CMSC 473/673
Natural Language Processing

Instructor: Lara J. Martin (she/they)

TA: Duong Ta (he)

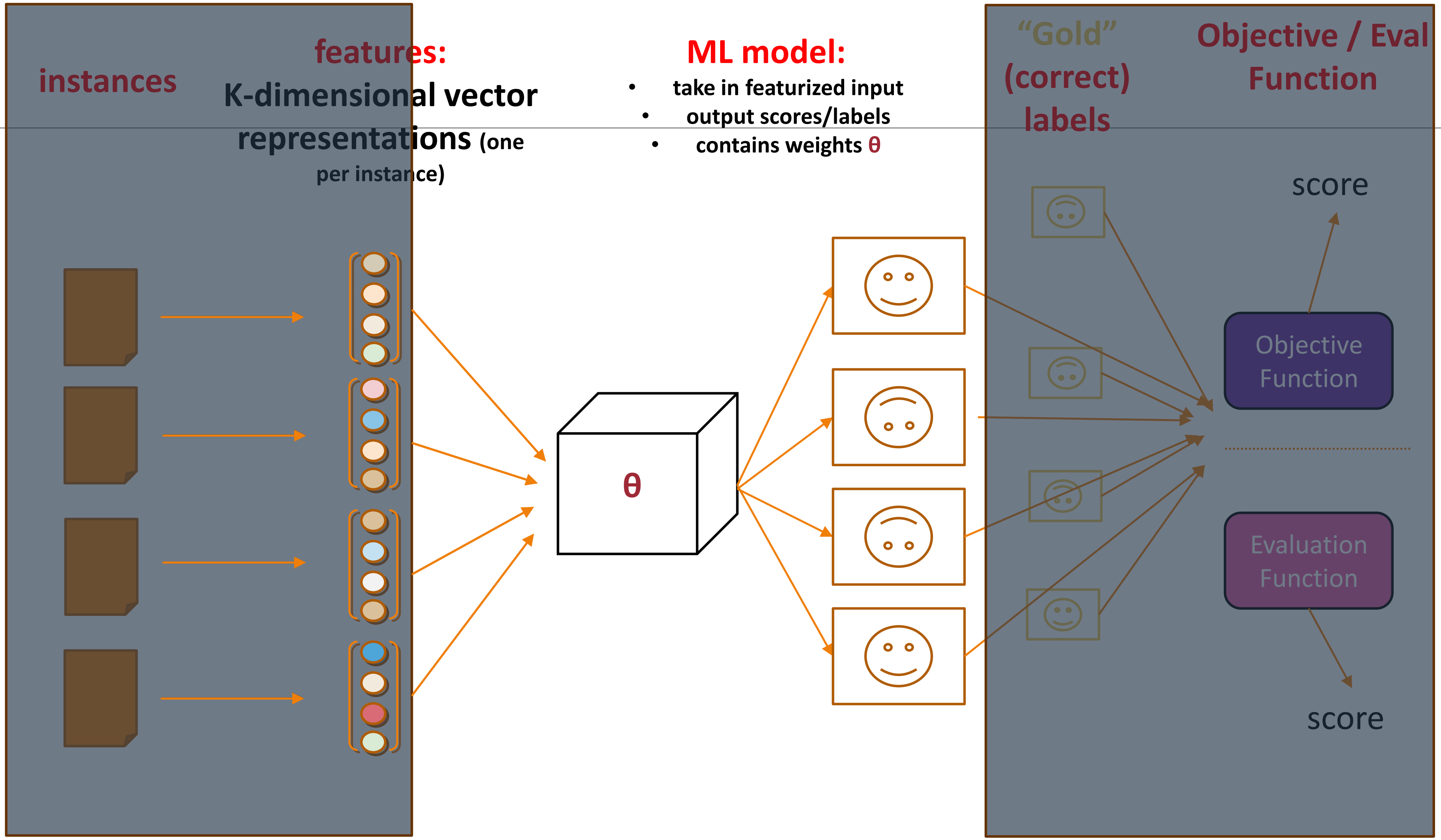
Slides modified from Dr. Frank Ferraro

Learning Objectives

Define the basic cell architecture of an RNN

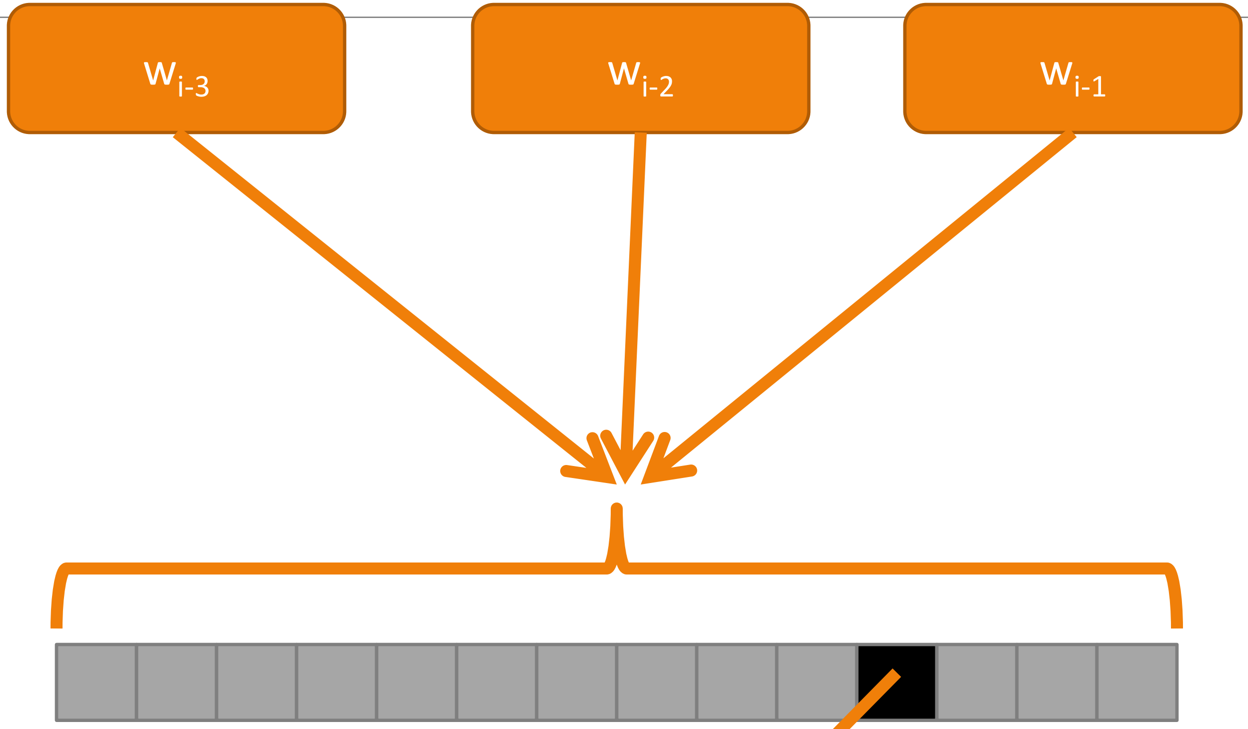
Backpropagate loss through an example RNN

Defining the Model



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)?

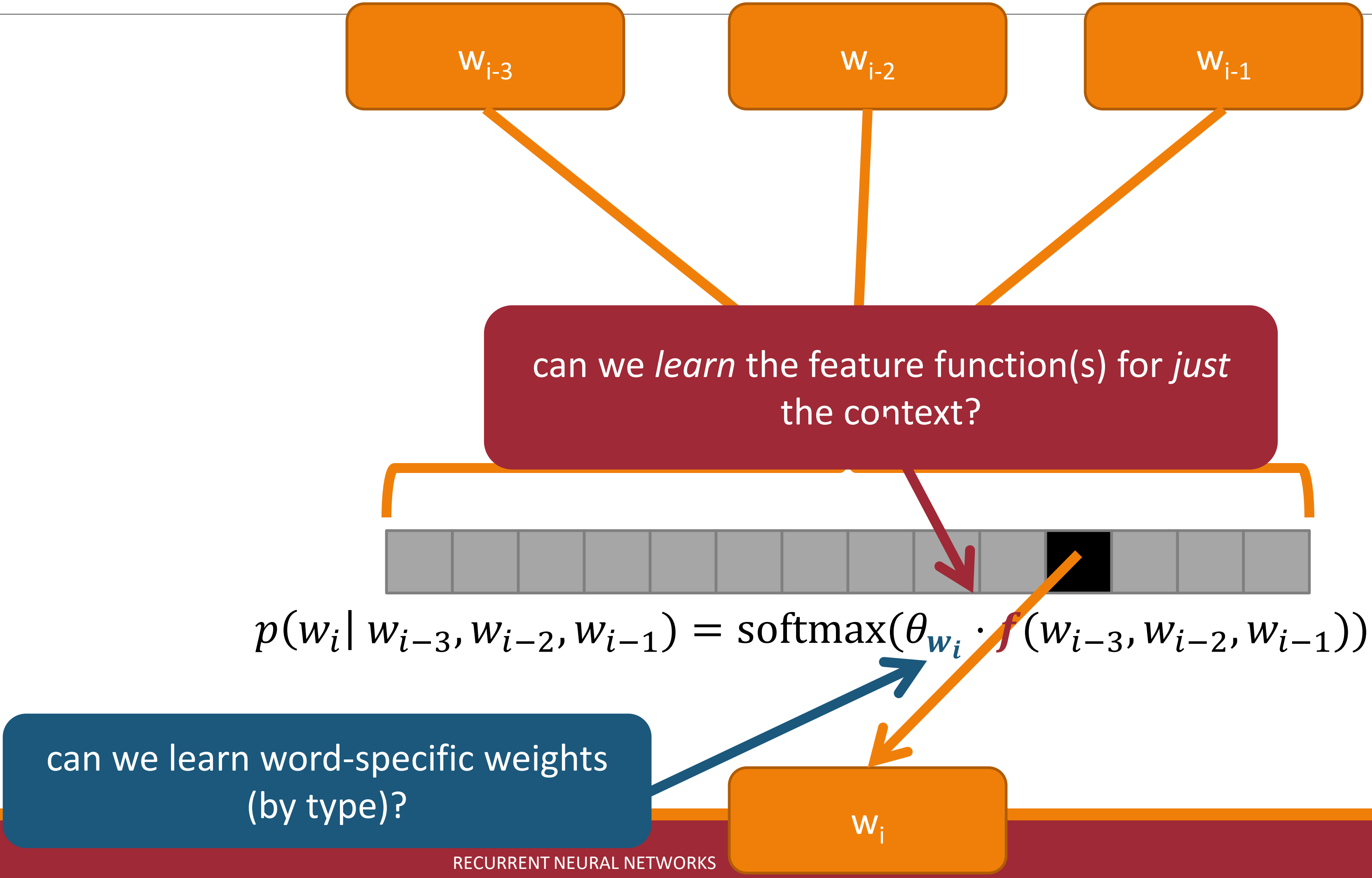
w_i

Review: Neural Language Models

given some context…

compute beliefs about what is likely…

predict the next word



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

Review: Neural Language Models

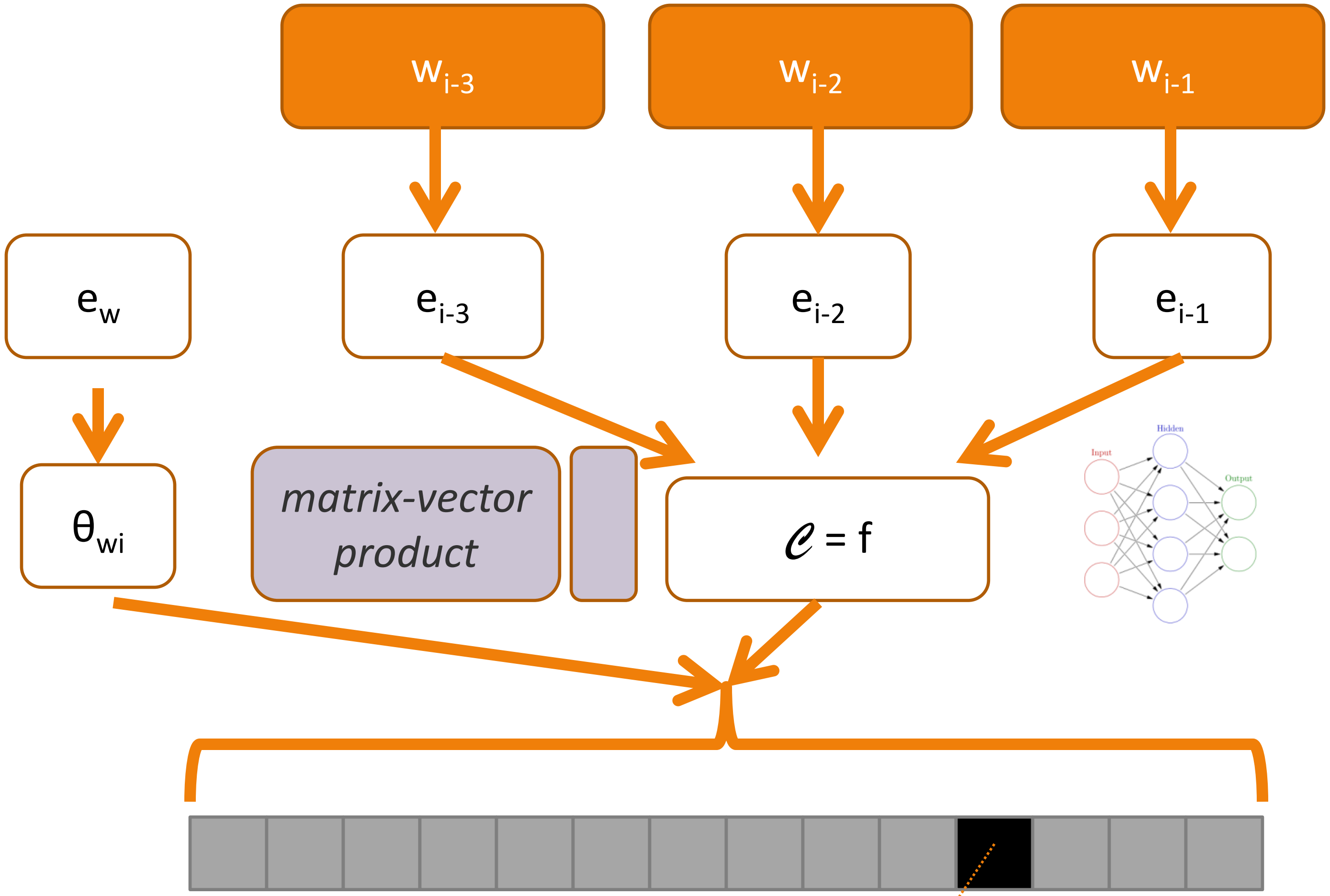
given some context…

create/use “distributed representations”…

combine these representations…

compute beliefs about what is likely…

predict the next word



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

w_i

LM Comparison

COUNT-BASED

Class-specific

MAXENT

Class-based

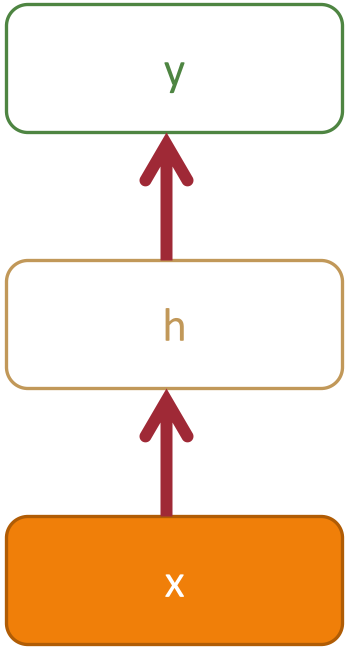
Uses features

NEURAL

Class-based

Uses *embedded* features

Network Types: Flat Input, Flat Output

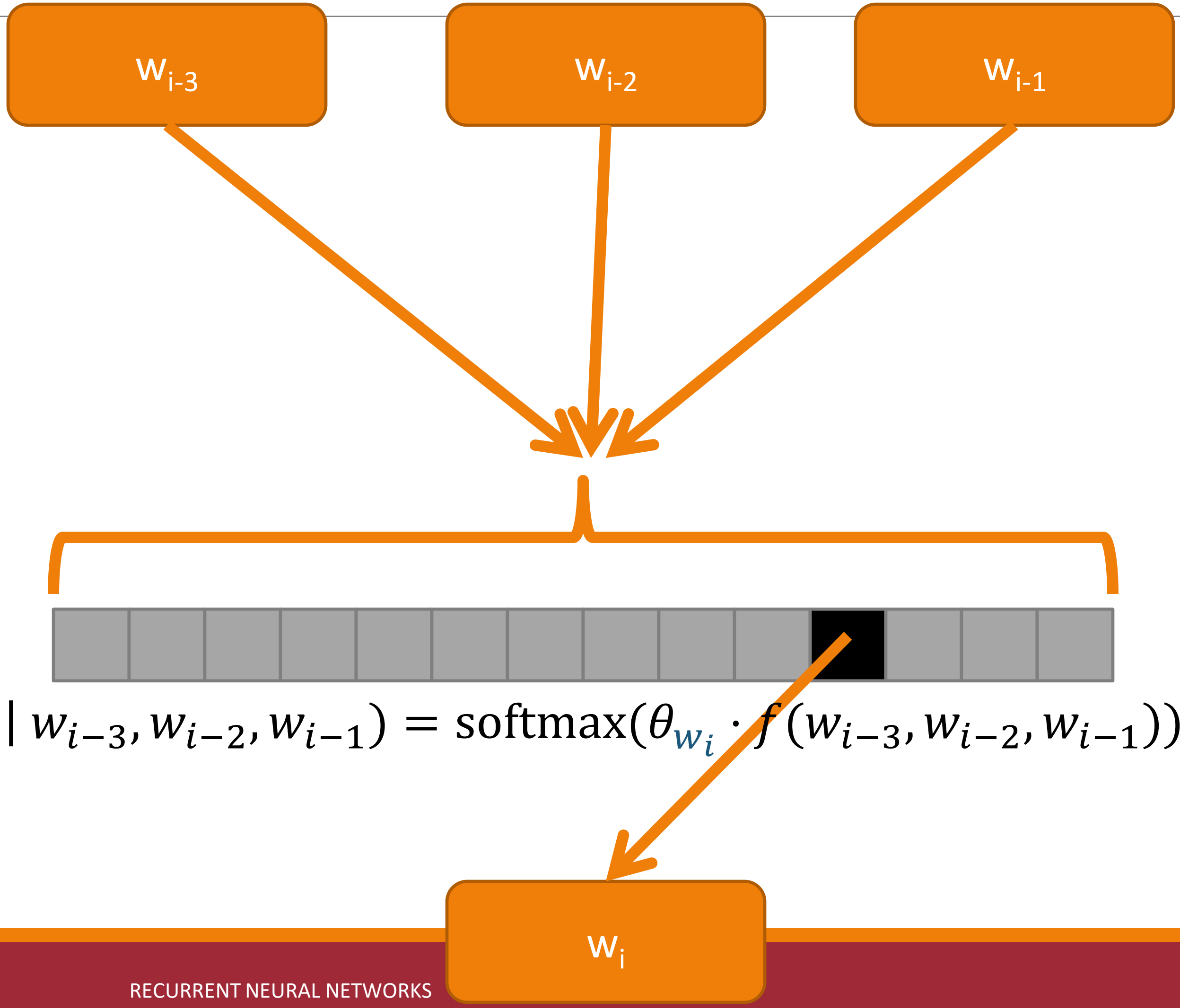


1. Feed forward

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

Maxent Language Models

given some context…



compute beliefs about what is likely…

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

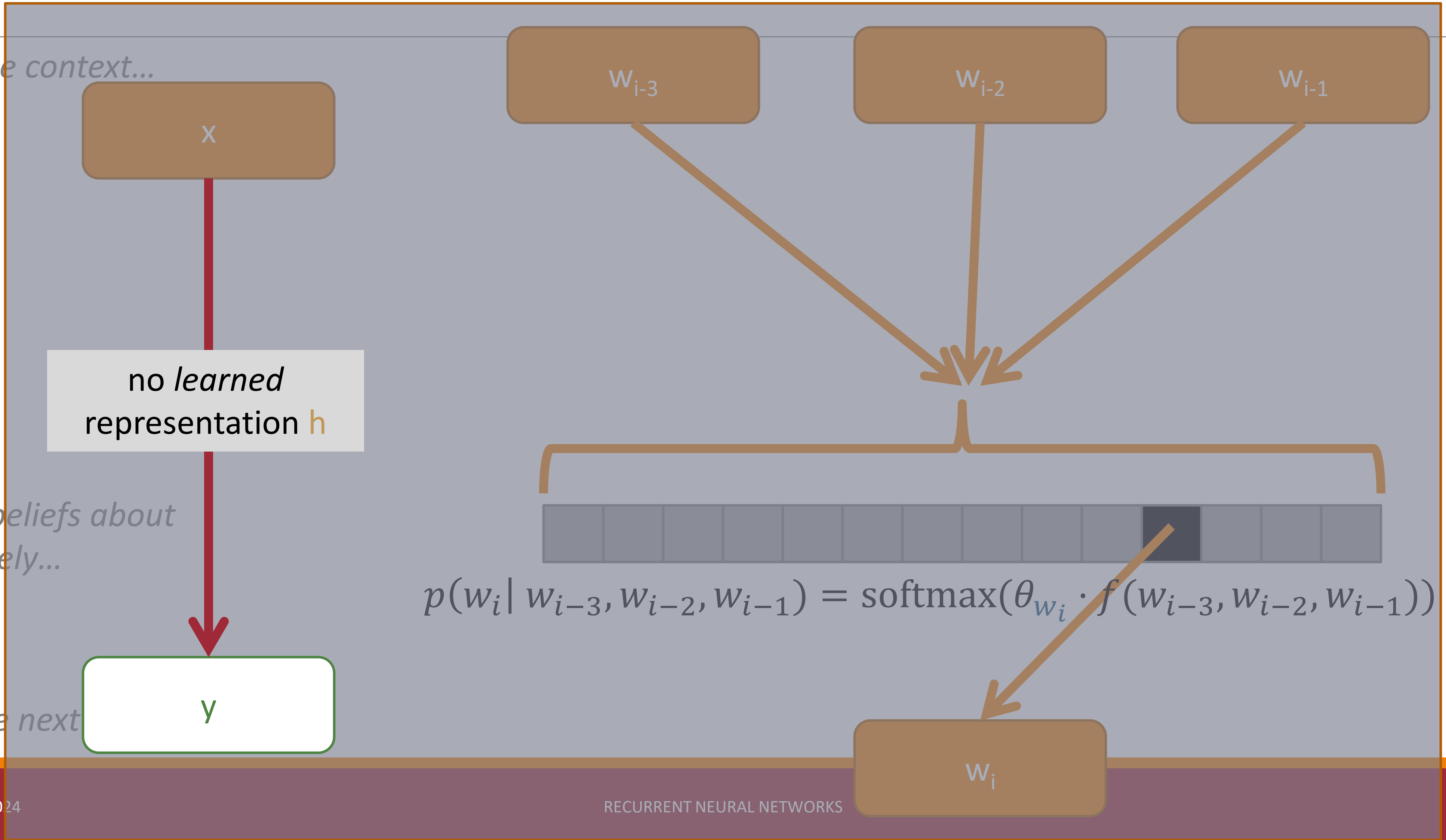
predict the next word

Maxent Language Models

given some context…

compute beliefs about what is likely…

predict the next



Neural Language Models

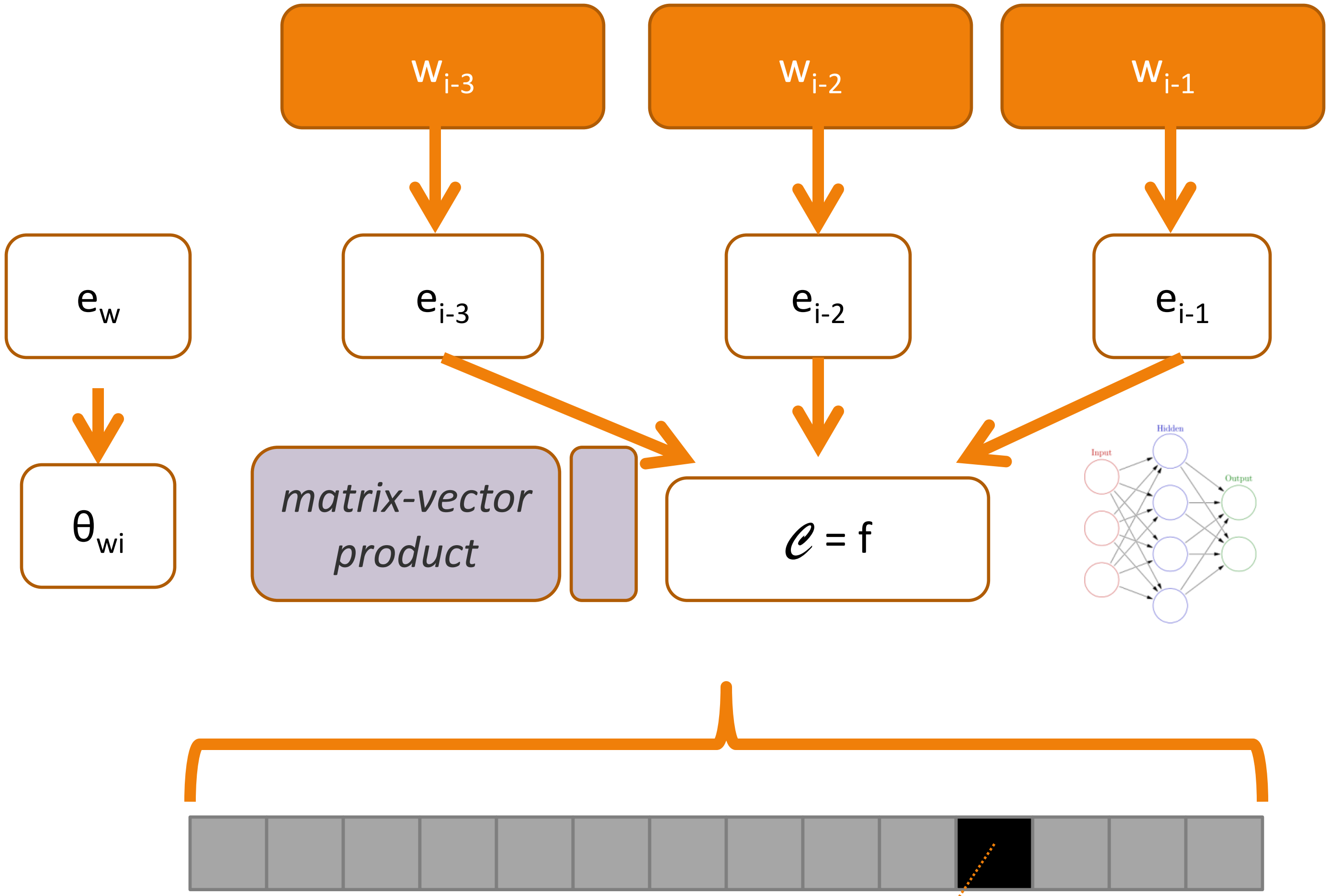
given some context…

create/use "distributed representations"…

combine these representations…

compute beliefs about what is likely…

predict the next word



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

w_i

Neural Language Models

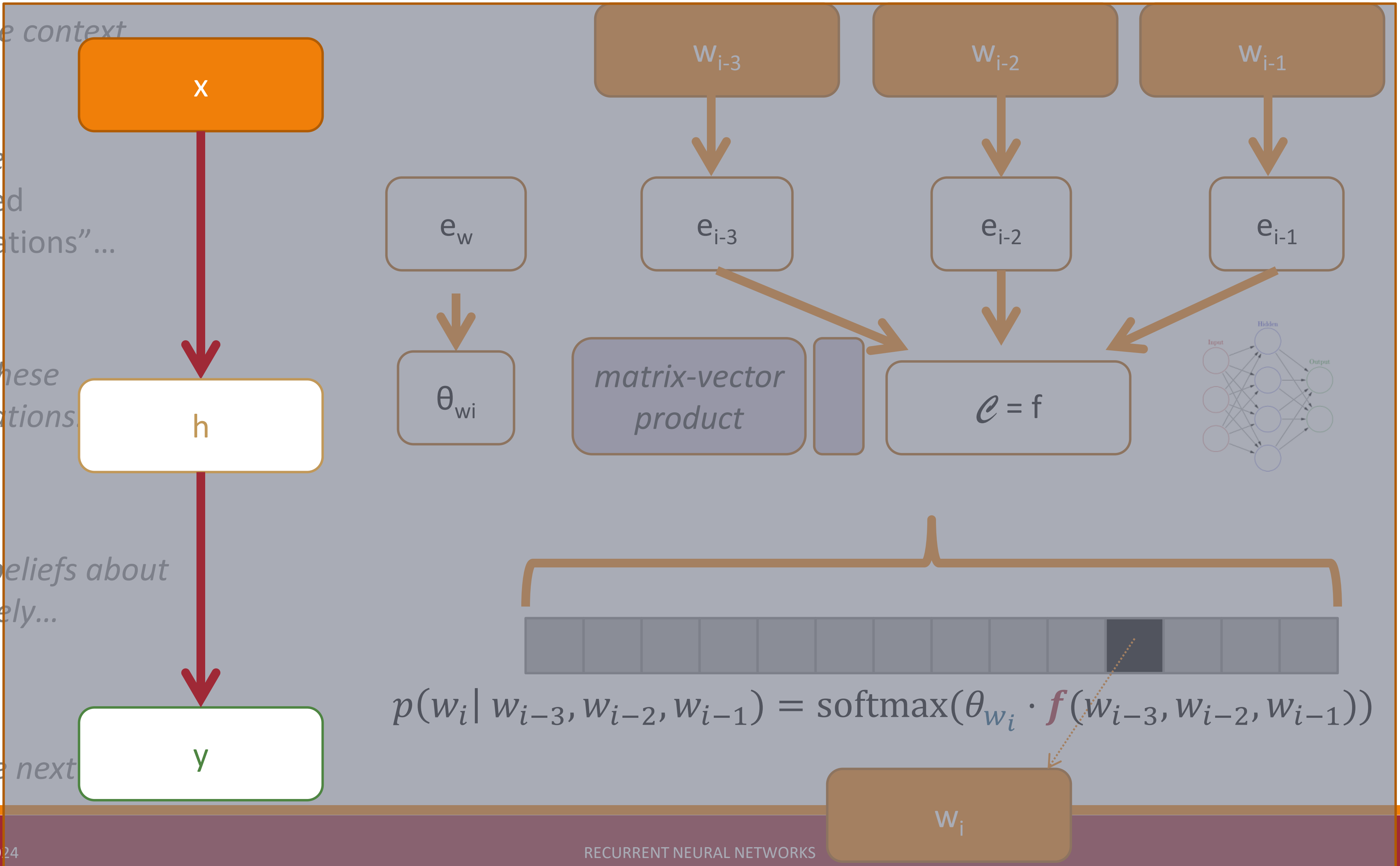
given some context…

create/use "distributed representations"…

combine these representations…

compute beliefs about what is likely…

predict the next…



Common Types of Flat Input, Flat Output

Feed forward networks

Multilayer perceptrons (MLPs)

General Formulation:

Input: x
Compute:

$h_0 = x$
for layer l = 1 to L:
$h_l = f_l(W_l h_{l-1} + b_l)$ linear layer

hidden state at layer l

(non-linear) activation function at l

return $\arg\max_y \text{softmax}(\theta h_L)$

In Pytorch (torch.nn):

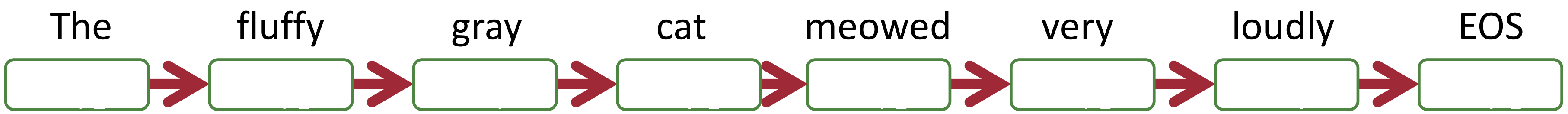
Activation functions:
https://pytorch.org/docs/stable/nn.html?highlight=activation#non-linear-activations-weighted-sum-nonlinearity

Linear layer:
https://pytorch.org/docs/stable/nn.html#linear-layers

torch.nn.Linear(
in_features=<dim of h_{l-1} >,
out_features=<dim of h_l >,
bias=<Boolean: include bias b_l >)

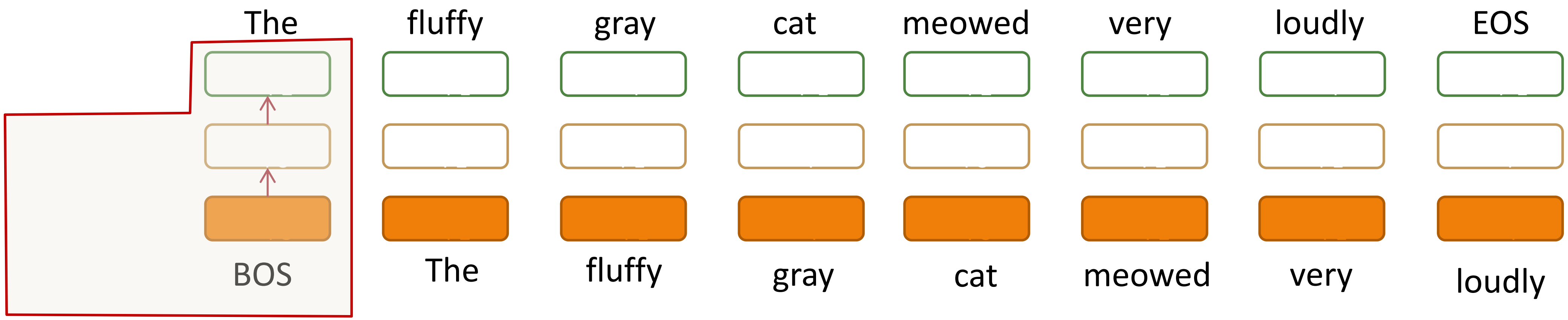
Review: A Neural N-Gram Model

The fluffy gray cat meowed very loudly



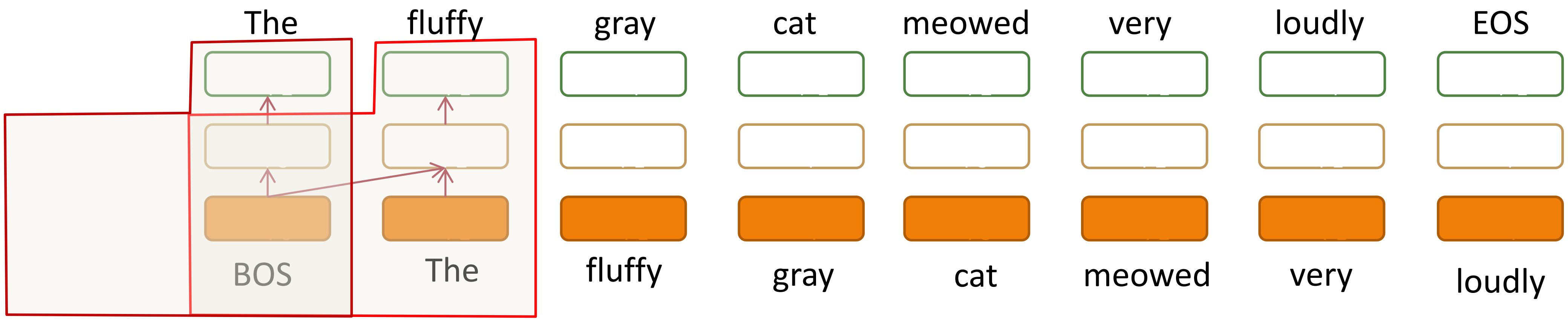
Review: A Neural N-Gram Model (N=3)

The fluffy gray cat meowed very loudly



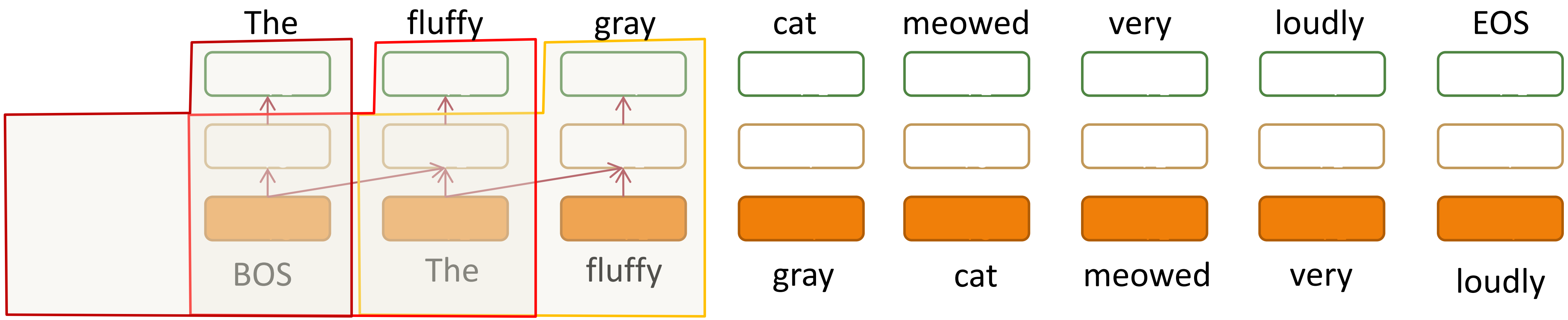
Review: A Neural N-Gram Model (N=3)

The fluffy gray cat meowed very loudly



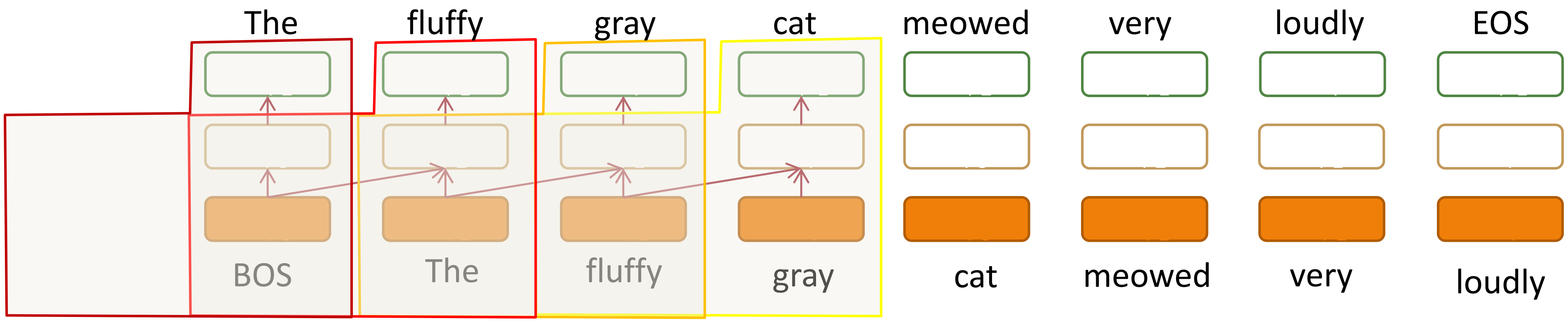
Review: A Neural N-Gram Model (N=3)

The fluffy gray cat meowed very loudly



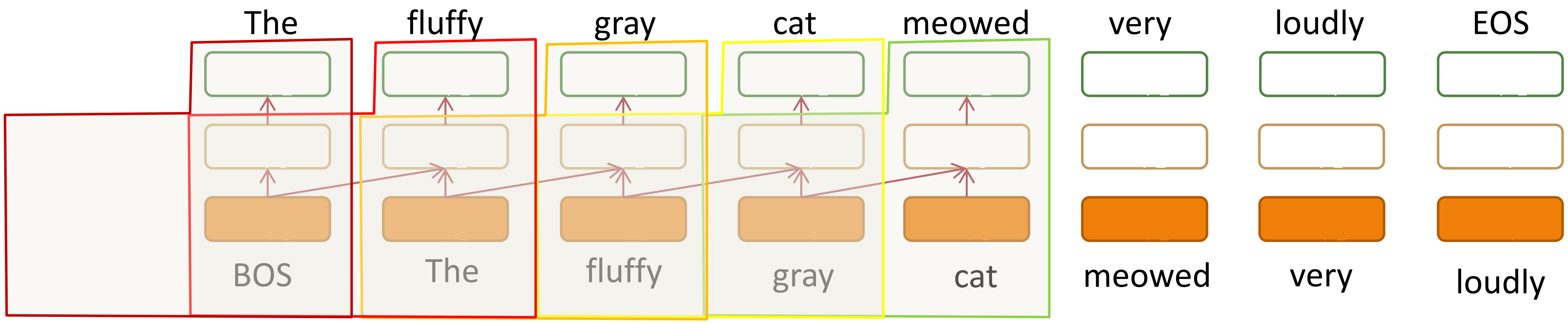
Review: A Neural N-Gram Model (N=3)

The fluffy gray cat meowed very loudly



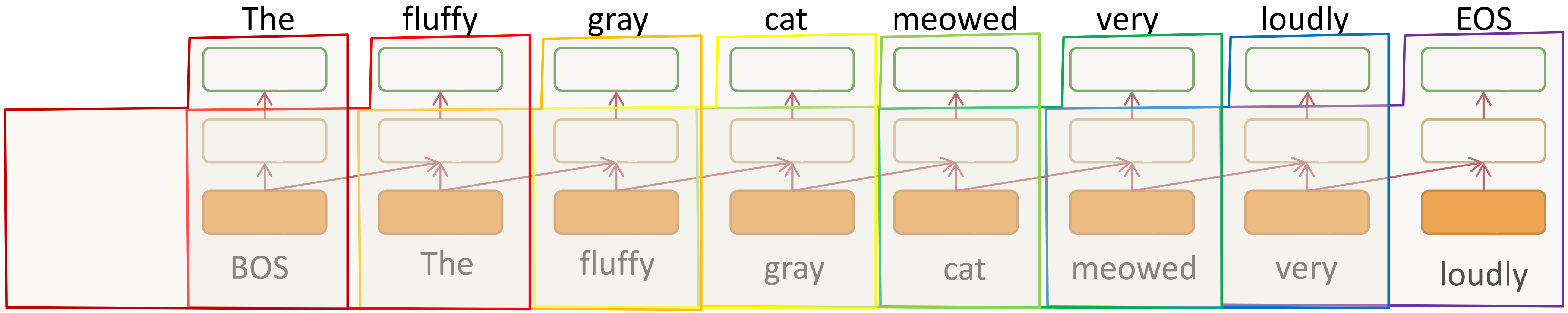
Review: A Neural N-Gram Model (N=3)

The fluffy gray cat meowed very loudly



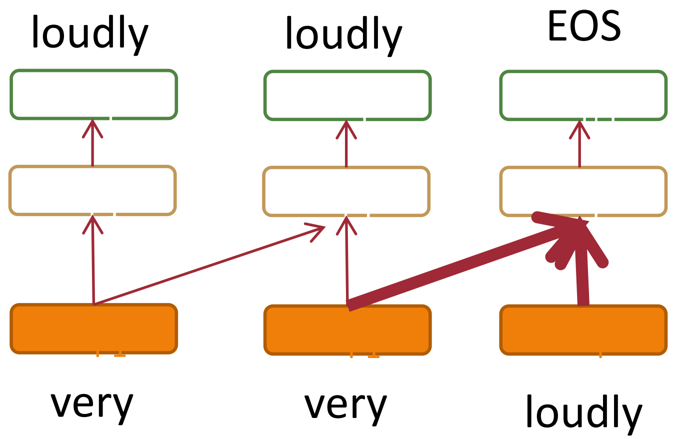
Review: A Neural N-Gram Model (N=3)

The fluffy gray cat meowed very loudly



A Neural N-Gram Model (N=3)

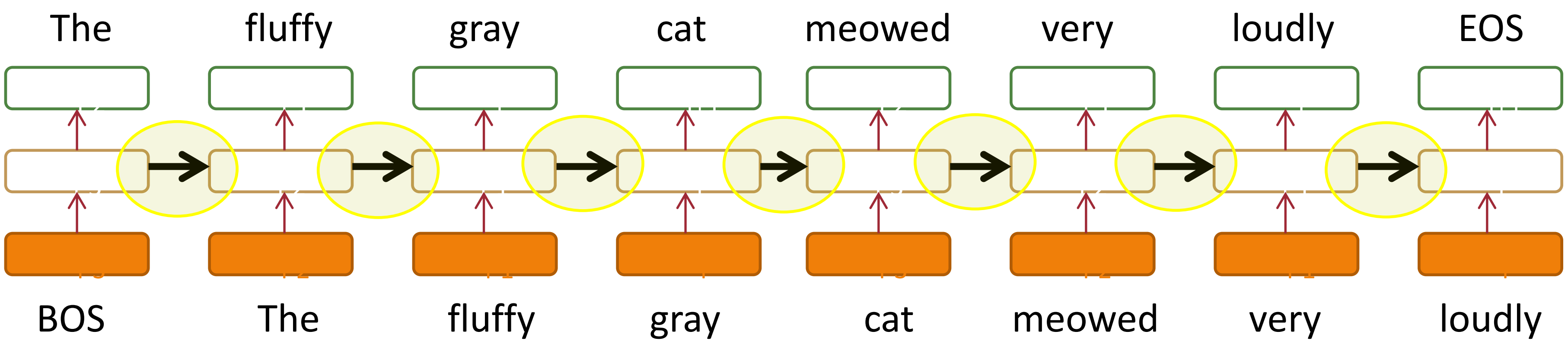
The fluffy gray cat meowed very loudly



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

A Recurrent Neural Language Model

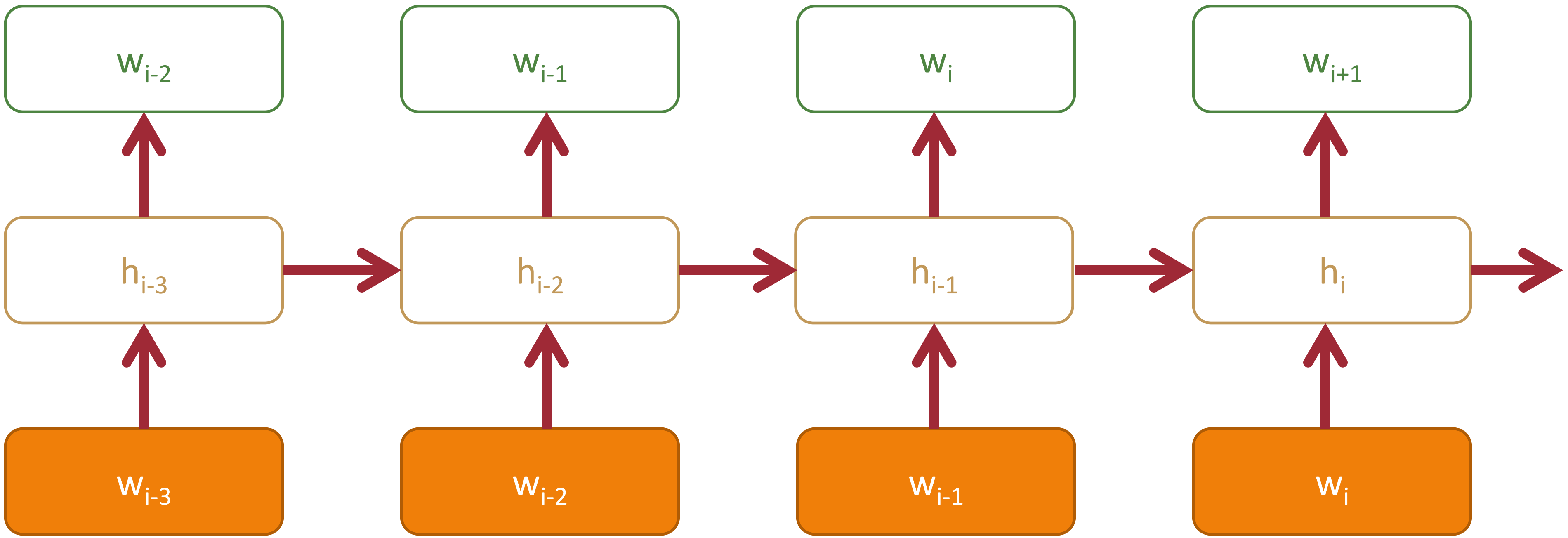
The fluffy gray cat meowed very loudly



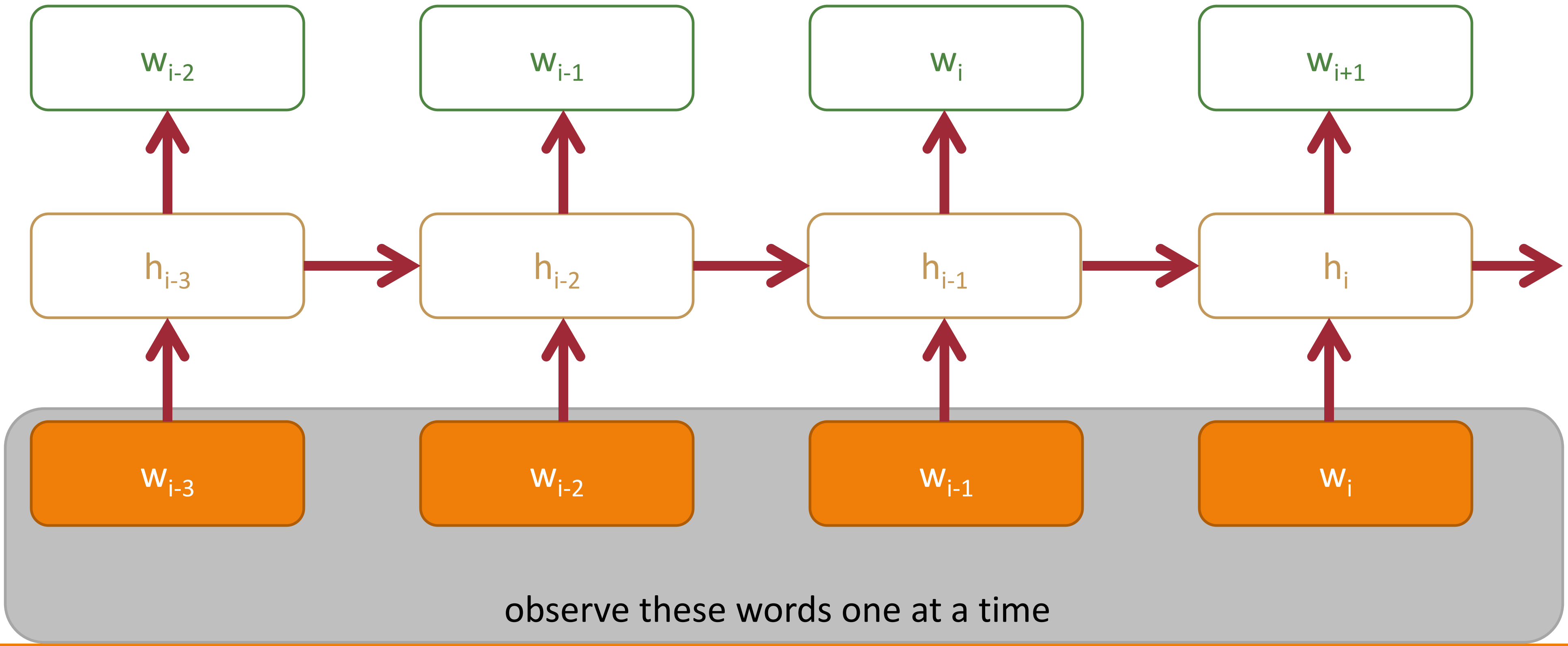
Critical issue: the amount of information flow is fundamentally restricted!!!

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

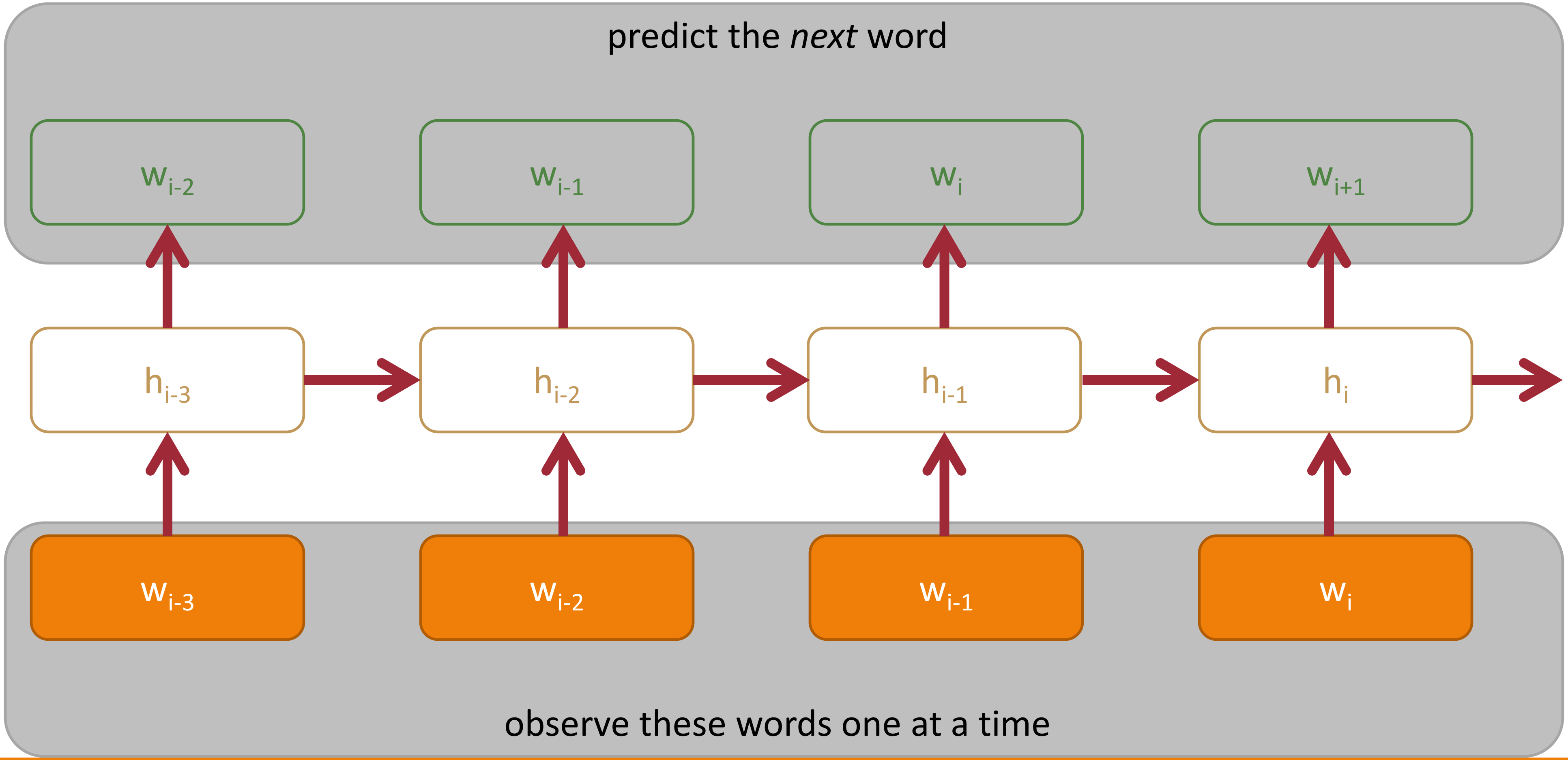
A Classic View of Recurrent Neural Language Modeling



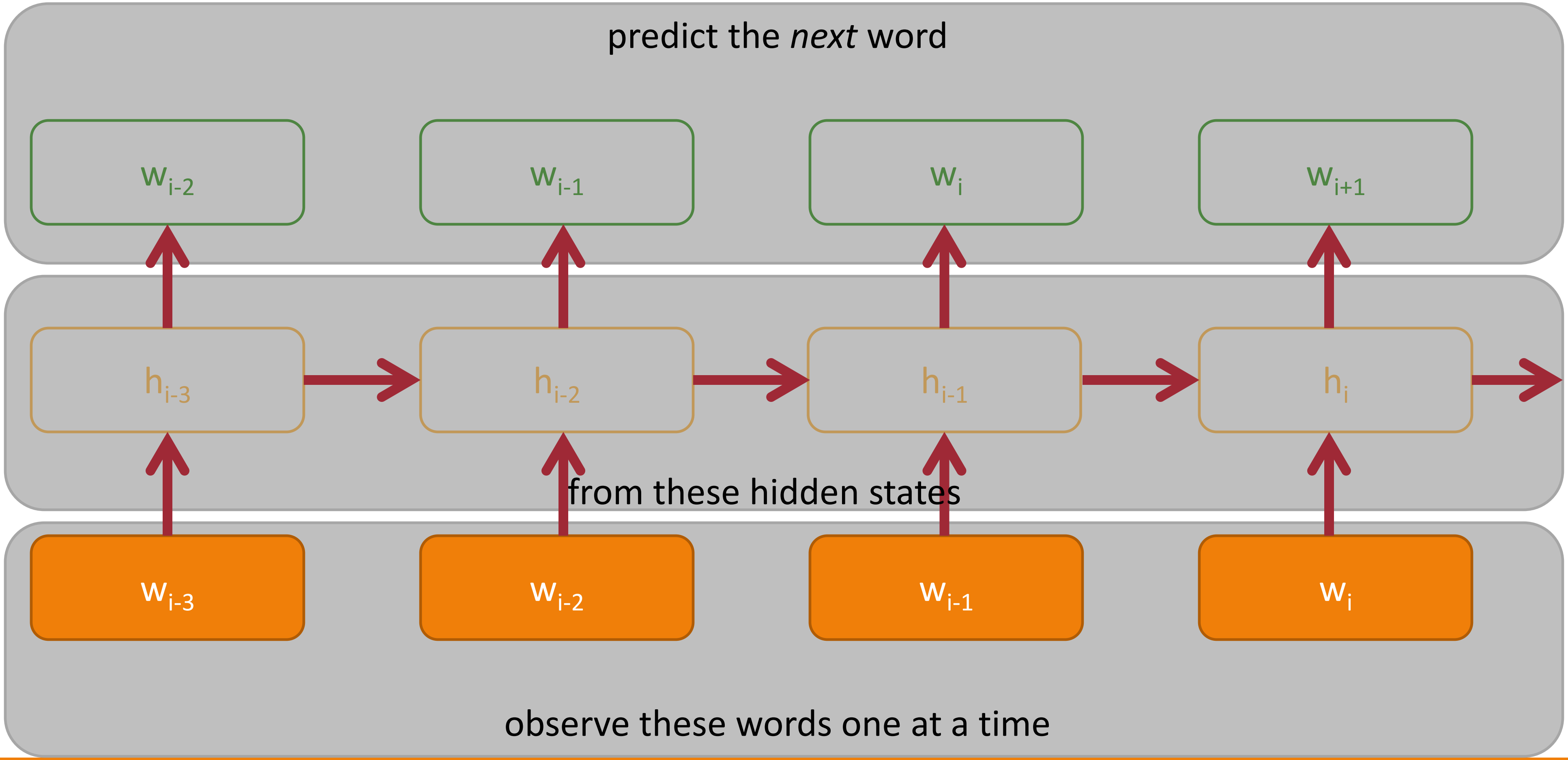
A Classic View of Recurrent Neural Language Modeling



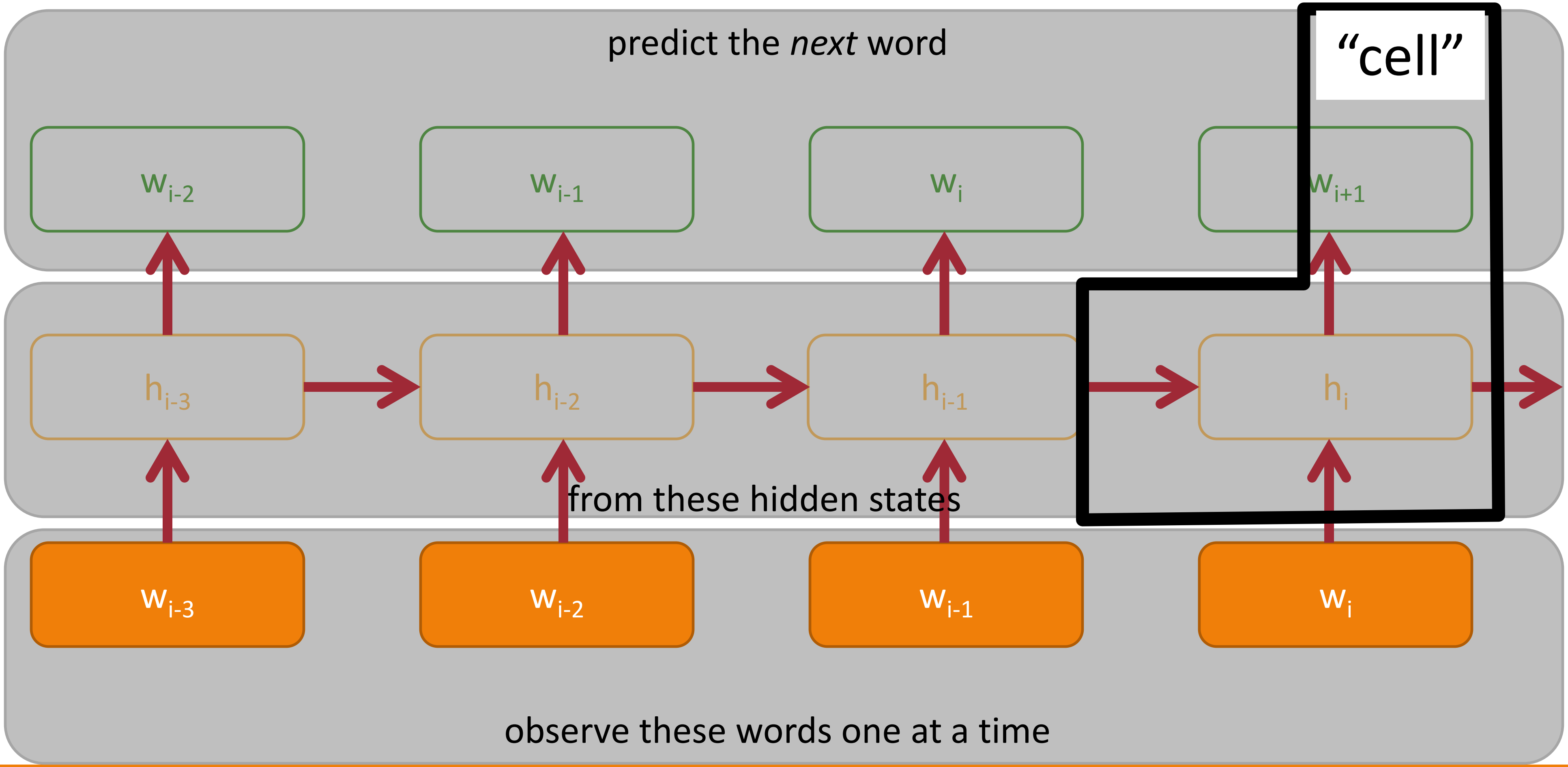
A Classic View of Recurrent Neural Language Modeling



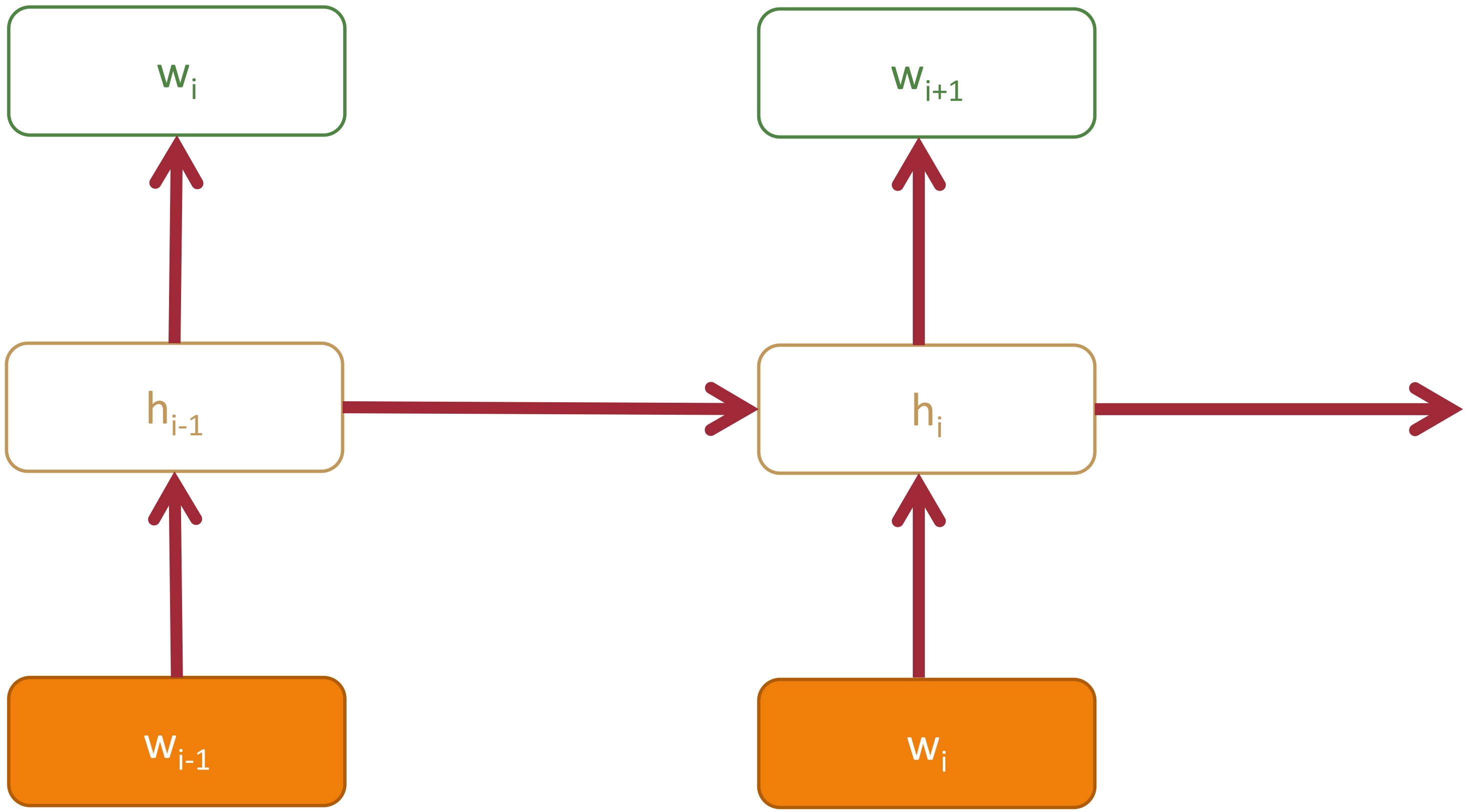
A Classic View of Recurrent Neural Language Modeling



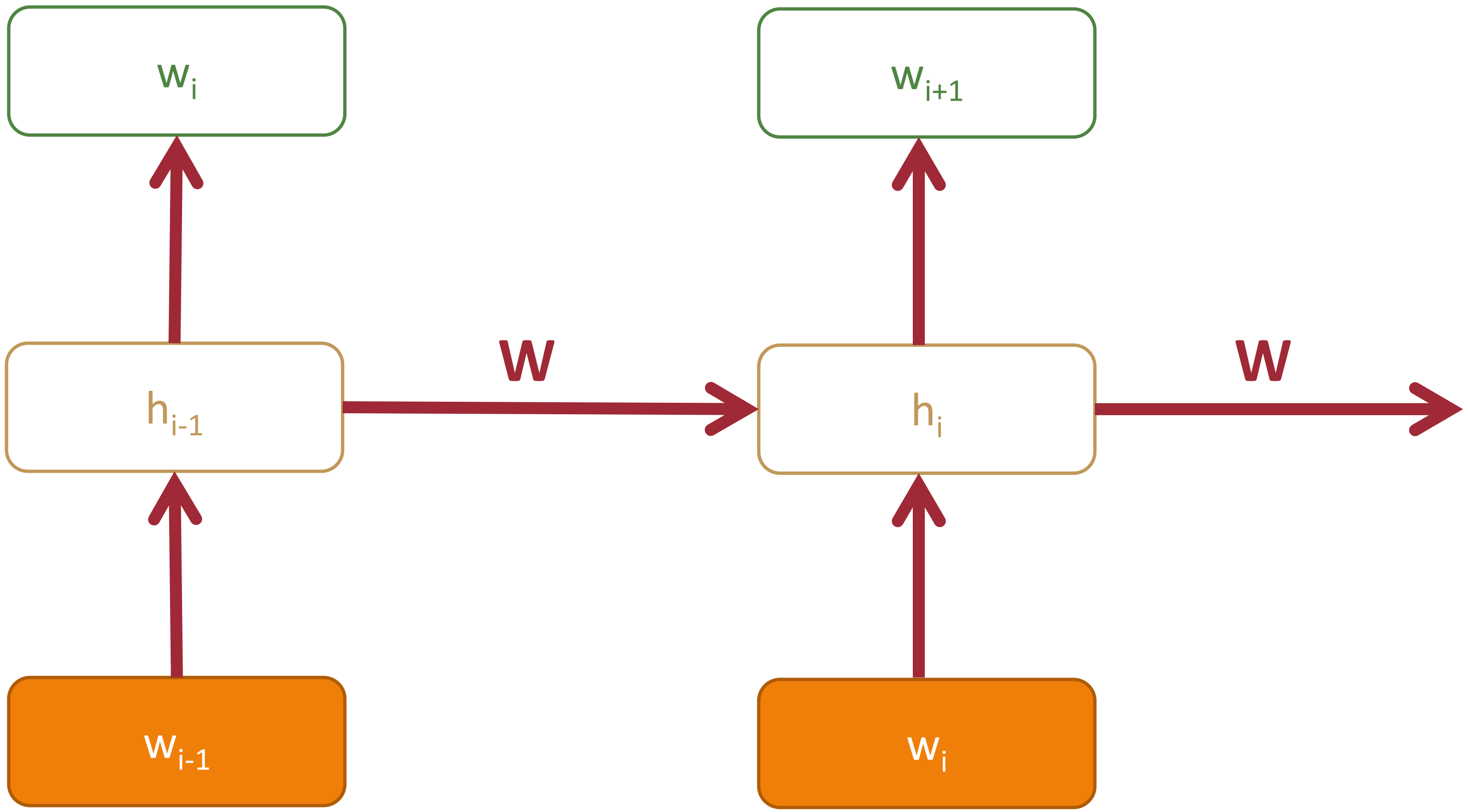
A Classic View of Recurrent Neural Language Modeling



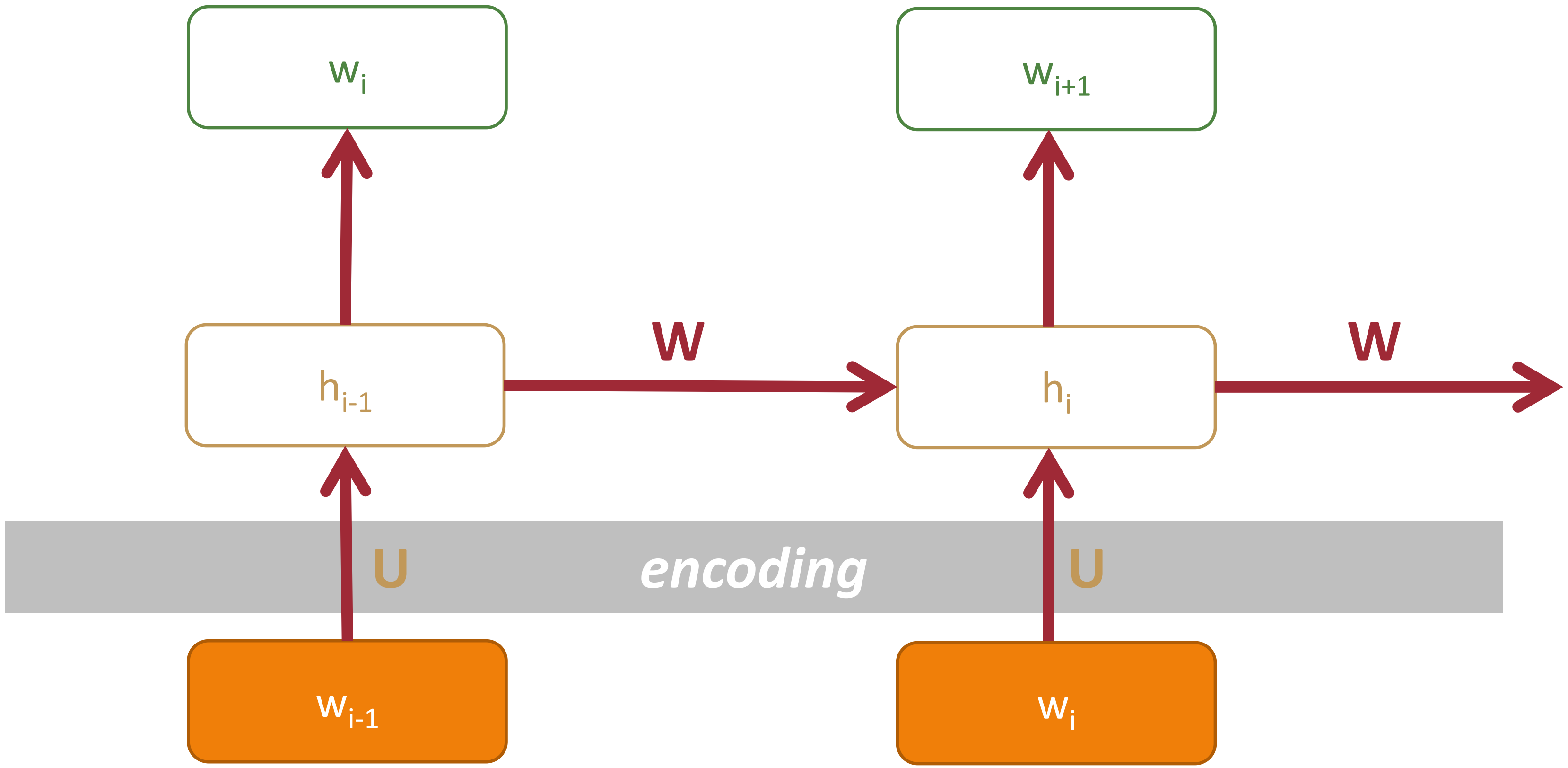
A Recurrent Neural Network Cell



A Recurrent Neural Network Cell

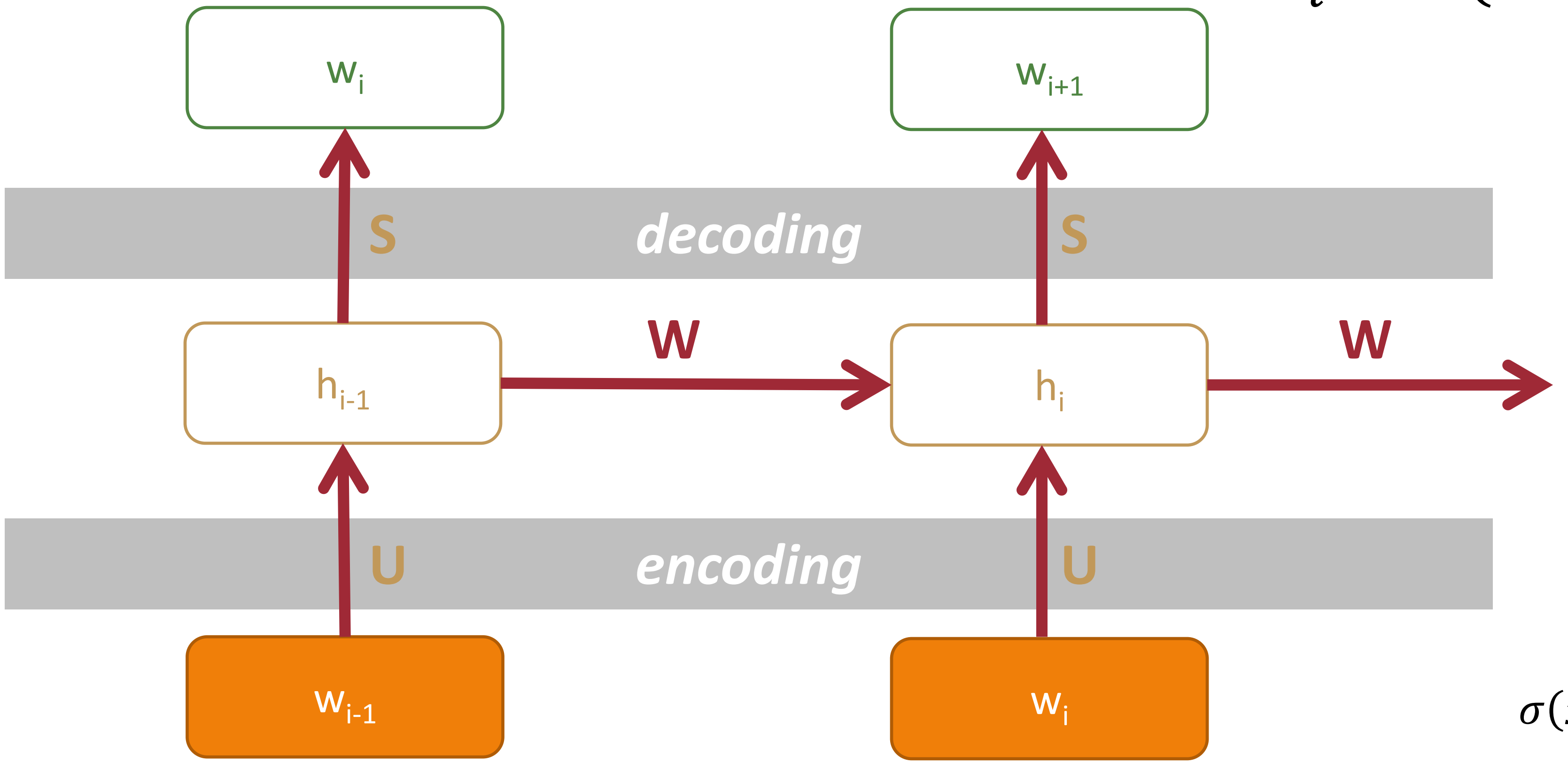


A Recurrent Neural Network Cell



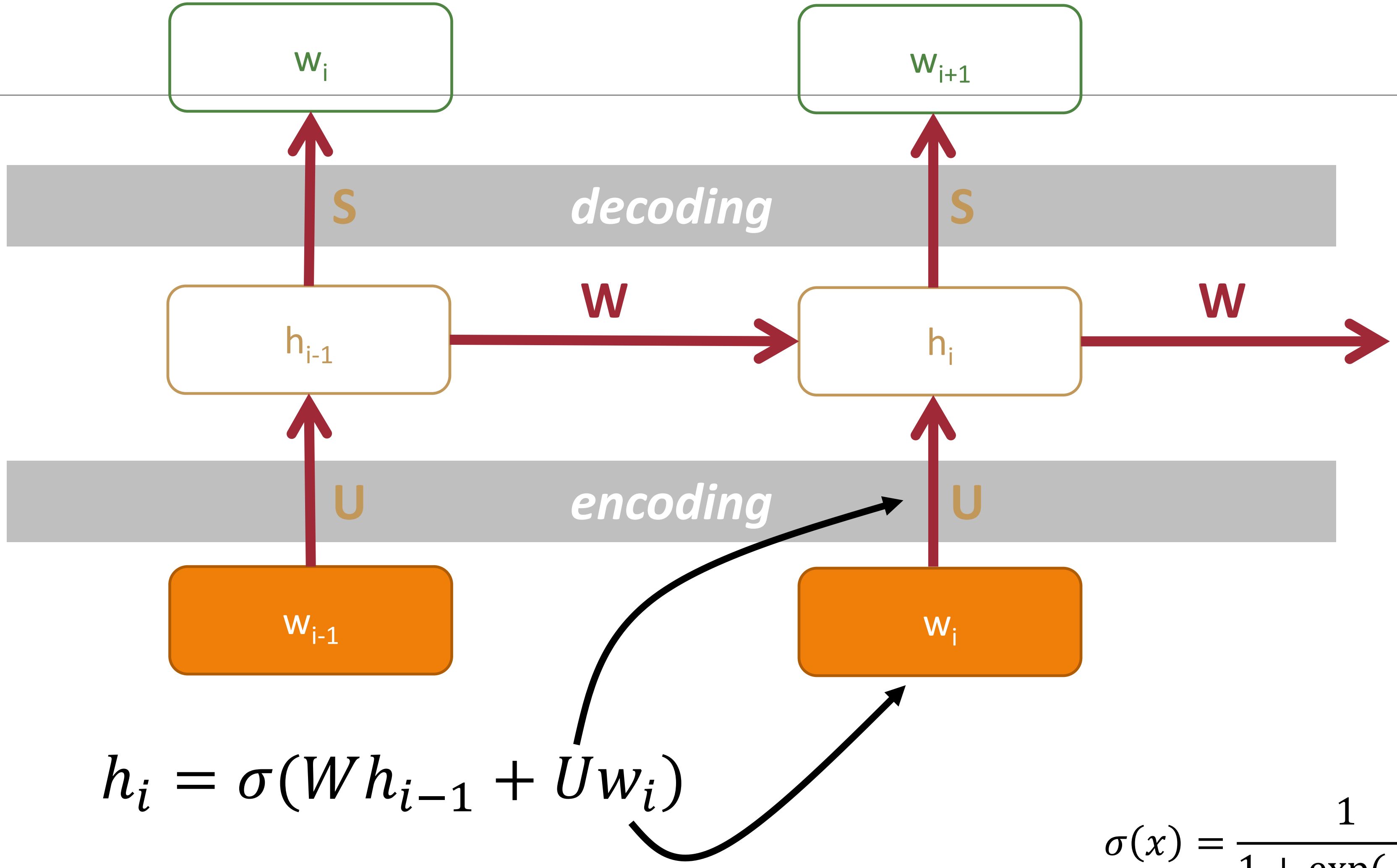
A Recurrent Neural Network Cell

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



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

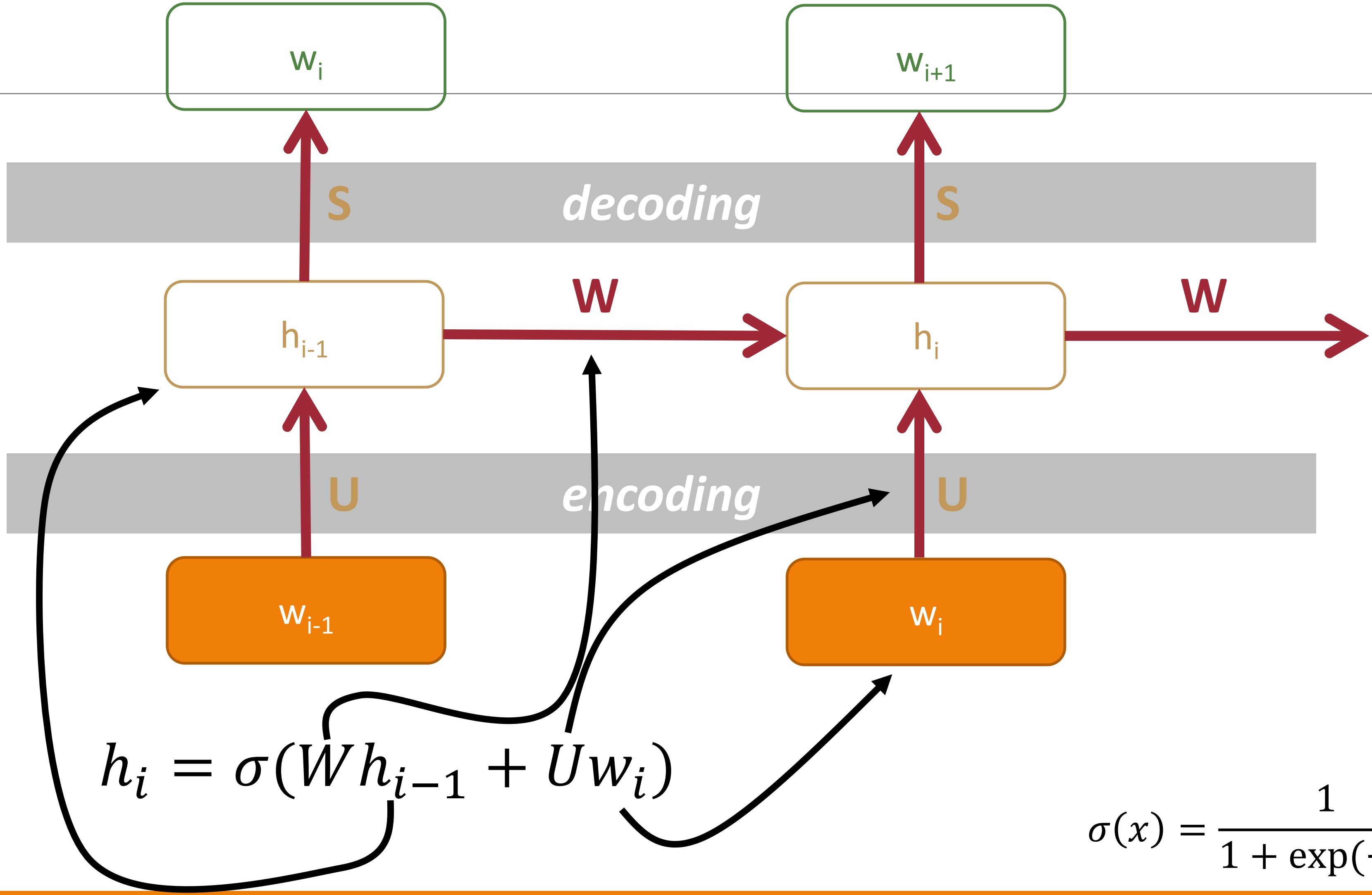
A *Simple* Recurrent Neural Network Cell



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

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

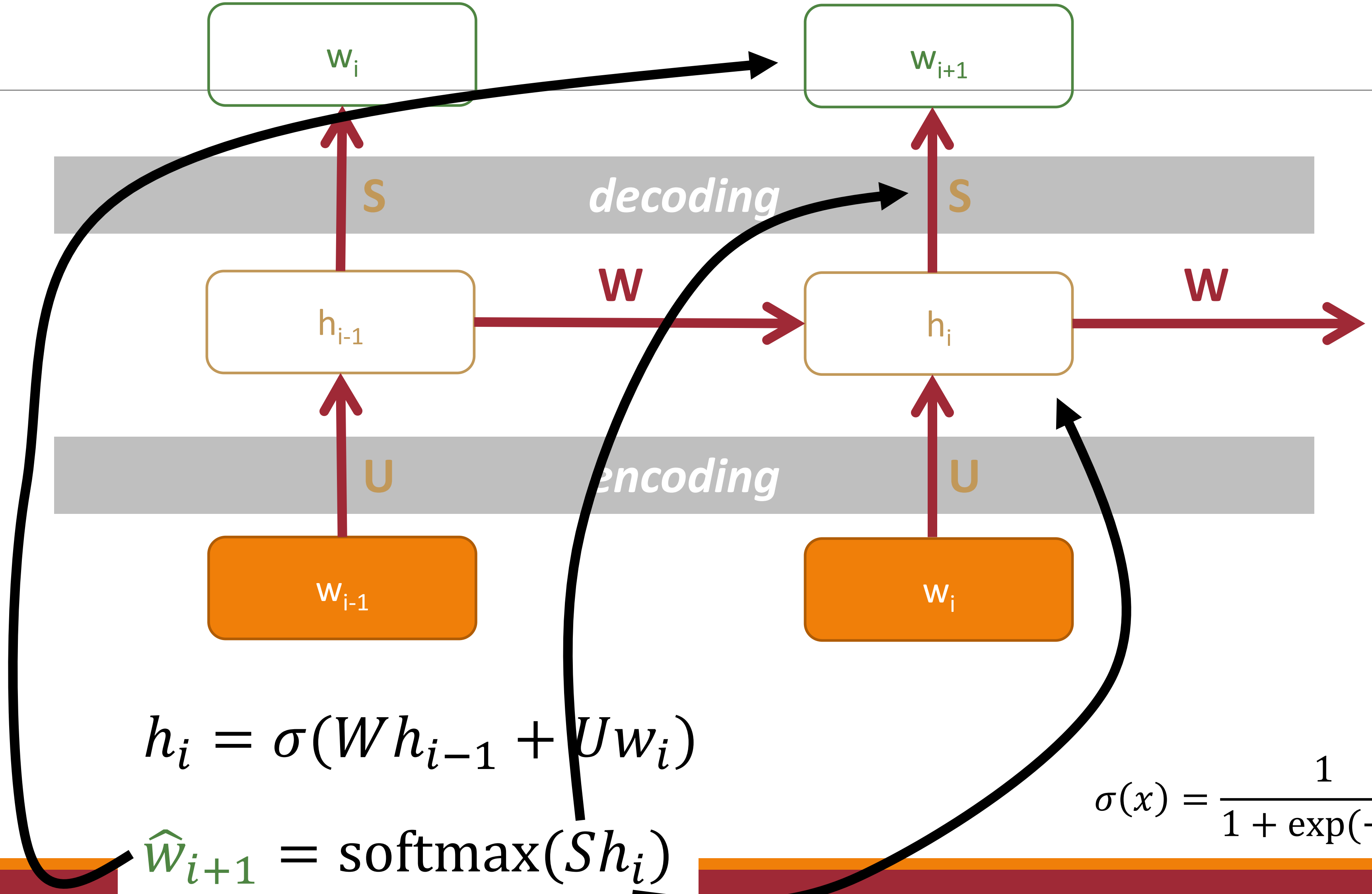
A *Simple* Recurrent Neural Network Cell



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

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

A *Simple* Recurrent Neural Network Cell

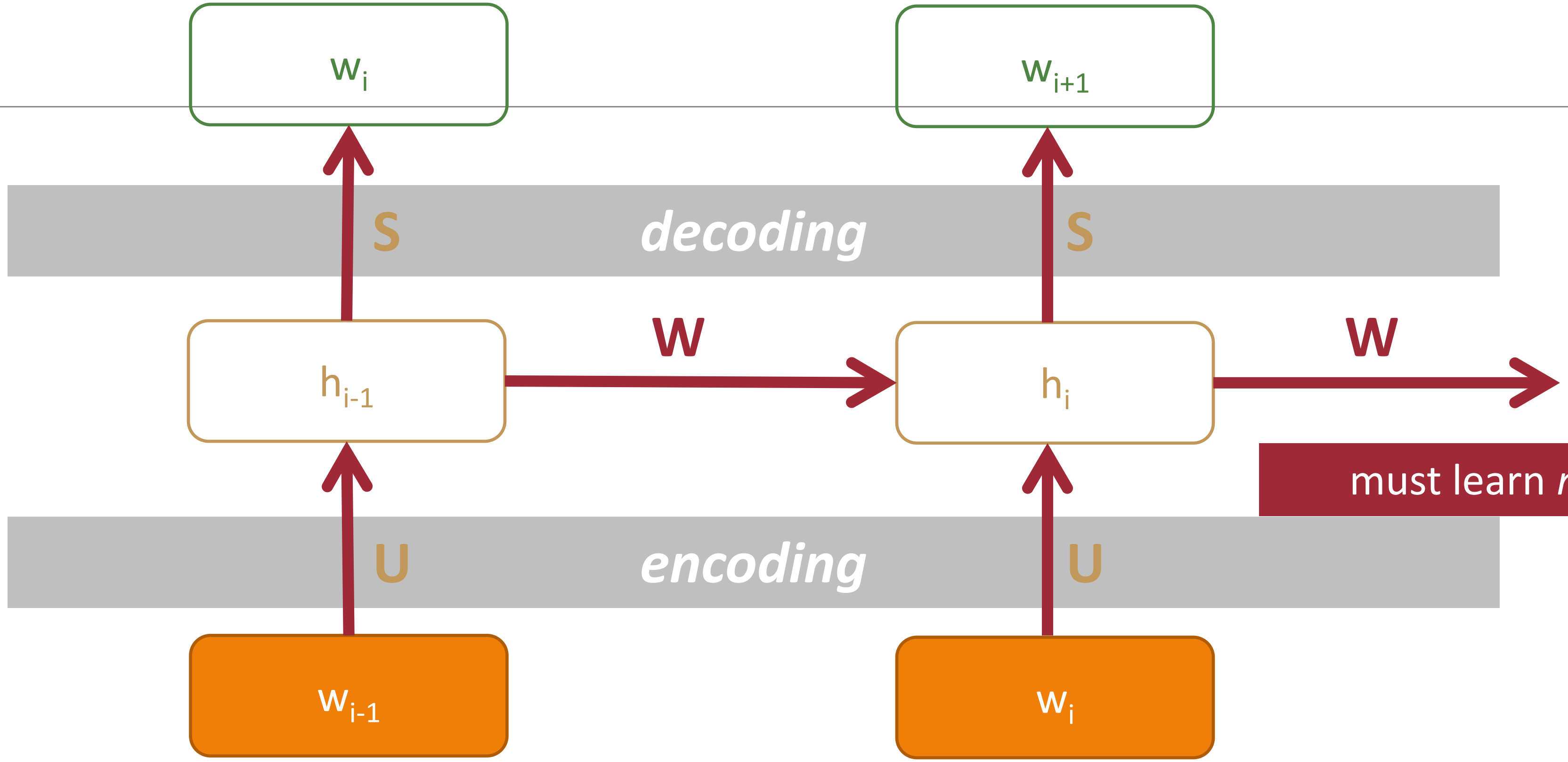


$$h_i = \sigma(W h_{i-1} + U w_i)$$

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

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

A *Simple* Recurrent Neural Network Cell

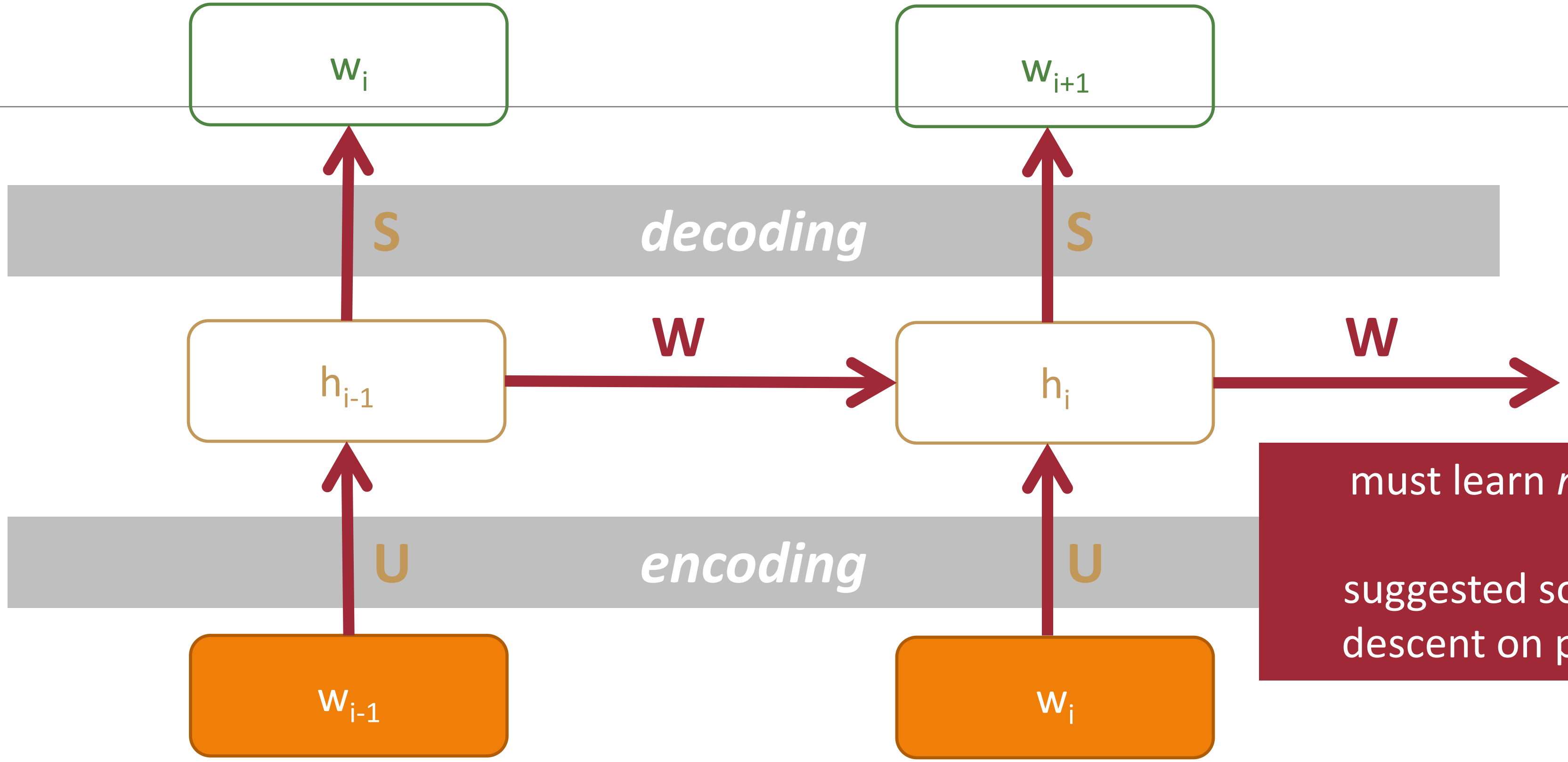


must learn *matrices* U, S, W

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

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

A *Simple* Recurrent Neural Network Cell



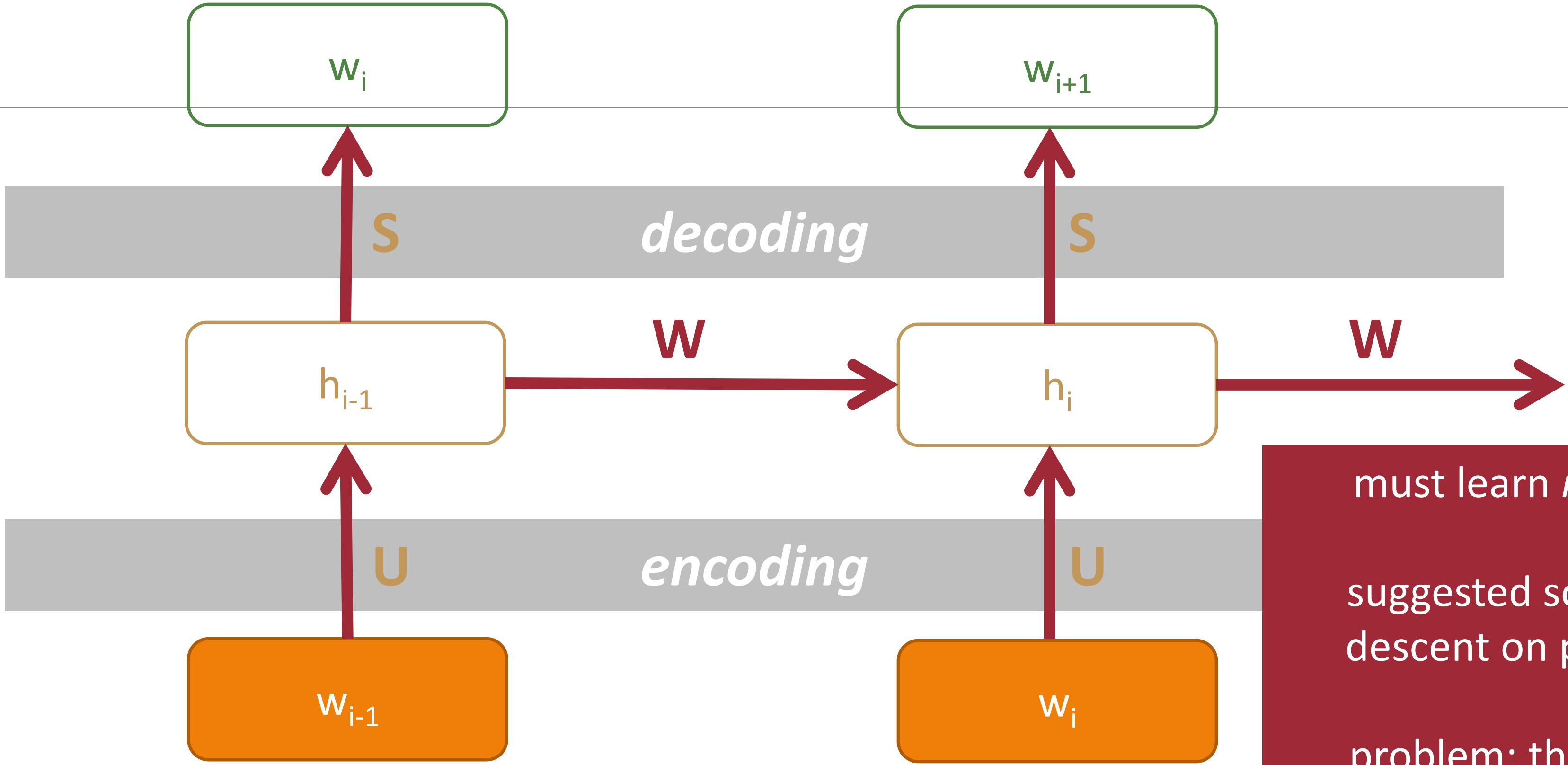
must learn *matrices* U, S, W

suggested solution: gradient descent on prediction ability

$$h_i = \sigma(Wh_{i-1} + Uw_i)$$

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

A *Simple* Recurrent Neural Network Cell



must learn *matrices* U, S, W

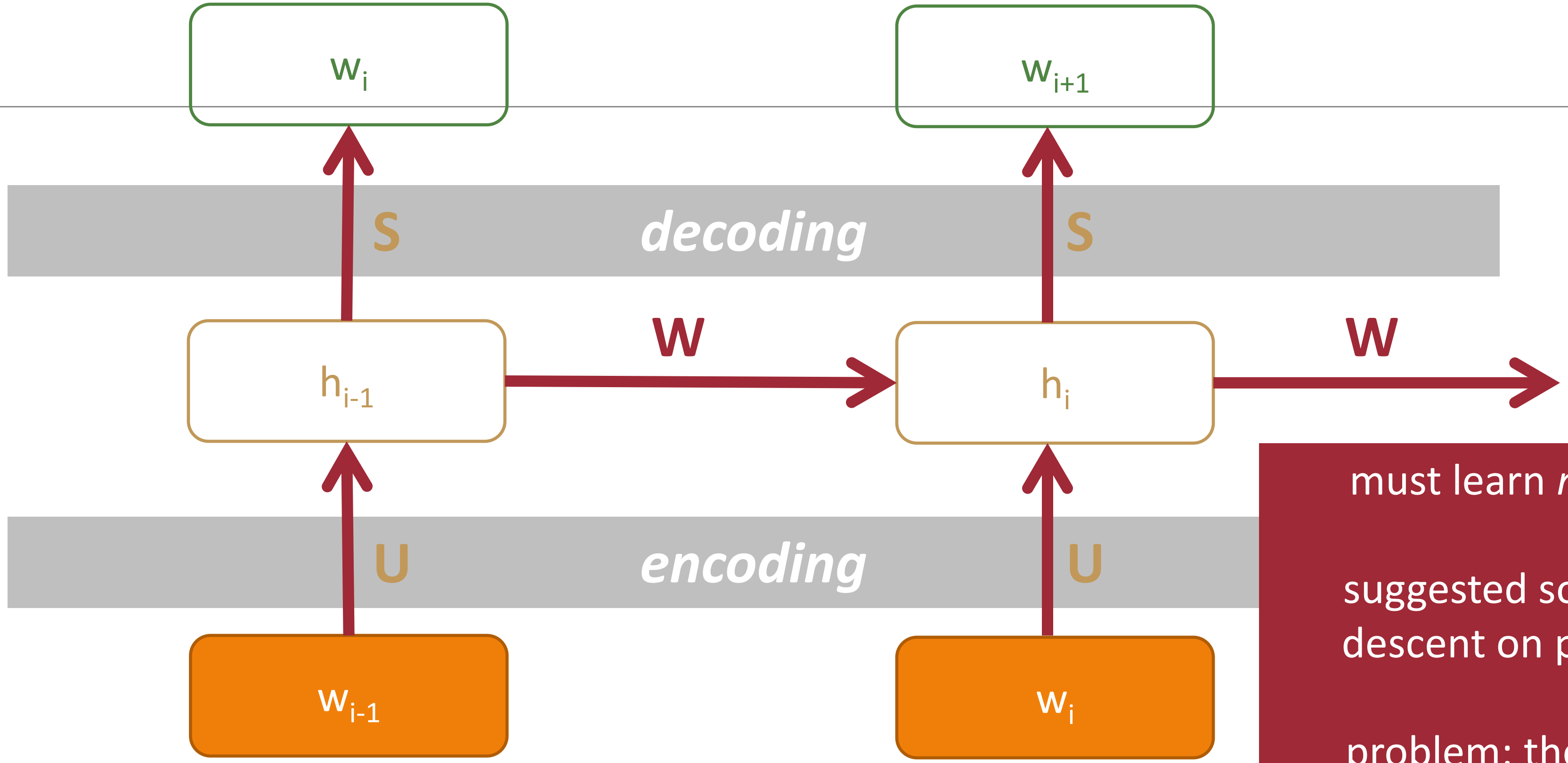
suggested solution: gradient descent on prediction ability

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

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

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

A *Simple* Recurrent Neural Network Cell



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

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

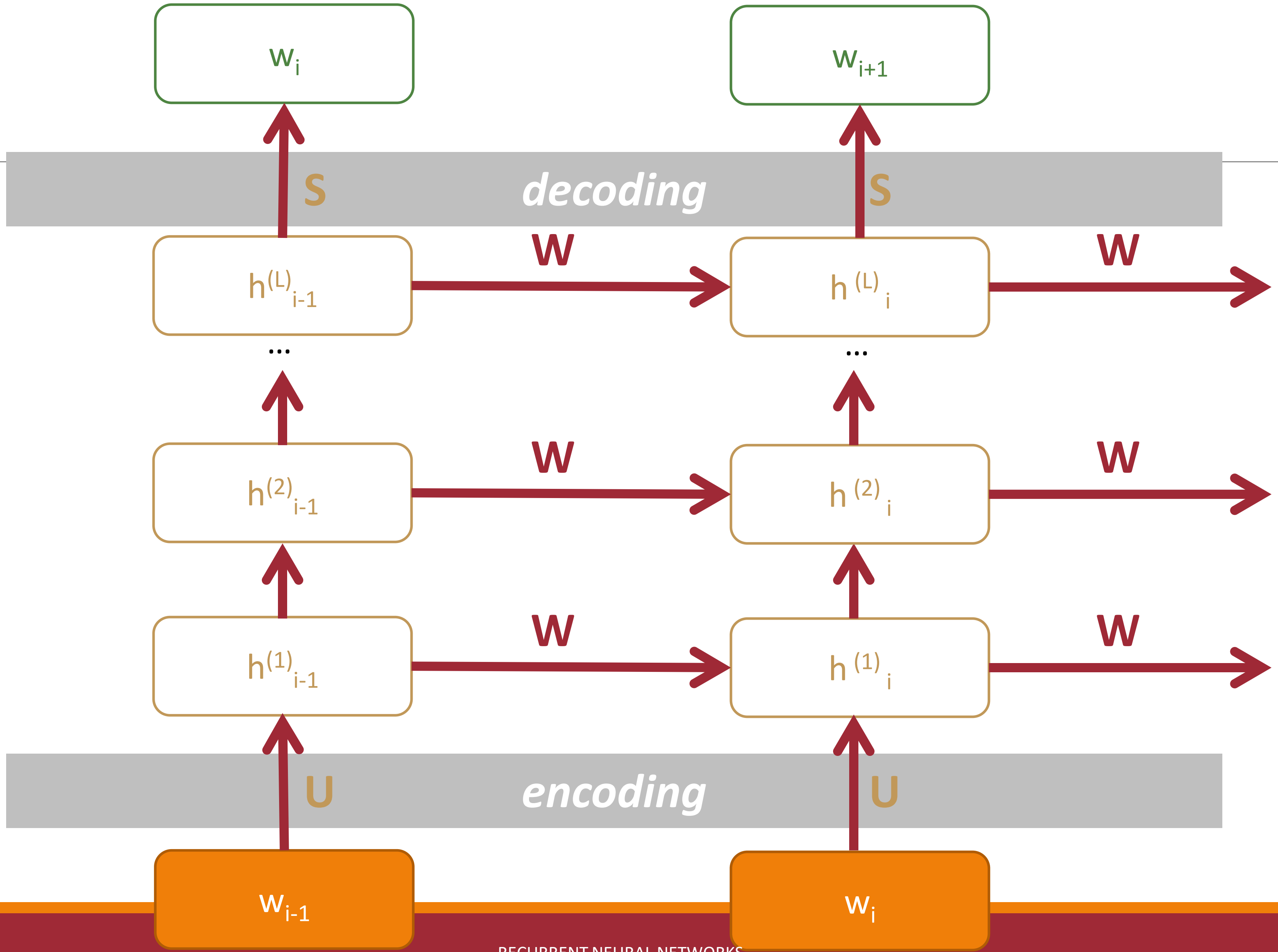
must learn *matrices* U, S, W

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



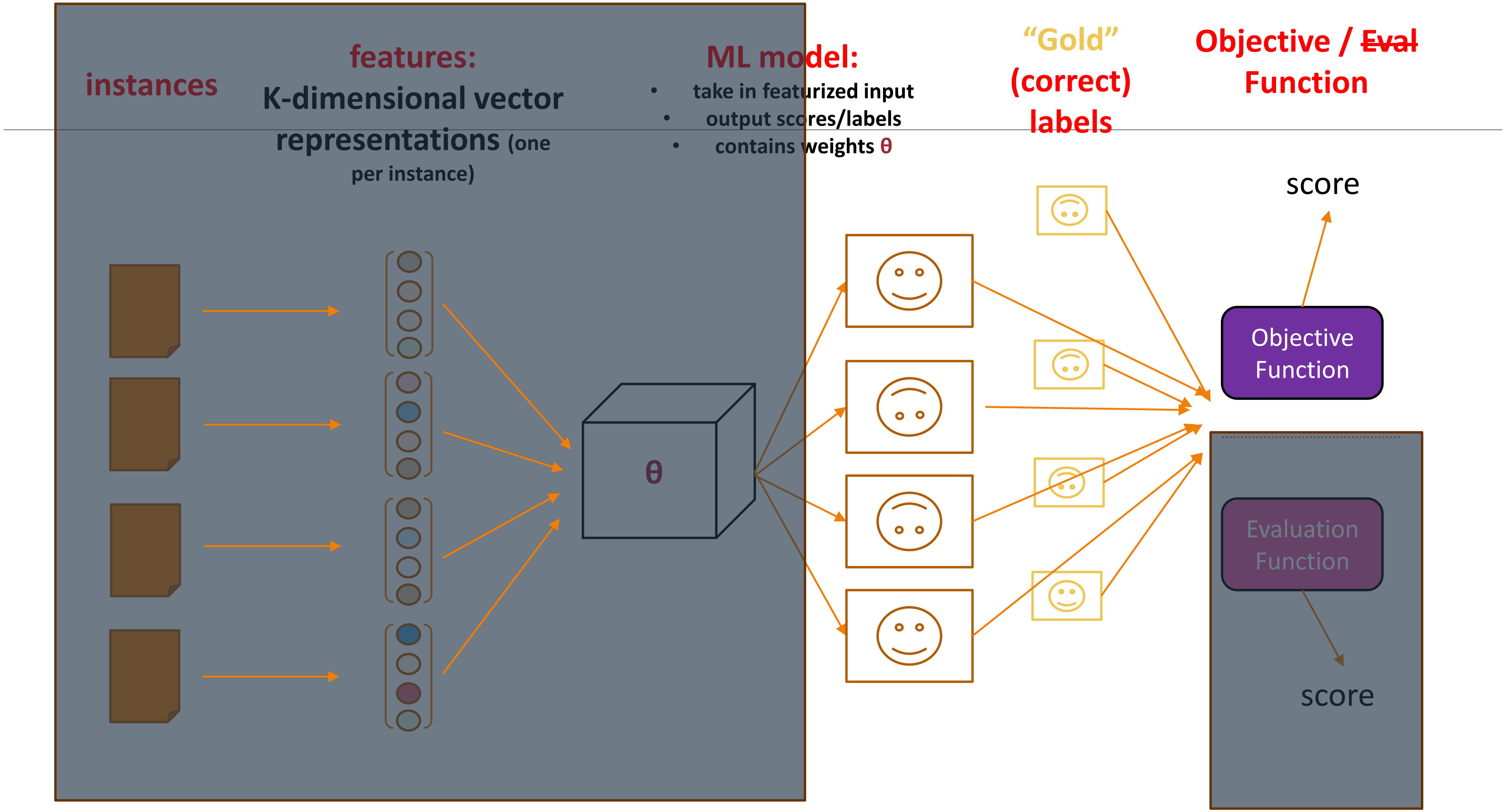
How do you learn an RNN?

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

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

Classifier output

True probability (i.e., correct output)

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

index of "1" indicates correct value

$$\begin{pmatrix} 0 \\ 0 \\ \dots \\ 1 \\ \dots \\ 0 \end{pmatrix}$$

one-hot vector

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
     $l = model(x_i)$
  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).

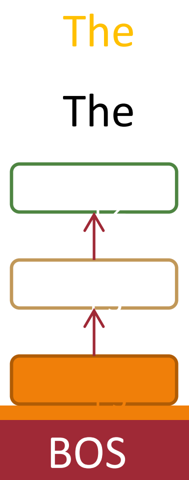
This **loss** is the sum of the per-token cross-entropy loss

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!



Recurrent NN Loss

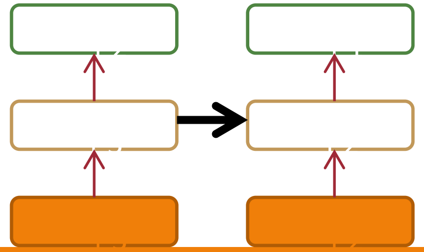
$\log .2 + \log .12$

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

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

The wet

The fluffy



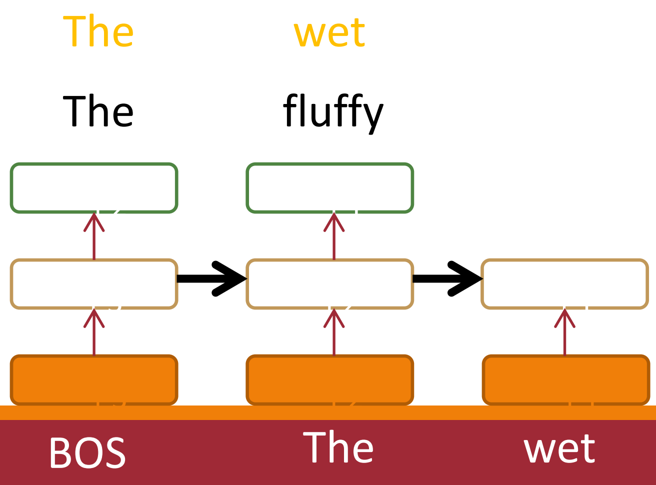
BOS The

Recurrent NN Loss

log .2 + log .12

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

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



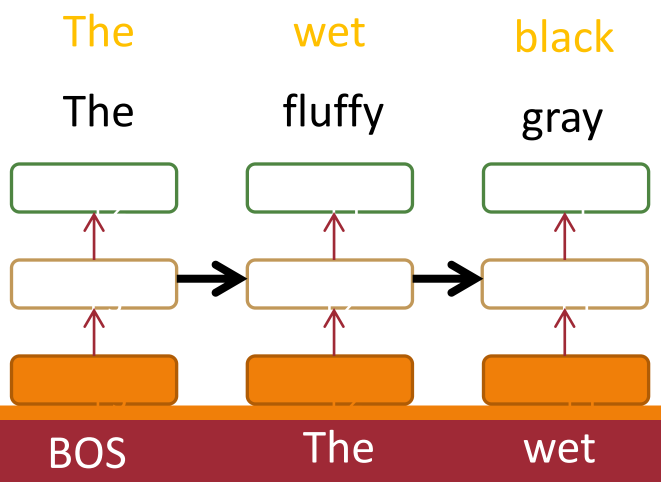
Recurrent NN Loss

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

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

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

word	prob.
black	.2
gray	.01
blue	.001
bald	.0005
wet	.0005
…	…



Recurrent NN Loss

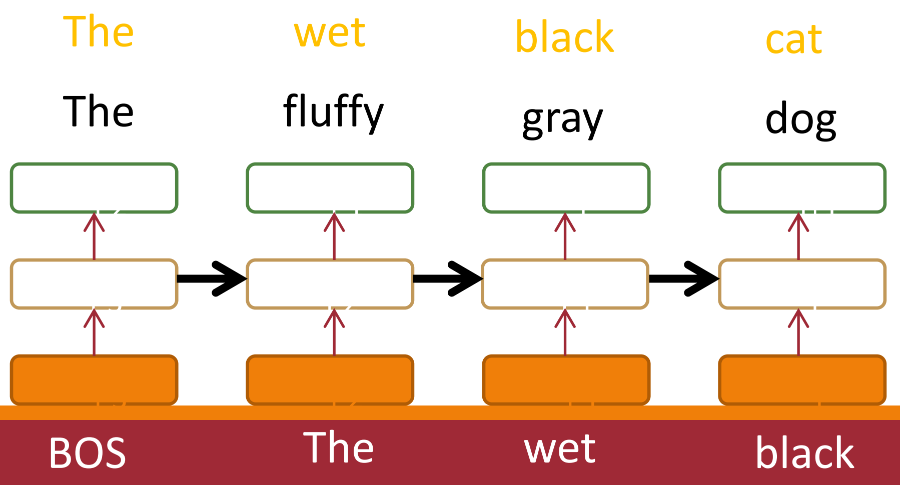
$\log .2 + \log .12 + \log .2 + \log .19$

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

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

word	prob.
black	.2
gray	.01
blue	.001
bald	.0005
wet	.0005
…	…

word	prob.
dog	.2
cat	.19
blue	.001
fluffy	.0005
wet	.0005
…	…



Recurrent NN Loss

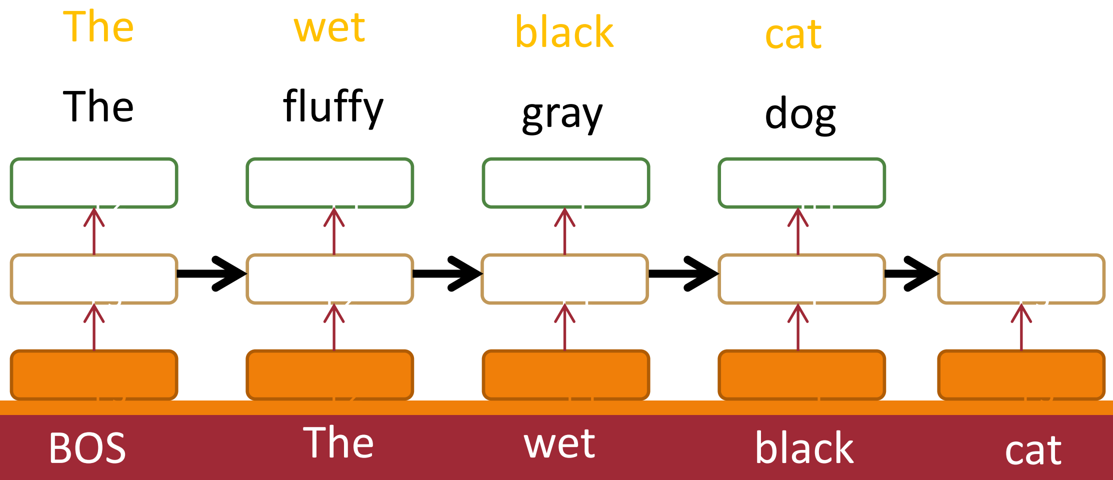
$\log .2 + \log .12 + \log .2 + \log .19$

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

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

word	prob.
black	.2
gray	.01
blue	.001
bald	.0005
wet	.0005
…	…

word	prob.
dog	.2
cat	.19
blue	.001
fluffy	.0005
wet	.0005
…	…



Recurrent NN Loss

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

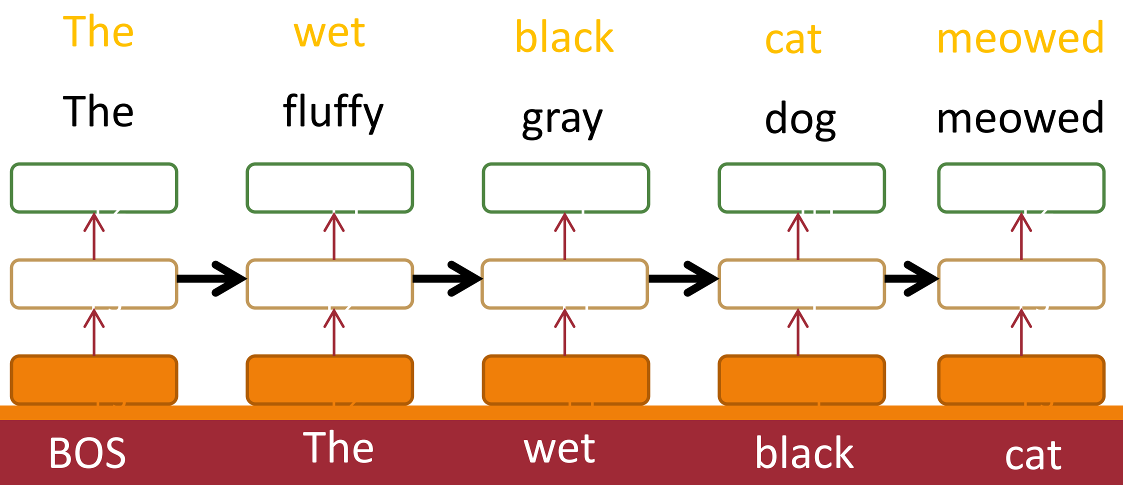
word	prob.
The	.2
gray	.01
blue	.001
fluffy	.0005
wet	.0005
…	…

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

word	prob.
black	.2
gray	.01
blue	.001
bald	.0005
wet	.0005
…	…

word	prob.
dog	.2
cat	.19
blue	.001
fluffy	.0005
wet	.0005
…	…

word	prob
meowed	.3
purred	.2
hissed	.1
fluffy	.001
wet	.001
…	…



Recurrent NN Loss

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

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

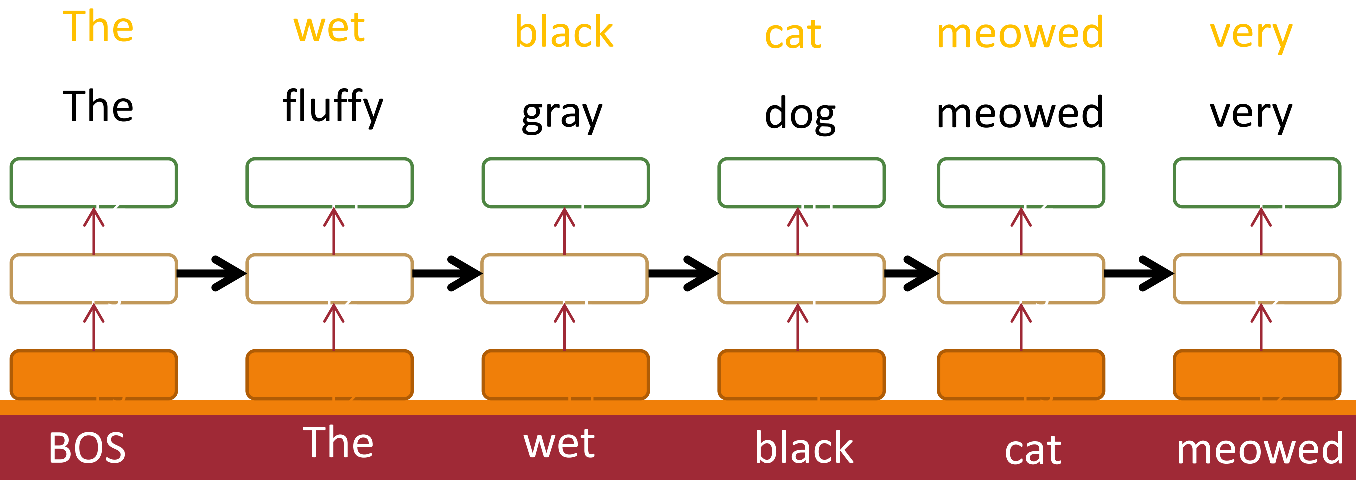
word	prob.
black	.2
wet	.12
blue	.001
fluffy	.0005
gray	.0005
…	…

word	prob.
black	.2
gray	.01
blue	.001
bald	.0005
wet	.0005
…	…

word	prob.
dog	.2
cat	.19
blue	.001
fluffy	.0005
wet	.0005
…	…

word	prob
meowed	.3
purred	.2
hissed	.1
fluffy	.001
wet	.001
…	…

word	prob.
very	.2
lots	.1
softly	. 1
fluffy	.0005
wet	.0005
…	…

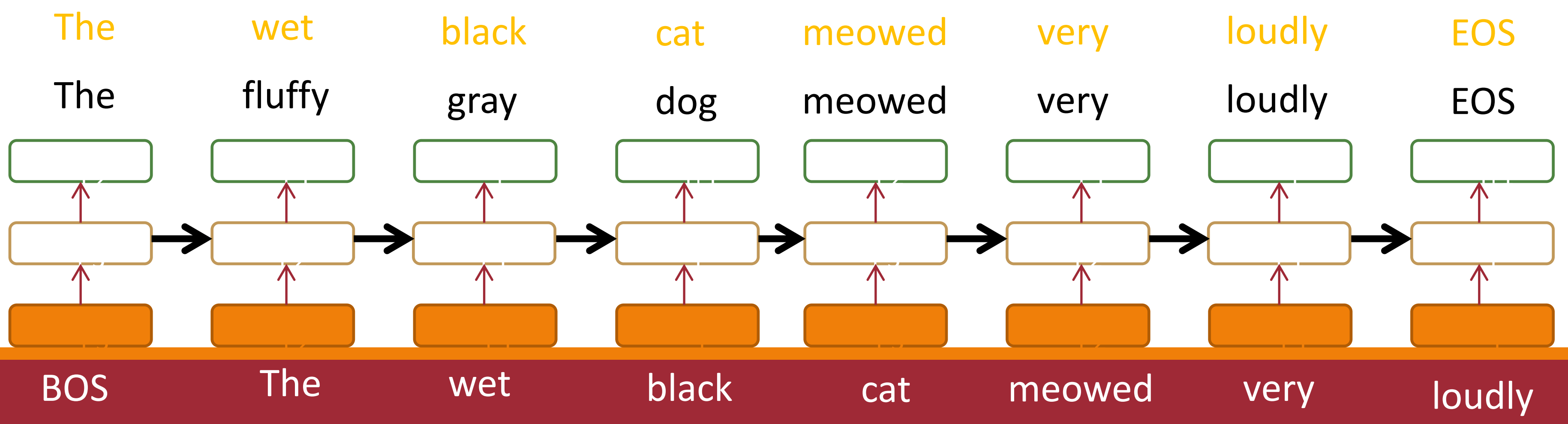


Recurrent NN Loss

(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

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 θ_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 ρ_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 θ_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
   $g_t \mathrel{+}= l'(x_i)$

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

"Batch"

Set t = 0
Pick a starting value θ_t
Until converged:

set $g_t = 0$
for example(s) i in full data:
1. Compute loss l on x_i
2. Accumulate gradient
   $g_t \mathrel{+}= l'(x_i)$

done
Get scaling factor ρ_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