Recurrent Neural Networks

CMSC 473/673 - NATURAL LANGUAGE PROCESSING

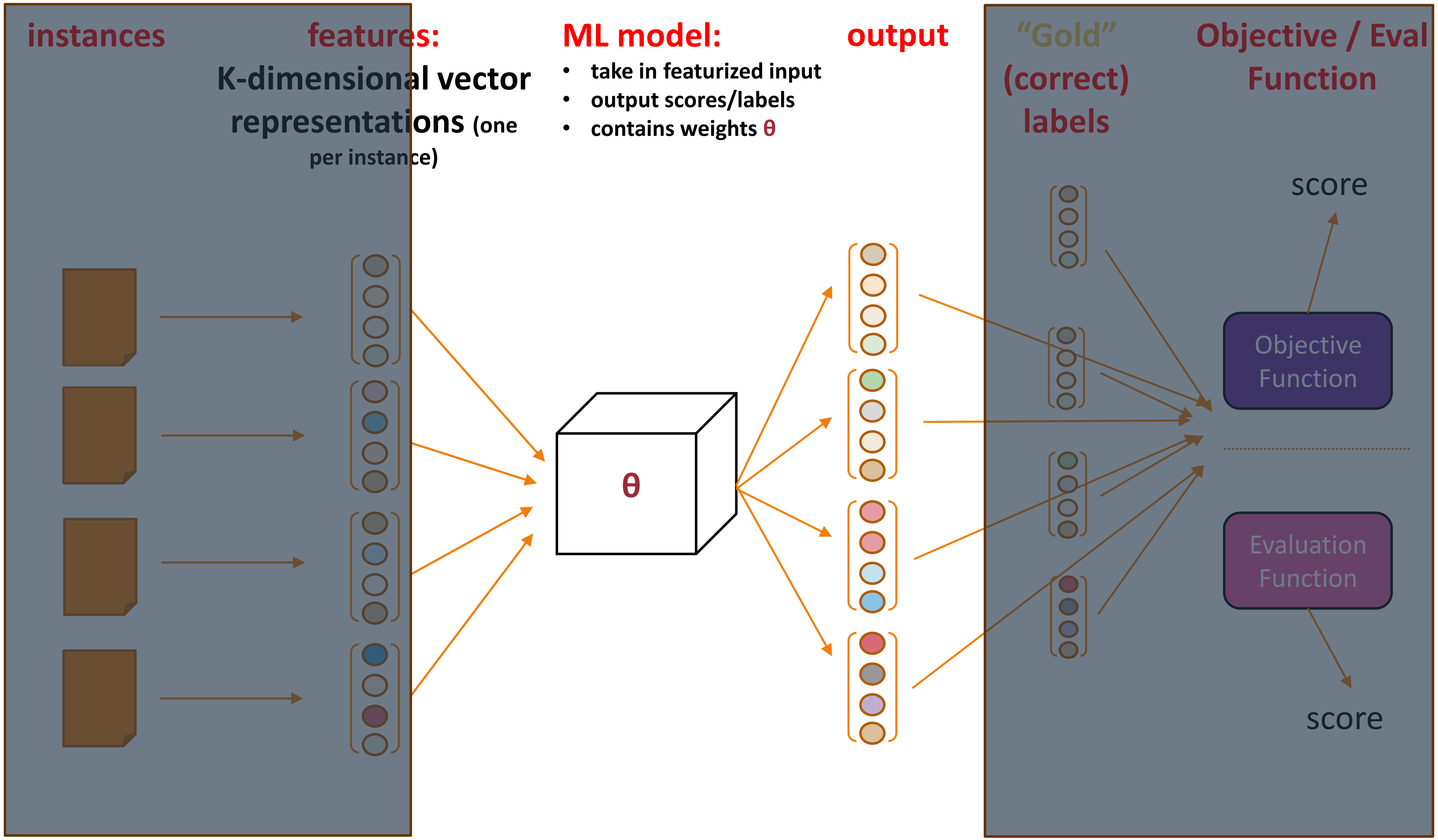
Slides modified from Dr. Frank Ferraro

Learning Objectives

Define the basic cell architecture of an RNN

Backpropagate loss through an example RNN

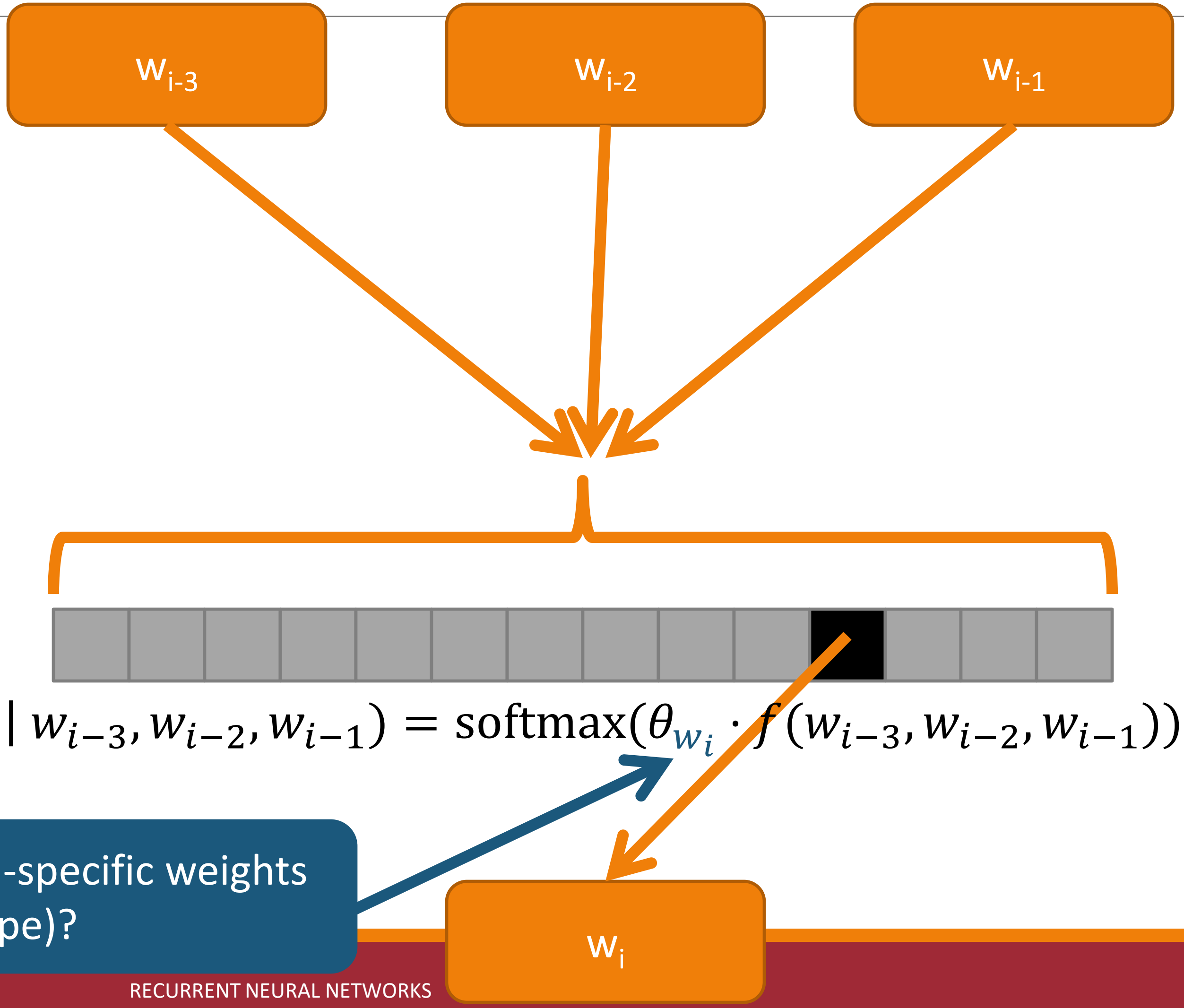
Create a simple RNN with PyTorch



instances
features:
K-dimensional vector representations (one per instance)
ML model:
• take in featurized input
• output scores/labels
• contains weights θ
θ
output
"Gold" (correct) labels
Objective / Eval Function
score
Objective Function
Evaluation Function
score

Review: 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

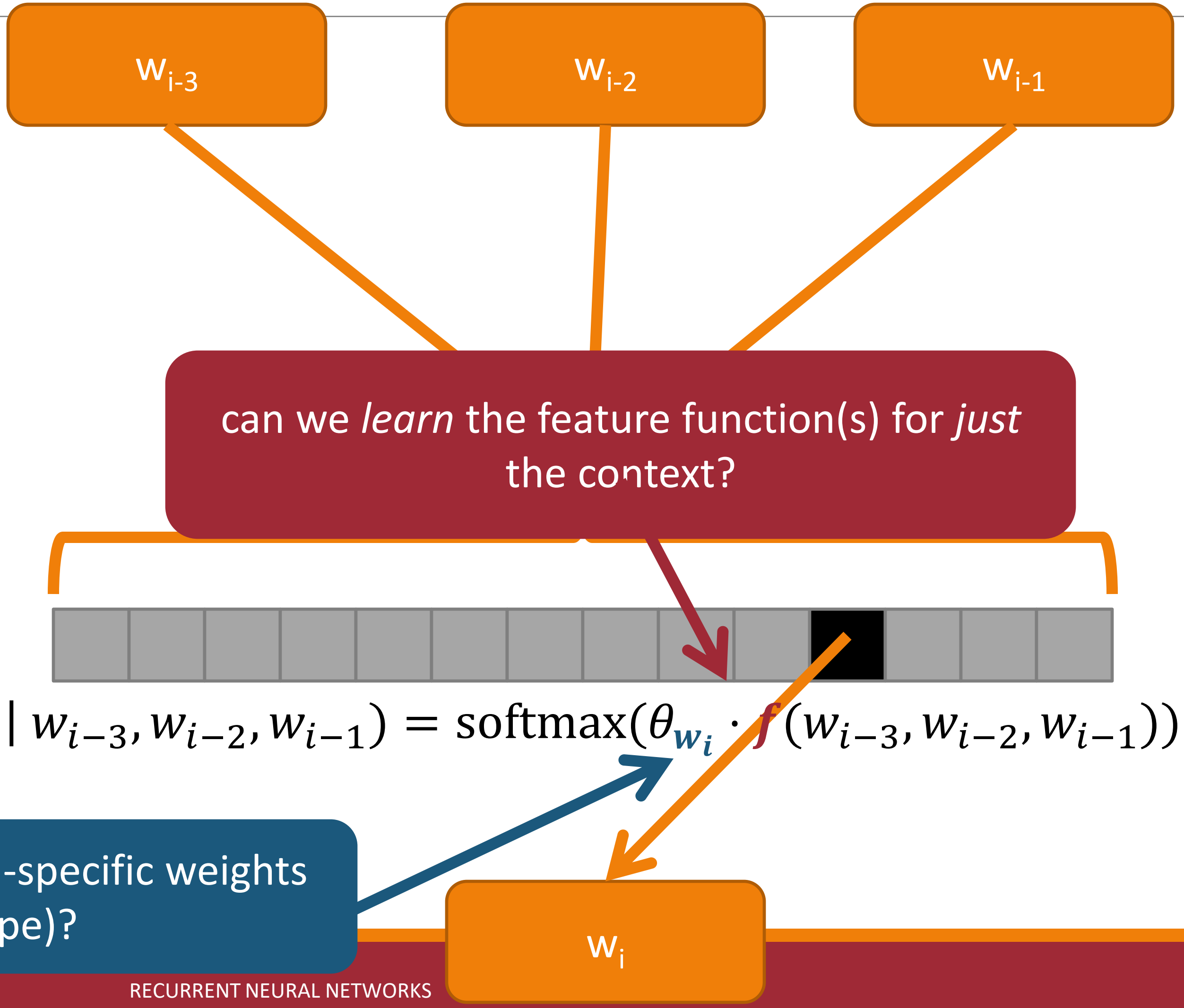
can we learn word-specific weights (by type)?

Review: Neural Language Models

given some context…

compute beliefs about what is likely…

predict the next word



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

Review: Neural Language Models

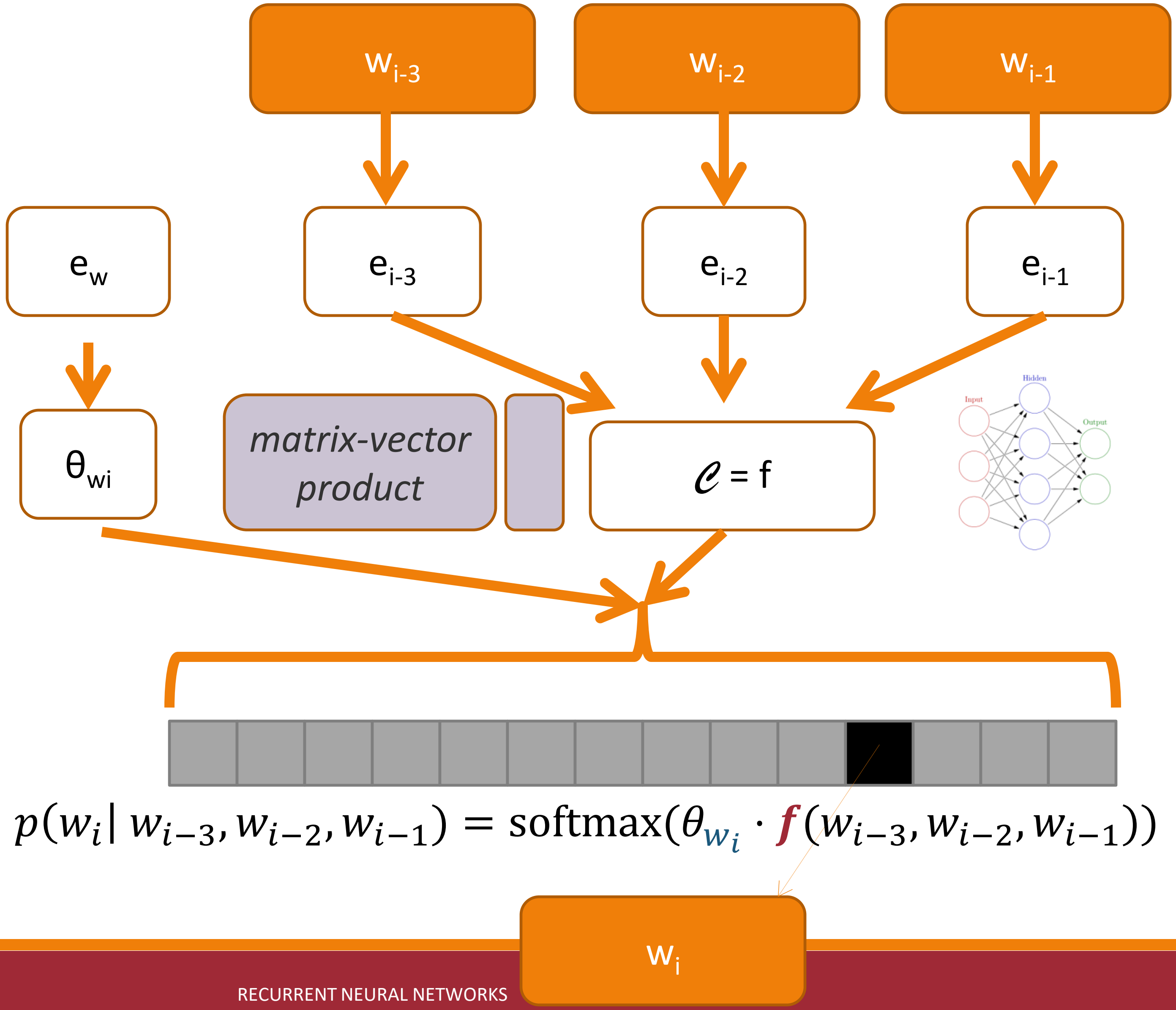
given some context…

create/use “distributed representations”…

combine these representations…

compute beliefs about what is likely…

predict the next word



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

Review: Neural Language Models

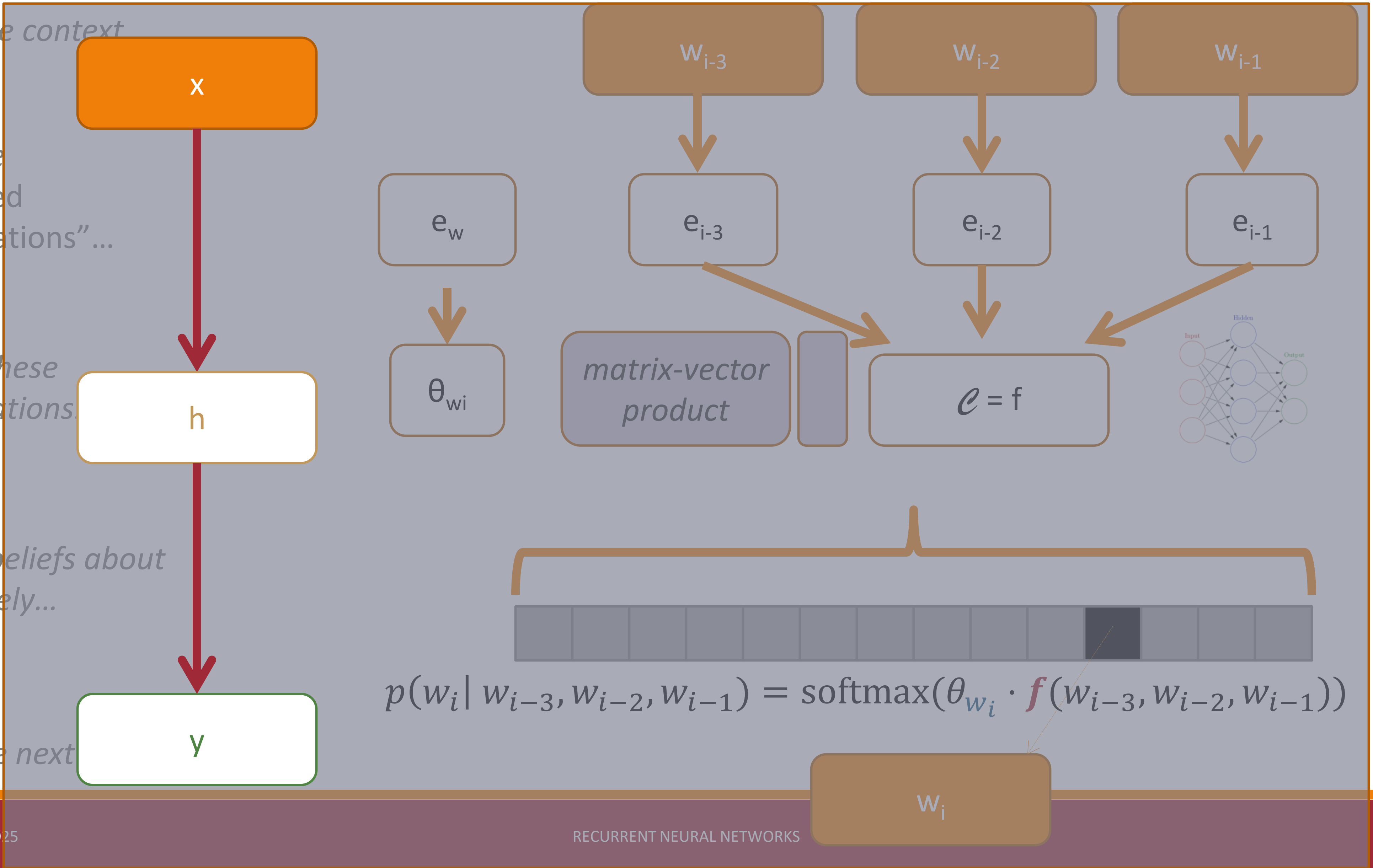
given some context…

create/use "distributed representations"…

combine these representations…

compute beliefs about what is likely…

predict the next

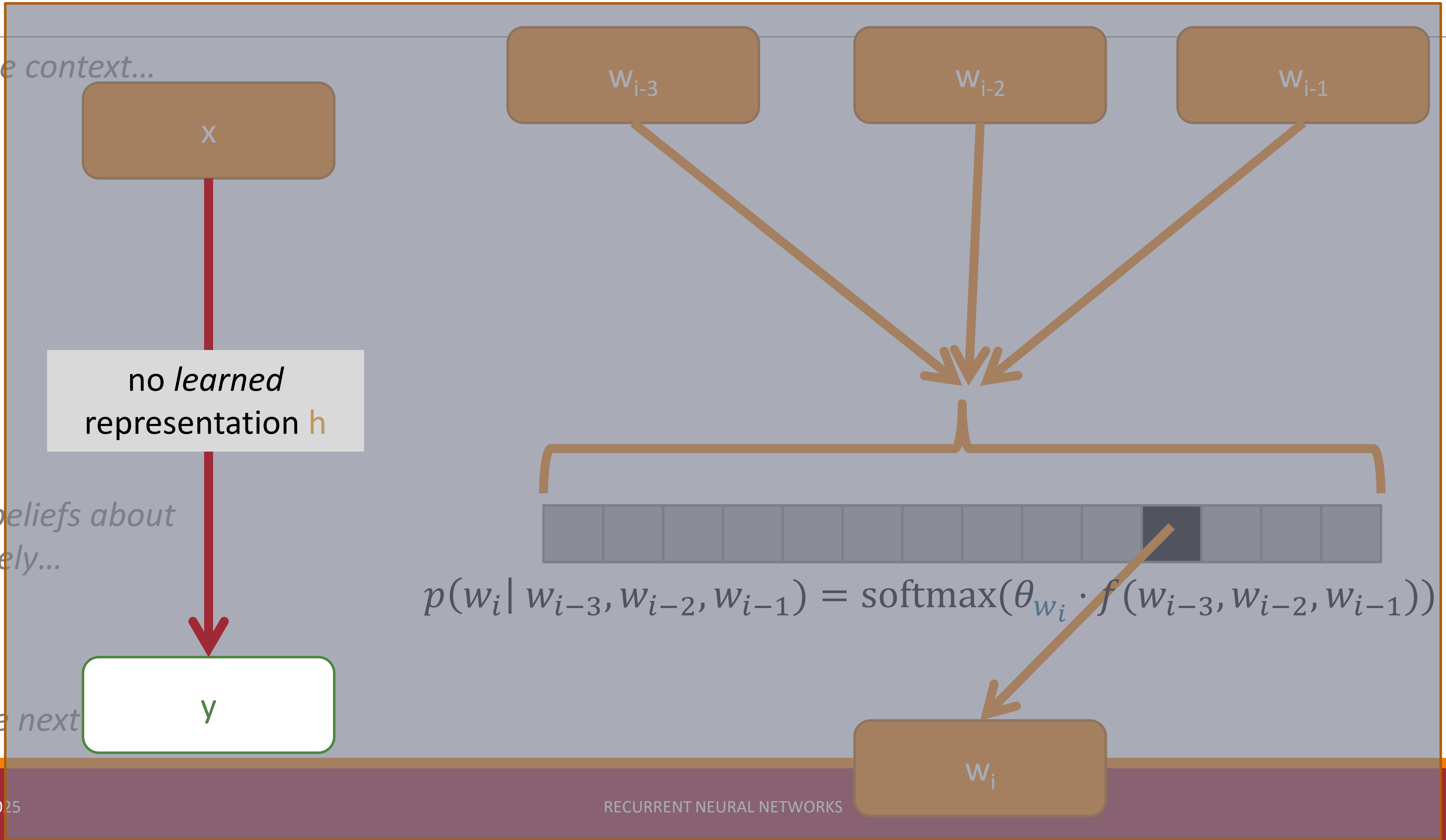


Review: Maxent Language Models

given some context…

compute beliefs about what is likely…

predict the next



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

Review: LM Comparison

COUNT-BASED

Class-specific

MAXENT

Class-based

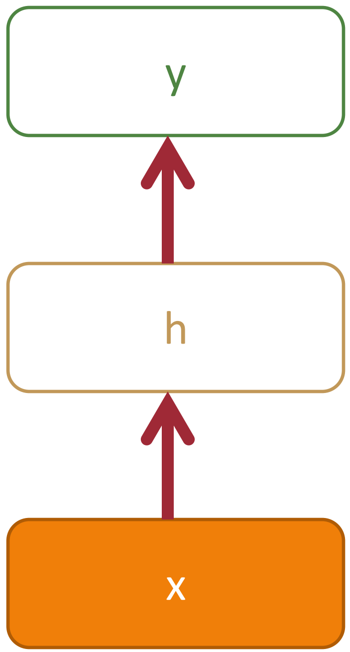
Uses features

NEURAL

Class-based

Uses *embedded* features

Review:
Network Types: Flat Input, Flat Output

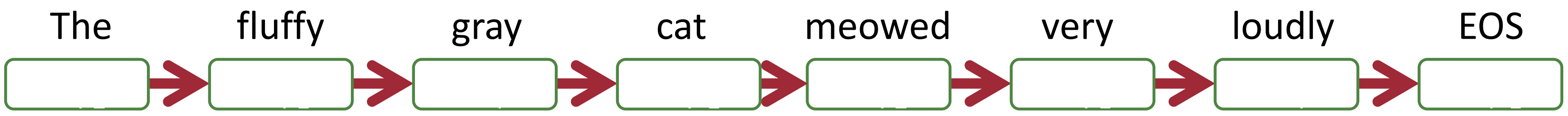


1. Feed forward

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

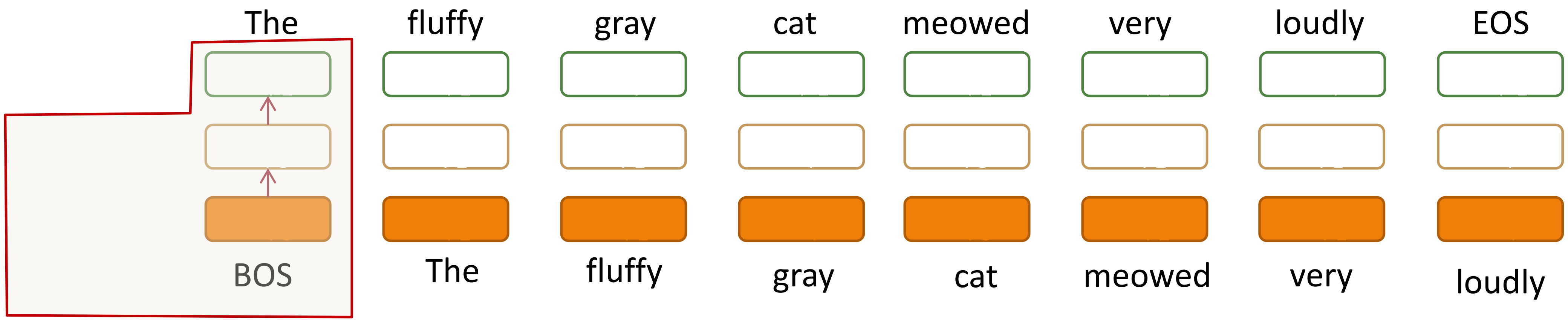
Review: A Neural N-Gram Model

The fluffy gray cat meowed very loudly



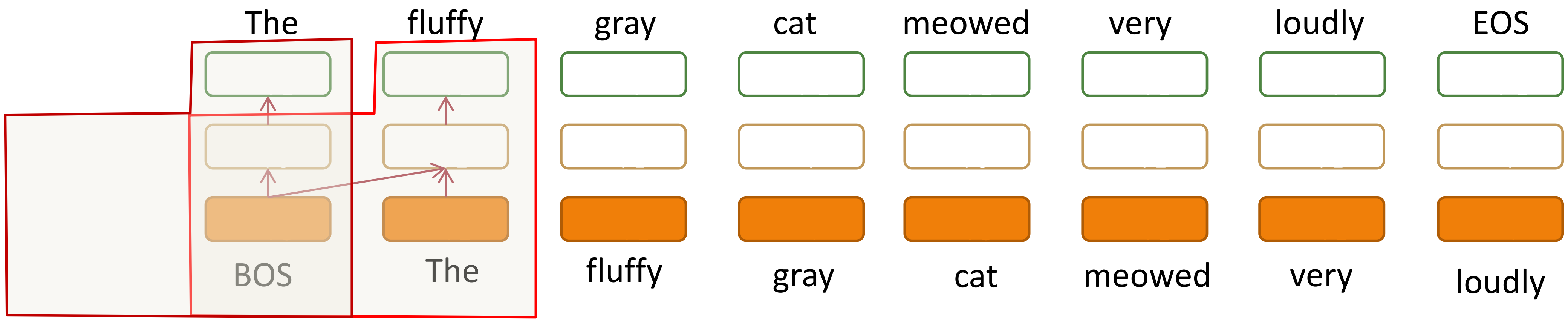
Review: A Neural N-Gram Model (N=3)

The fluffy gray cat meowed very loudly



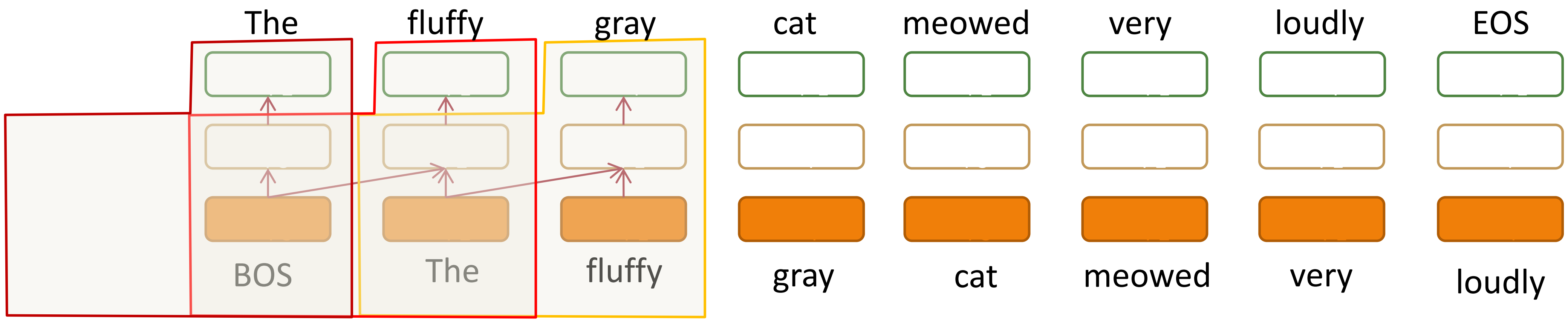
Review: A Neural N-Gram Model (N=3)

The fluffy gray cat meowed very loudly



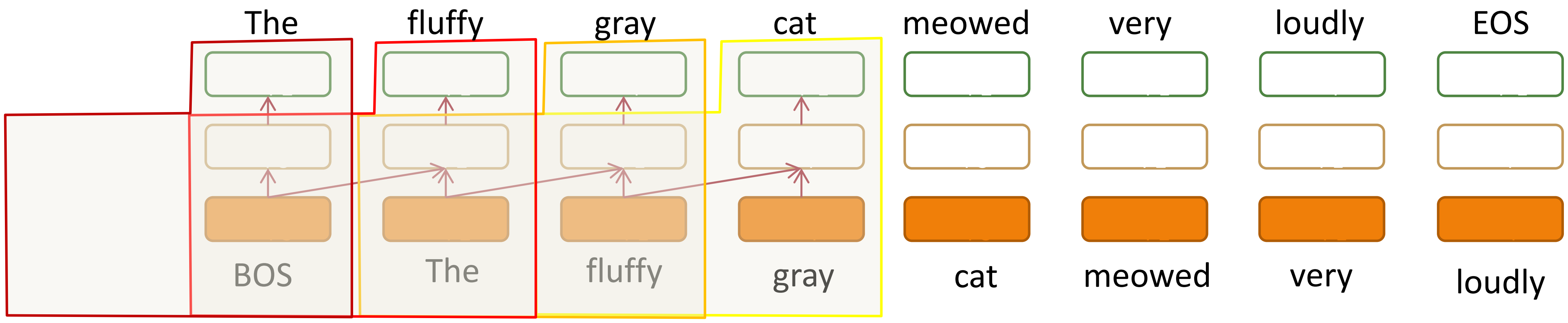
Review: A Neural N-Gram Model (N=3)

The fluffy gray cat meowed very loudly



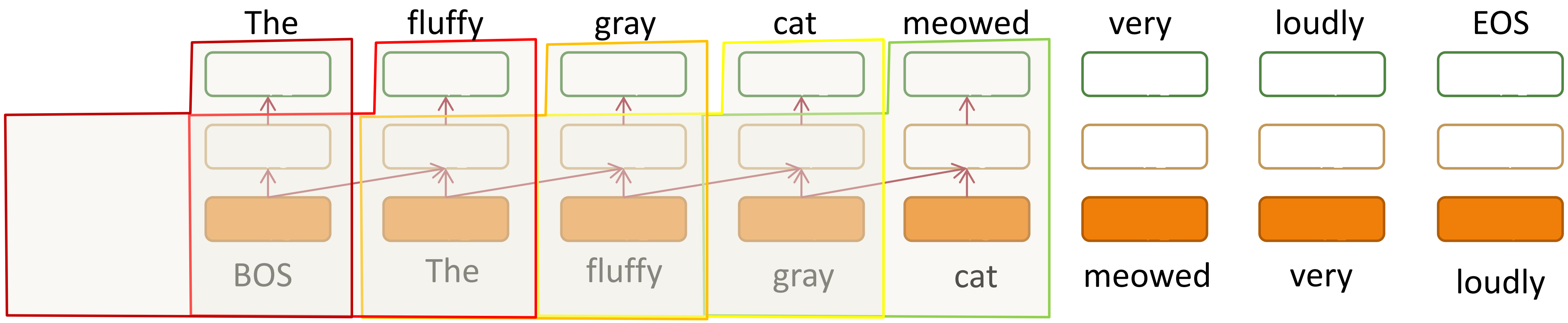
Review: A Neural N-Gram Model (N=3)

The fluffy gray cat meowed very loudly



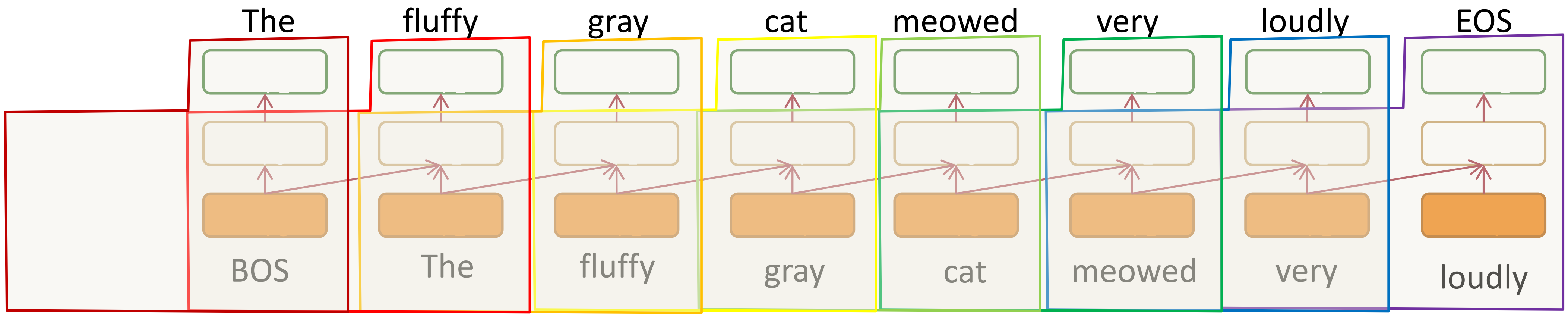
Review: A Neural N-Gram Model (N=3)

The fluffy gray cat meowed very loudly



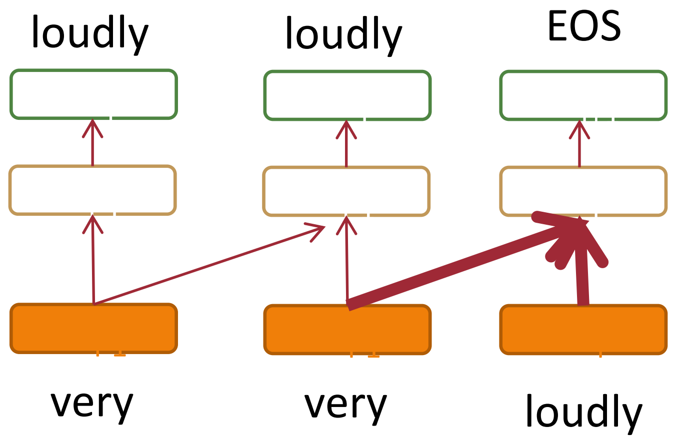
Review: A Neural N-Gram Model (N=3)

The fluffy gray cat meowed very loudly



A Neural N-Gram Model (N=3)

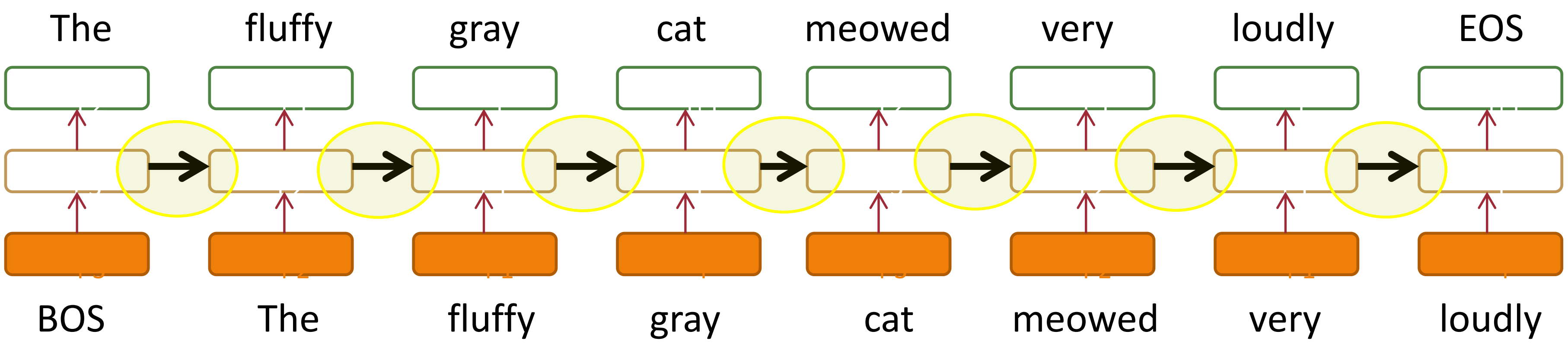
The fluffy gray cat meowed very loudly



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

A Recurrent Neural Language Model

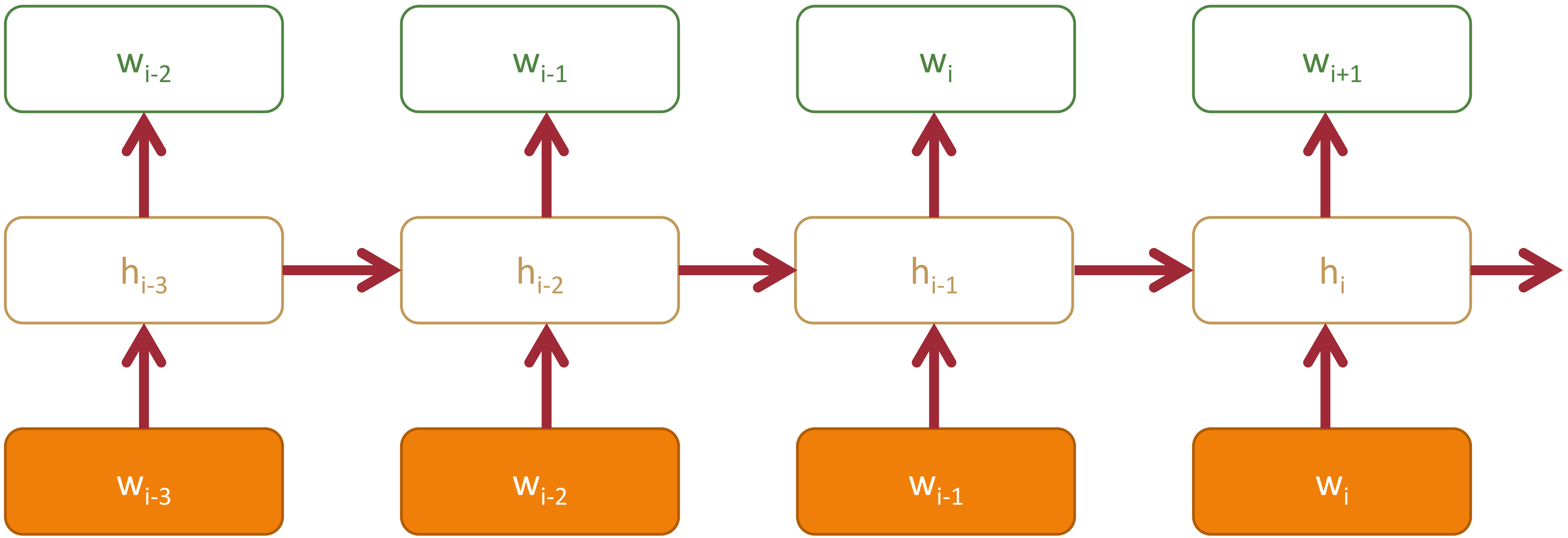
The fluffy gray cat meowed very loudly



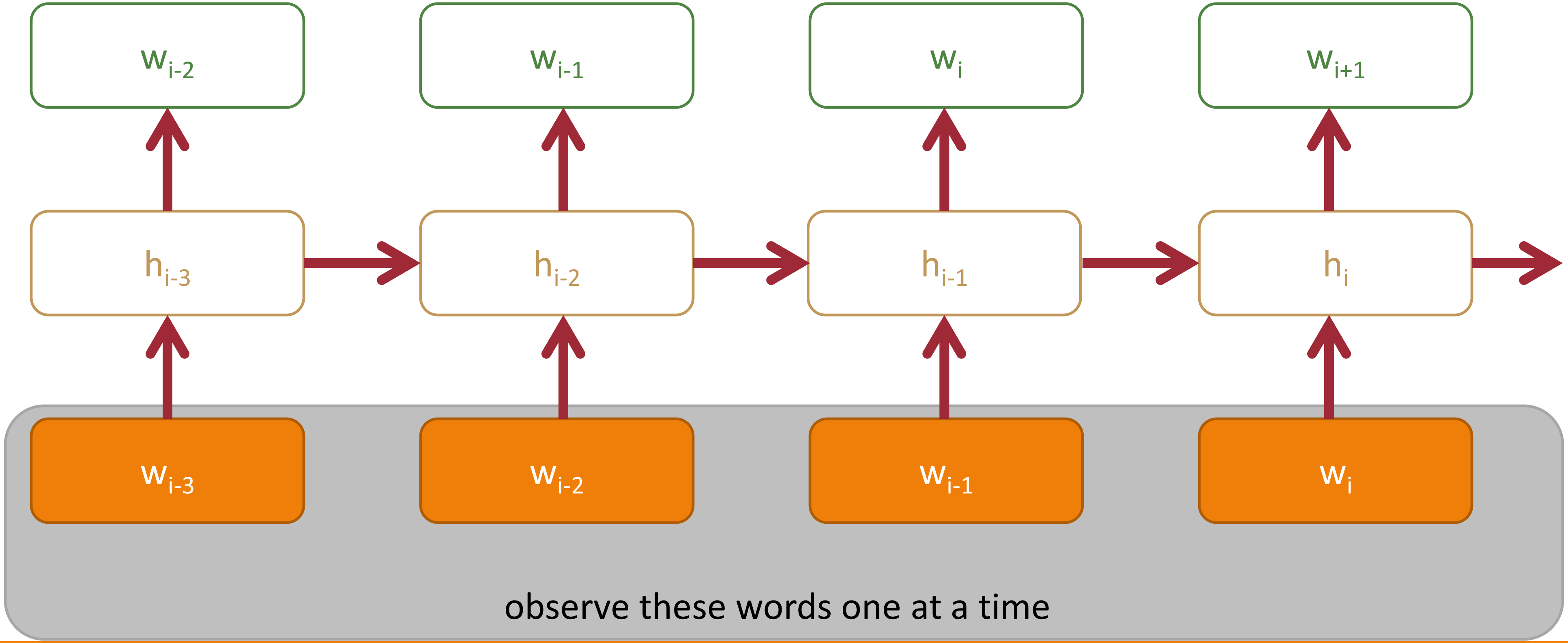
Critical issue: the amount of information flow is fundamentally restricted!!!

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

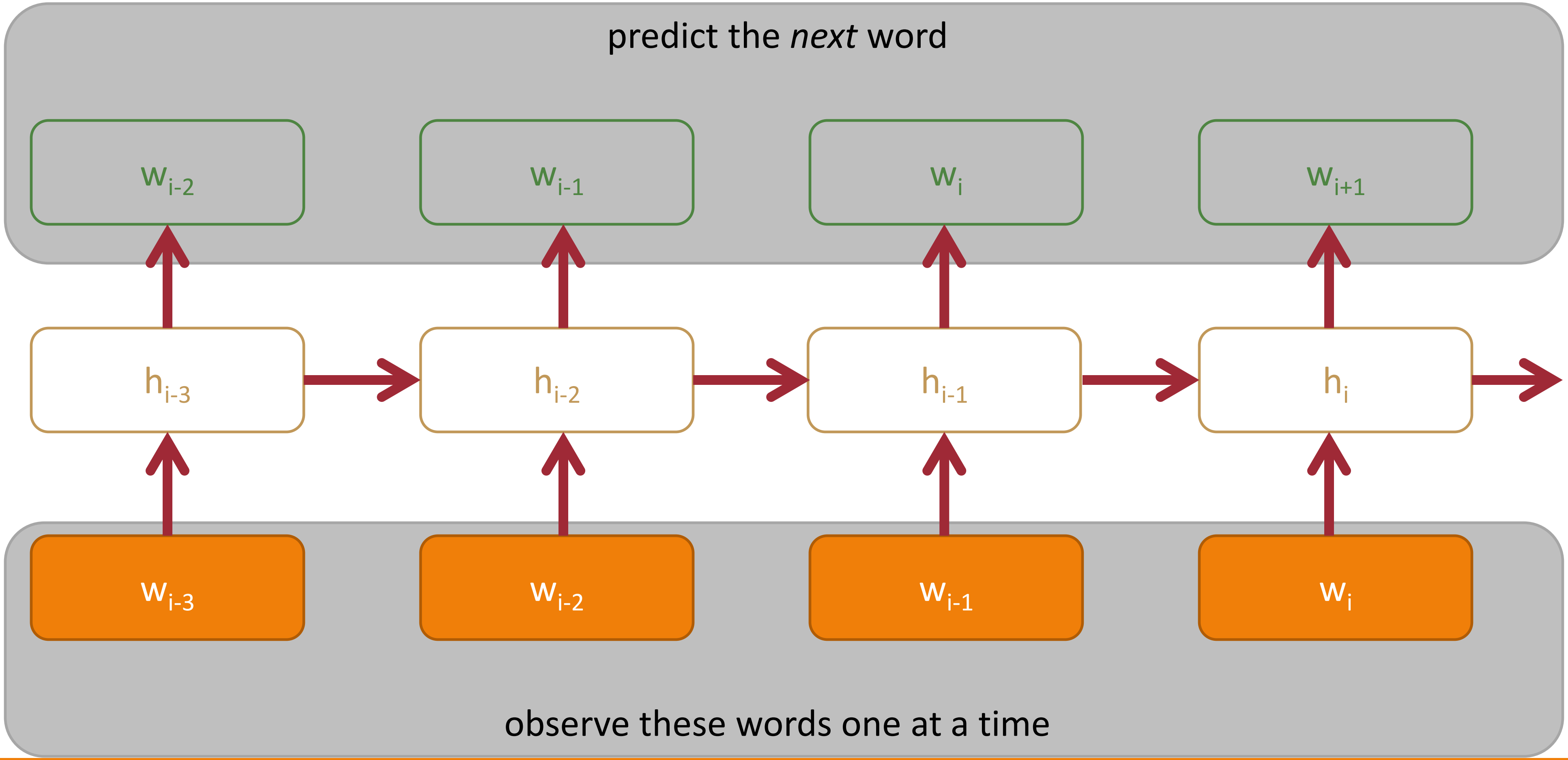
A Classic View of Recurrent Neural Language Modeling



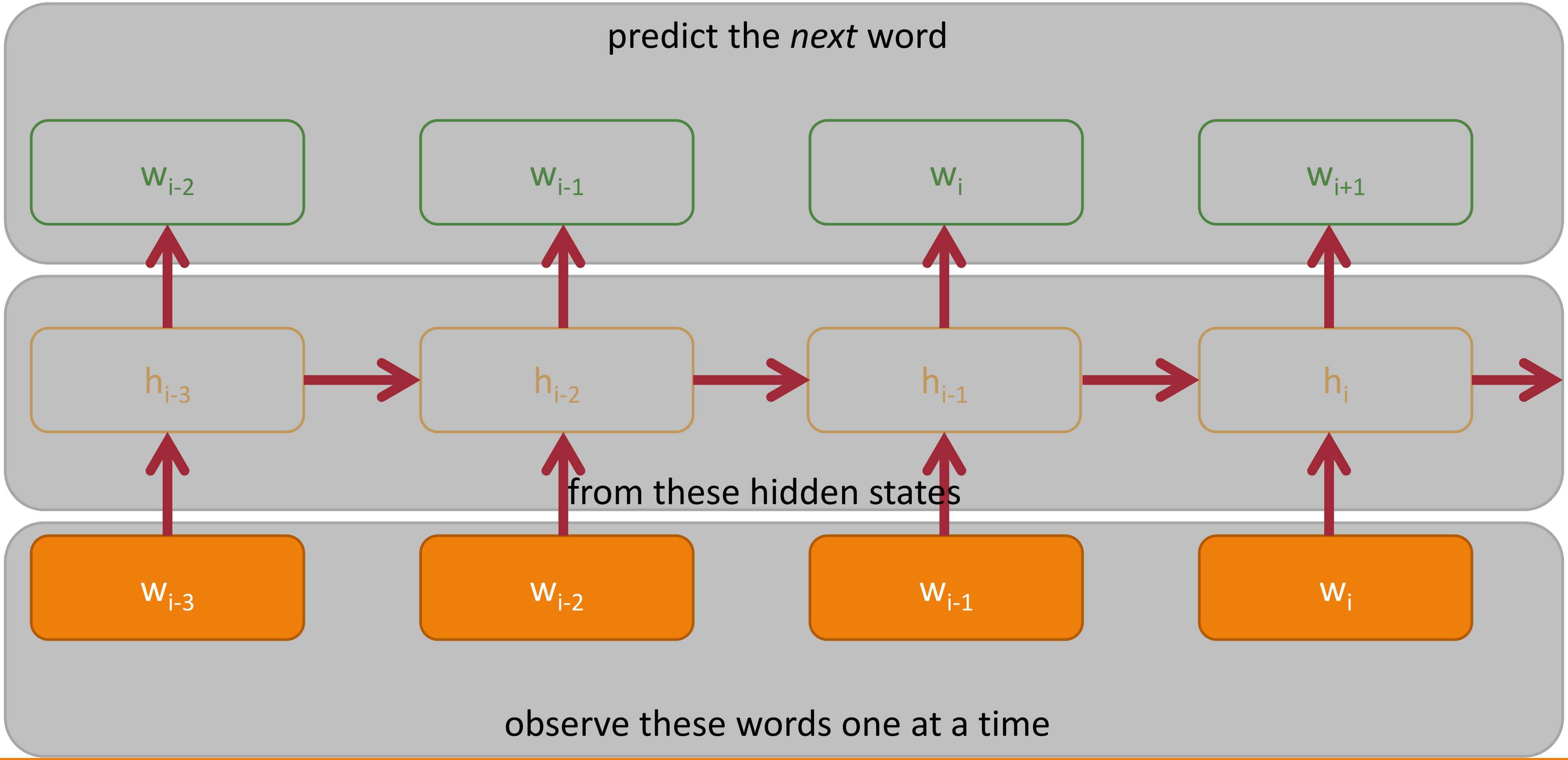
A Classic View of Recurrent Neural Language Modeling



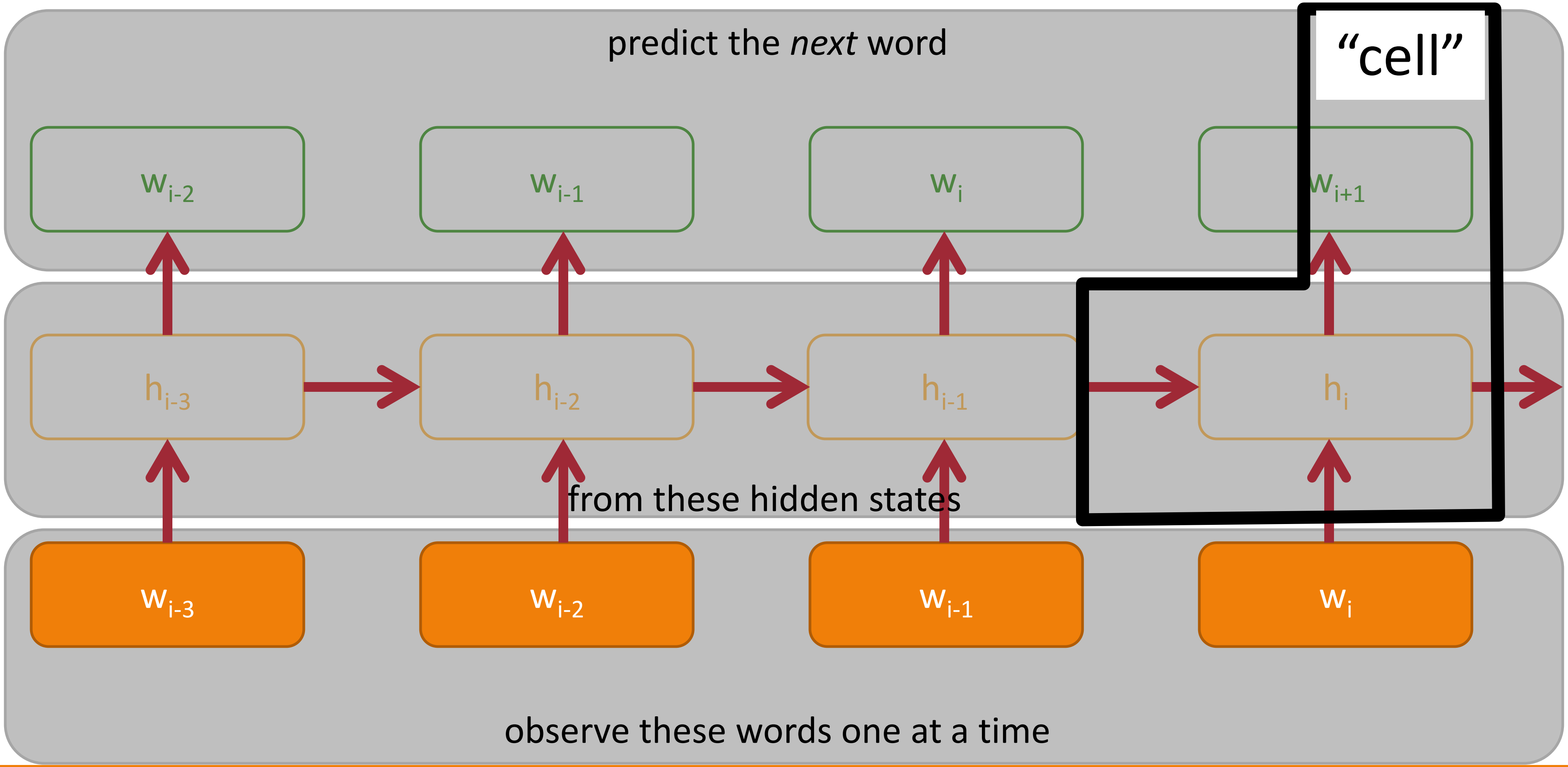
A Classic View of Recurrent Neural Language Modeling



A Classic View of Recurrent Neural Language Modeling



A Classic View of Recurrent Neural Language Modeling



Review: Forward Propagation Example

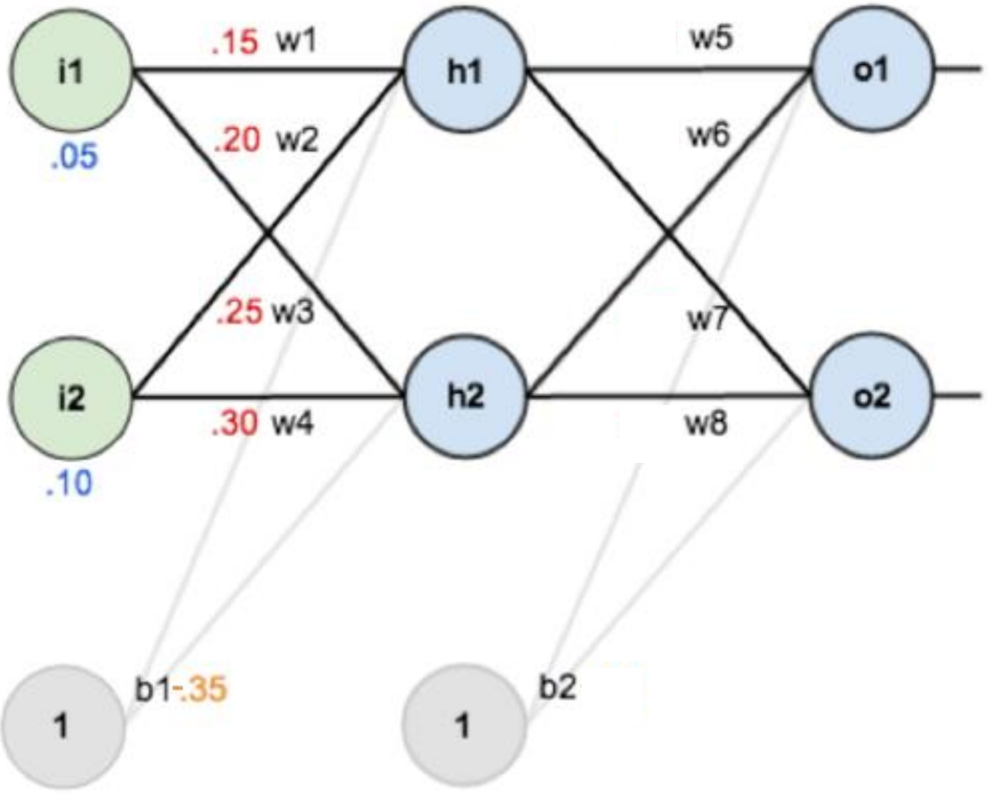
Calculate outputs to the hidden layer (units h1 and h2):

How do we do this?
Use our activation function!

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

What will be our *x*?

$in_{h1} = -.3225$
$in_{h2} = -.3075$

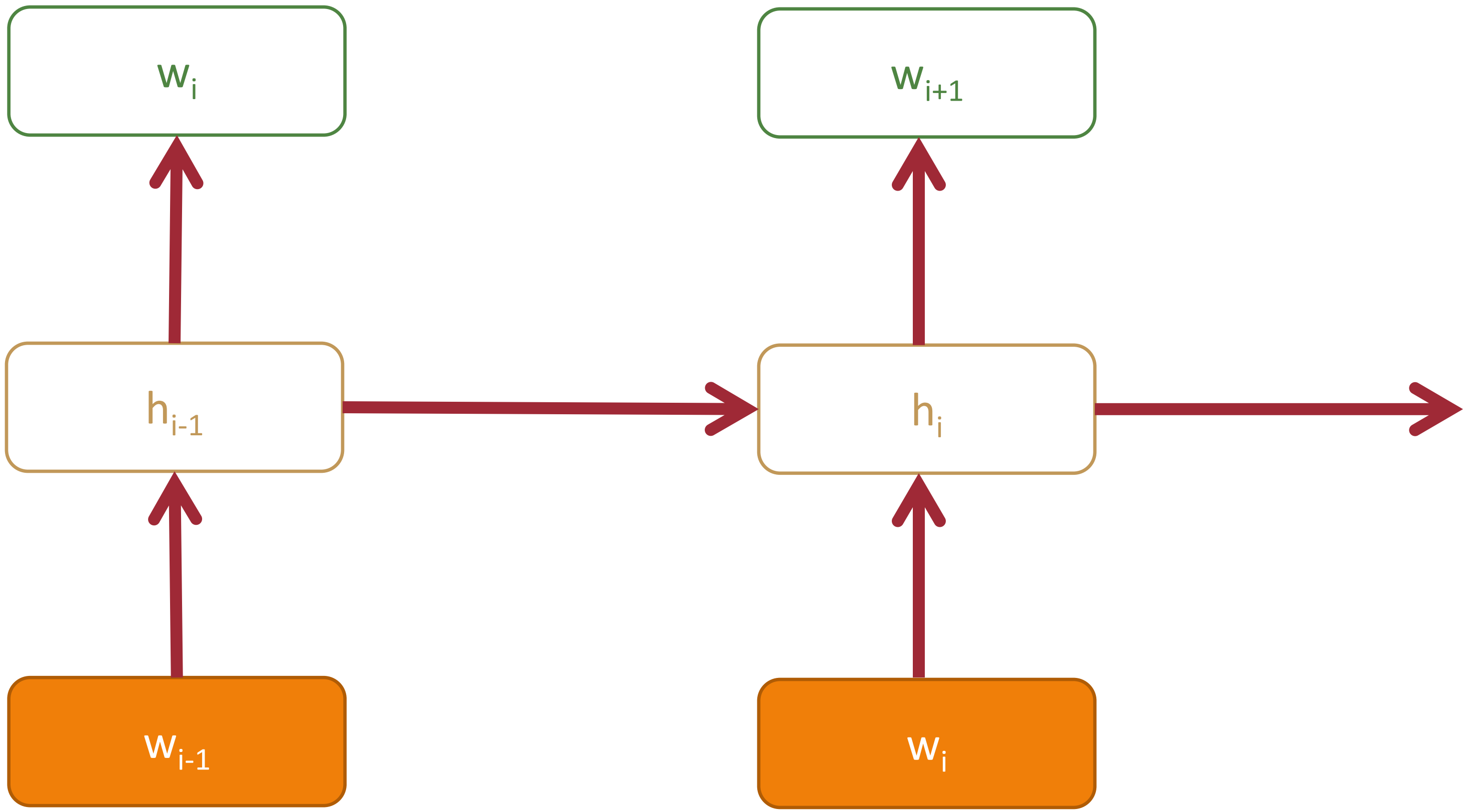


$out_{h1} = g(in_{h1})$
$= \frac{1}{1+e^{-in_{h1}}}$
$= \frac{1}{1+e^{-(-.3275)}}$
$= .4188$

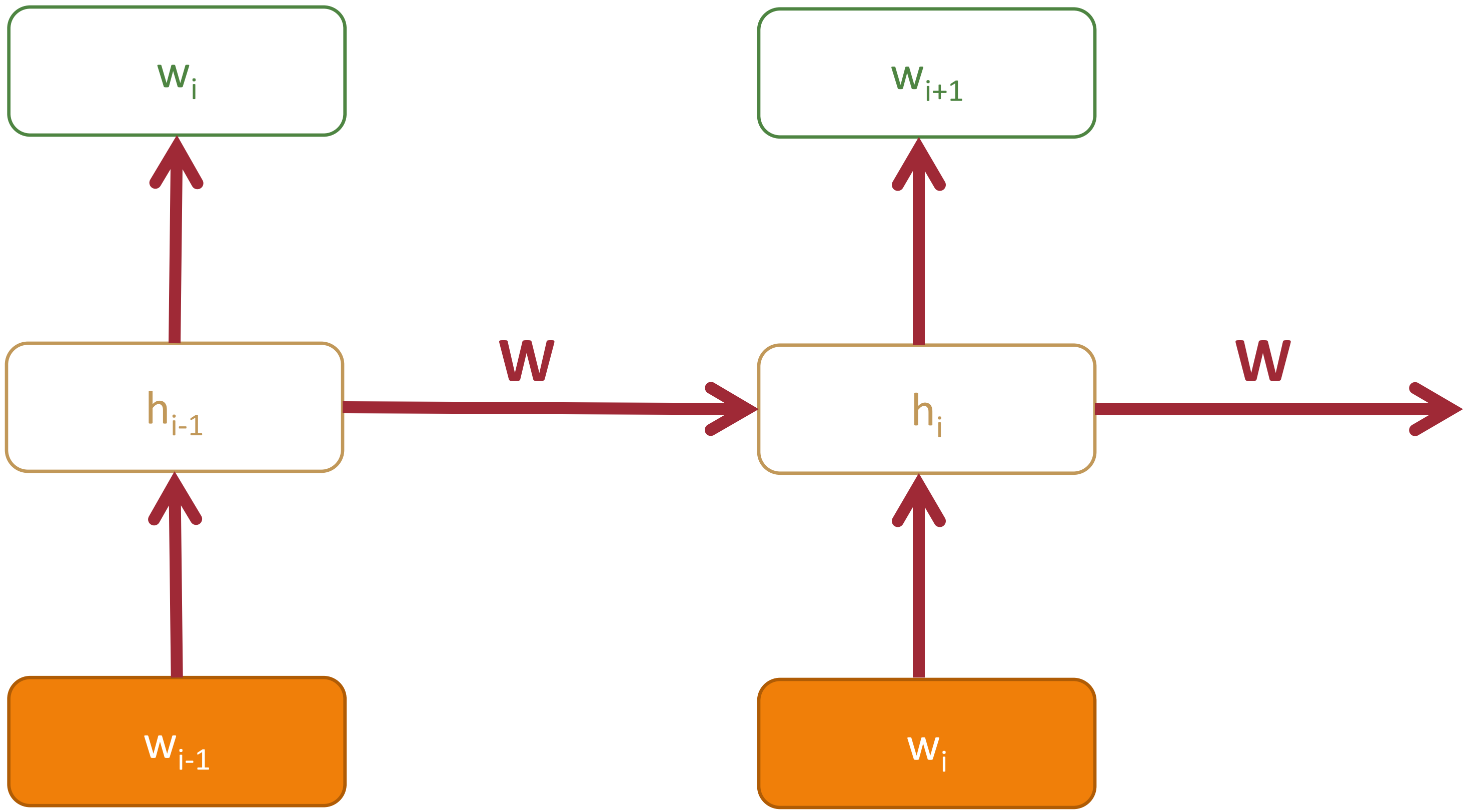
$out_{h2} = g(in_{h2})$
$= \frac{1}{1+e^{-in_{h2}}}$
$= \frac{1}{1+e^{-(-.3075)}}$
$= .4237$

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

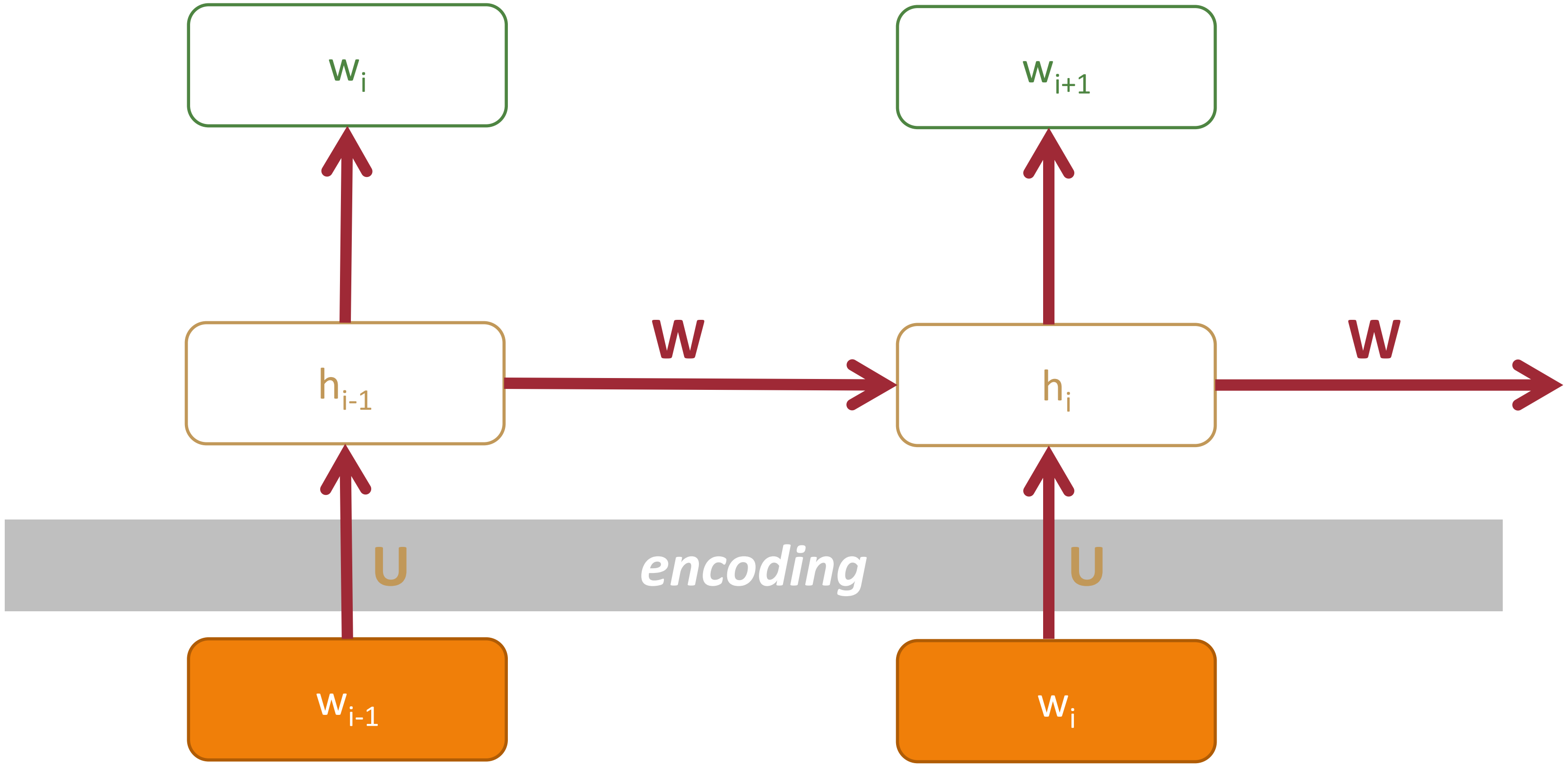
A Recurrent Neural Network Cell



A Recurrent Neural Network Cell

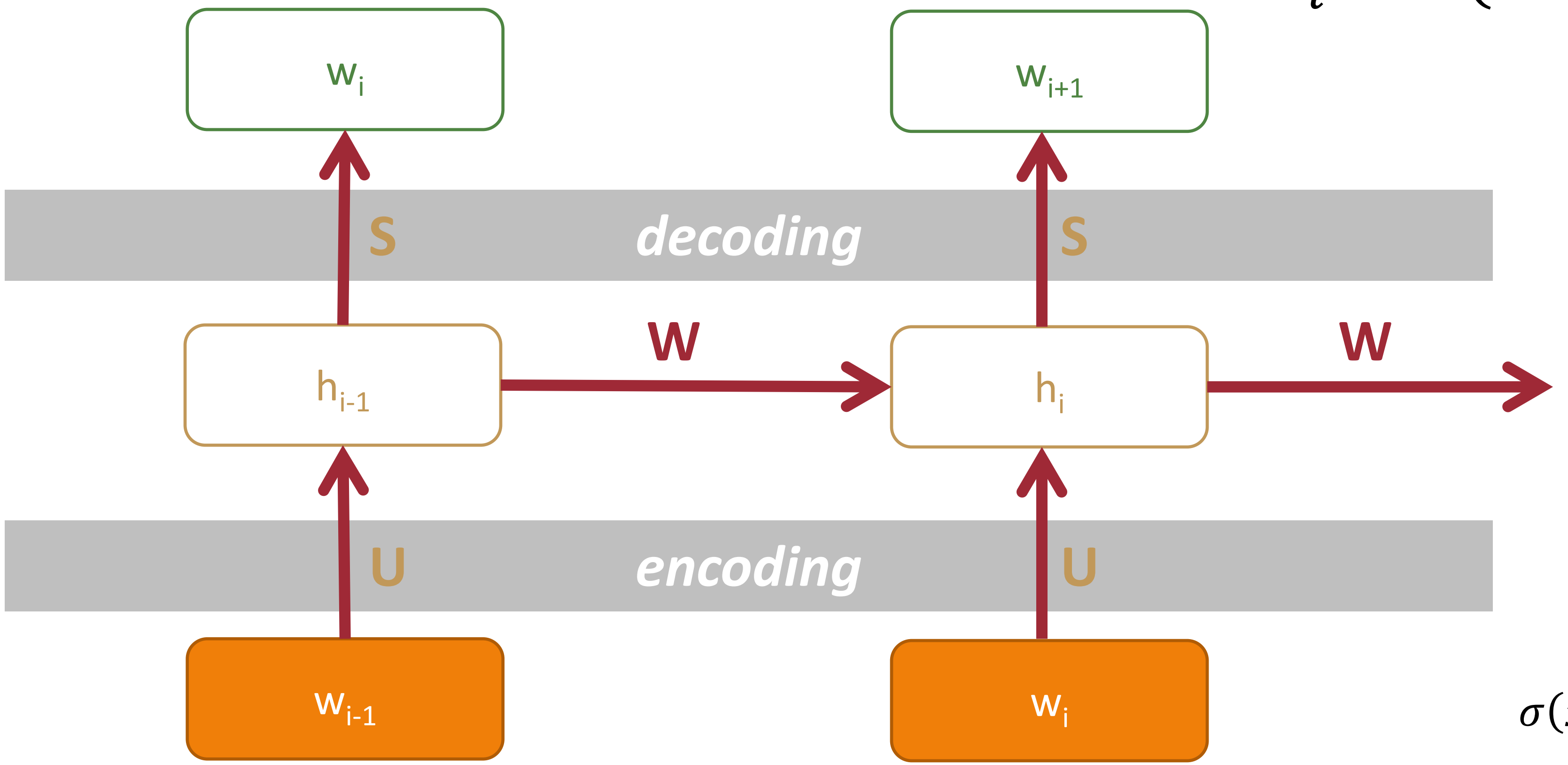


A Recurrent Neural Network Cell



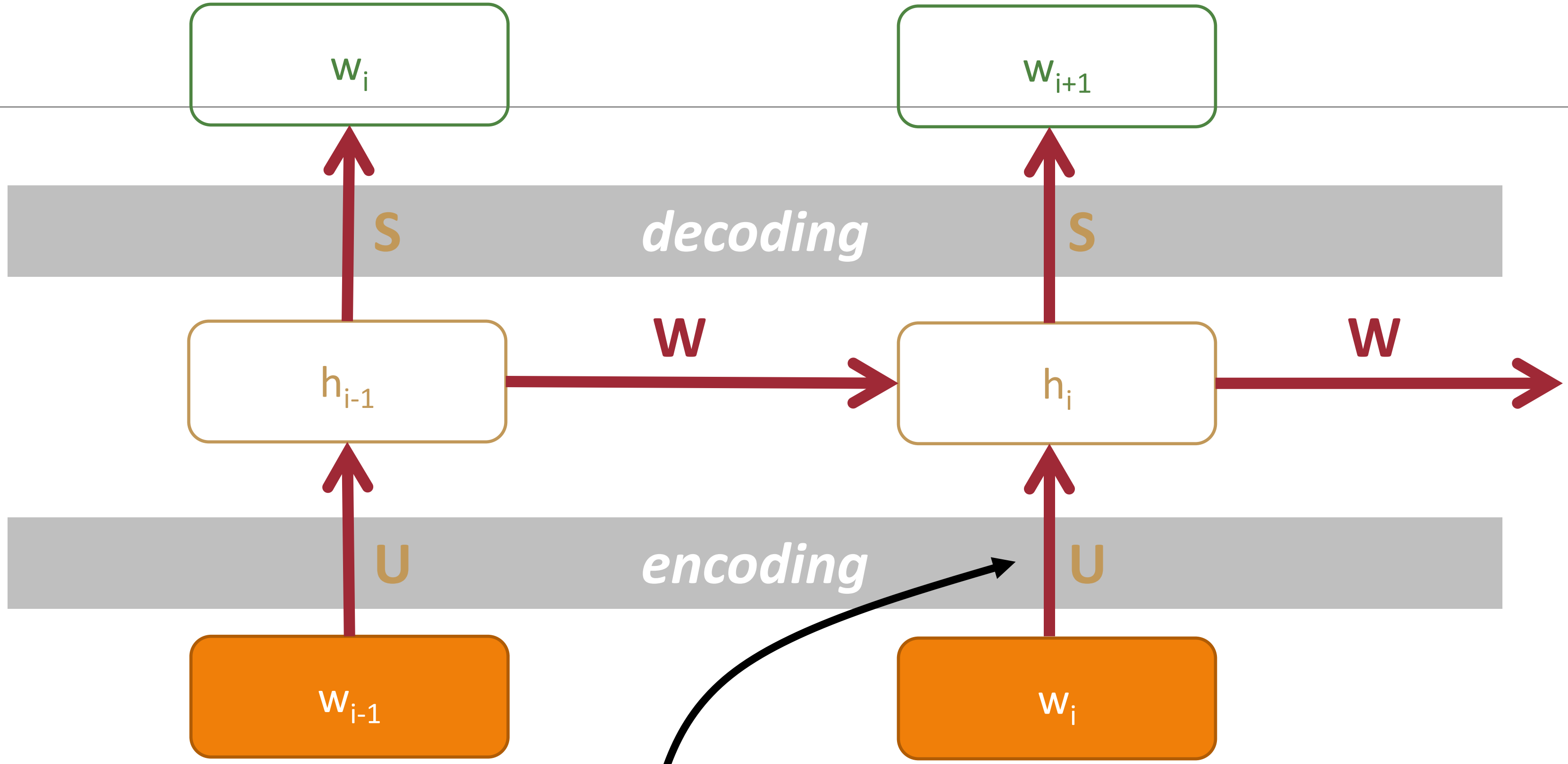
A Recurrent Neural Network Cell

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



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

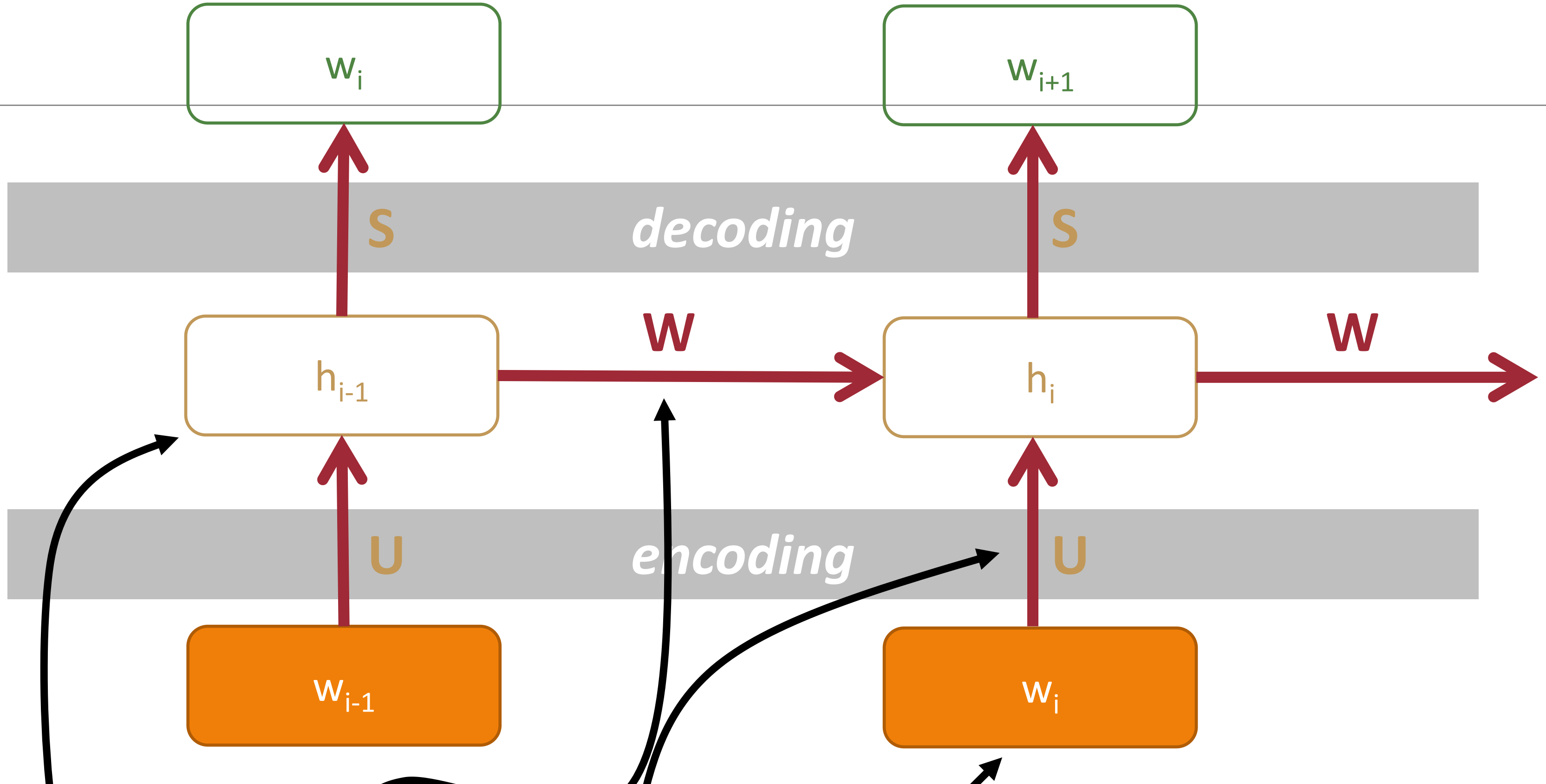
A *Simple* Recurrent Neural Network Cell



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

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

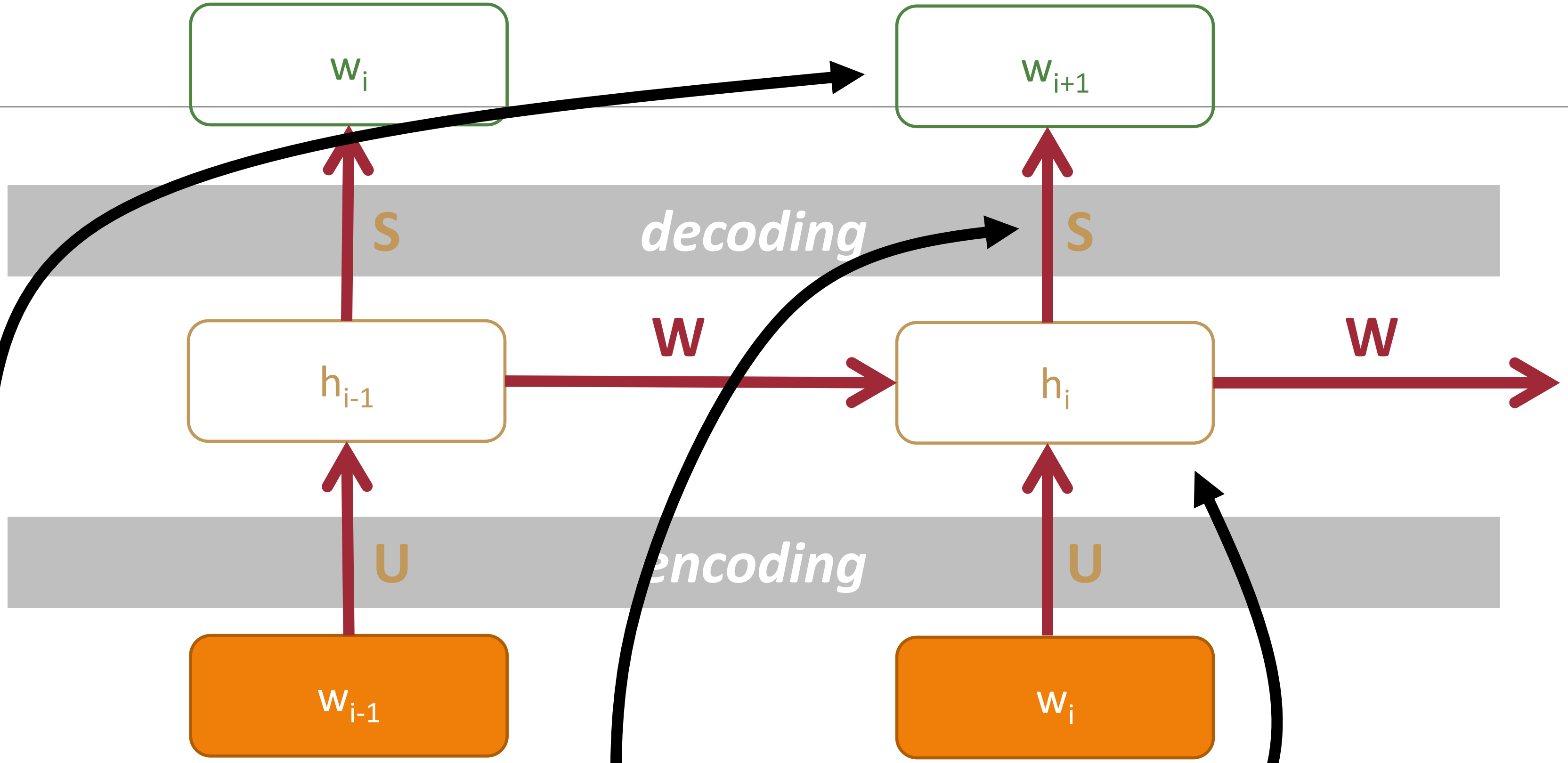
A *Simple* Recurrent Neural Network Cell



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

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

A *Simple* Recurrent Neural Network Cell

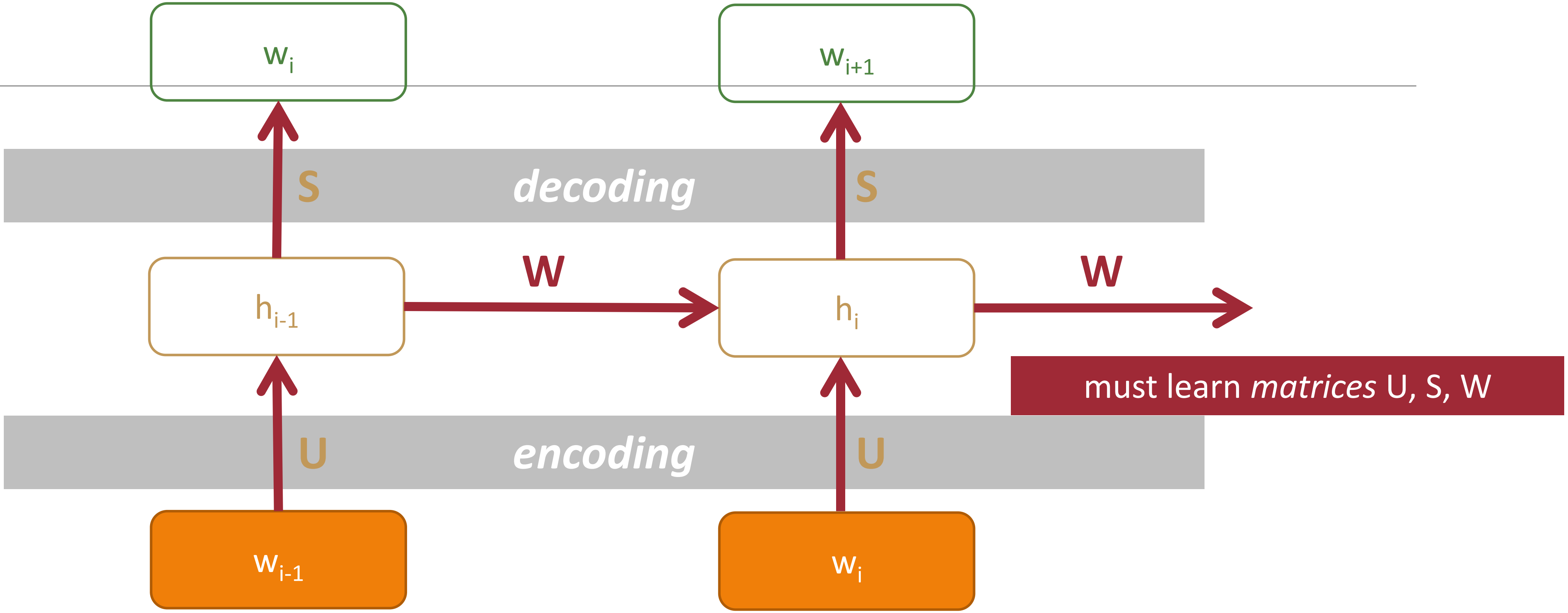


$$h_i = \sigma(W h_{i-1} + U w_i)$$

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

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

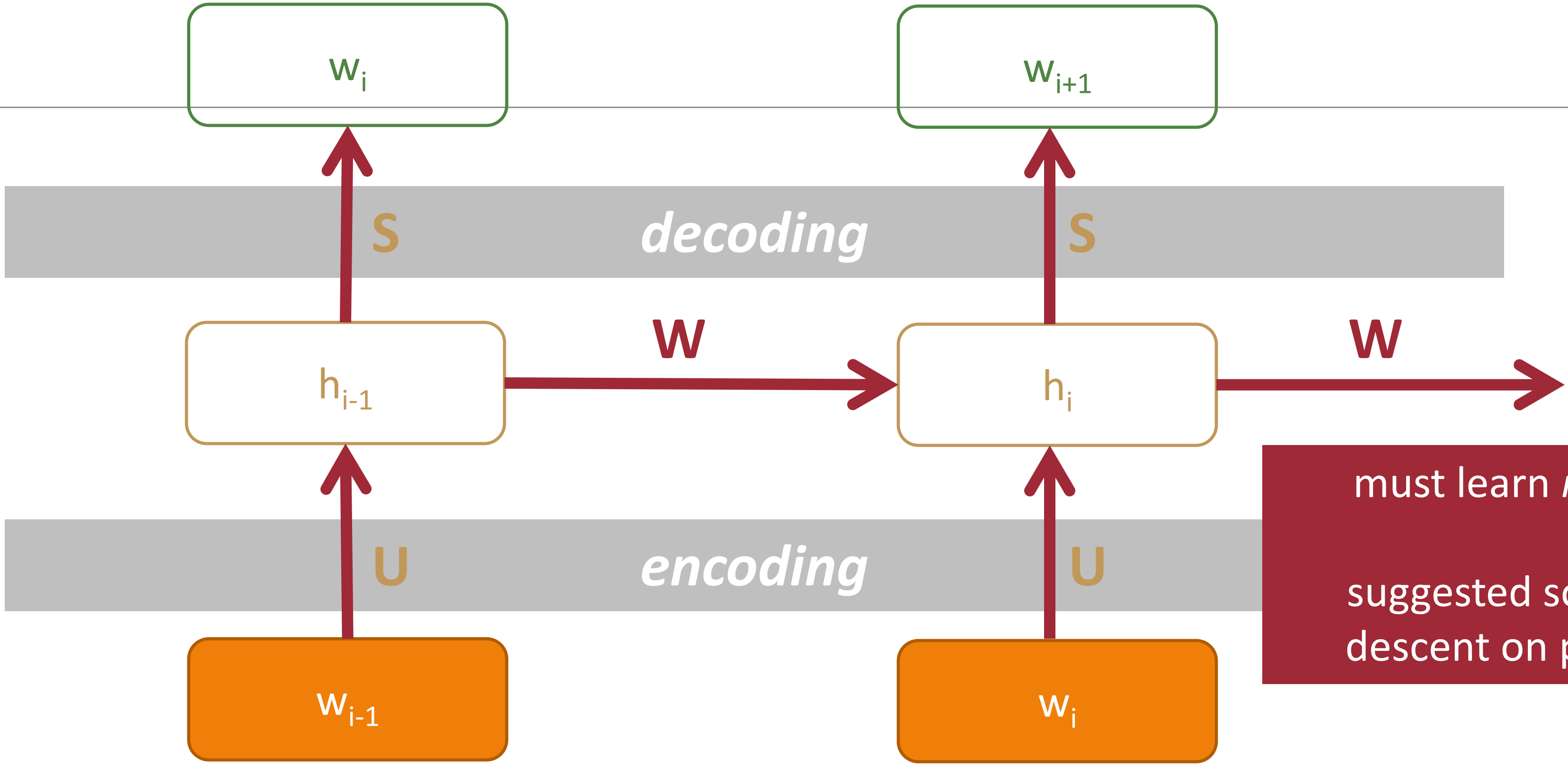
A *Simple* Recurrent Neural Network Cell



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

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

A *Simple* Recurrent Neural Network Cell



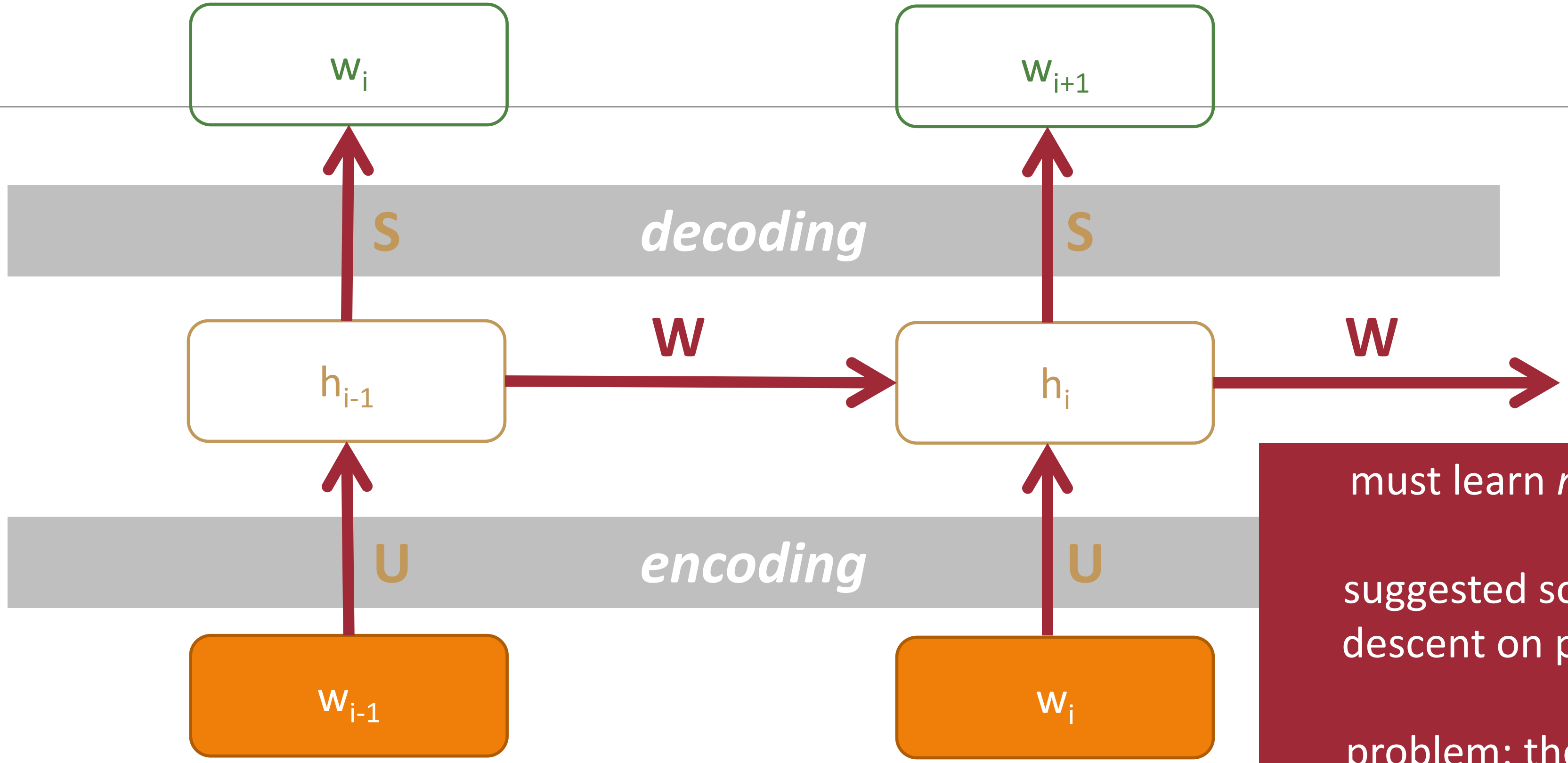
must learn *matrices* U, S, W

suggested solution: gradient descent on prediction ability

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

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

A *Simple* Recurrent Neural Network Cell



must learn *matrices* U, S, W

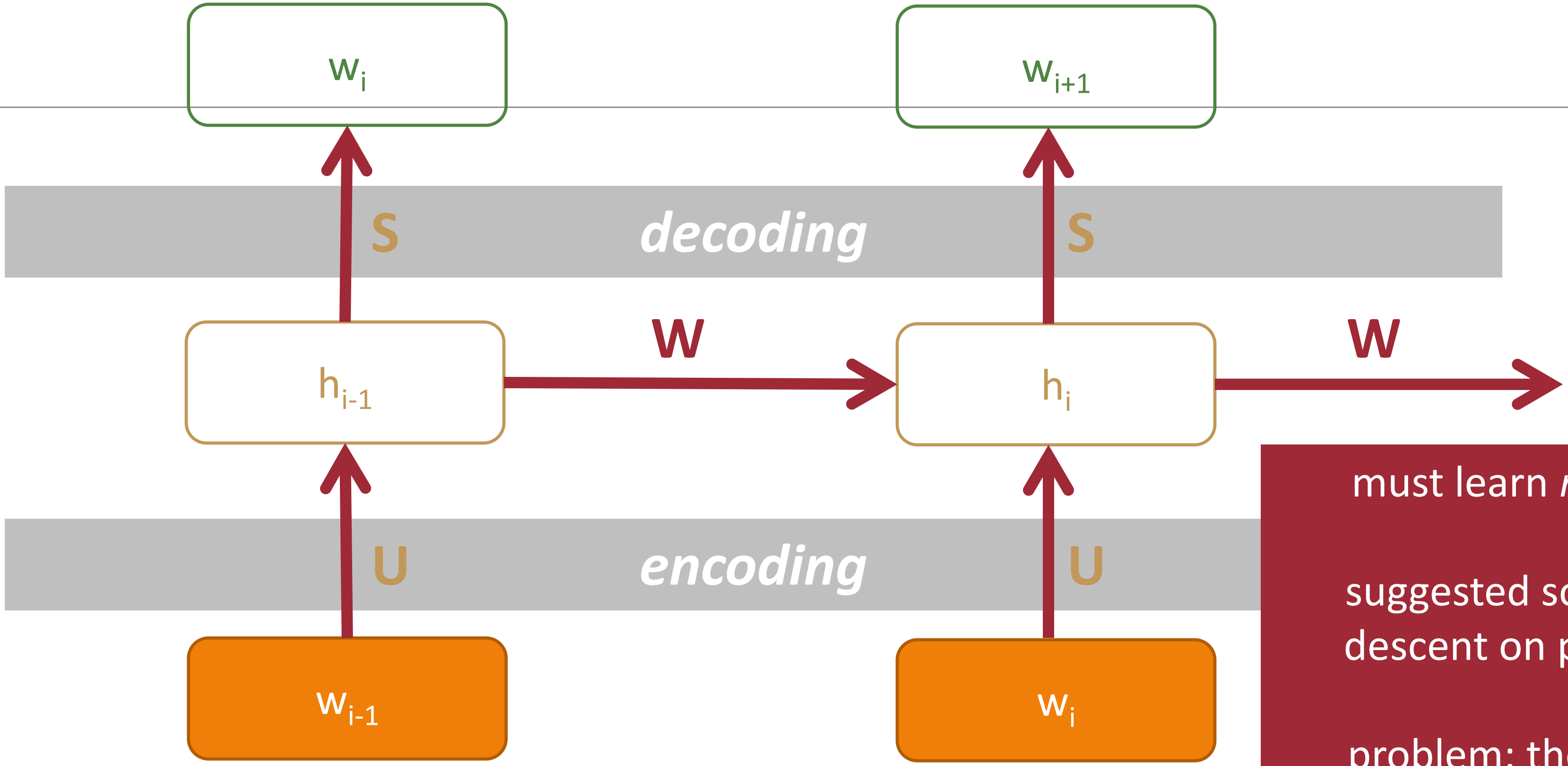
suggested solution: gradient descent on prediction ability

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

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

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

A *Simple* Recurrent Neural Network Cell



must learn *matrices* U, S, W

suggested solution: gradient descent on prediction ability

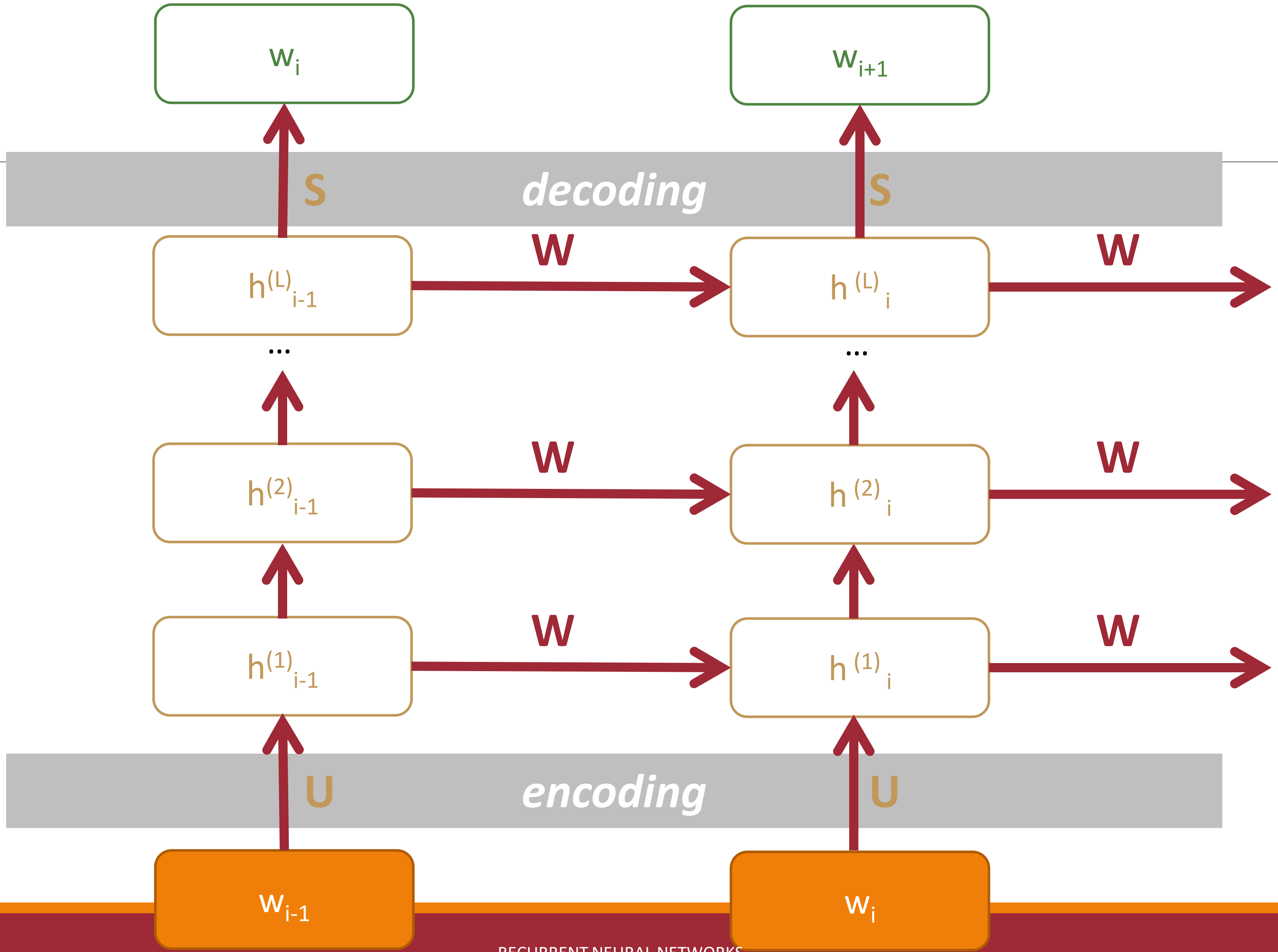
problem: they're *tied* across inputs/timesteps

good news for you: many toolkits do this automatically

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

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

A Multi-Layer *Simple* Recurrent Neural Network Cell



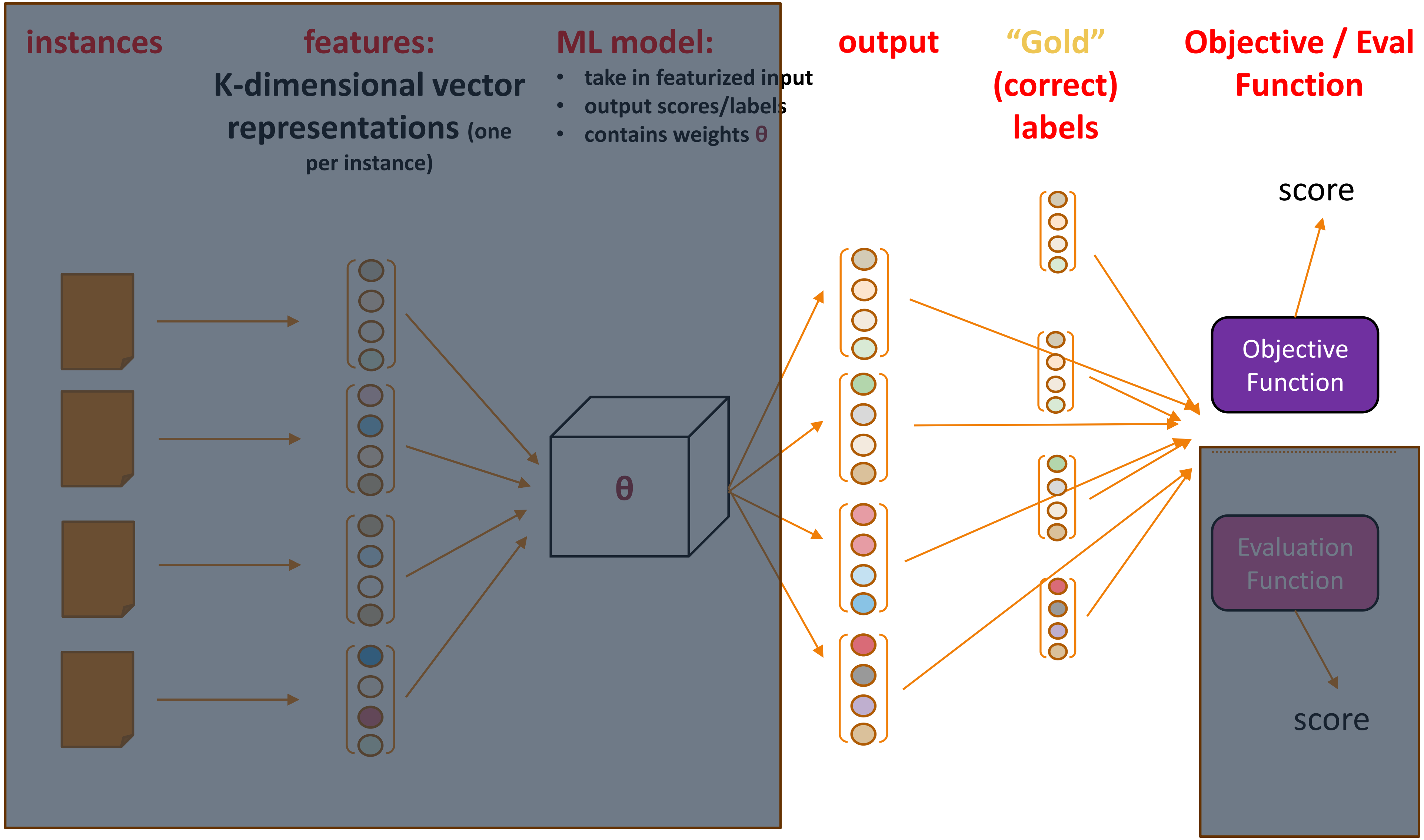
How do you learn an RNN?

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

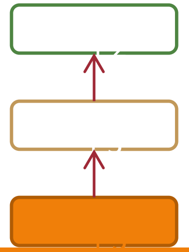
$\log .2$

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



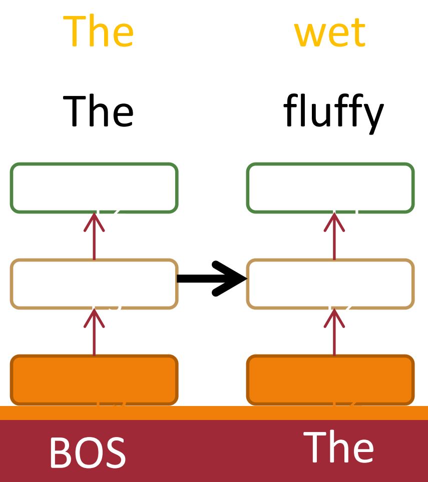
BOS

Recurrent NN Loss

$\log .2 + \log .12$

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

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



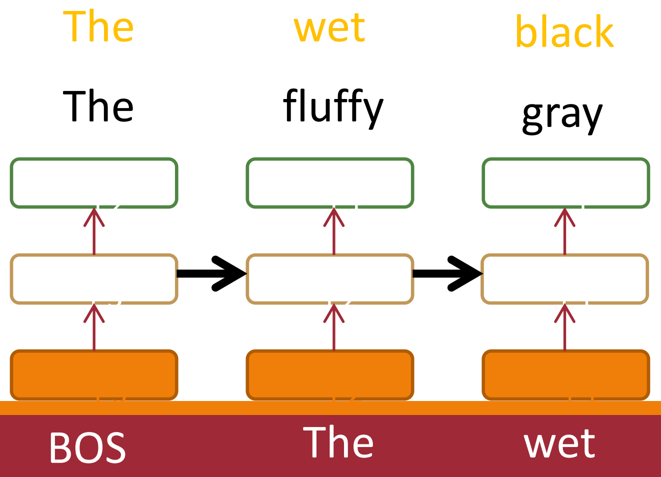
Recurrent NN Loss

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

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

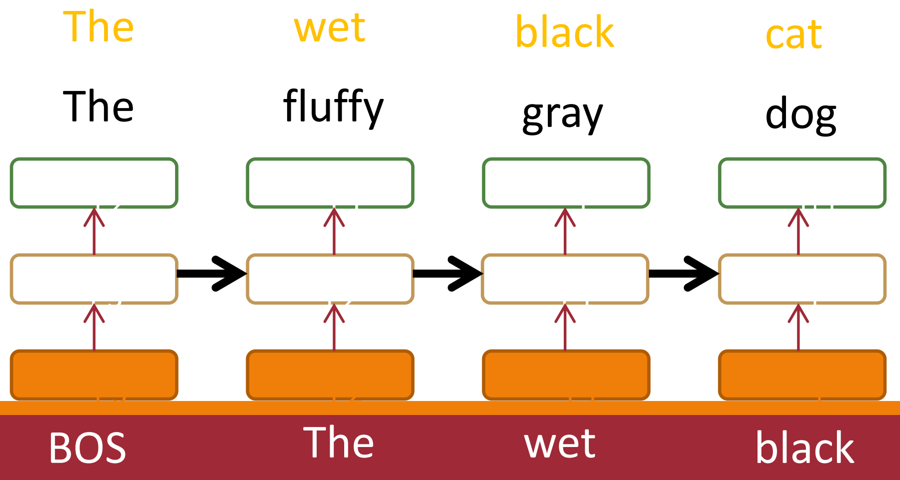
$\log .2 + \log .12 + \log .2 + \log .19$

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

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

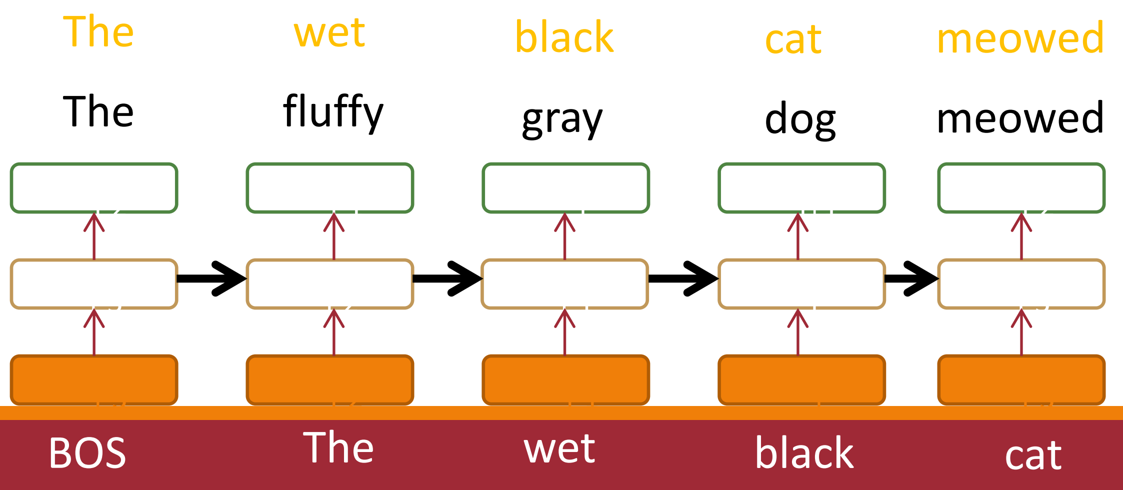
word	prob.
The	.2
gray	.01
blue	.001
fluffy	.0005
wet	.0005
…	…

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
…	…

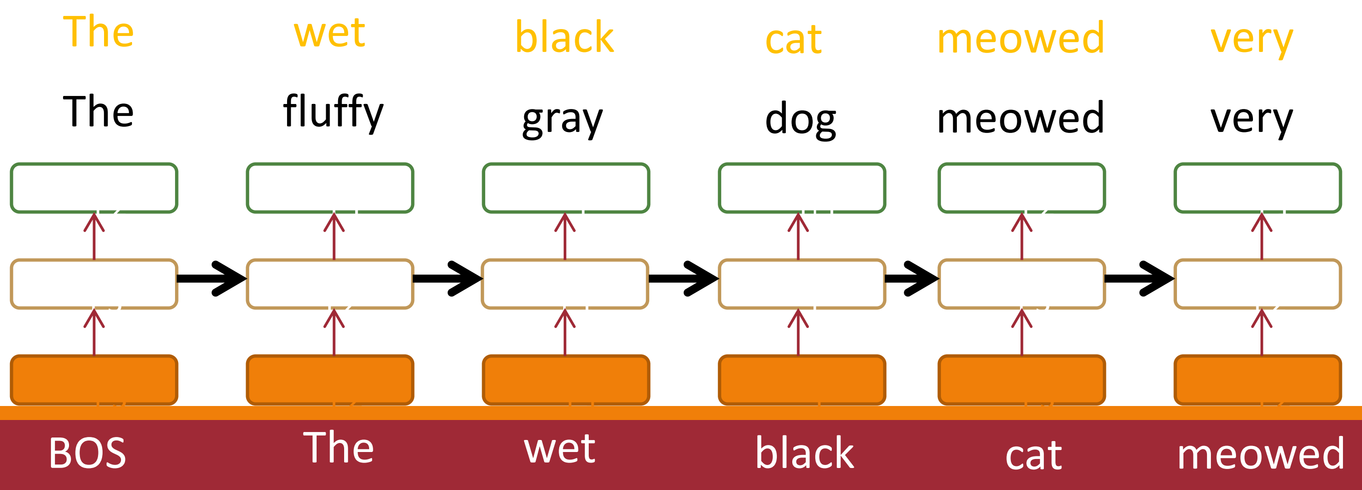
word	prob
meowed	.3
purred	.2
hissed	.1
fluffy	.001
wet	.001
…	…



Recurrent NN Loss

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

word	prob.	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
…	…	…	…	…	…	…	…	…	…	…	…

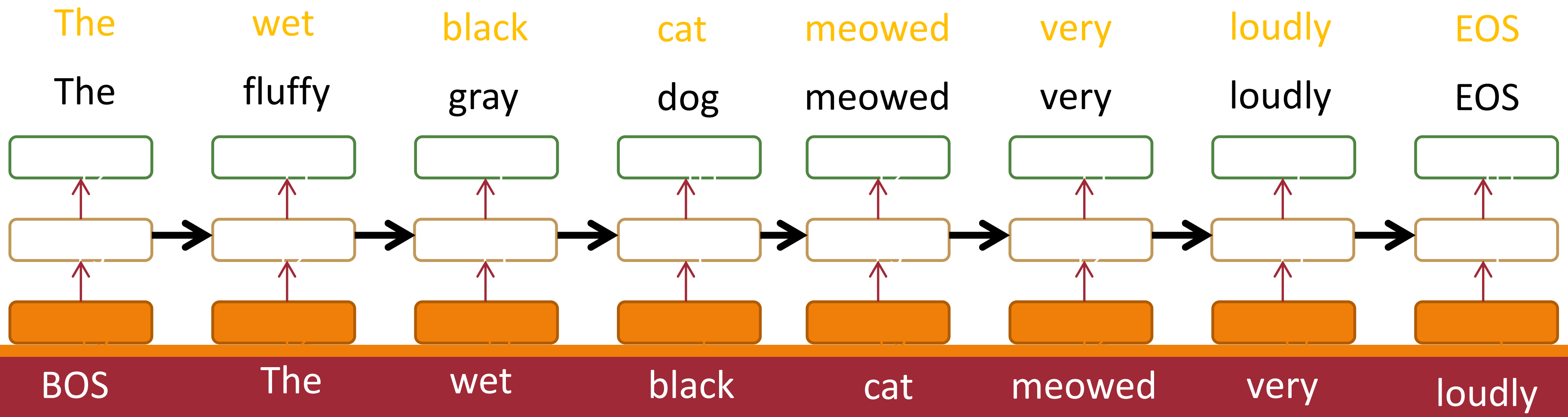


Recurrent NN Loss

(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:

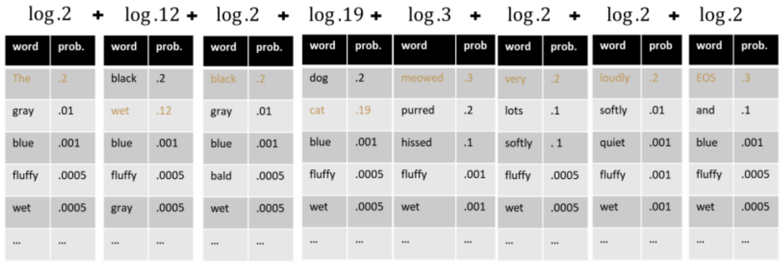
1. Compute loss l on x_i

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



Gradient Descent: Backpropagate the Error

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

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

PyTorch RNN LMs

Pick Your Toolkit

PyTorch

Deeplearning4j

TensorFlow

Caffe

Keras

MXNet

Torch

…

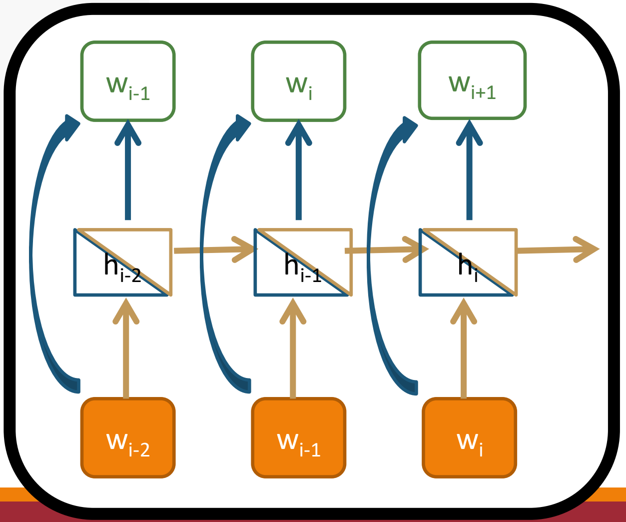
Comparisons:

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

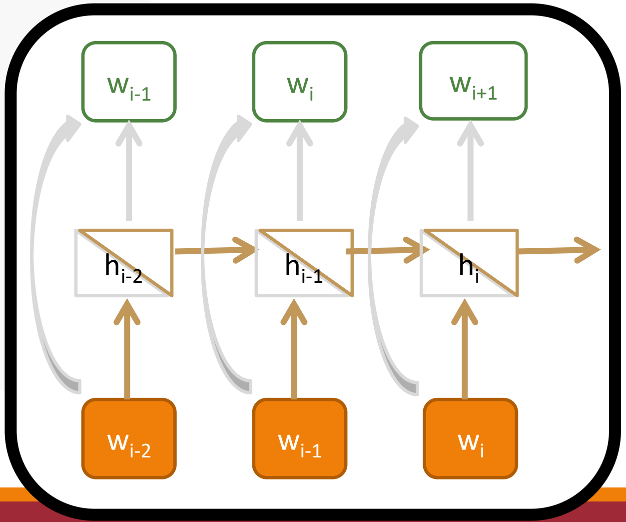
```
import torch.nn as nn
import torch.nn.functional as F

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

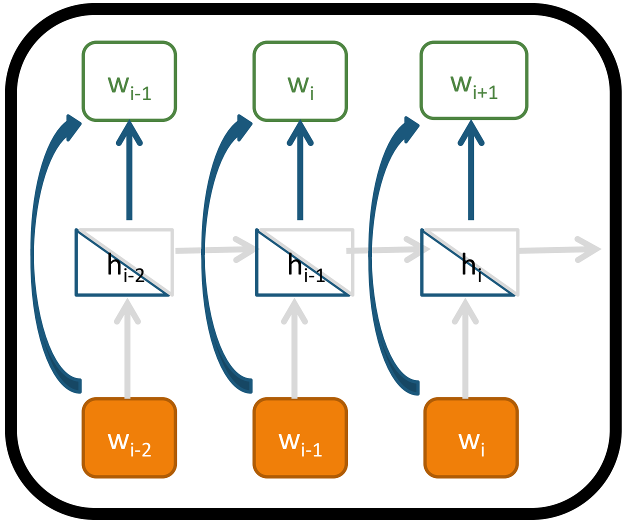
```
import torch.nn as nn
import torch.nn.functional as F

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

```
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

```
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, hidd
        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:

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

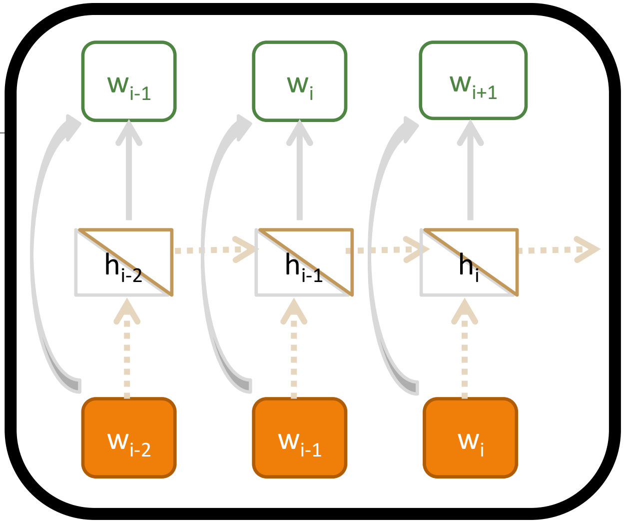
When the input Tensor is a sparse tensor then the unspecified values are treated as `-inf`.

Defining A Simple RNN in Python

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

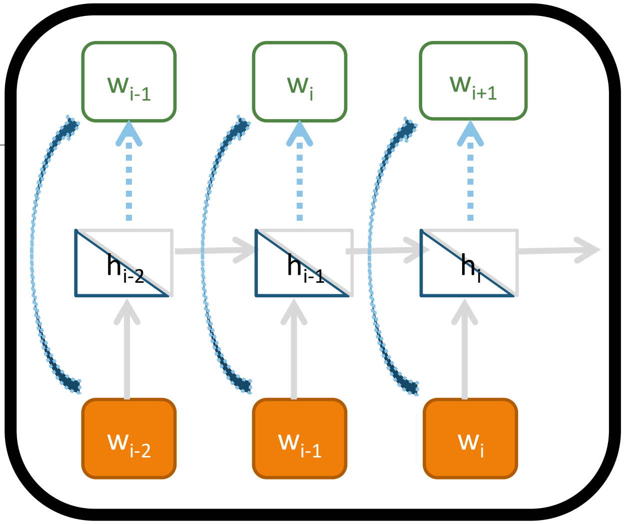
```
import torch.nn as nn
import torch.nn.functional as F

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




```python
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

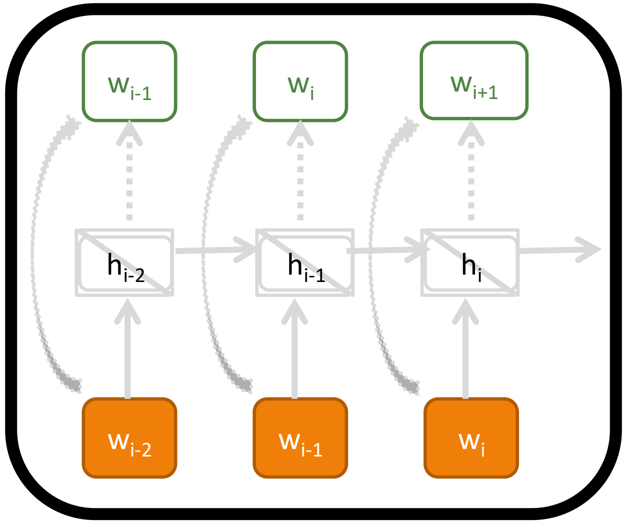
http://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

Training A Simple RNN in Python

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

```
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

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

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

```
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 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

```
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 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

```python
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

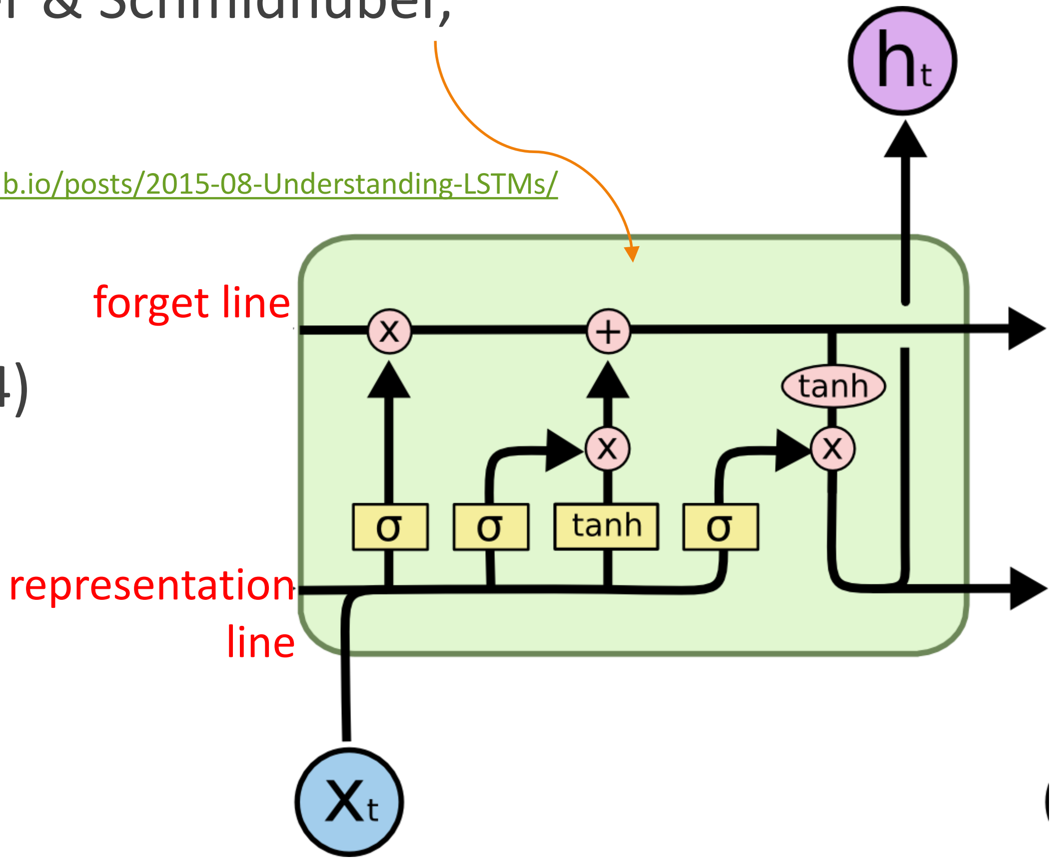
```python
def train(rnn, training_data, n_epoch = 50, learning_rate = 0.2, criterion = nn.NLLLoss()):
    """
    Learn on a batch of training_data for a specified number of iterations and reporting thresholds
    """
    # Keep track of losses for plotting
    current_loss = 0
    all_losses = []
    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)}")
```

nn.CrossEntropyLoss()

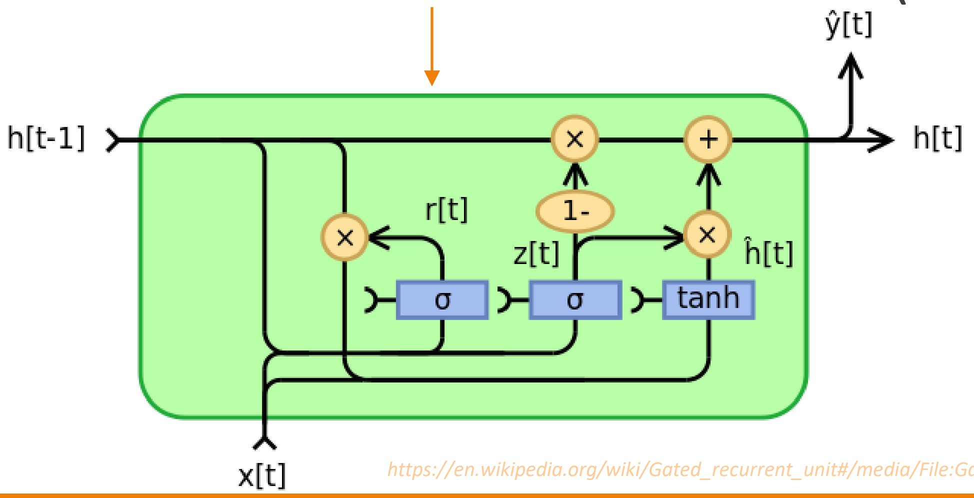
Another Solution: LSTMs/GRUs

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

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



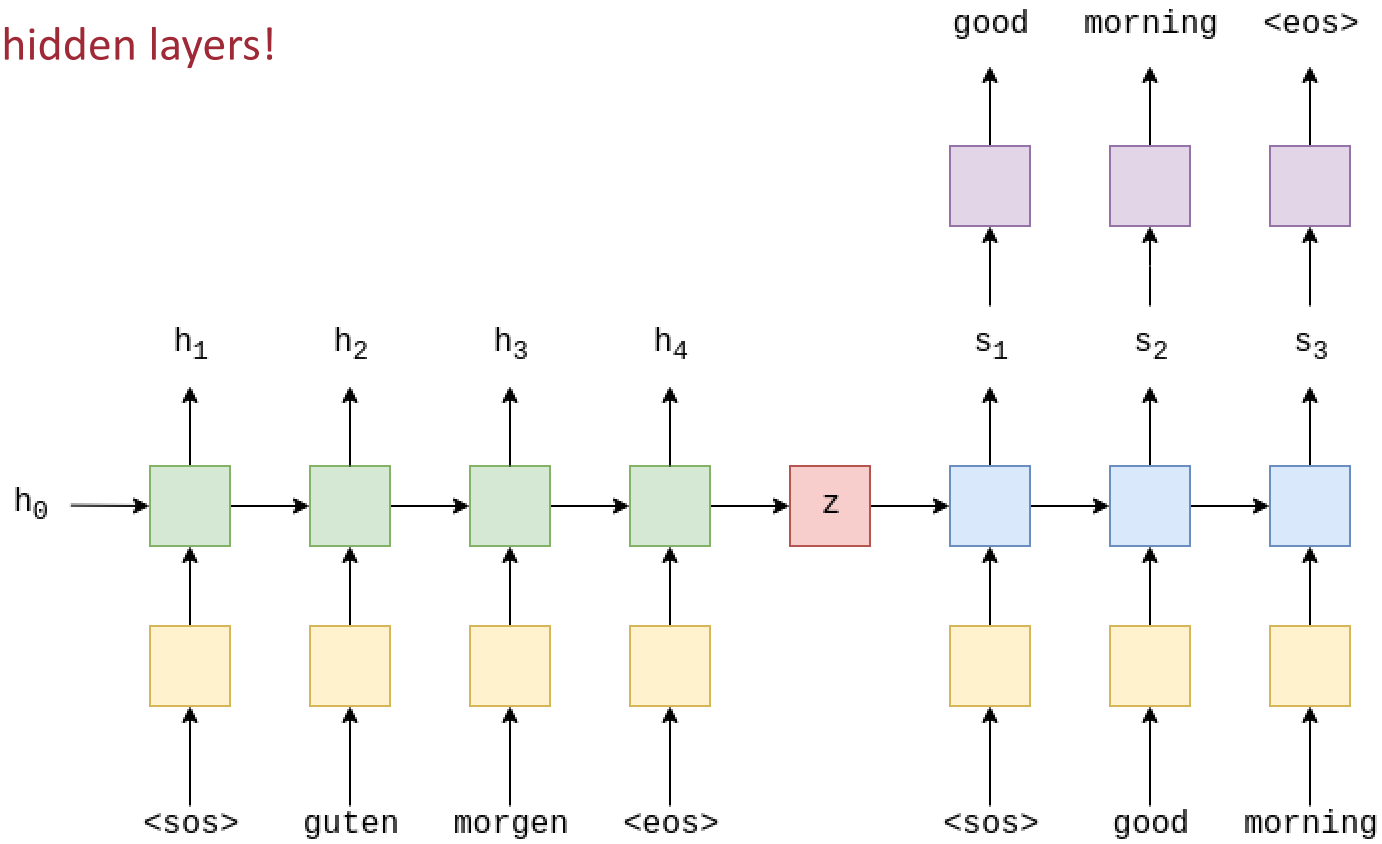
GRU: Gated Recurrent Unit (Cho et al., 2014)



https://en.wikipedia.org/wiki/Gated_recurrent_unit#/media/File:Gated_Recurrent_Unit,_base_type.svg

Sequence-to-Sequence

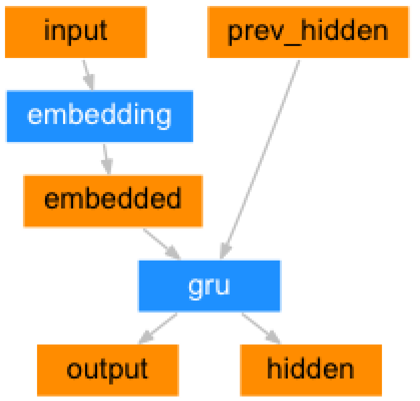
Note that this still has hidden layers!



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

I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to Sequence Learning with Neural Networks,” in *Conference on Advances in Neural Information Processing Systems (NeurIPS)*, Montréal, Canada, 2014, pp. 3104–3112. https://proceedings.neurips.cc/paper_files/paper/2014/hash/a14ac55a4f27472c5d894ec1c3c743d2-Abstract.html

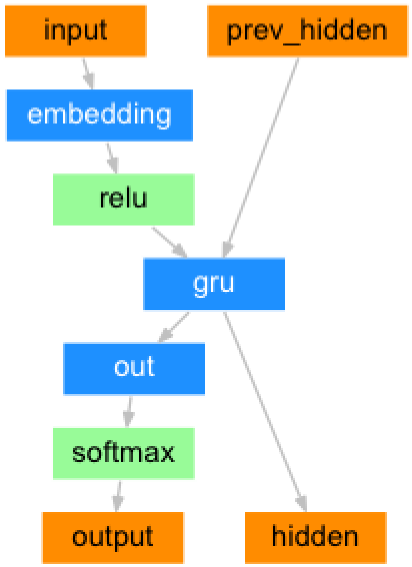
Encoder




```
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



https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html

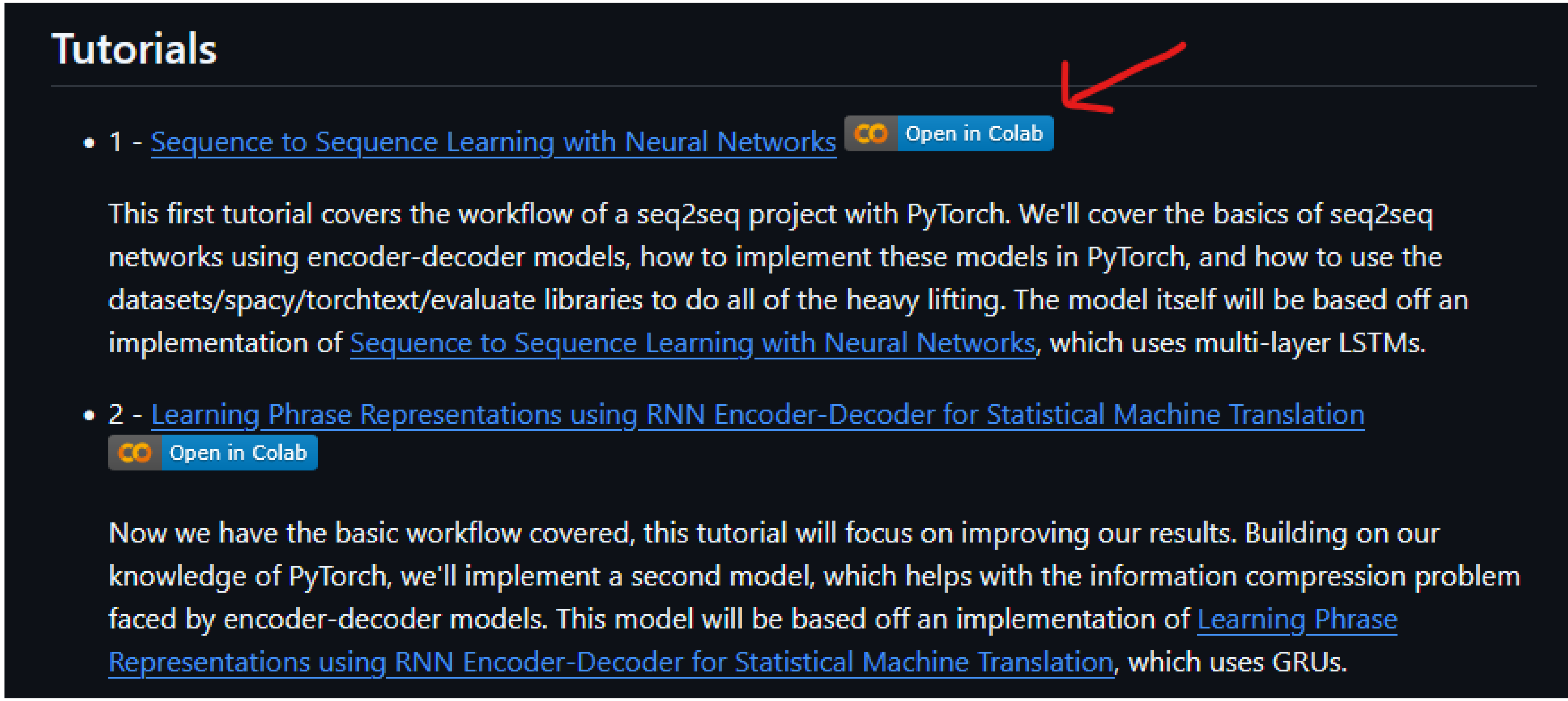
```
class DecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size):
        super(DecoderRNN, self).__init__()
        self.embedding = nn.Embedding(output_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size, batch_first=True)
        self.out = nn.Linear(hidden_size,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

Seq2Seq Tutorial

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

Direct link:

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



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.