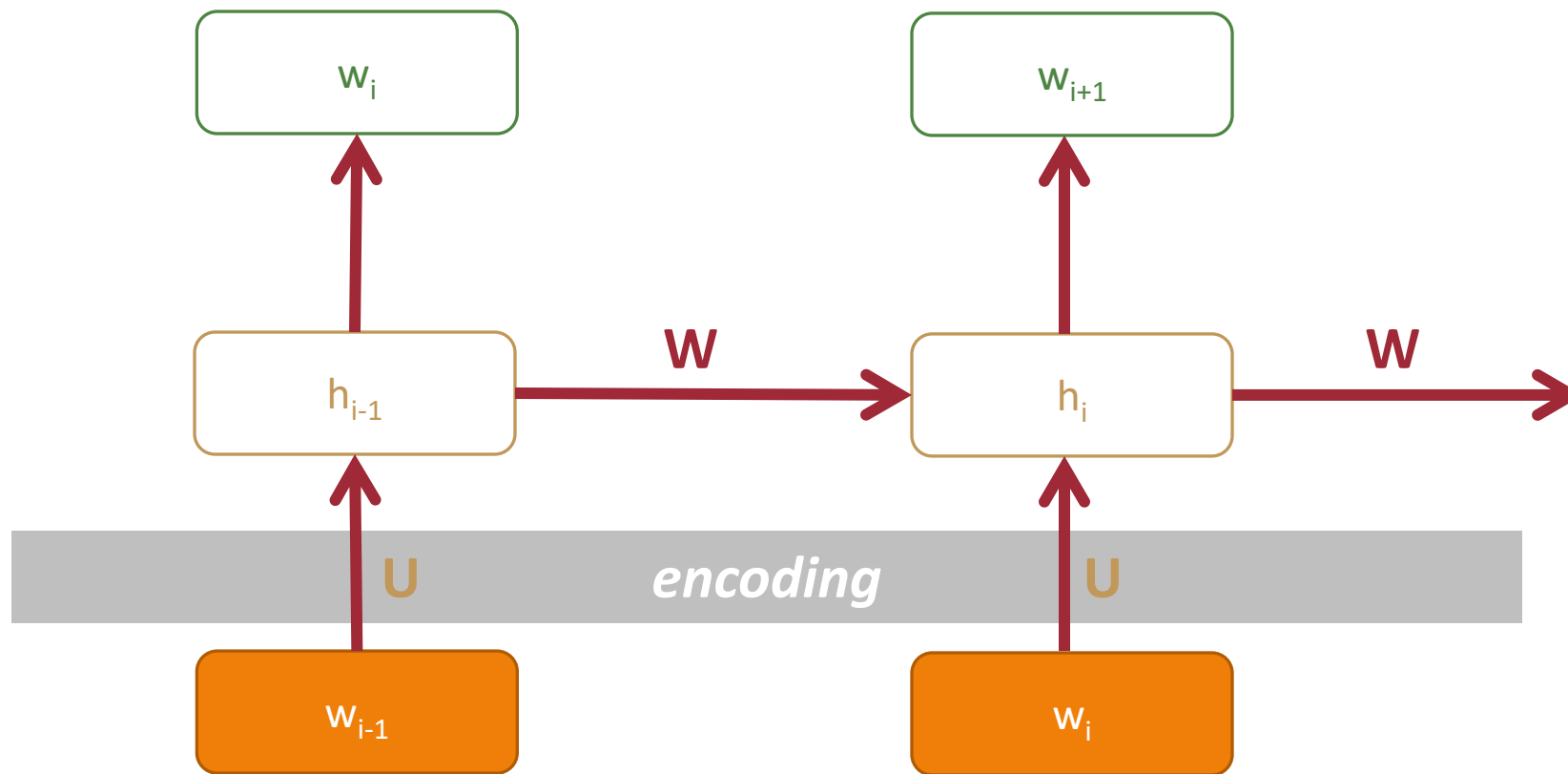
A Recurrent Neural Network Cell



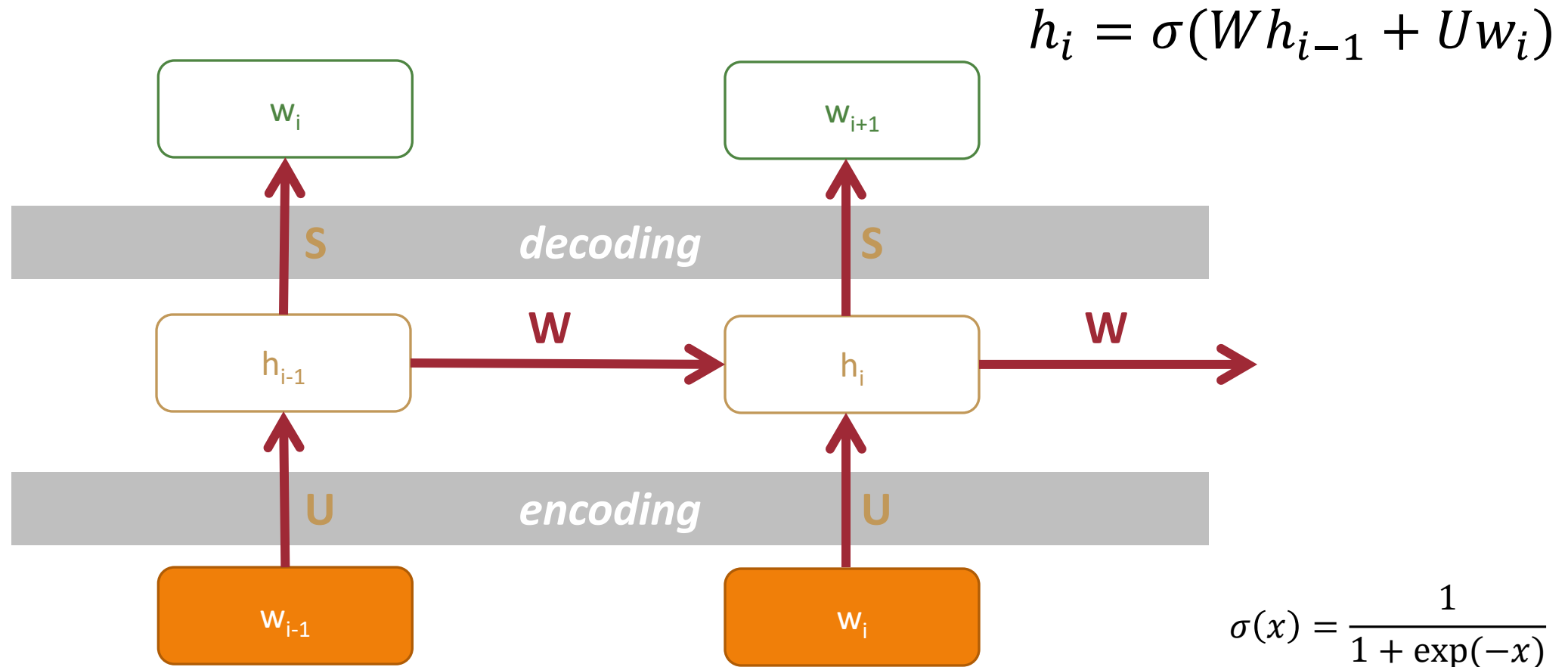
A Recurrent Neural Network Cell



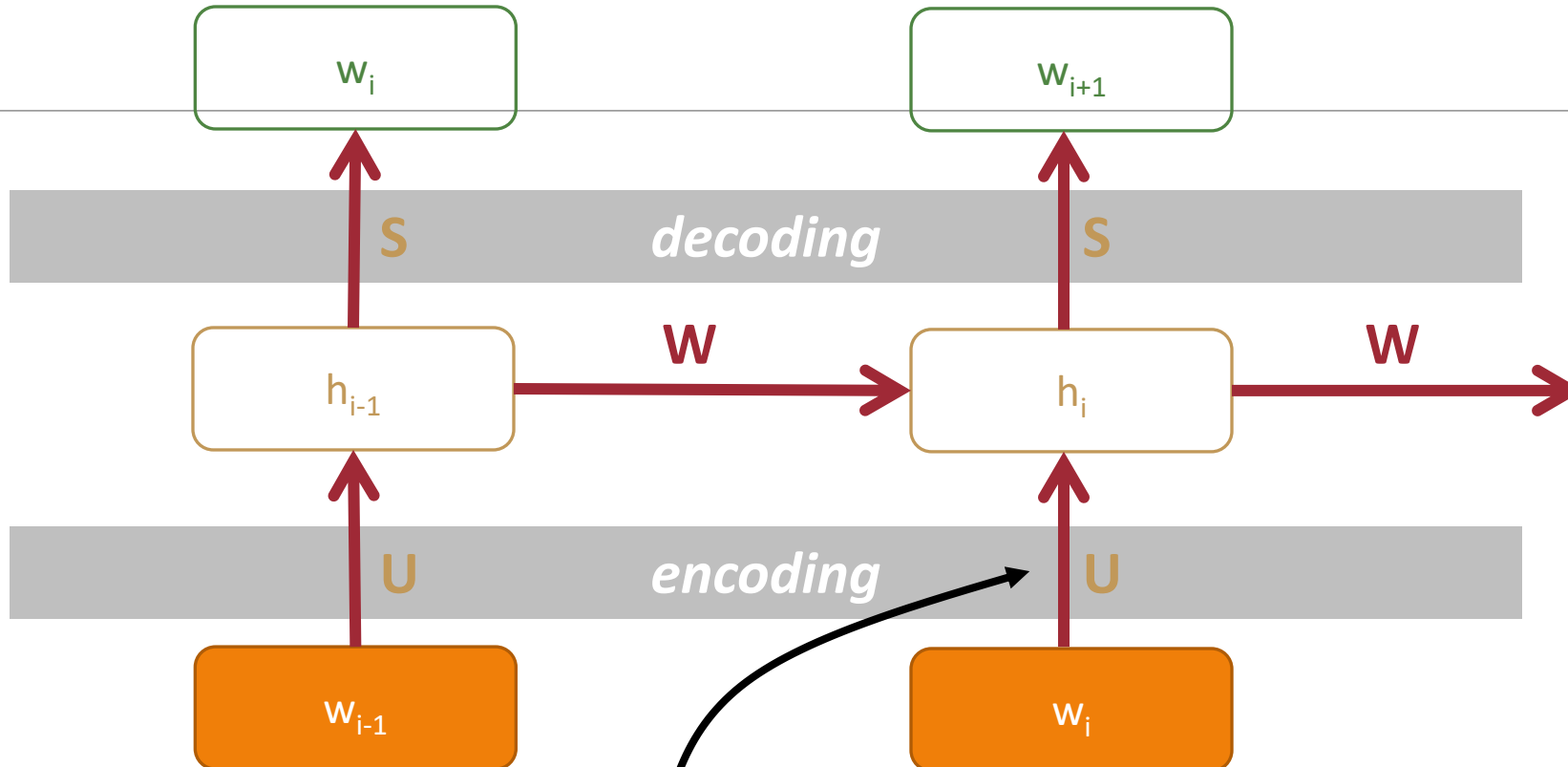
A Recurrent Neural Network Cell



A Recurrent Neural Network Cell



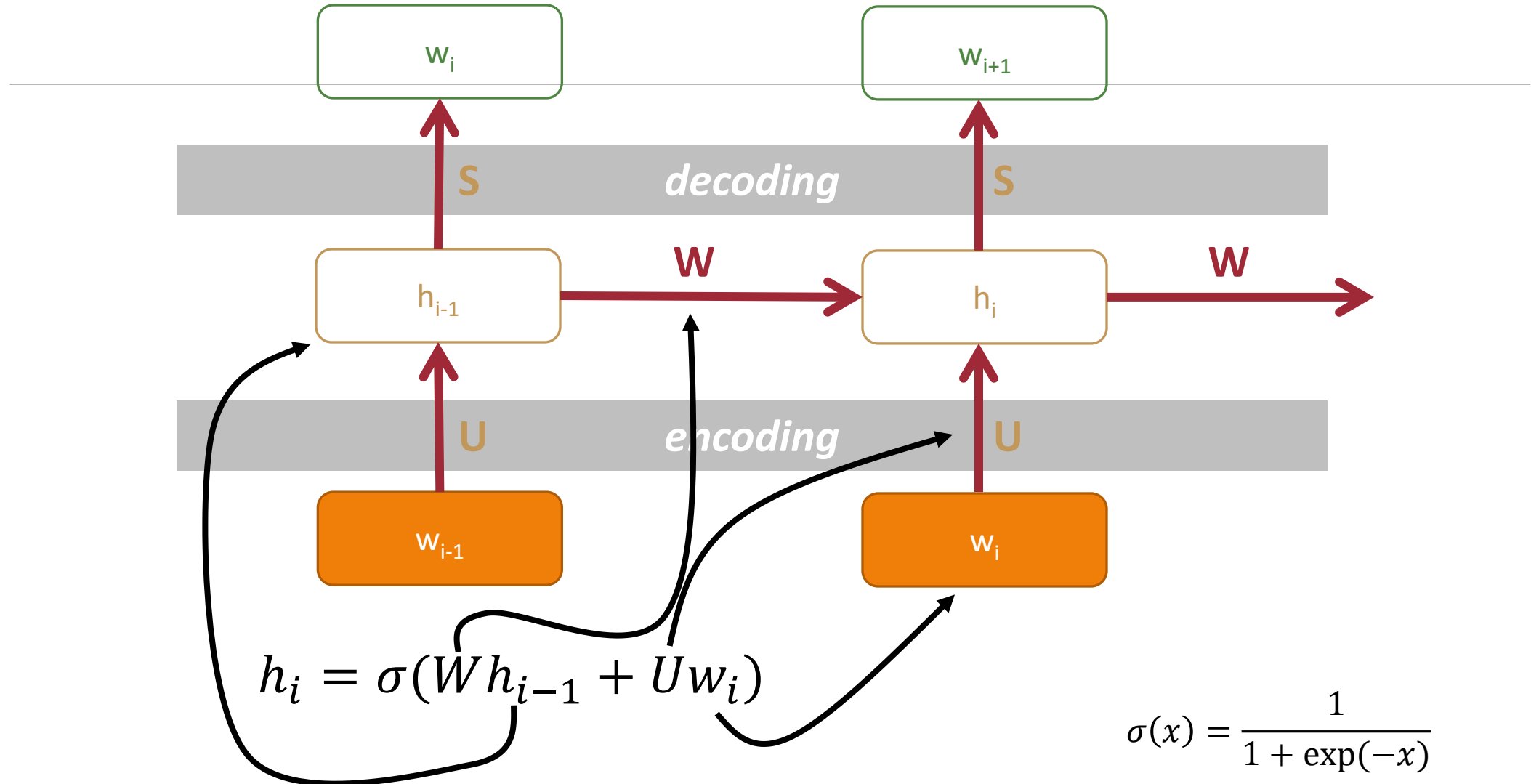
A Simple Recurrent Neural Network Cell



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

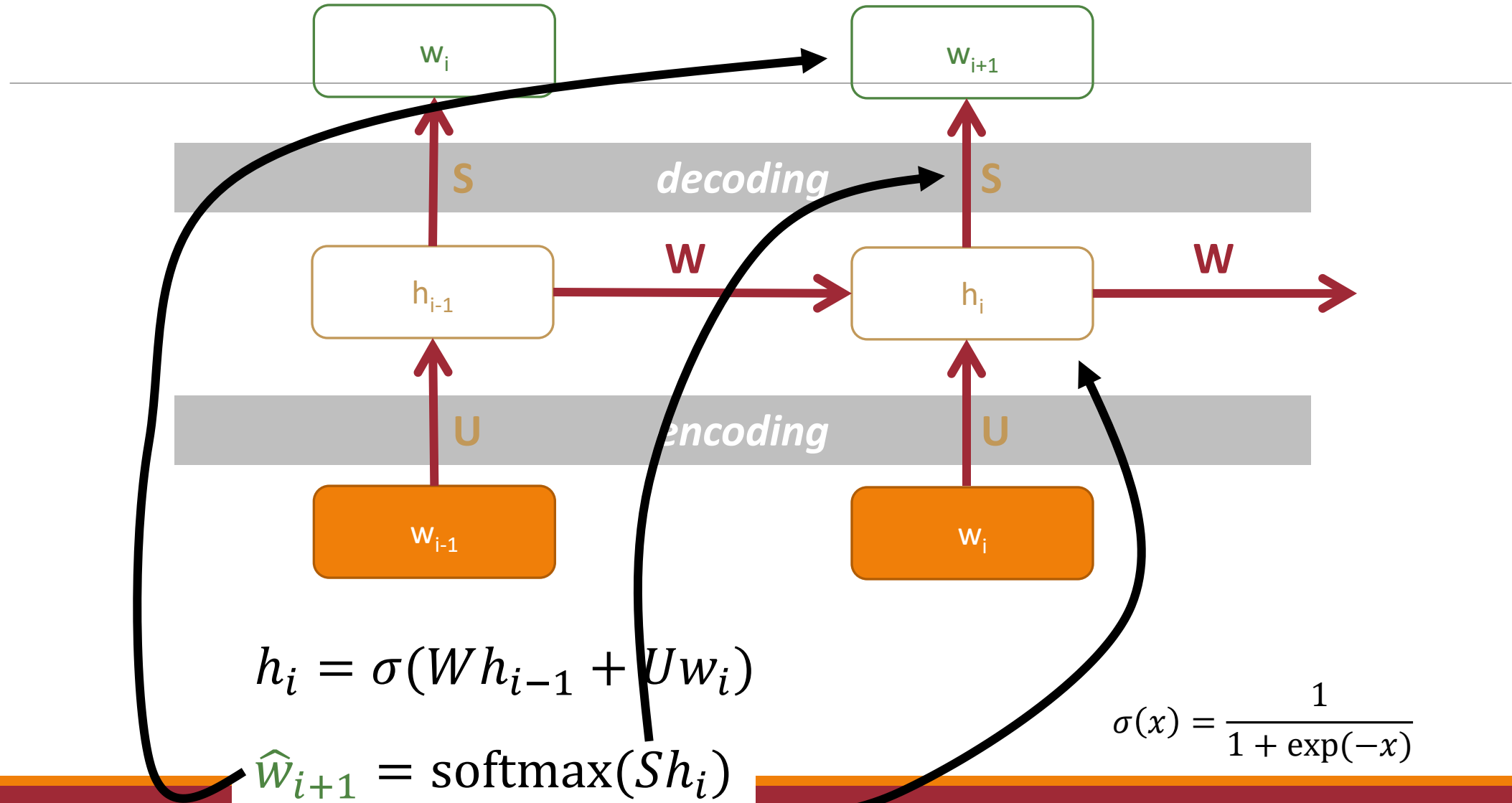
$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

A Simple Recurrent Neural Network Cell

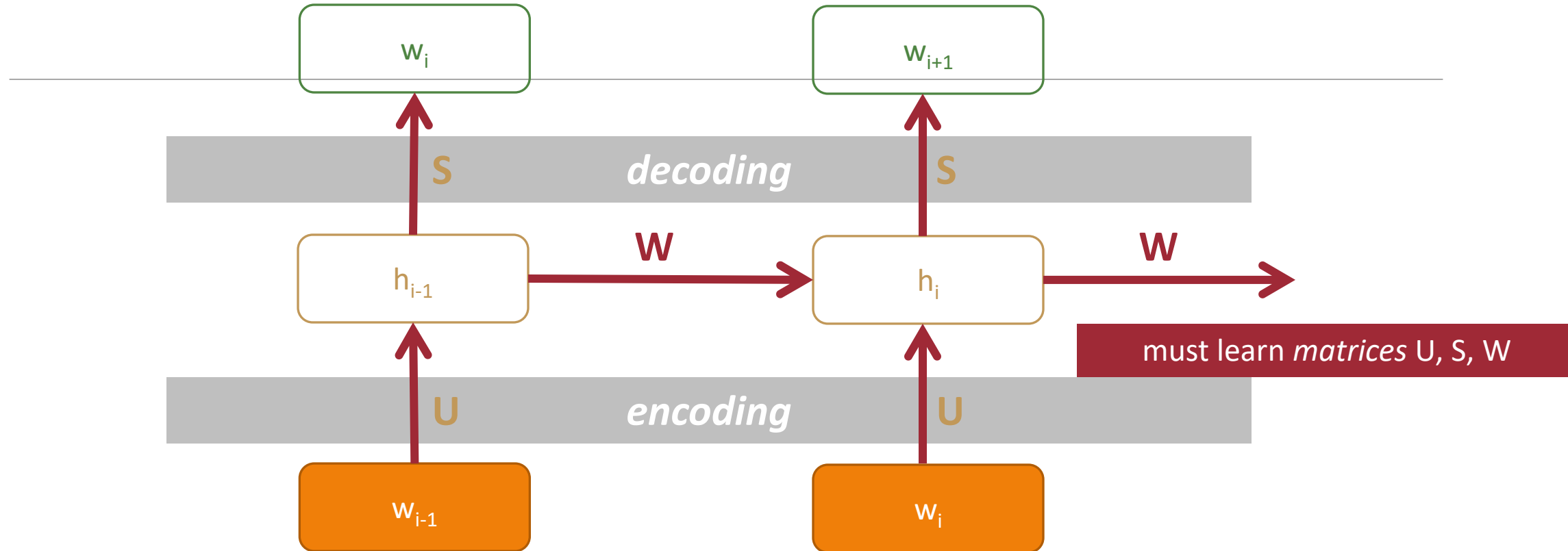


$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

A Simple Recurrent Neural Network Cell



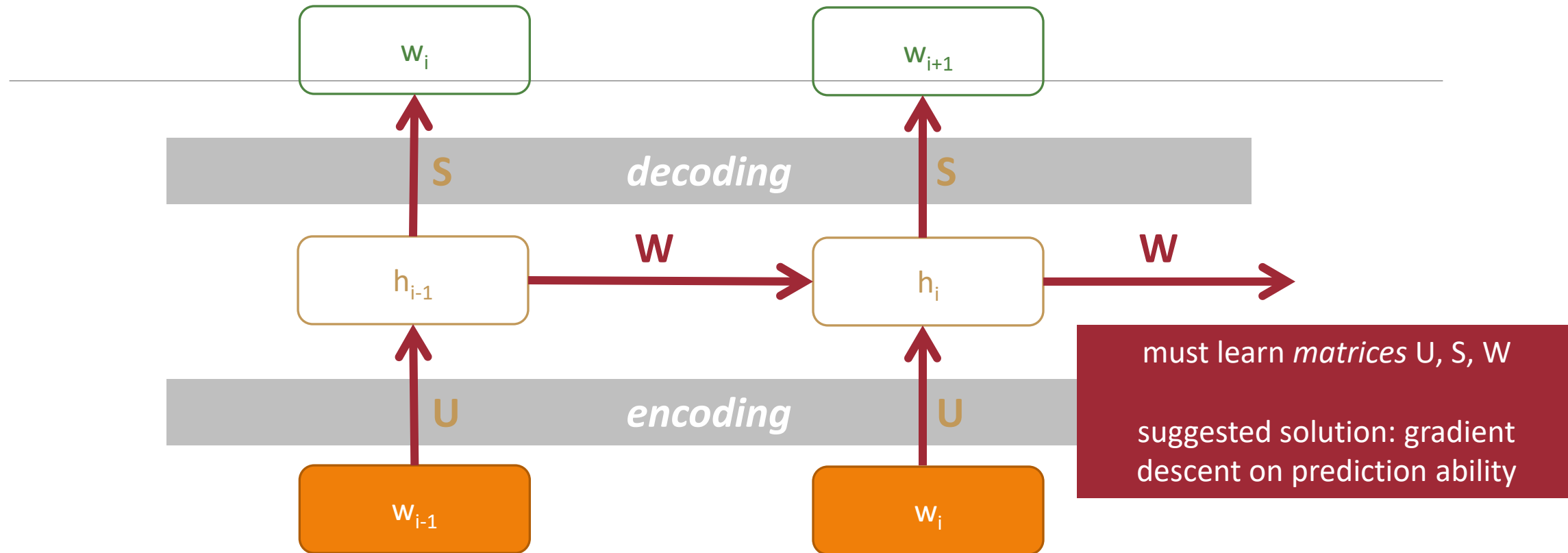
A Simple Recurrent Neural Network Cell



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

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

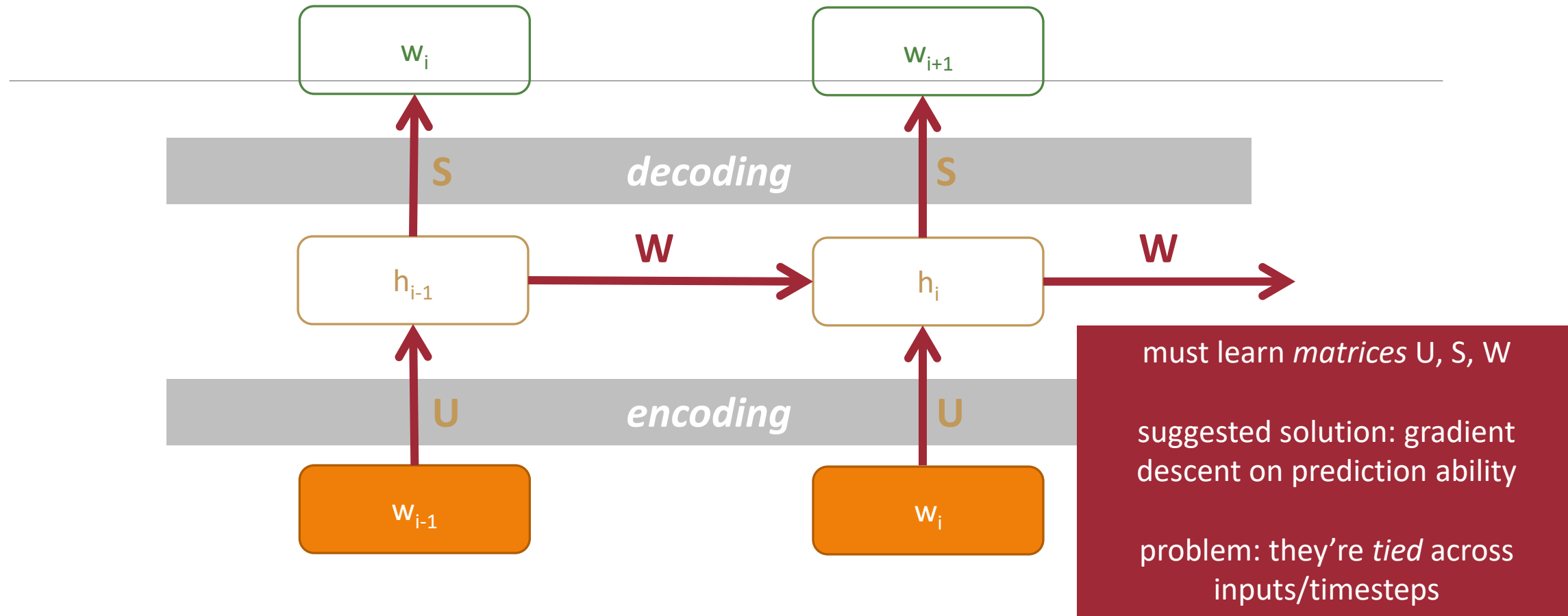
A Simple Recurrent Neural Network Cell



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

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

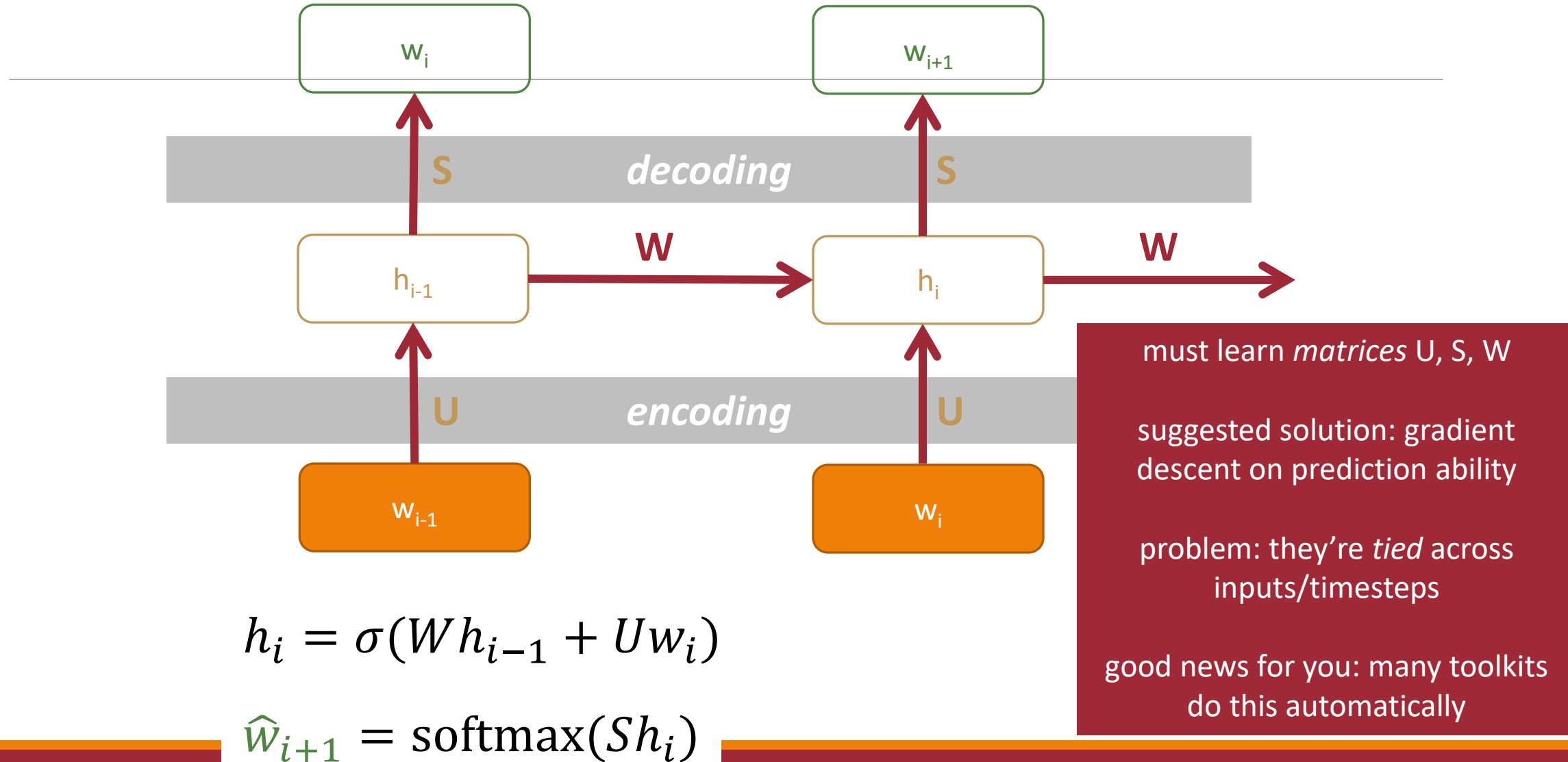
A Simple Recurrent Neural Network Cell



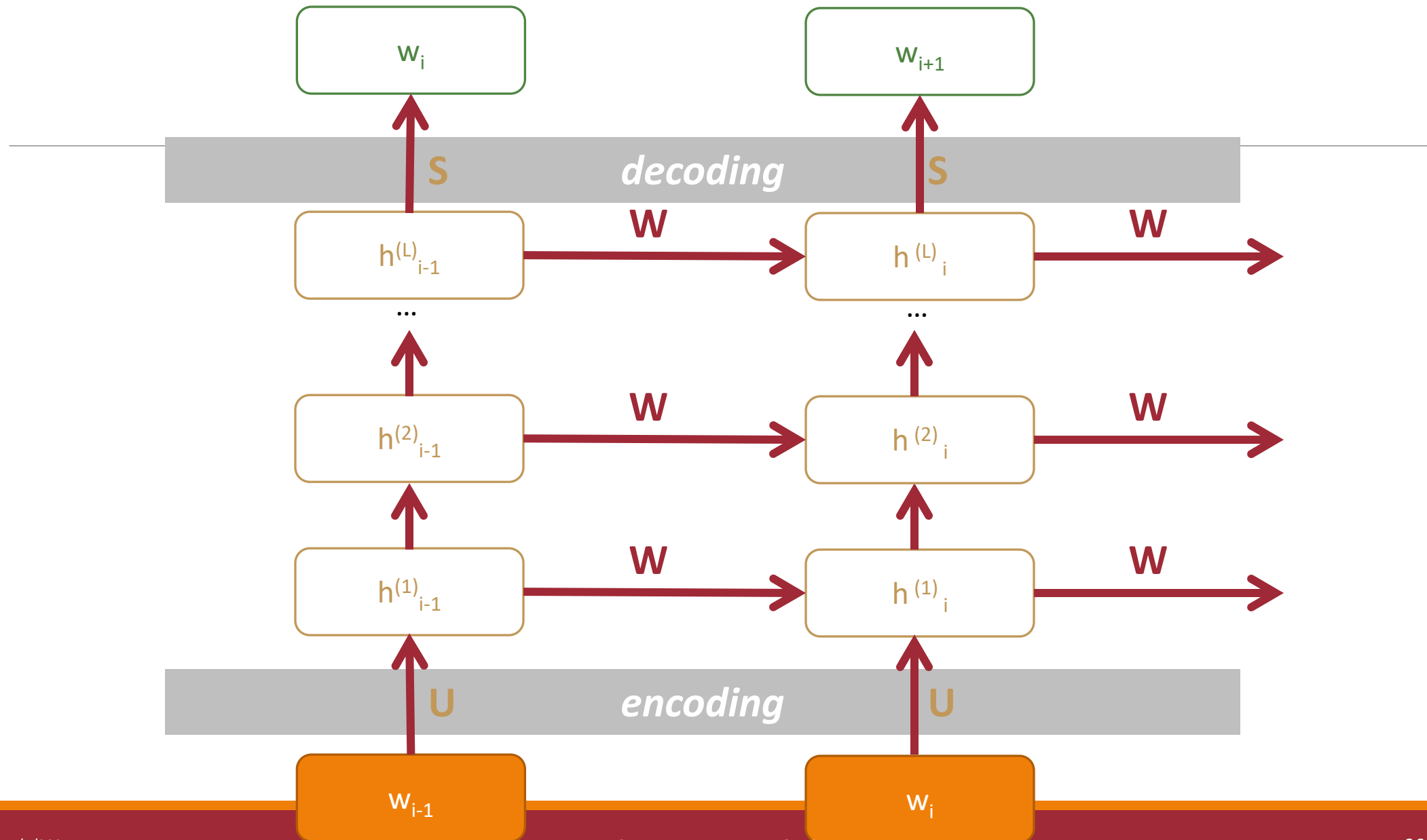
$$h_i = \sigma(W h_{i-1} + U w_i)$$

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

A Simple Recurrent Neural Network Cell



A Multi-Layer Simple Recurrent Neural Network Cell



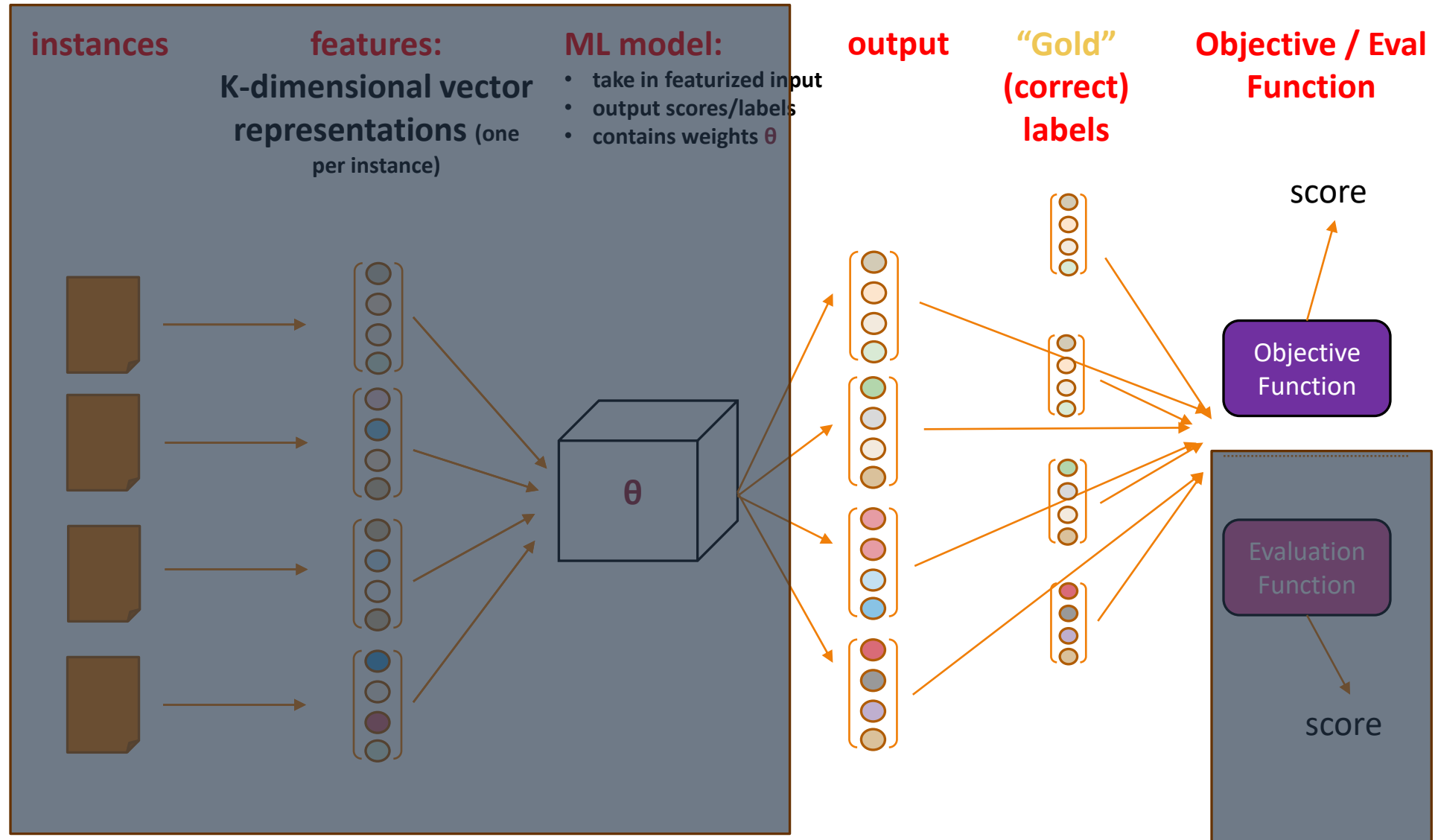
How do you learn an RNN?

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 cross entropy loss formula. The predicted output \hat{y} is shown as a vector, and the true output y is shown as a one-hot vector. The loss is calculated as the negative sum over all labels k of the product of the predicted probability $\hat{y}[k]$ and the logarithm of the true probability $p(y = k|x)$. The one-hot vector y has a value of 1 at the index corresponding to the correct class, and 0 elsewhere.

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 θ_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 ρ_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 θ_t

Until converged:

for example(s) sentence i:

1. Compute loss l on x_i

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

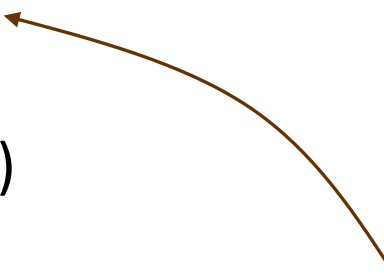
2. Get gradient $g_t = l'(x_i)$

3. Get scaling factor ρ_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

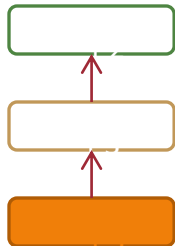
$\log .2$

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

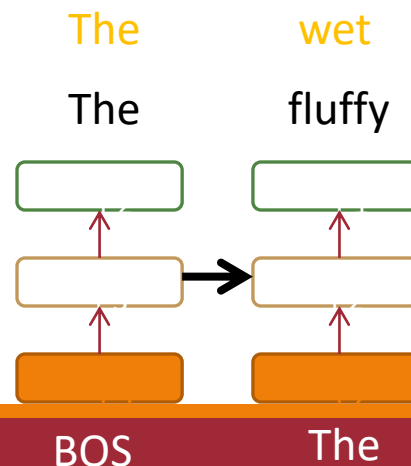


BOS

Recurrent NN Loss

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

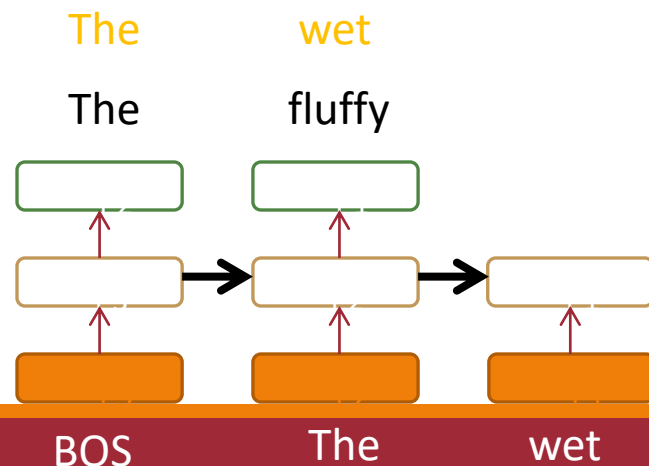
word	prob.	word	prob.
The	.2	black	.2
gray	.01	wet	.12
blue	.001	blue	.001
fluffy	.0005	fluffy	.0005
wet	.0005	gray	.0005
...



Recurrent NN Loss

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

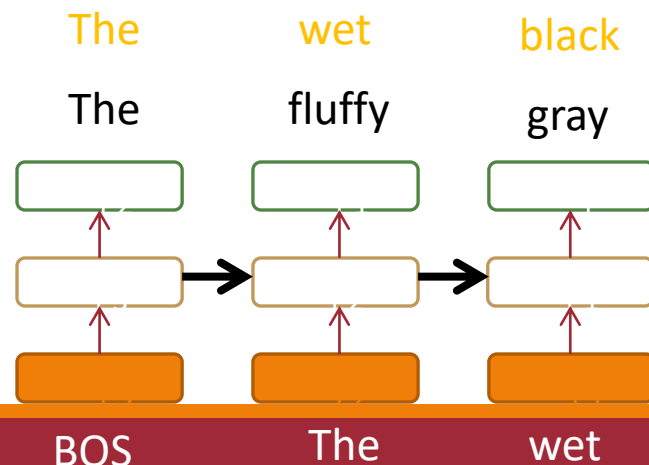
word	prob.	word	prob.
The	.2	black	.2
gray	.01	wet	.12
blue	.001	blue	.001
fluffy	.0005	fluffy	.0005
wet	.0005	gray	.0005
...



Recurrent NN Loss

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

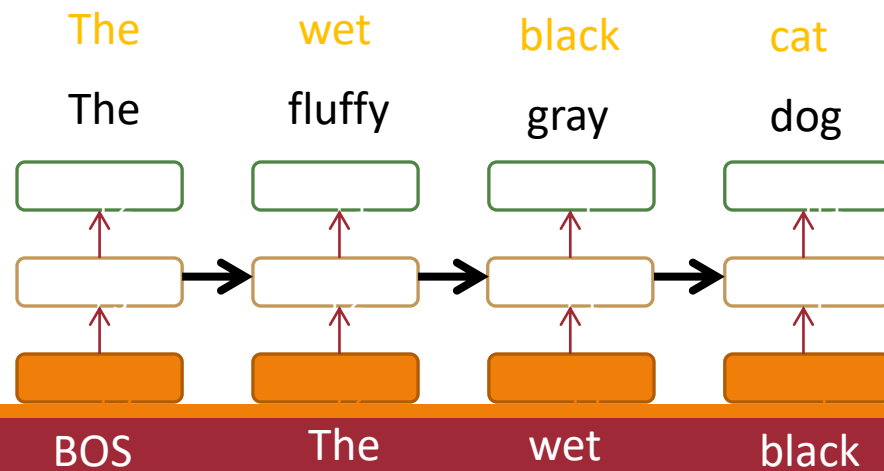
word	prob.	word	prob.	word	prob.
The	.2	black	.2	black	.2
gray	.01	wet	.12	gray	.01
blue	.001	blue	.001	blue	.001
fluffy	.0005	fluffy	.0005	bald	.0005
wet	.0005	gray	.0005	wet	.0005
...



Recurrent NN Loss

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

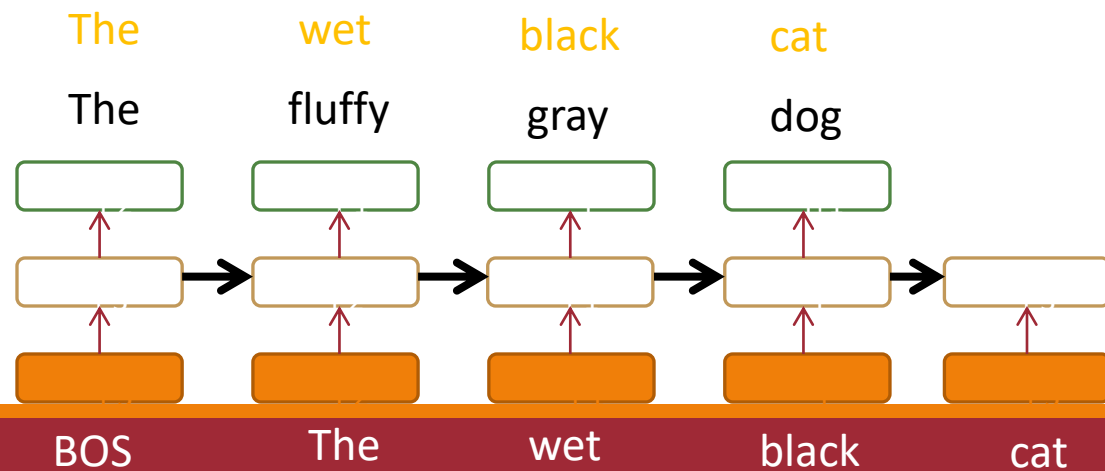
word	prob.	word	prob.	word	prob.	word	prob.
The	.2	black	.2	black	.2	dog	.2
gray	.01	wet	.12	gray	.01	cat	.19
blue	.001	blue	.001	blue	.001	blue	.001
fluffy	.0005	fluffy	.0005	bald	.0005	fluffy	.0005
wet	.0005	gray	.0005	wet	.0005	wet	.0005
...



Recurrent NN Loss

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

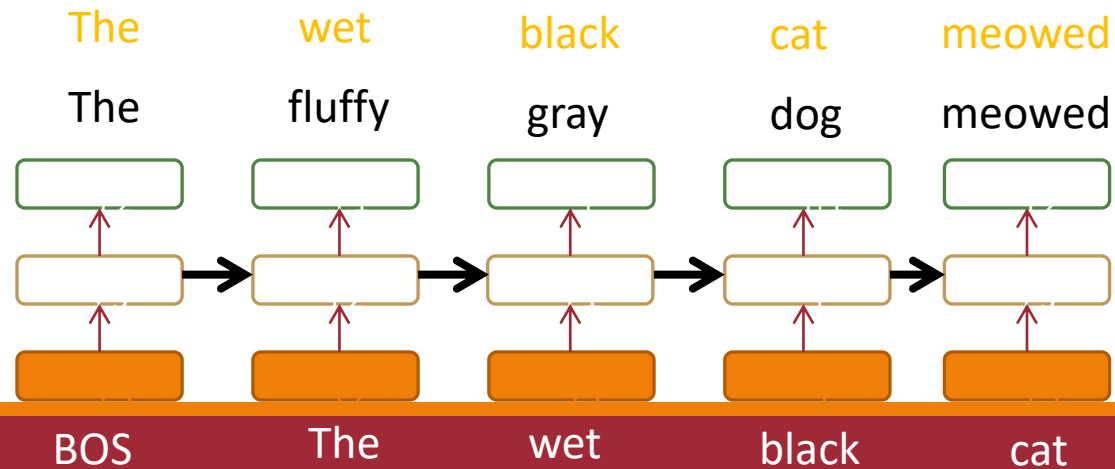
word	prob.	word	prob.	word	prob.	word	prob.
The	.2	black	.2	black	.2	dog	.2
gray	.01	wet	.12	gray	.01	cat	.19
blue	.001	blue	.001	blue	.001	blue	.001
fluffy	.0005	fluffy	.0005	bald	.0005	fluffy	.0005
wet	.0005	gray	.0005	wet	.0005	wet	.0005
...



Recurrent NN Loss

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

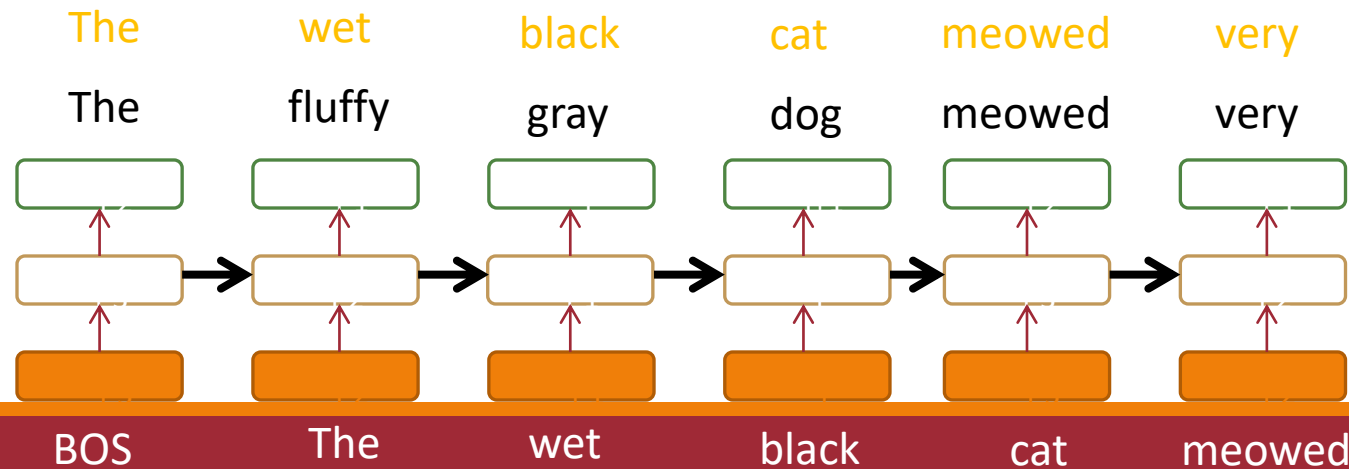
word	prob.	word	prob.	word	prob.	word	prob.	word	prob.
The	.2	black	.2	black	.2	dog	.2	meowed	.3
gray	.01	wet	.12	gray	.01	cat	.19	purred	.2
blue	.001	blue	.001	blue	.001	blue	.001	hissed	.1
fluffy	.0005	fluffy	.0005	bald	.0005	fluffy	.0005	fluffy	.001
wet	.0005	gray	.0005	wet	.0005	wet	.0005	wet	.001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



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
...

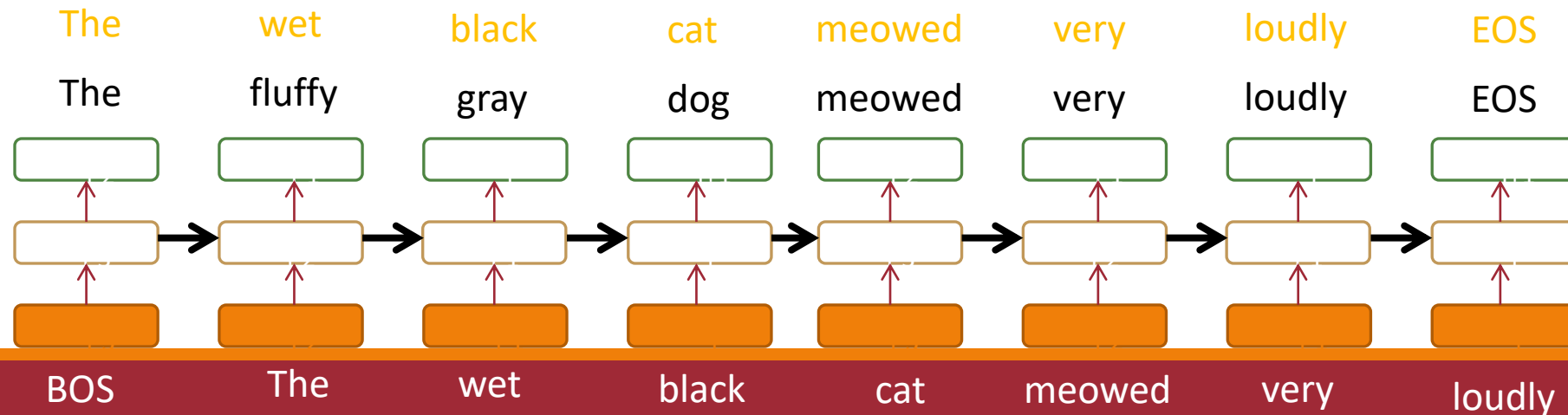


Recurrent NN Loss

(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 θ_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 ρ_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)

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

Set $t = 0$

Pick a starting value θ_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 ρ_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

One solution: clip the gradients to a max value

PyTorch RNN LMs

Pick Your Toolkit

PyTorch
Deeplearning4j
TensorFlow
Caffe
Keras

MXNet
Torch
...

Comparisons:

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

```
import torch.nn as nn

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

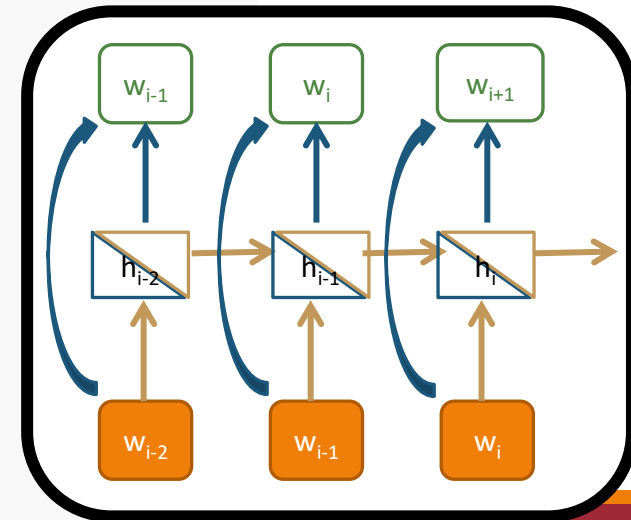
        self.hidden_size = hidden_size

        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.h2o = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        hidden = self.i2h(combined)
        output = self.h2o(hidden)
        output = self.softmax(output)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, self.hidden_size)

n_hidden = 128
rnn = RNN(n_letters, n_hidden, n_categories)
```



Defining A Simple RNN in Python

http://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

```
import torch.nn as nn

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

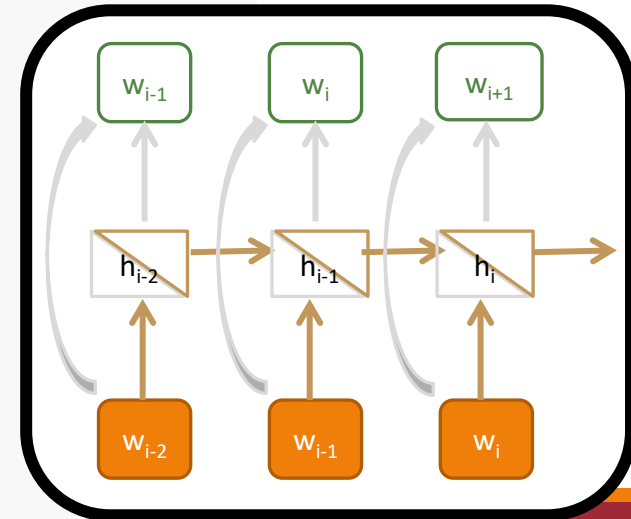
        self.hidden_size = hidden_size

        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.h2o = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        hidden = self.i2h(combined)
        output = self.h2o(hidden)
        output = self.softmax(output)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, self.hidden_size)

n_hidden = 128
rnn = RNN(n_letters, n_hidden, n_categories)
```



Defining A Simple RNN in Python

http://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

```
import torch.nn as nn

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

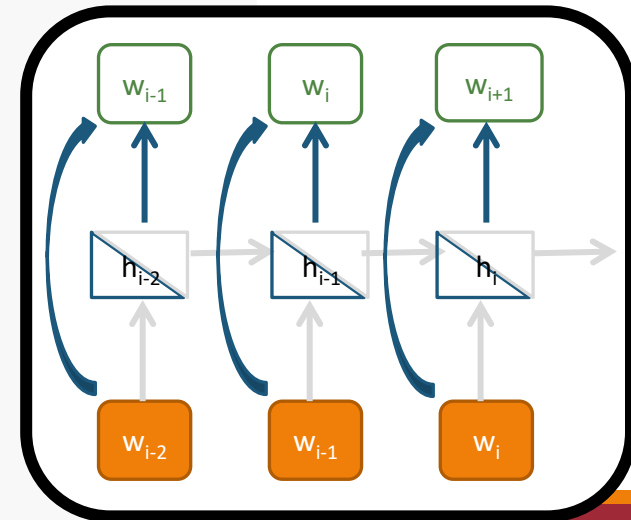
        self.hidden_size = hidden_size

        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.h2o = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        hidden = self.i2h(combined)
        output = self.h2o(hidden)
        output = self.softmax(output)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, self.hidden_size)

n_hidden = 128
rnn = RNN(n_letters, n_hidden, n_categories)
```



Defining A Simple RNN in Python

http://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

```
import torch.nn as nn

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

        self.hidden_size = hidden_size

        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.h2o = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)
```

```
def forward(self, input):
    combined = torch.cat((input, self.hidden), 1)
    hidden = self.i2h(combined)
    output = self.h2o(hidden)
    return output
```

```
def initHidden():
    return torch.zeros(1, self.hidden_size, 1)
```

```
n_hidden = 128
rnn = RNN(n_letters, n_hidden, n_classes)
```

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

```
import torch.nn as nn

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

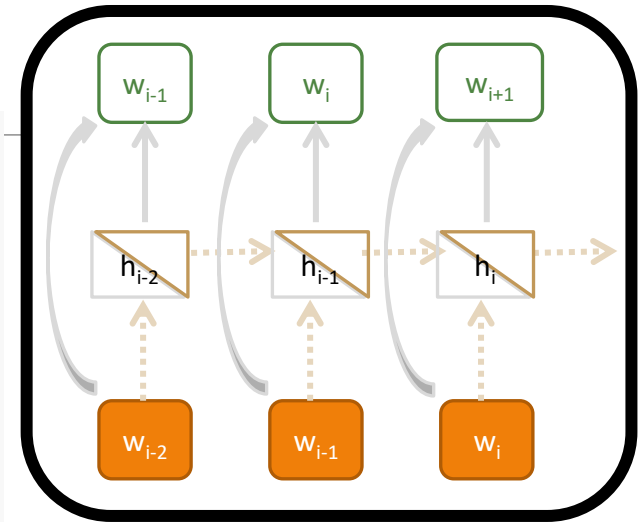
        self.hidden_size = hidden_size

        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.h2o = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        hidden = self.i2h(combined)
        output = self.h2o(hidden)
        output = self.softmax(output)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, self.hidden_size)

n_hidden = 128
rnn = RNN(n_letters, n_hidden, n_categories)
```



encode

Defining A Simple RNN in Python

http://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

```
import torch.nn as nn

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

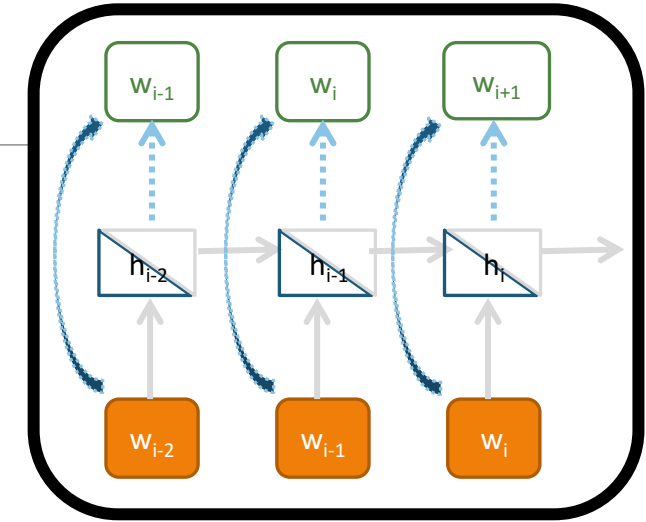
        self.hidden_size = hidden_size

        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.h2o = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        hidden = self.i2h(combined)
        output = self.h2o(hidden)
        output = self.softmax(output)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, self.hidden_size)

n_hidden = 128
rnn = RNN(n_letters, n_hidden, n_categories)
```



decode

Training A Simple RNN in Python

http://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

Negative log-likelihood

(we'll talk about this)

```
criterion = nn.NLLLoss()

learning_rate = 0.005 # If you set this too high, it might explode. If too low, it might not learn

def train(category_tensor, line_tensor):
    hidden = rnn.initHidden()

    rnn.zero_grad()

    for i in range(line_tensor.size()[0]):
        output, hidden = rnn(line_tensor[i], hidden)

    loss = criterion(output, category_tensor)
    loss.backward()

    # Add parameters' gradients to their values, multiplied by learning rate
    for p in rnn.parameters():
        p.data.add_(-learning_rate, p.grad.data)

    return output, loss.data[0]
```

Training A Simple RNN in Python

http://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

Negative log-likelihood

```
criterion = nn.NLLLoss()

learning_rate = 0.005 # If you set this too high, it might explode. If too low,

def train(category_tensor, line_tensor):
    hidden = rnn.initHidden()

    rnn.zero_grad()

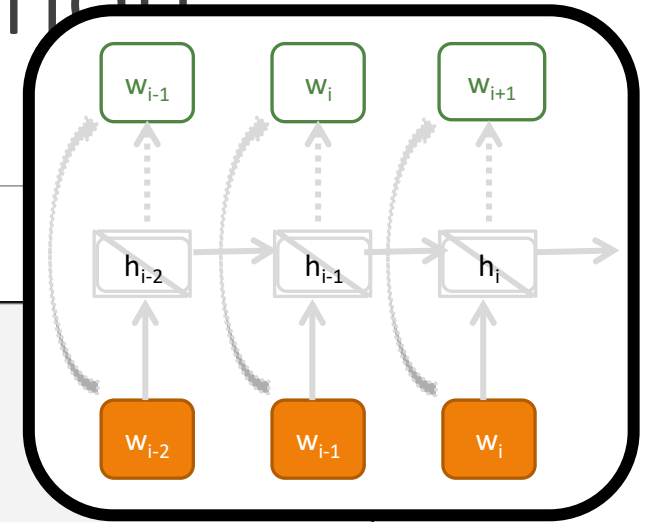
    for i in range(line_tensor.size()[0]):
        output, hidden = rnn(line_tensor[i], hidden)

    loss = criterion(output, category_tensor)
    loss.backward()

    # Add parameters' gradients to their values, multiplied by learning rate
    for p in rnn.parameters():
        p.data.add_(-learning_rate, p.grad.data)

    return output, loss.data[0]
```

get predictions



Training A Simple RNN in Python

http://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

Negative log-likelihood

```
criterion = nn.NLLLoss()

learning_rate = 0.005 # If you set this too high, it might explode. If too low, it might not learn

def train(category_tensor, line_tensor):
    hidden = rnn.initHidden()

    rnn.zero_grad()

    for i in range(line_tensor.size()[0]):
        output, hidden = rnn(line_tensor[i], hidden)

    loss = criterion(output, category_tensor)
    loss.backward()

    # Add parameters' gradients to their values, multiplied by learning rate
    for p in rnn.parameters():
        p.data.add_(-learning_rate, p.grad.data)

    return output, loss.data[0]
```

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 θ_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 ρ_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

Negative log-likelihood

```
criterion = nn.NLLLoss()

learning_rate = 0.005 # If you set this too high, it might explode. If too low, it might not learn

def train(category_tensor, line_tensor):
    hidden = rnn.initHidden()

    rnn.zero_grad()

    for i in range(line_tensor.size()[0]):
        output, hidden = rnn(line_tensor[i], hidden)

    loss = criterion(output, category_tensor)
    loss.backward()

    # Add parameters' gradients to their values, multiplied by learning rate
    for p in rnn.parameters():
        p.data.add_(-learning_rate, p.grad.data)

    return output, loss.data[0]
```

get predictions

eval predictions

compute gradient

Set $t = 0$

Pick a starting value θ_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 ρ_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

Negative log-likelihood

```
criterion = nn.NLLLoss()

learning_rate = 0.005 # If you set this too high, it might explode. If too low, it might not learn

def train(category_tensor, line_tensor):
    hidden = rnn.initHidden()

    rnn.zero_grad()

    for i in range(line_tensor.size()[0]):
        output, hidden = rnn(line_tensor[i], hidden)

    loss = criterion(output, category_tensor)
    loss.backward()

    # Add parameters' gradients to their values, multiplied by learning rate
    for p in rnn.parameters():
        p.data.add_(-learning_rate, p.grad.data)

    return output, loss.data[0]
```

get predictions

eval predictions

compute gradient

perform SGD

Set $t = 0$
Pick a starting value θ_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 ρ_t
4. Set $\theta_{t+1} = \theta_t - \rho_t * g_t$
5. Set $t += 1$

Suggested Implementation Changes

```
import torch.nn as nn
from torch.autograd import Variable

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

        self.hidden_size = hidden_size

        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.i2o = nn.Linear(input_size + hidden_size, output_size)
        self.softmax = nn.LogSoftmax()

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        hidden = self.i2h(combined)
        output = self.i2o(combined)
        output = self.softmax(output)
        return output, hidden

    def initHidden(self):
        return Variable(torch.zeros(1, self.hidden_size))

n_hidden = 128
rnn = RNN(n_letters, n_hidden, n_classes)
```

current Pytorch refers
to this a “cell”

nn.CrossEntropyLoss()

```
criterion = nn.NLLLoss() nn.CrossEntropyLoss()

learning_rate = 0.005 # If you set this too high, it might explode. If too low, it might not learn

def train(category_tensor, line_tensor):
    hidden = rnn.initHidden()

    rnn.zero_grad()

    for i in range(line_tensor.size()[0]):
        output, hidden = rnn(line_tensor[i], hidden)

    loss = criterion(output, category_tensor)
    loss.backward()

    # Add parameters' gradients to their values, multiplied by learning rate
    for p in rnn.parameters():
        p.data.add_(-learning_rate, p.grad.data)

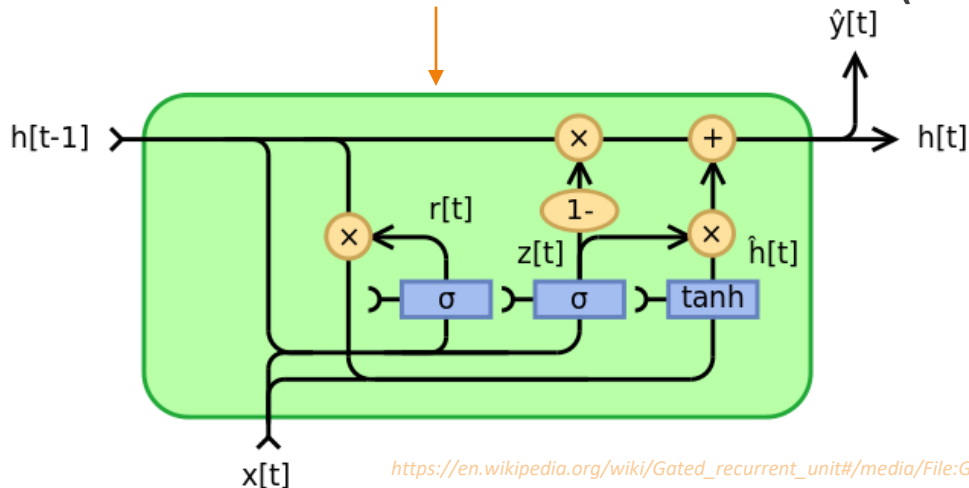
    return output, loss.data[0]
```

Another Solution: LSTMs/GRUs

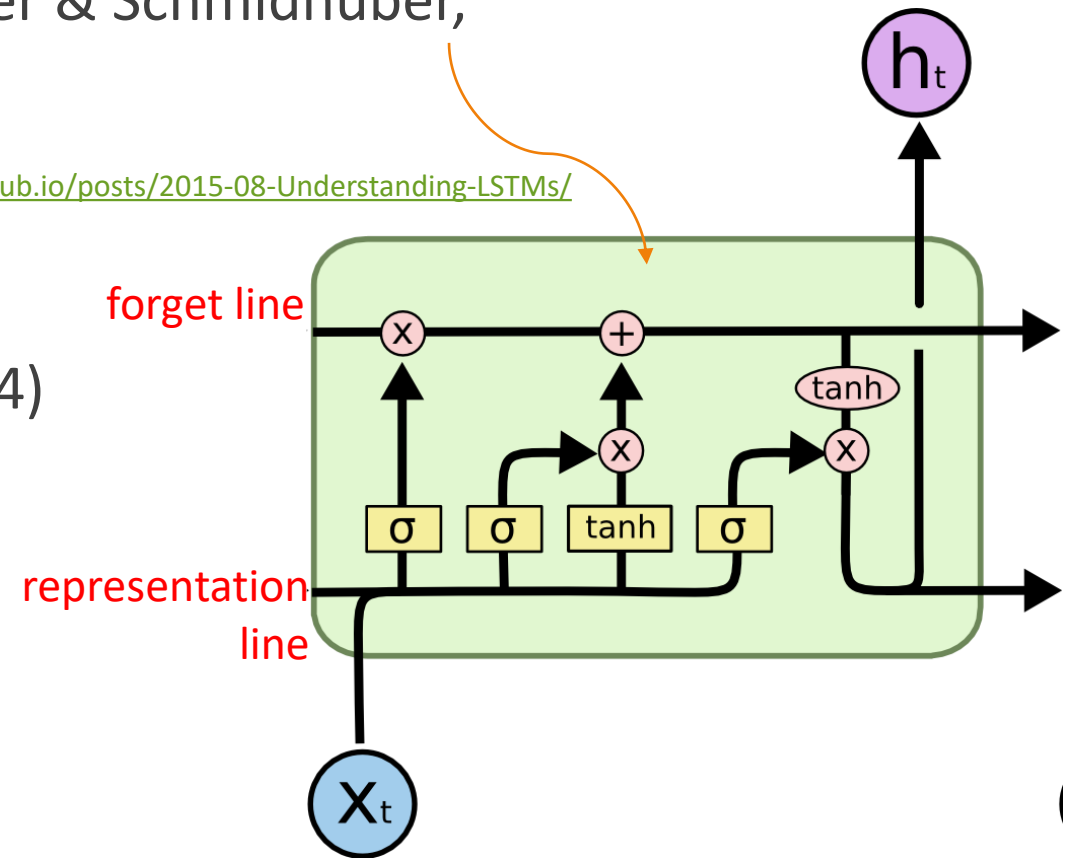
LSTM: Long Short-Term Memory (Hochreiter & Schmidhuber, 1997)

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

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

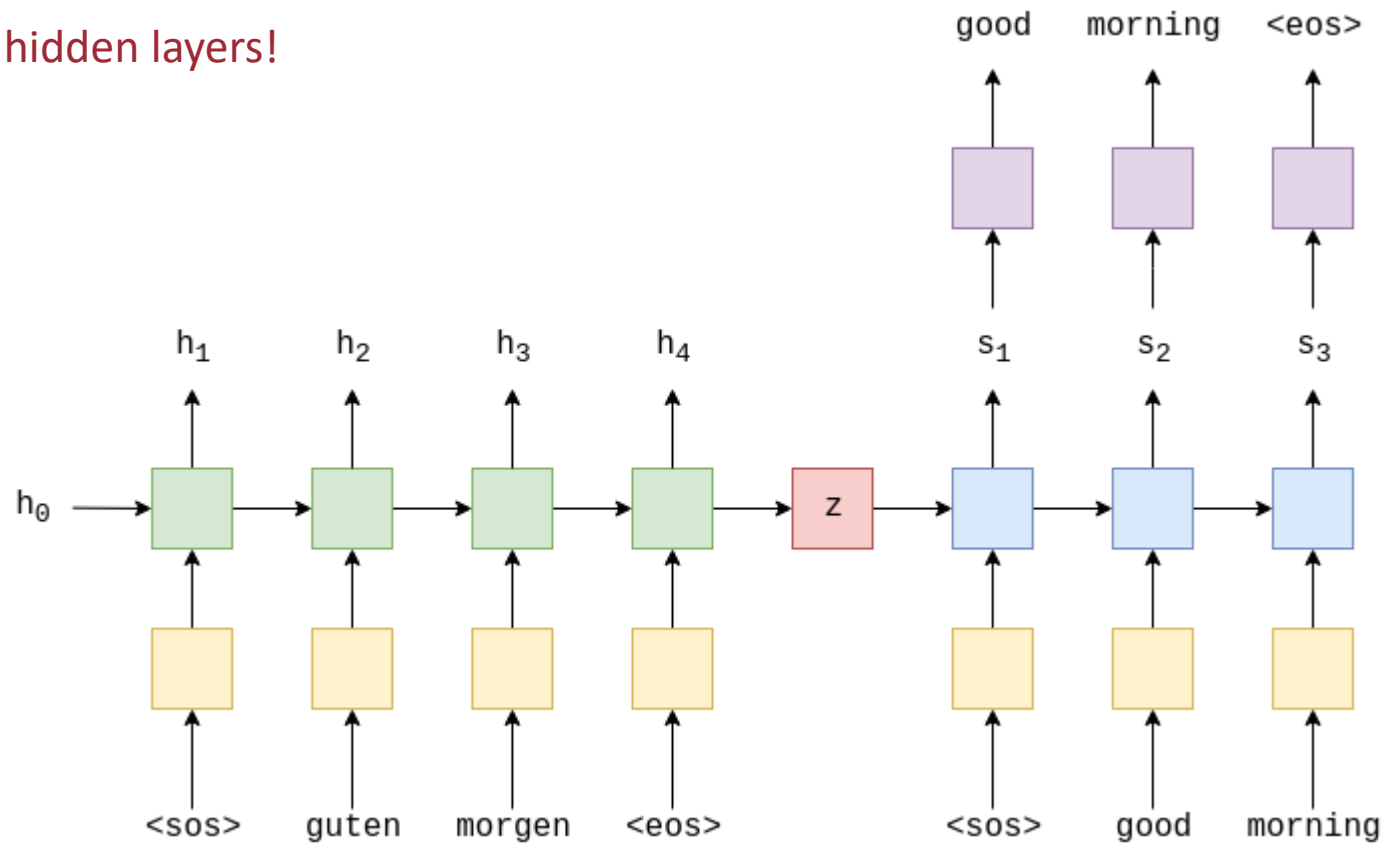


https://en.wikipedia.org/wiki/Gated_recurrent_unit#/media/File:Gated_Recurrent_Unit_base_type.svg



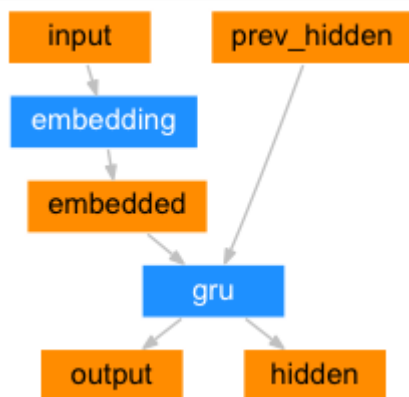
Sequence-to-Sequence

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



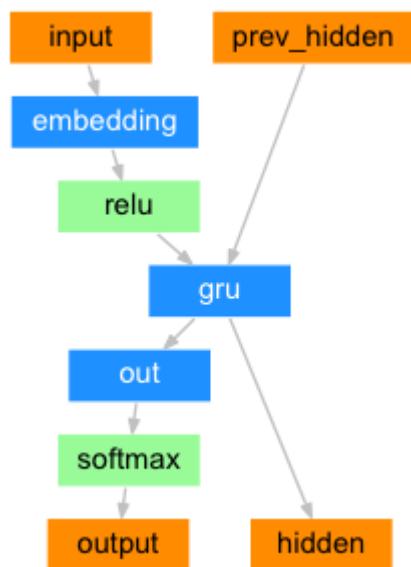
```
class EncoderRNN(nn.Module):
    def __init__(self, input_size, hidden_size, dropout_p=0.1):
        super(EncoderRNN, self).__init__()
        self.hidden_size = hidden_size

        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

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


Seq2Seq Tutorial

Direct link:

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


<https://github.com/bentrevett/pytorch-seq2seq>

Tutorials

- 1 - [Sequence to Sequence Learning with Neural Networks](#)  [Open in Colab](#)

This first tutorial covers the workflow of a seq2seq project with PyTorch. We'll cover the basics of seq2seq networks using encoder-decoder models, how to implement these models in PyTorch, and how to use the datasets/spacy/torchtext/evaluate libraries to do all of the heavy lifting. The model itself will be based off an implementation of [Sequence to Sequence Learning with Neural Networks](#), which uses multi-layer LSTMs.

- 2 - [Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation](#)

 [Open in Colab](#)

Now we have the basic workflow covered, this tutorial will focus on improving our results. Building on our knowledge of PyTorch, we'll implement a second model, which helps with the information compression problem faced by encoder-decoder models. This model will be based off an implementation of [Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation](#), which uses GRUs.