Recurrent Neural Models: Language Models and Generation

CMSC 473/673

Frank Ferraro

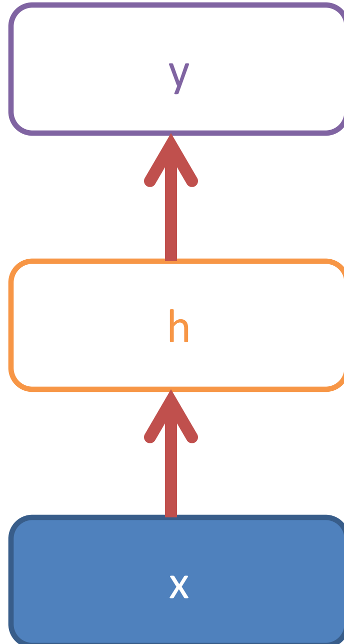
Outline

Core Problem

Basic cell definition

Example in PyTorch

Network Types: Flat Input, Flat Output



1. **Feed forward**

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

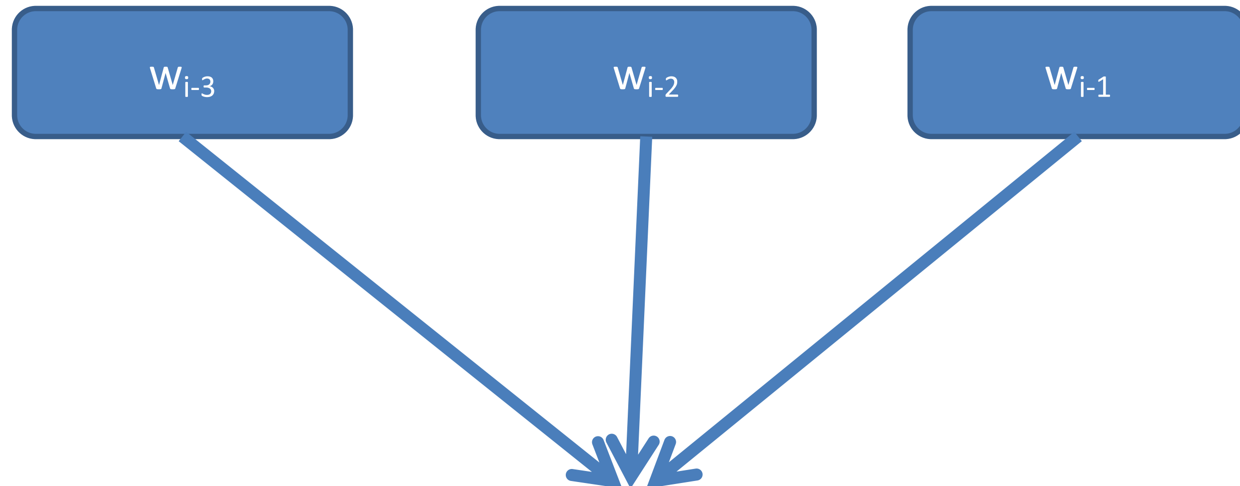
We've already seen some instances of this

Terminology

common NLP term	Log-Linear Models
as statistical regression	(Multinomial) logistic regression
	Softmax regression
based in information theory	Maximum Entropy models (MaxEnt)
a form of	Generalized Linear Models
viewed as	Discriminative Naïve Bayes
to be cool today :)	Very shallow (sigmoidal) neural nets

Recall: N-gram to Maxent to Neural Language Models

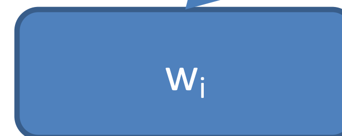
given some context…



compute beliefs about what is likely…

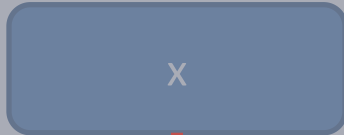
$$p(w_i | w_{i-3}, w_{i-2}, w_{i-1}) = \text{softmax}(\theta_{w_i} \cdot f(w_{i-3}, w_{i-2}, w_{i-1}))$$

predict the next word



Recall: N-gram to Maxent to Neural Language Models

given some context…

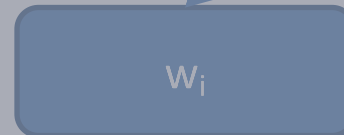


no *learned* representation h

compute beliefs about what is likely…

predict the next word

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



Recall: N-gram to Maxent to Neural Language Models

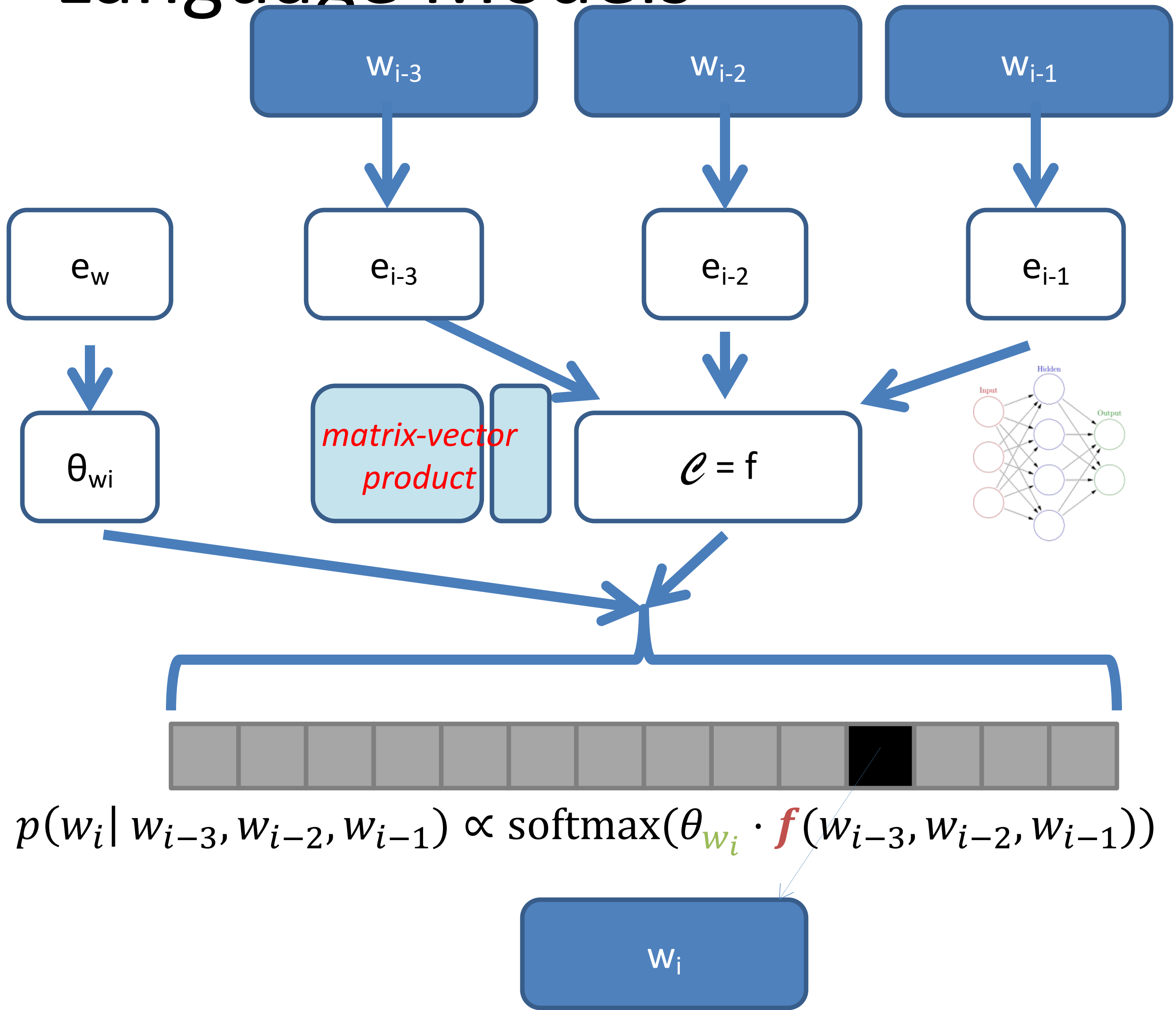
given some context…

create/use "distributed representations"…

combine these representations…

compute beliefs about what is likely…

predict the next word



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

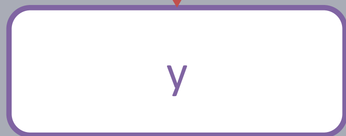
Recall: N-gram to Maxent to Neural Language Models

given some context…

create/use "distributed representations"…

combine these representations…

compute beliefs about what is likely…



predict the next word

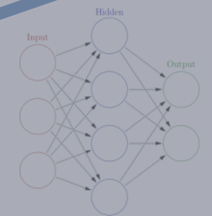
w_{i-3} w_{i-2} w_{i-1}

e_w e_{i-3} e_{i-2} e_{i-1}

θ_{wi}

matrix-vector product

$\mathcal{C} = f$



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

w_i

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:
  $h_l = f_l(W_l h_{l-1} + b_l)$

return $\operatorname{argmax}_y \operatorname{softmax}(\theta h_L)$

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

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 $\underset{y}{\operatorname{argmax}} \operatorname{softmax}(\theta h_L)$

In Pytorch (torch.nn):

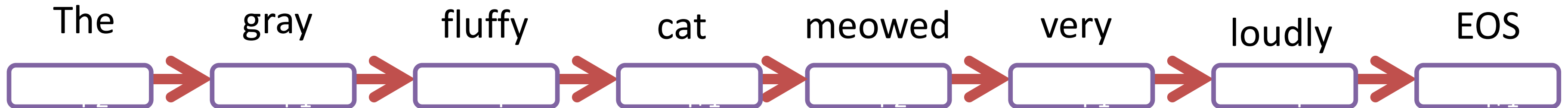
Activation functions:
https://pytorch.org/docs/stable/nn.html?highlight=activation#non-linear-activations-weighted-sum-nonlinearity

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>)
```

A Neural N-Gram Model

The gray fluffy cat meowed very loudly



A Neural N-Gram Model (N=3)

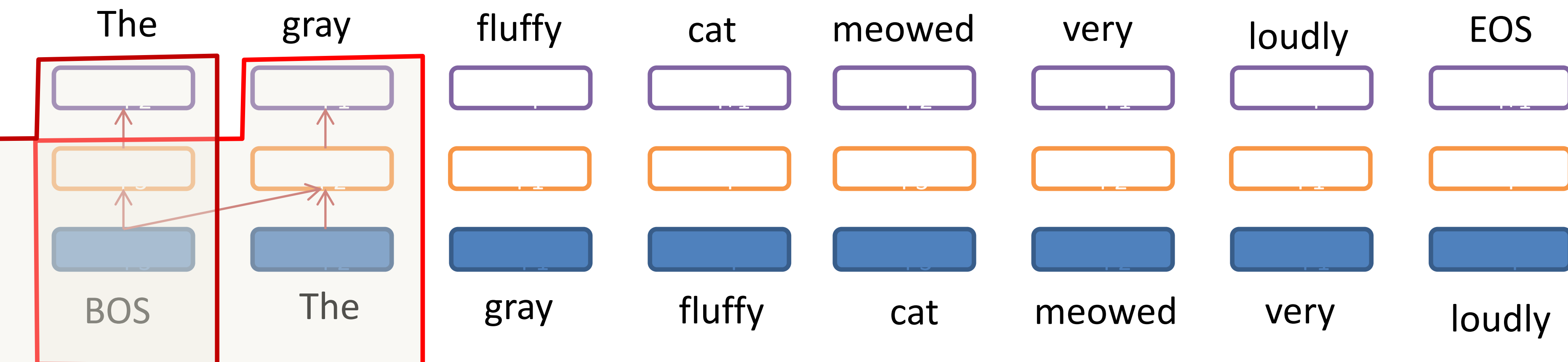
The gray fluffy cat meowed very loudly

The gray fluffy cat meowed very loudly EOS

BOS The gray fluffy cat meowed very loudly

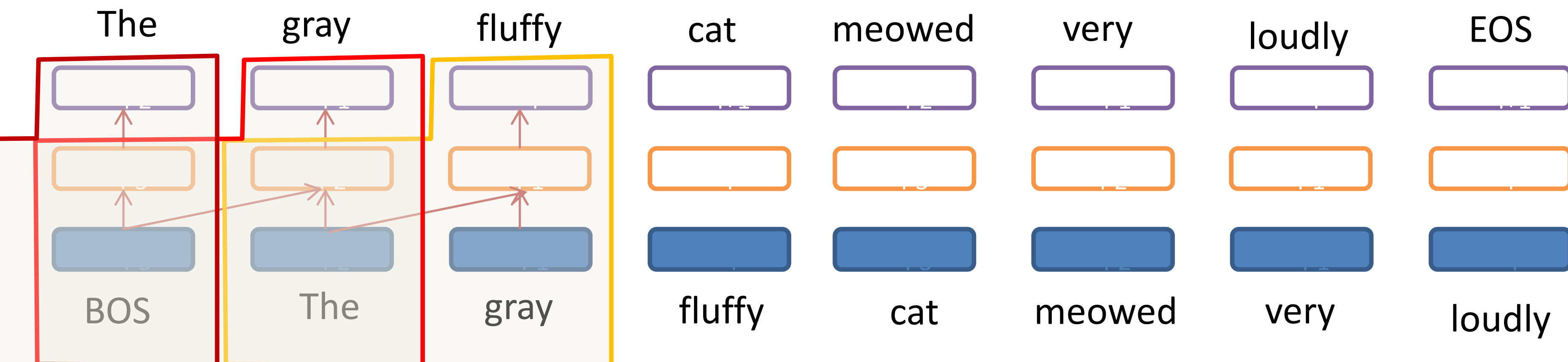
A Neural N-Gram Model (N=3)

The gray fluffy cat meowed very loudly



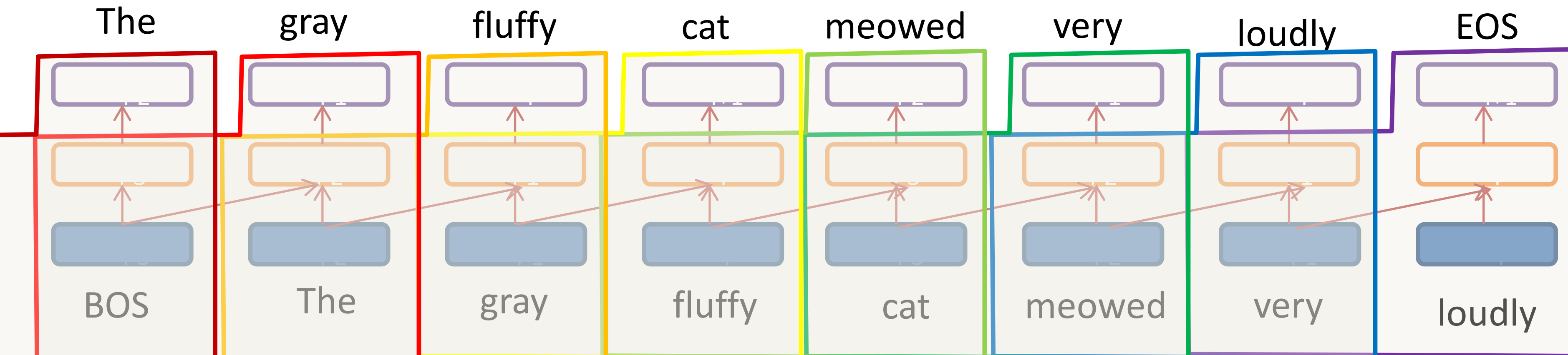
A Neural N-Gram Model (N=3)

The gray fluffy cat meowed very loudly



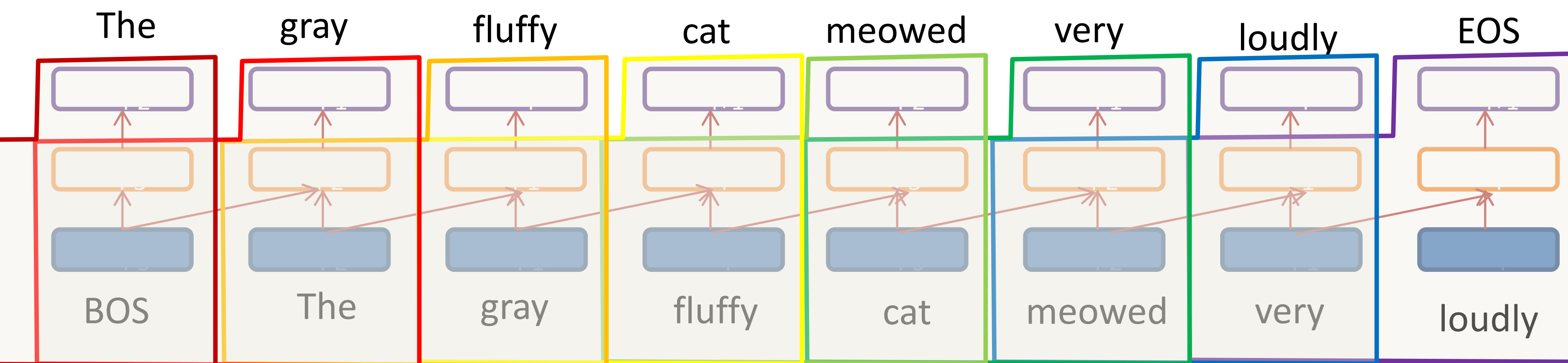
A Neural N-Gram Model (N=3)

The gray fluffy cat meowed very loudly



A Neural N-Gram Model (N=3)

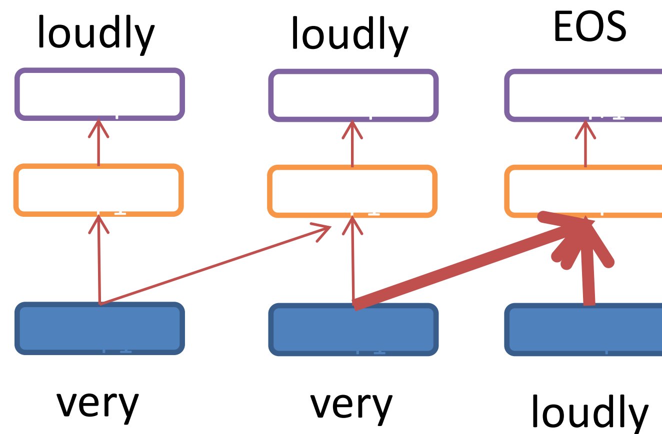
The gray fluffy cat meowed very loudly



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

A Neural N-Gram Model (N=3)

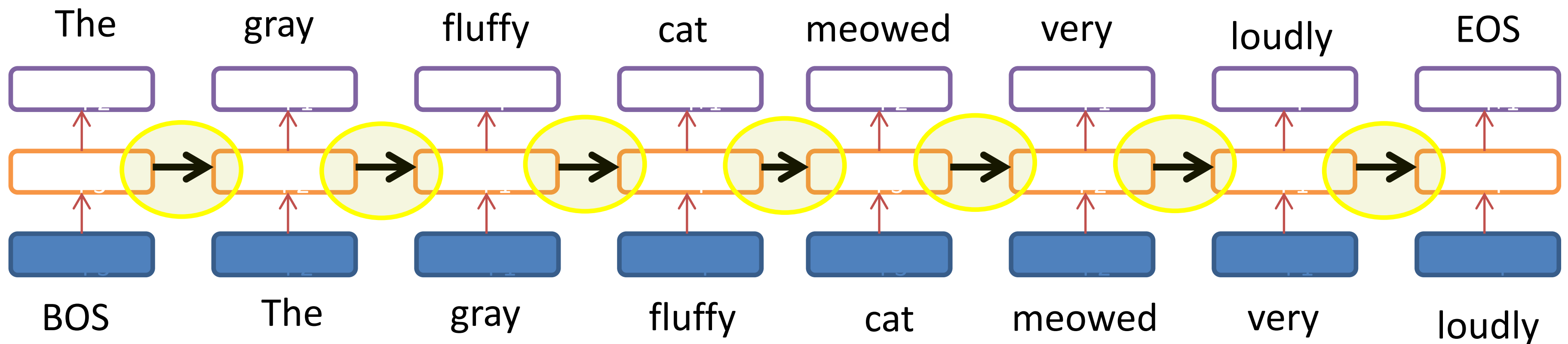
The gray fluffy cat meowed very loudly



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

A Recurrent Neural Language Model

The gray fluffy 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!

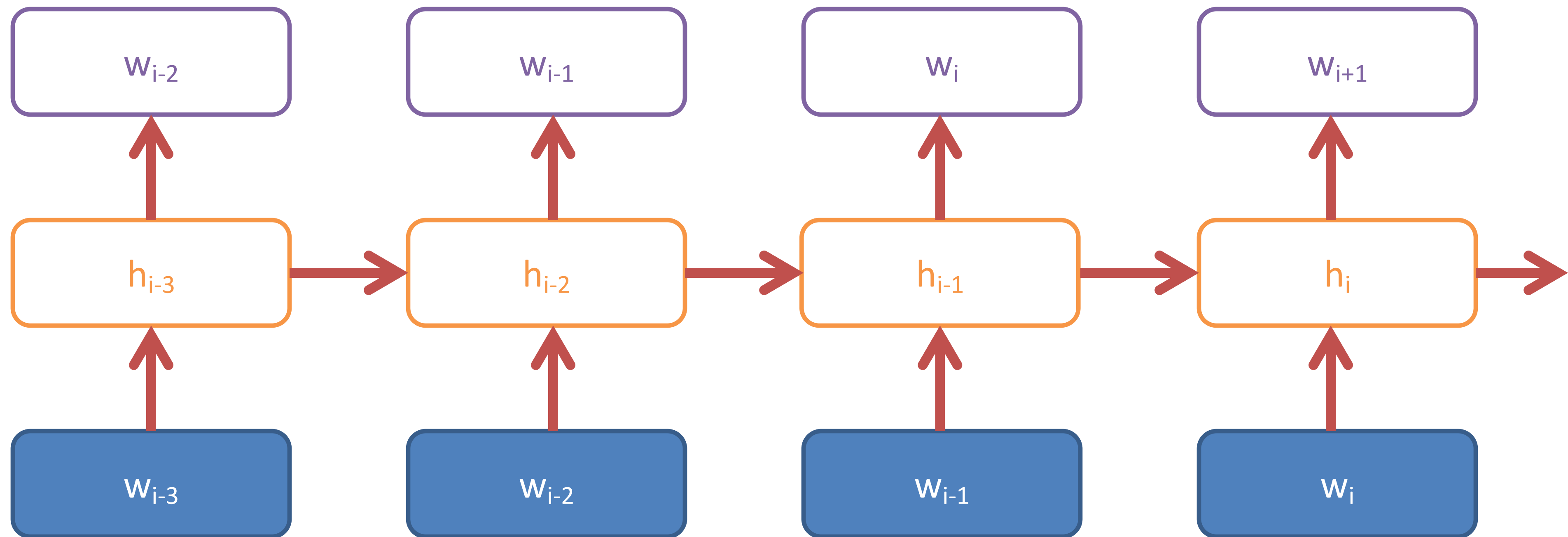
Outline

Types of networks

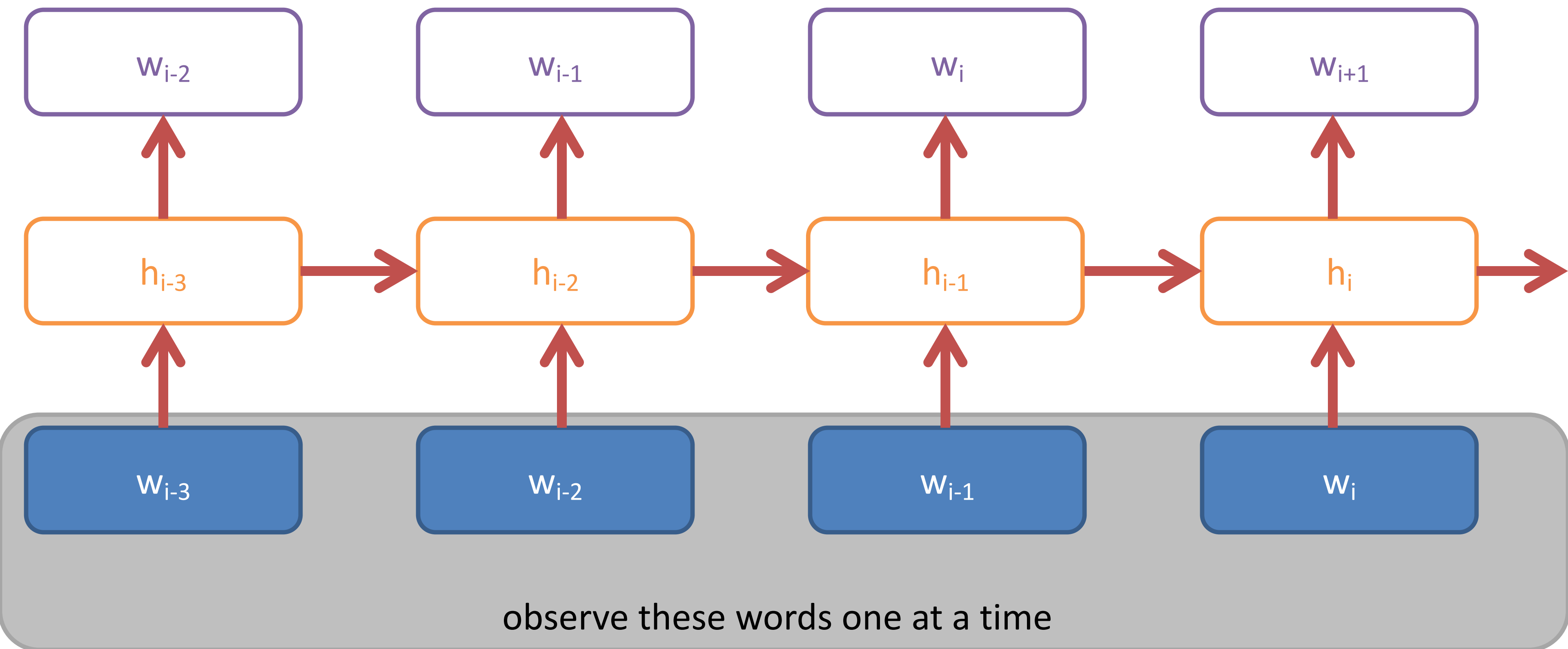
Basic cell definition

Example in PyTorch

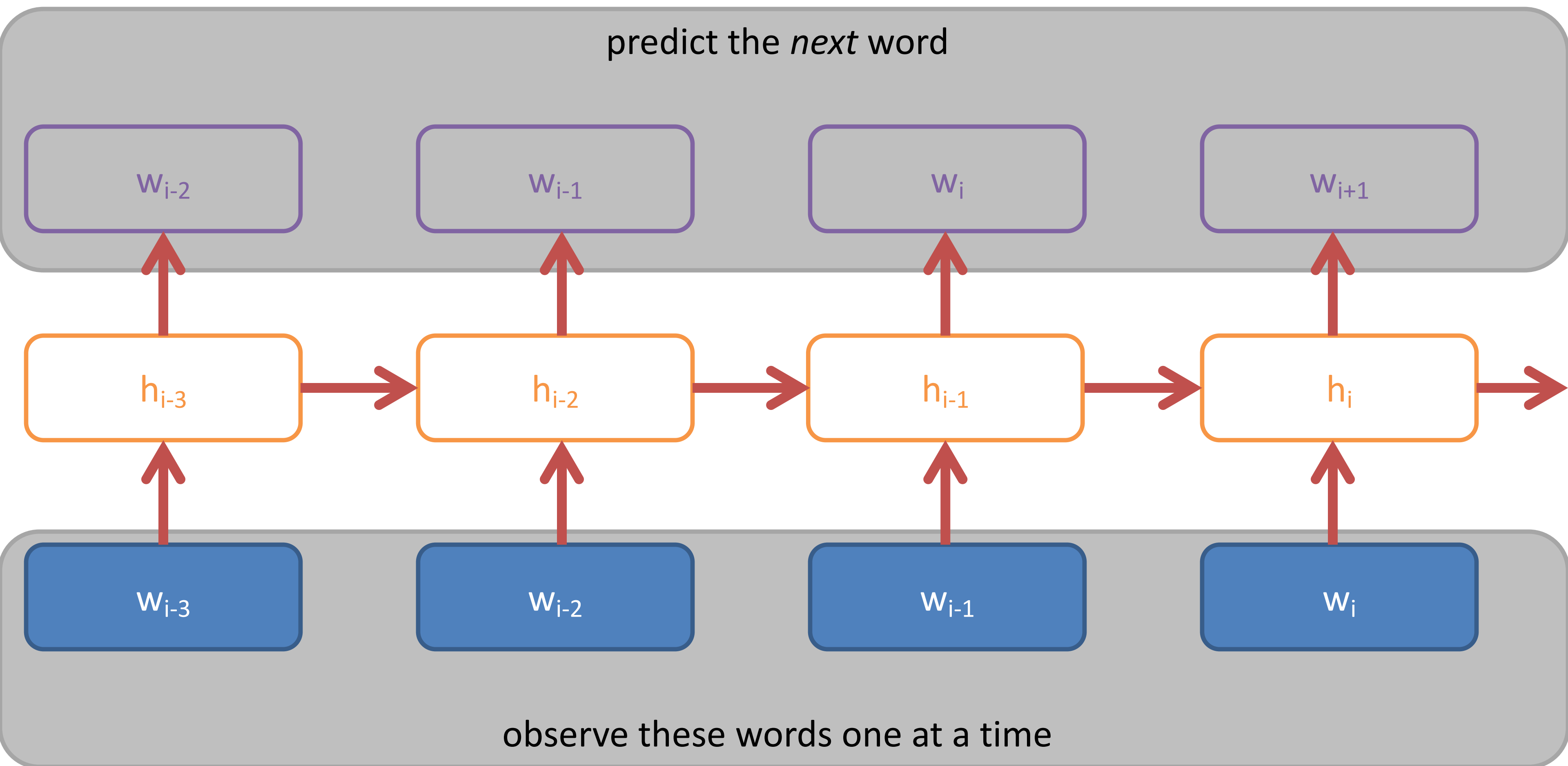
A Classic View of Recurrent Neural Language Modeling



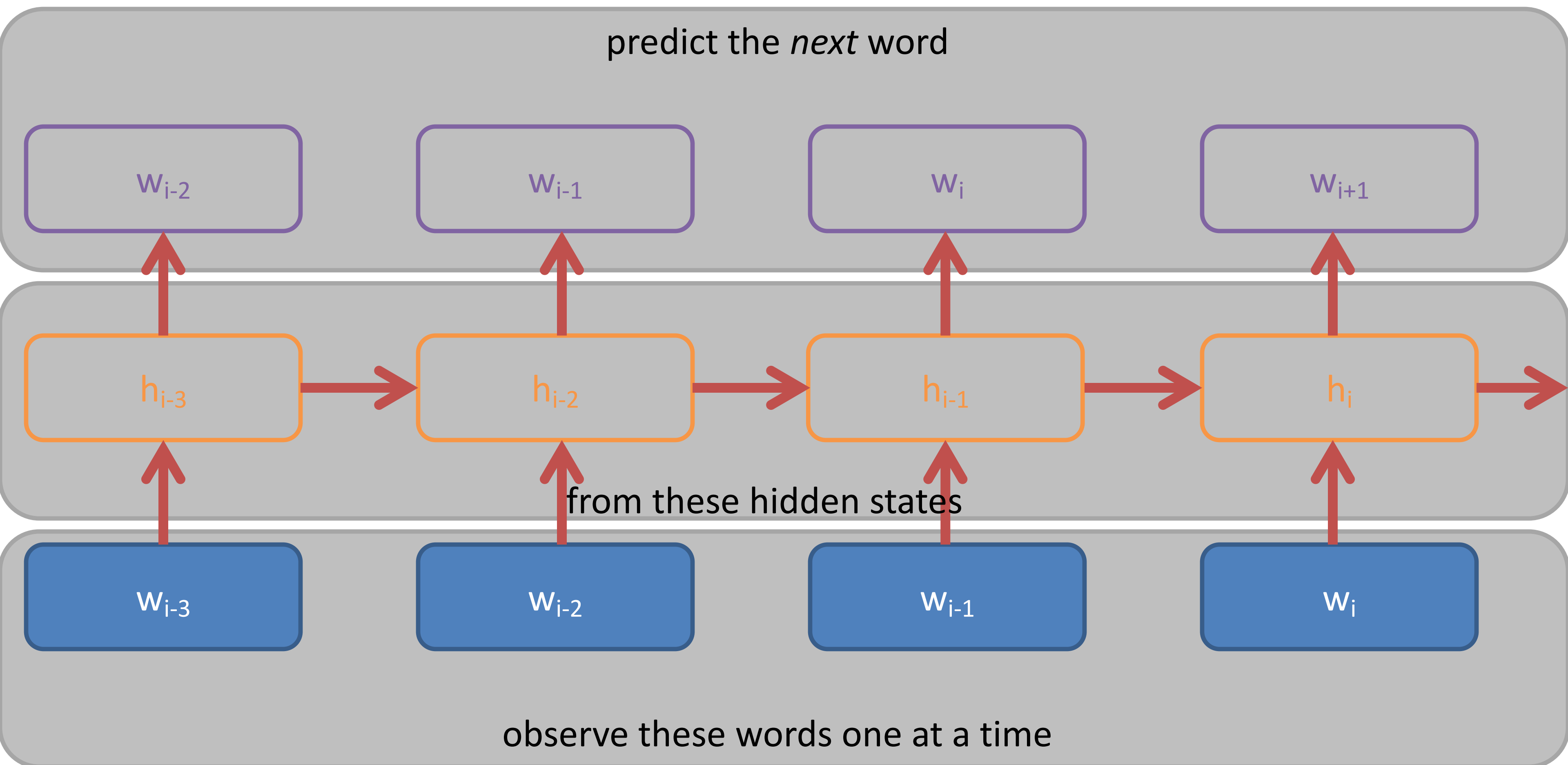
A Classic View of Recurrent Neural Language Modeling



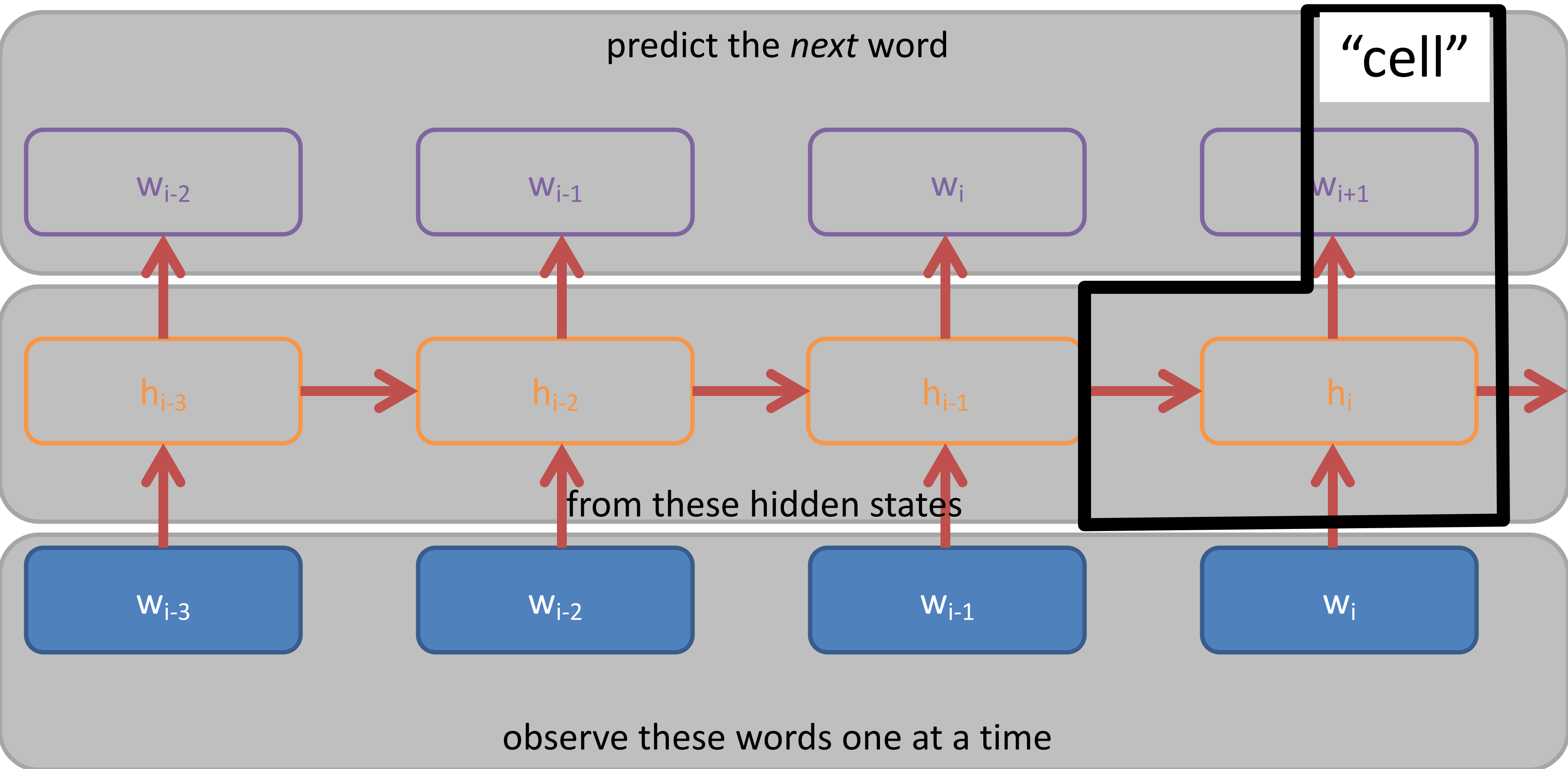
A Classic View of Recurrent Neural Language Modeling



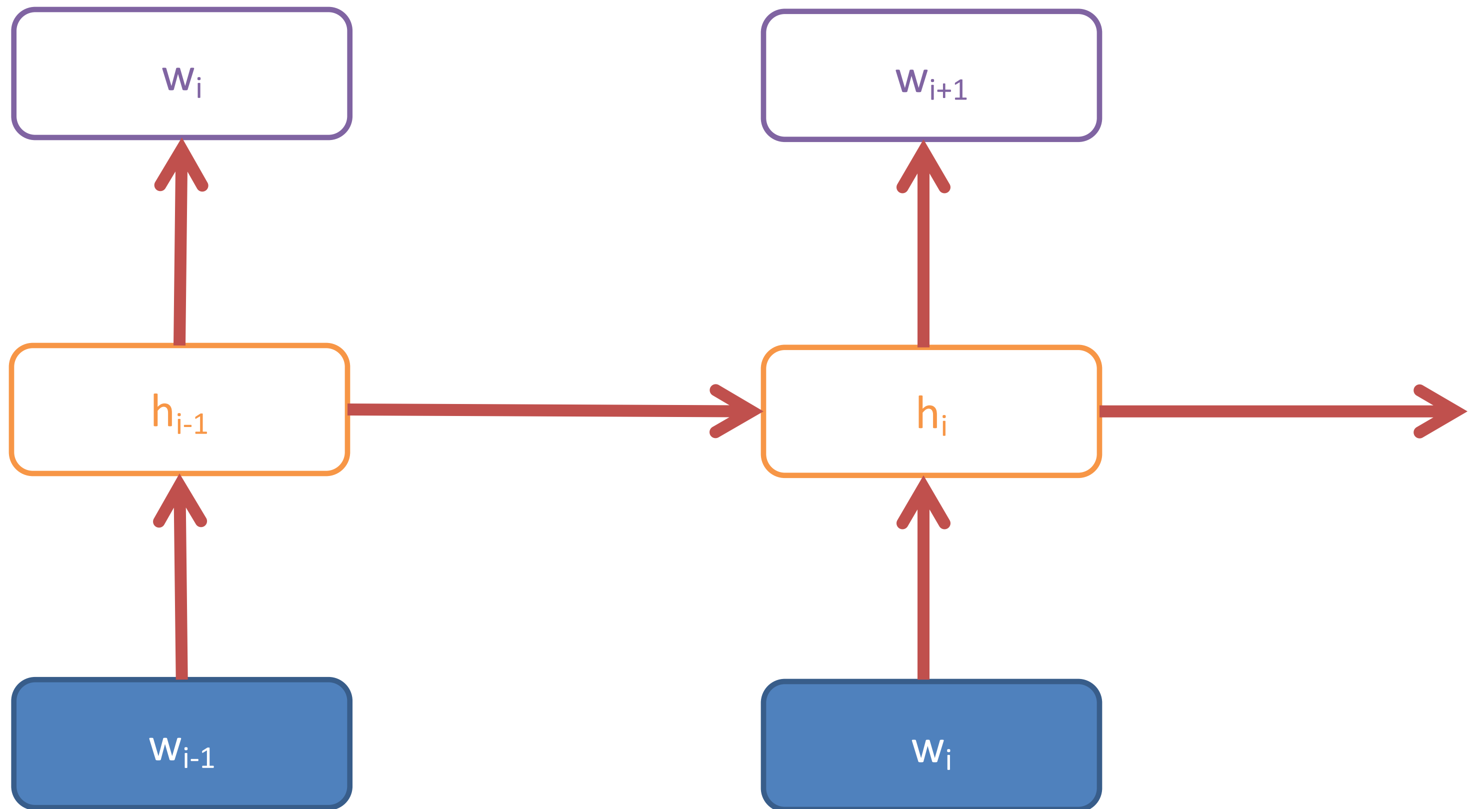
A Classic View of Recurrent Neural Language Modeling



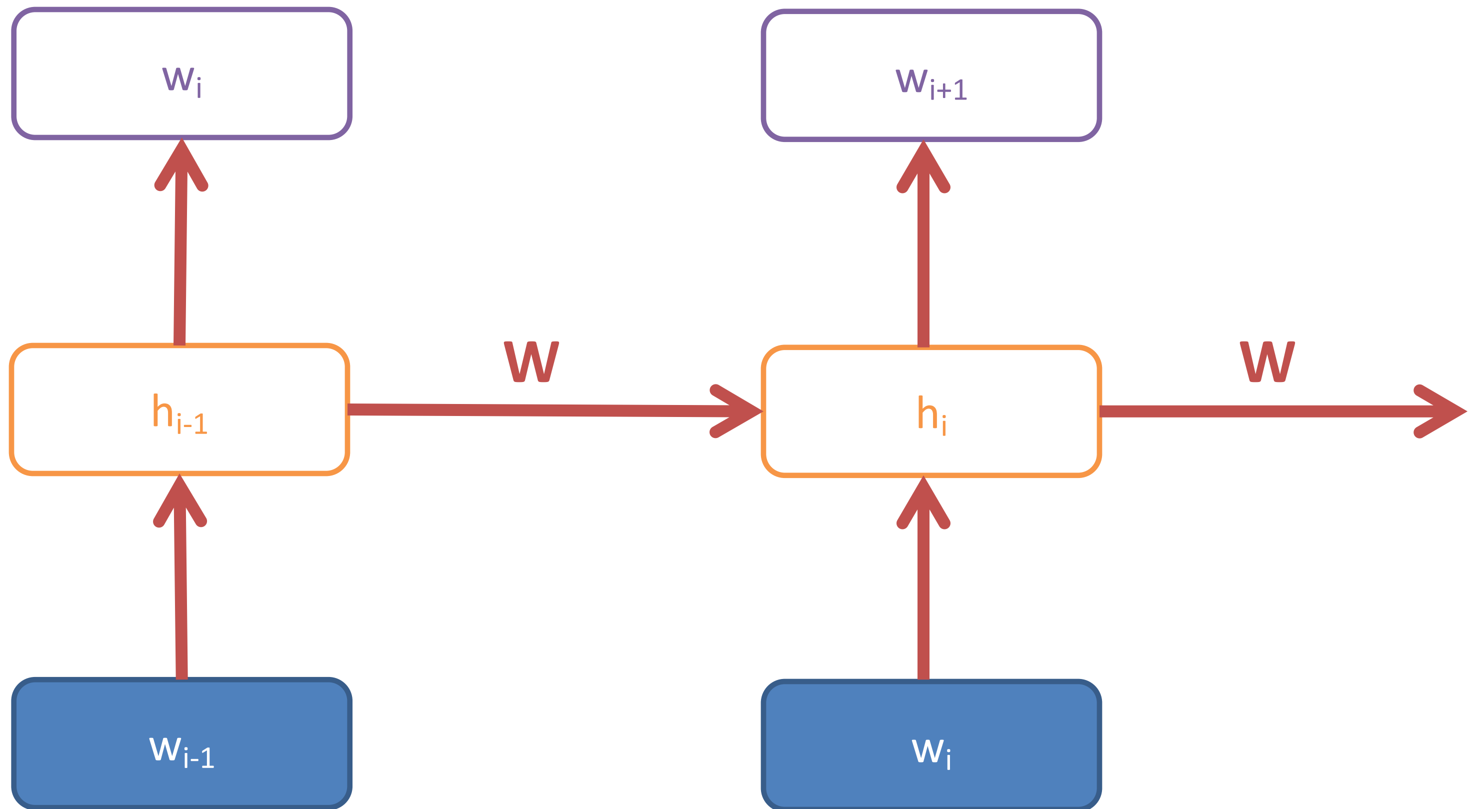
A Classic View of Recurrent Neural Language Modeling



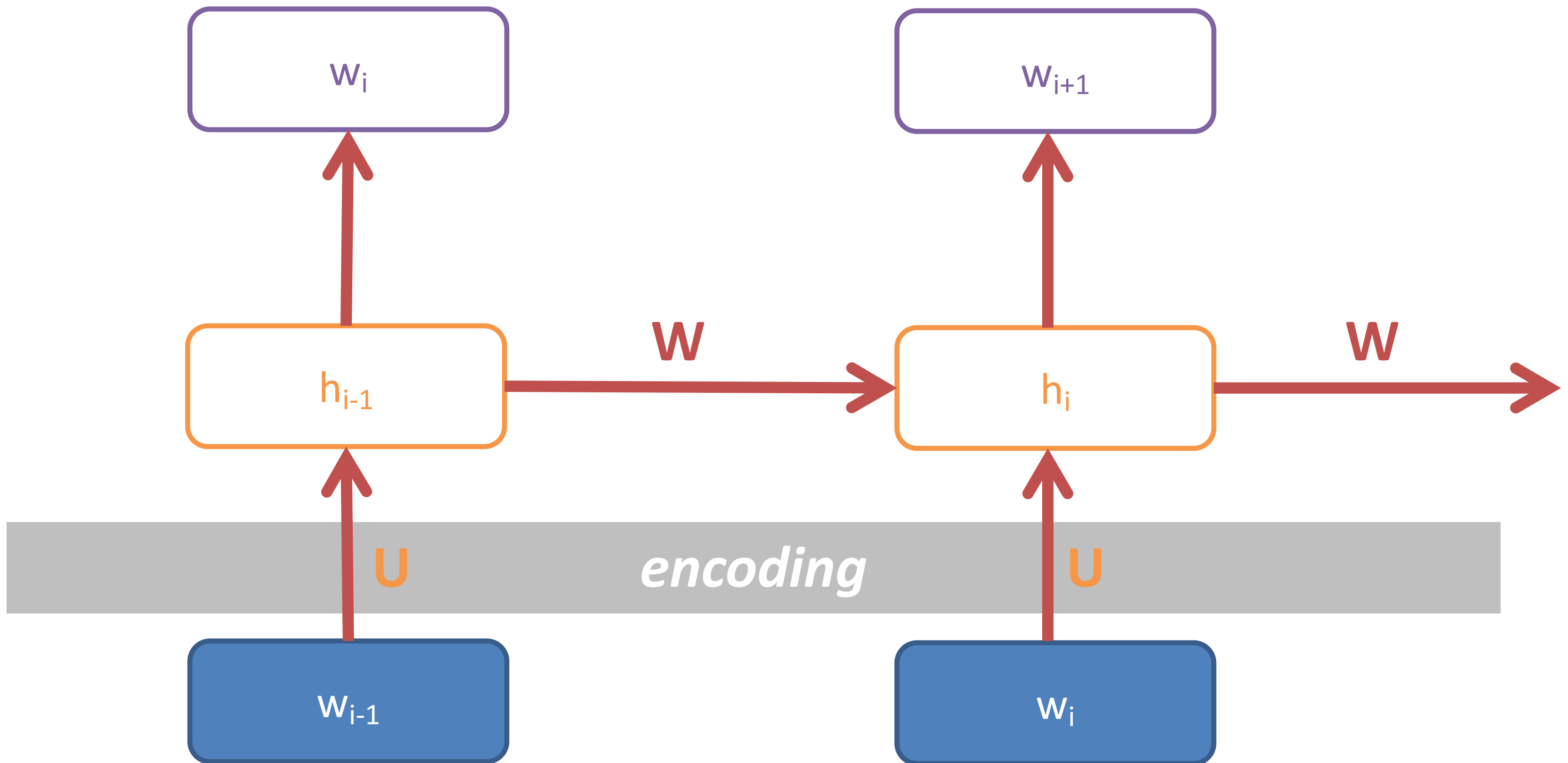
A Recurrent Neural Network Cell



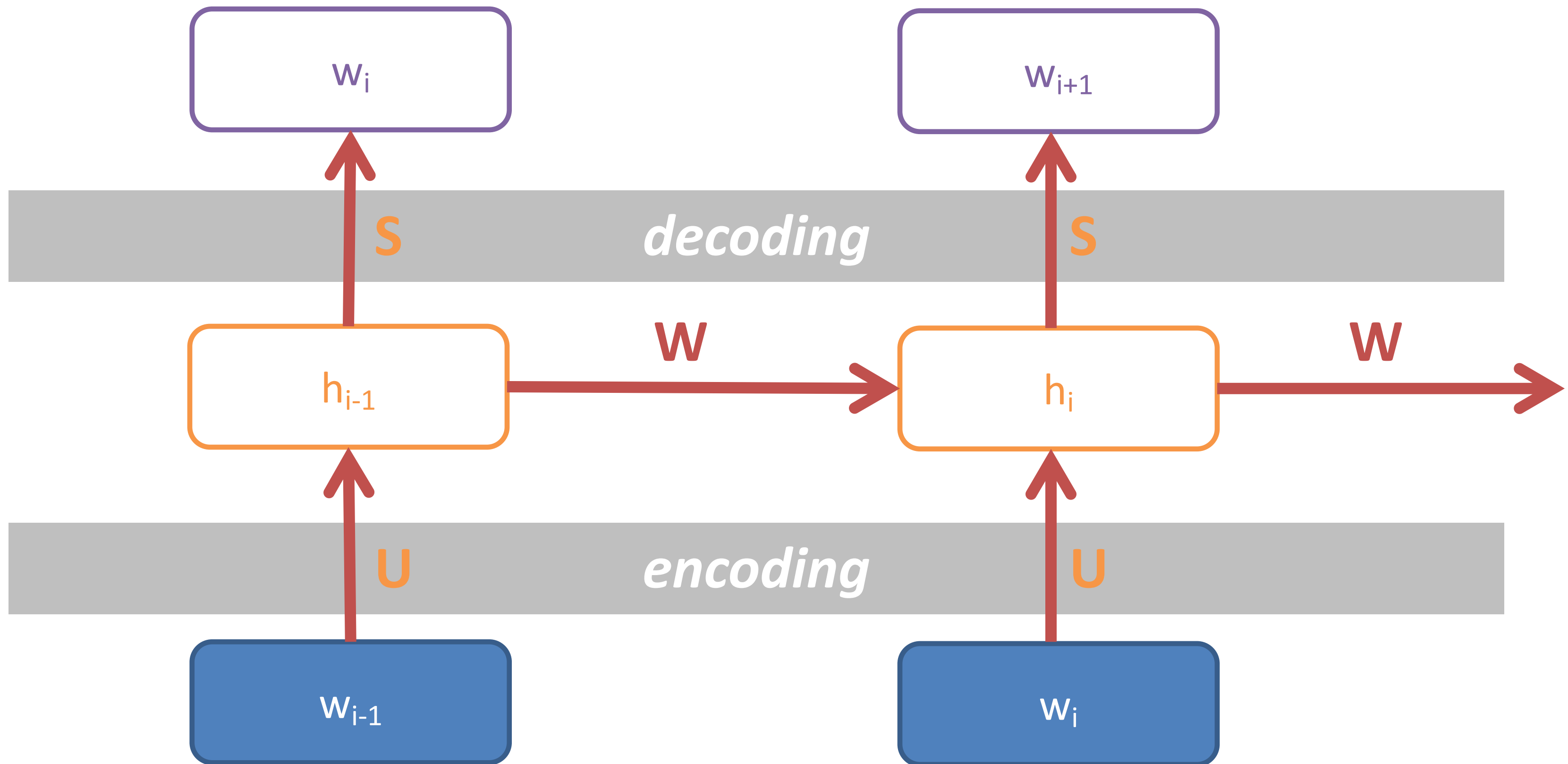
A Recurrent Neural Network Cell



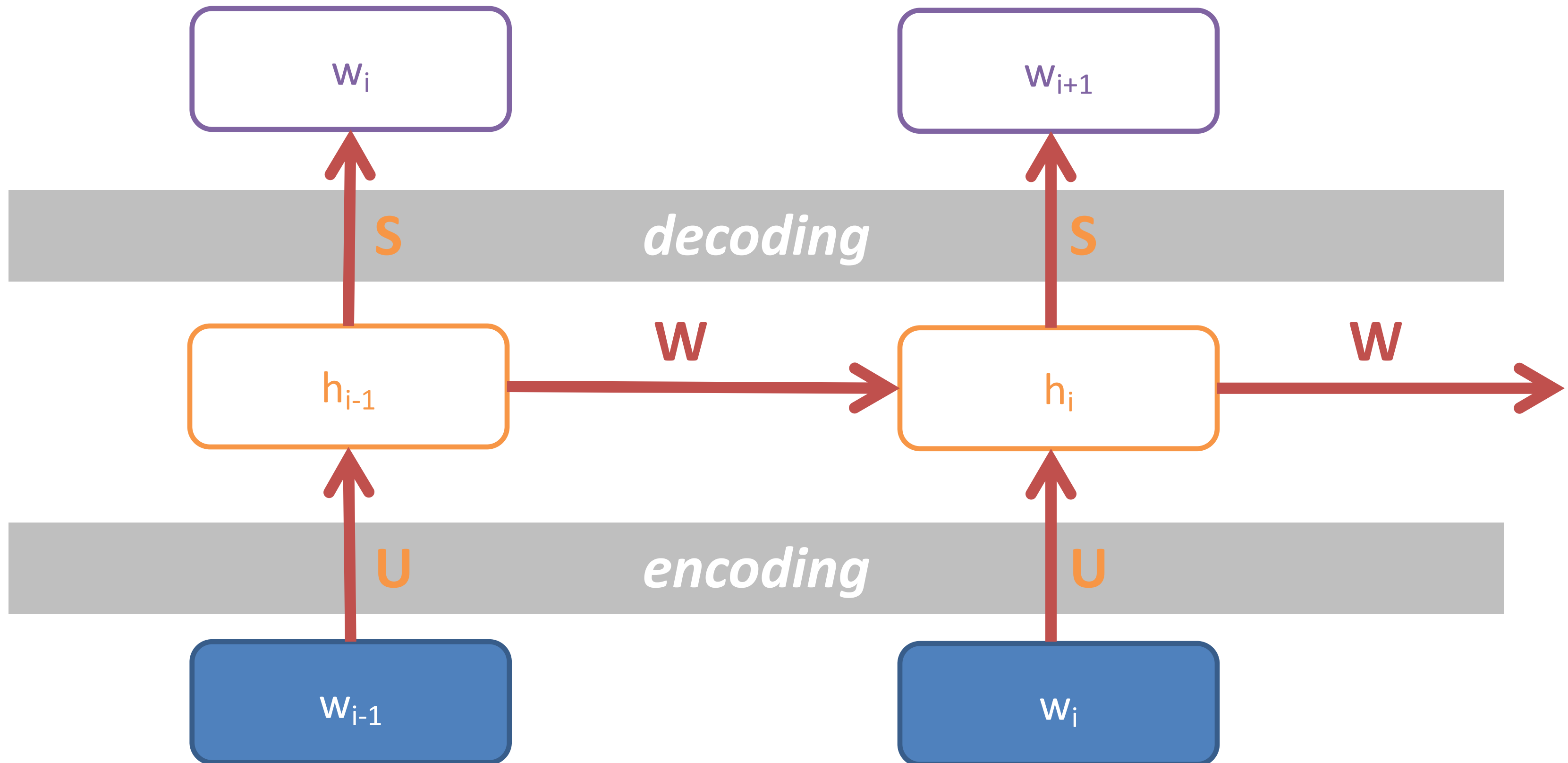
A Recurrent Neural Network Cell



A Recurrent Neural Network Cell

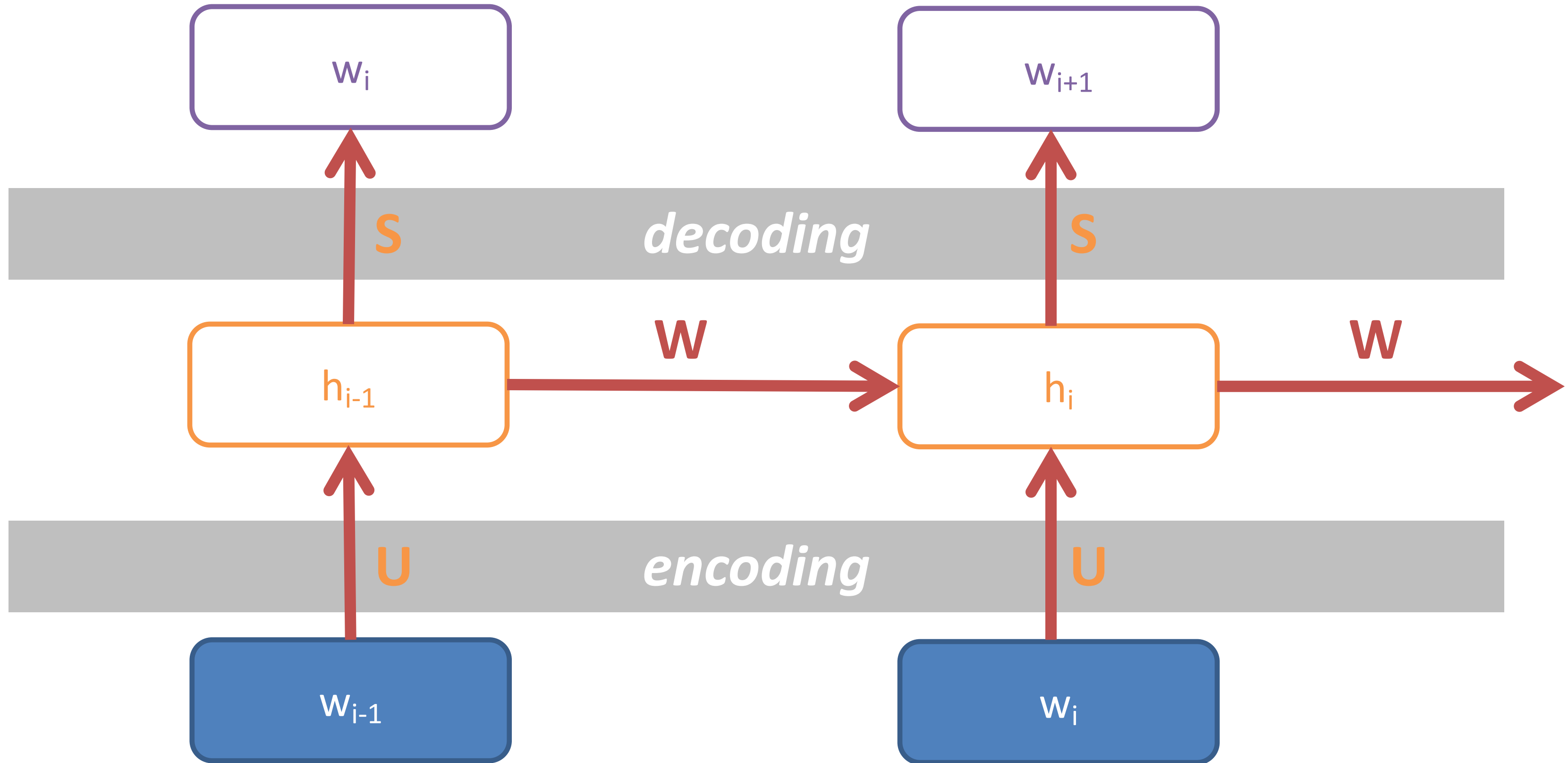


A *Simple* Recurrent Neural Network Cell



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

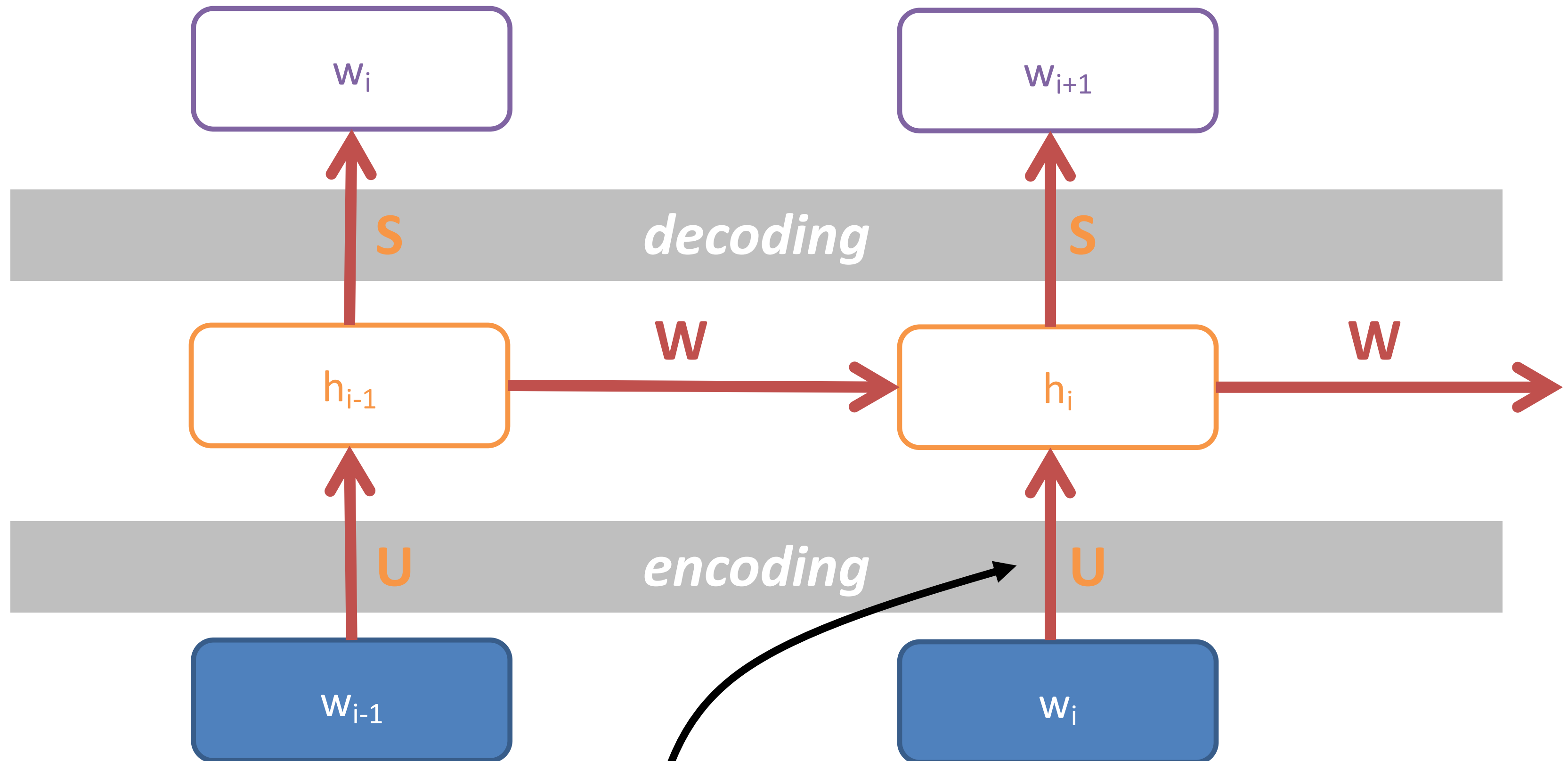
A *Simple* Recurrent Neural Network Cell



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

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

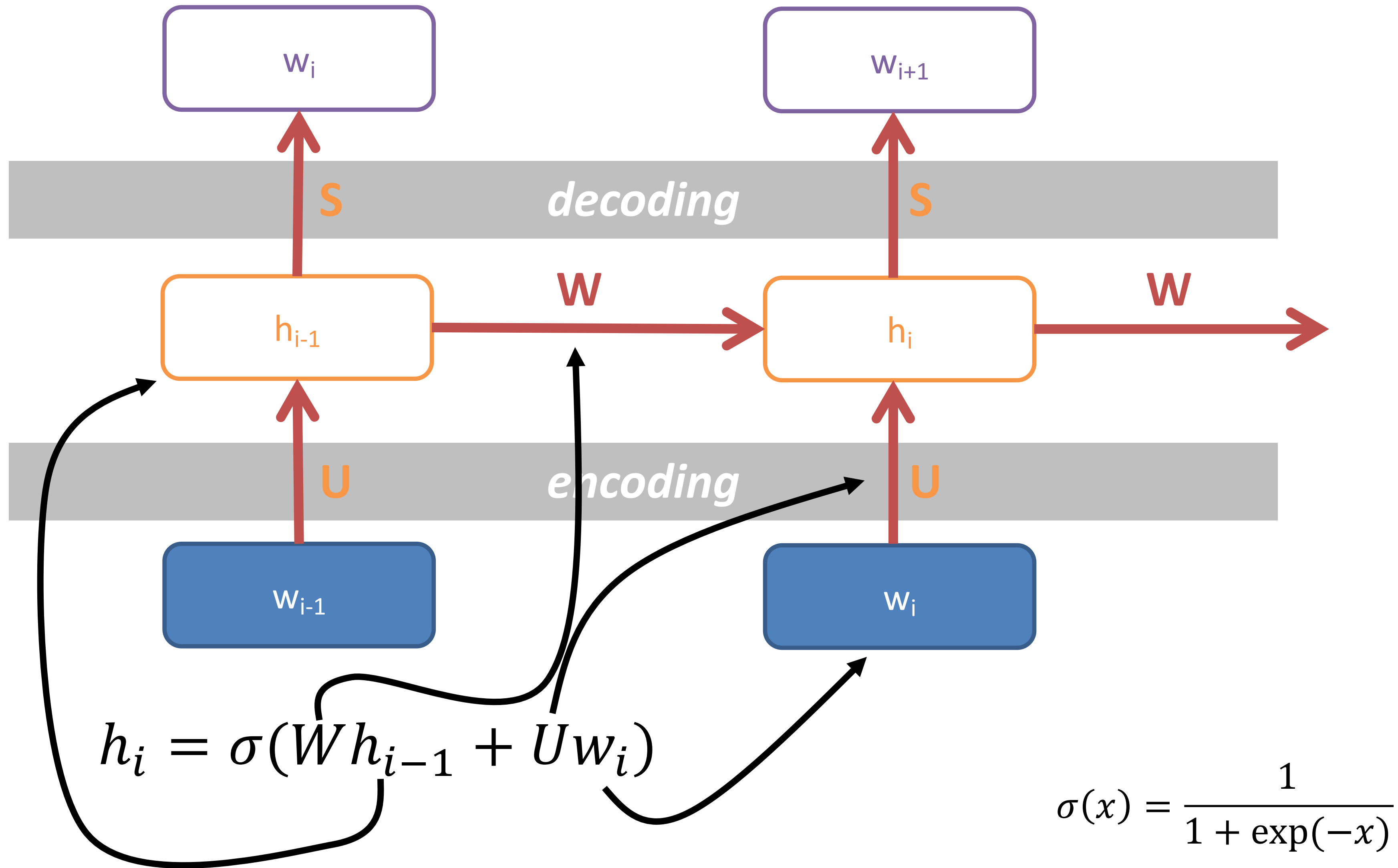
A *Simple* Recurrent Neural Network Cell



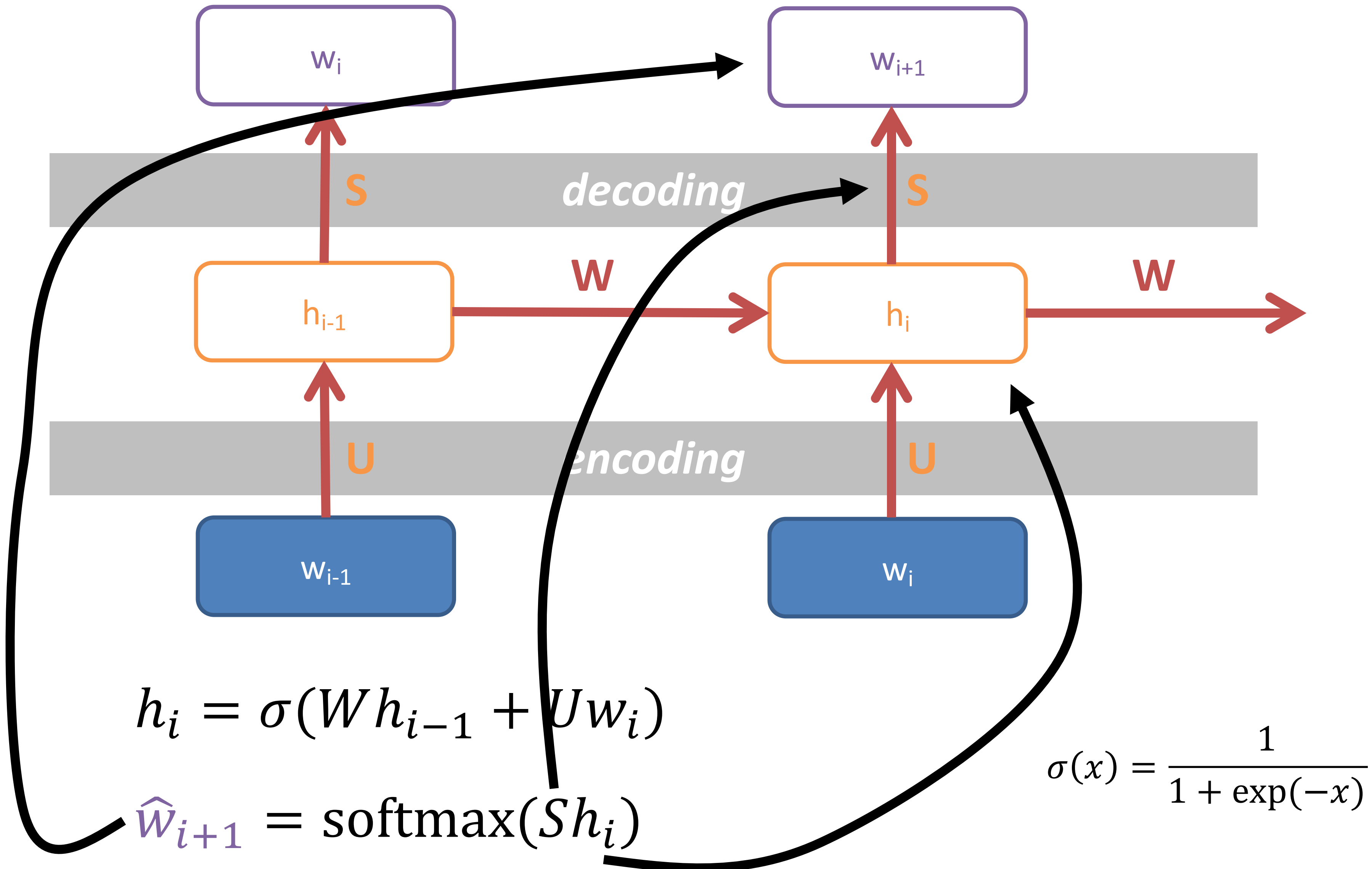
$$h_i = \sigma(W h_{i-1} + U w_i)$$

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

A *Simple* Recurrent Neural Network Cell



A *Simple* Recurrent Neural Network Cell

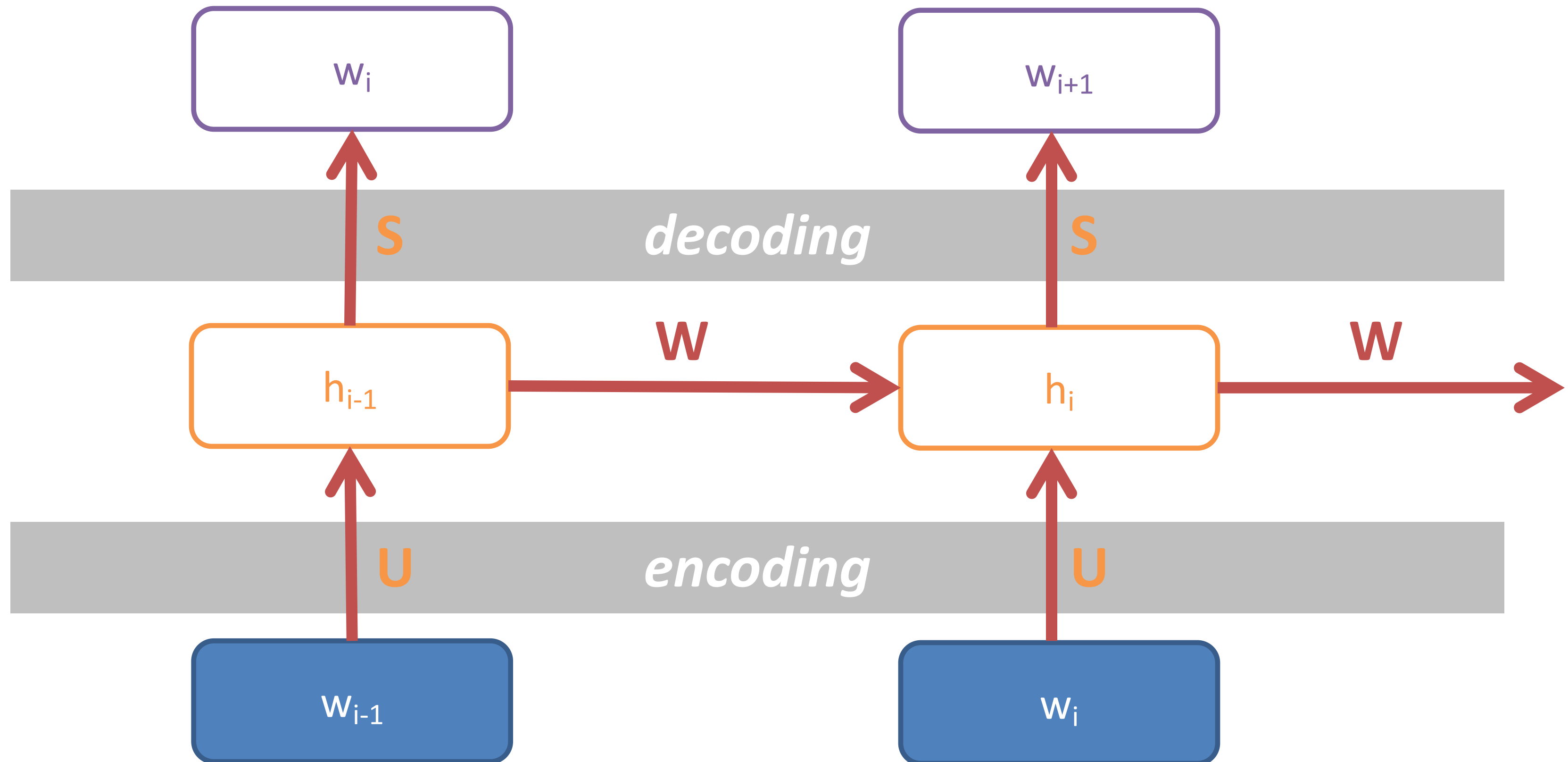


$$h_i = \sigma(W h_{i-1} + U w_i)$$

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

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

A *Simple* Recurrent Neural Network Cell

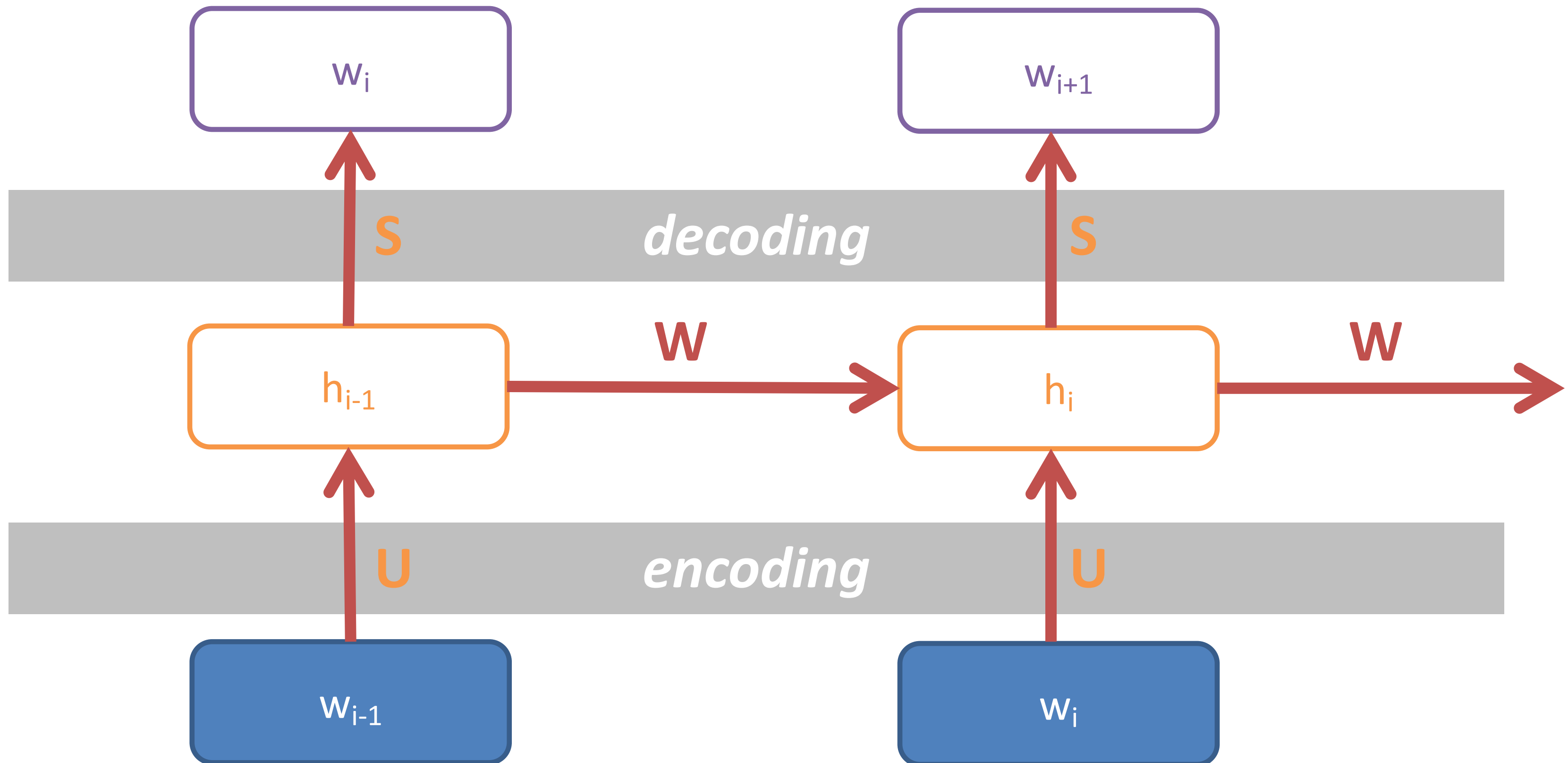


must learn *matrices* U, S, W

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

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

A *Simple* Recurrent Neural Network Cell



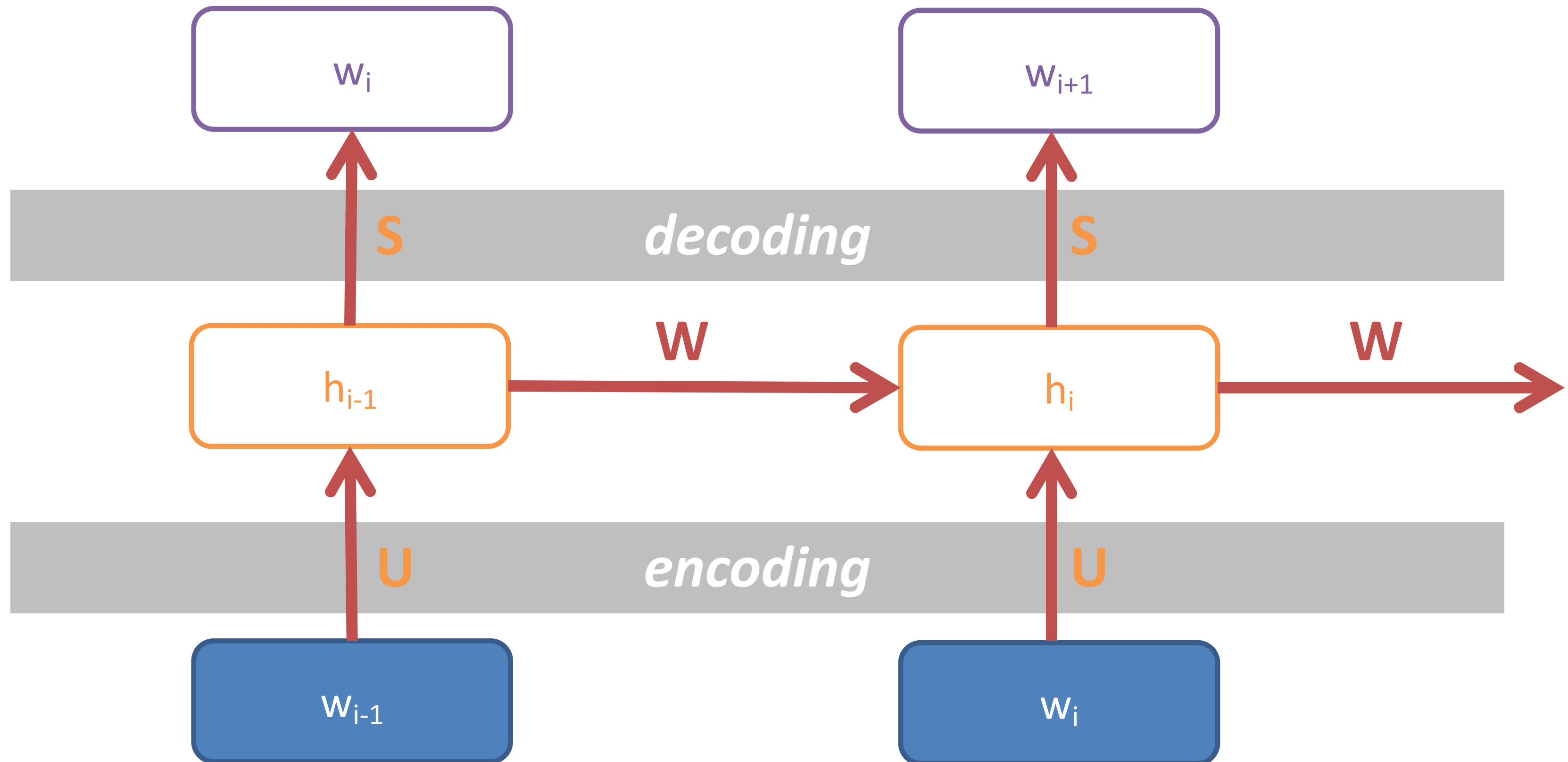
must learn *matrices* U, S, W

suggested solution: gradient descent on prediction ability

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

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

A *Simple* Recurrent Neural Network Cell



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

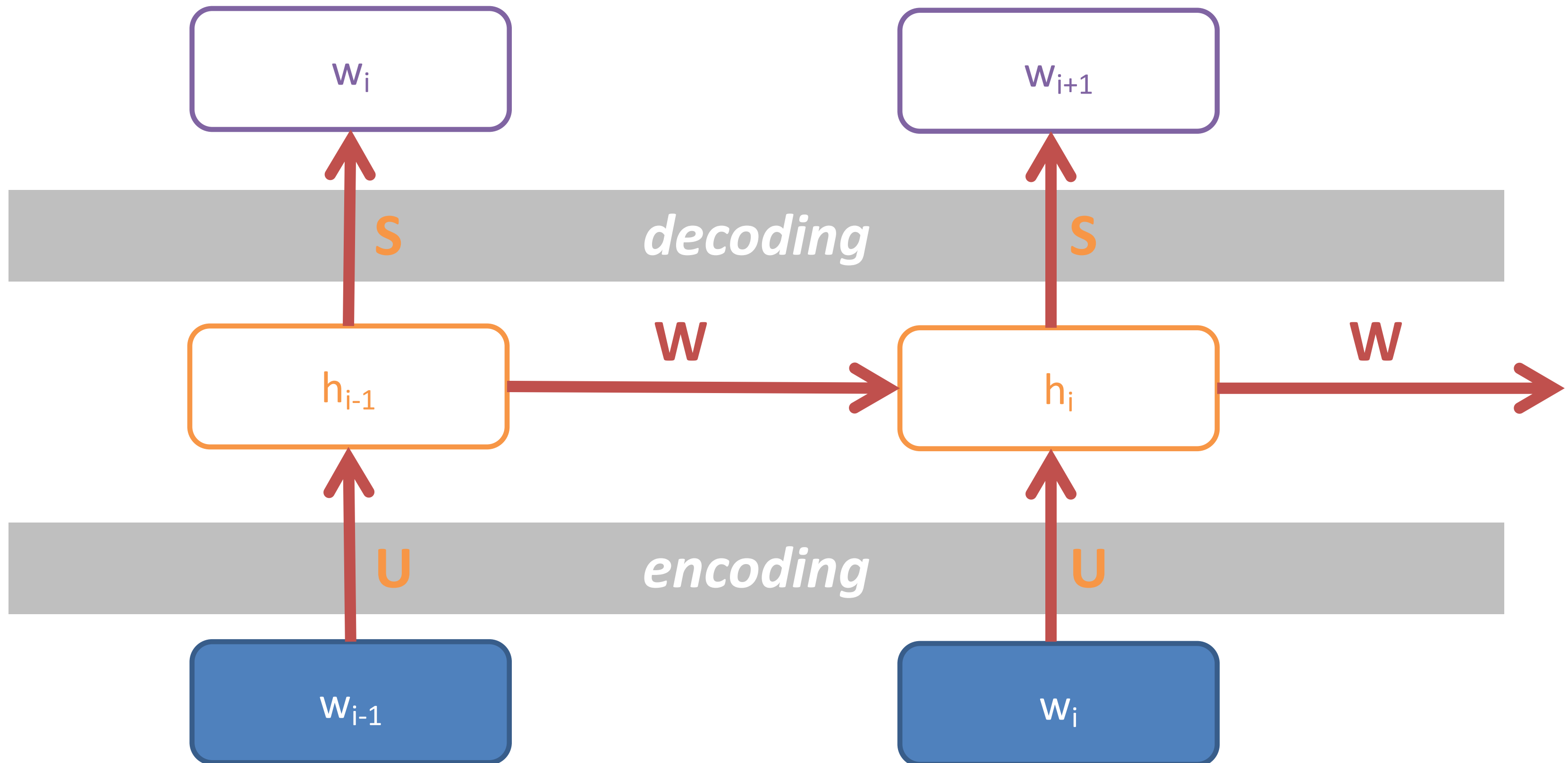
$$\hat{w}_{i+1} = \text{softmax}(S h_i)$$

must learn *matrices* U, S, W

suggested solution: gradient descent on prediction ability

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

A *Simple* Recurrent Neural Network Cell



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

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

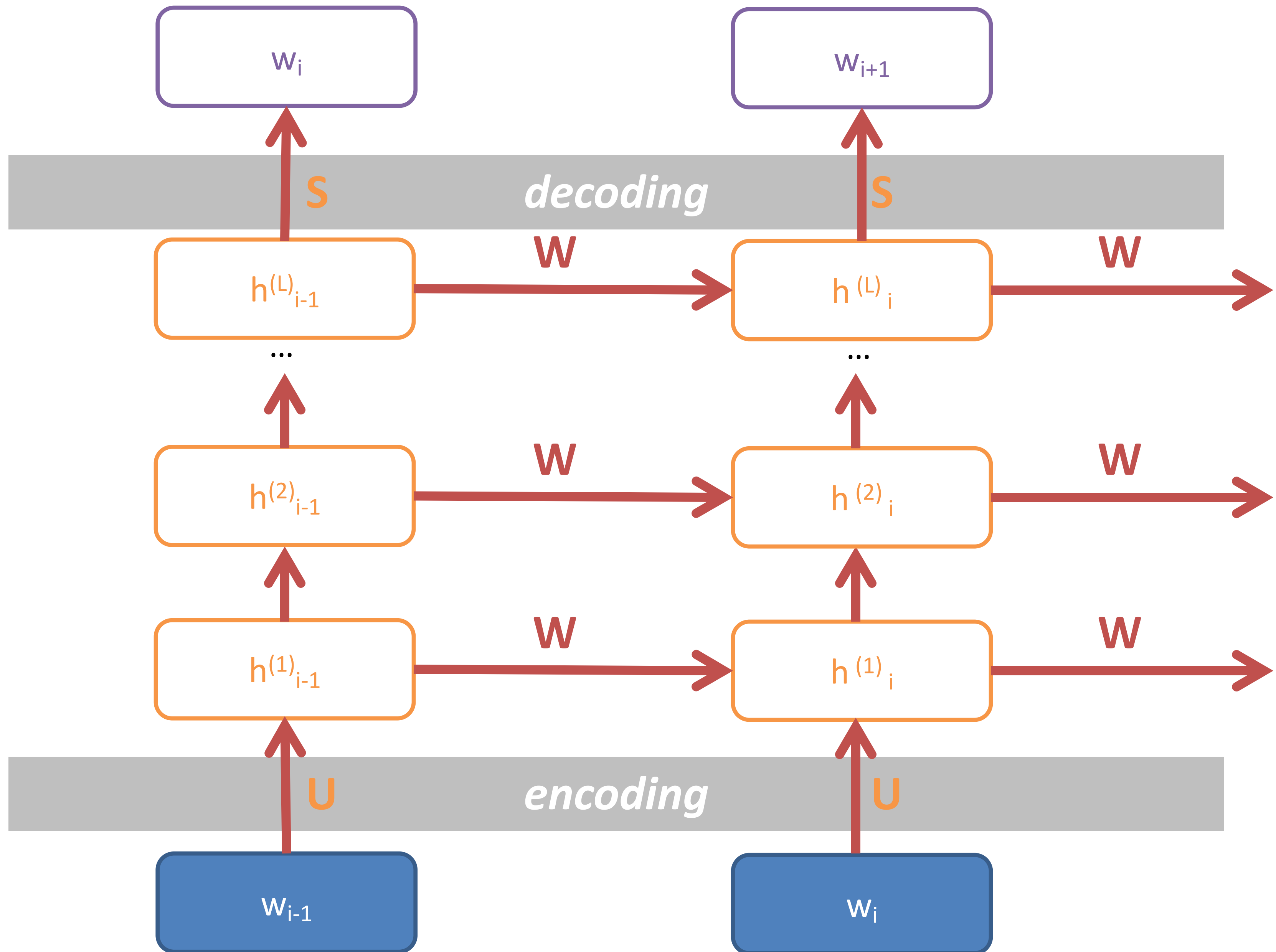
must learn *matrices* U, S, W

suggested solution: gradient descent on prediction ability

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

good news for you: many toolkits do this automatically

A Multi-Layer *Simple* Recurrent Neural Network Cell



How do you learn an RNN?

- As with other approaches: Compute the loss and perform gradient descent
- Loss: Cross-entropy, computed per output word
  - Just as with prior LM approaches!

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

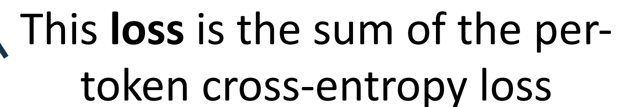
Until converged:

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).



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

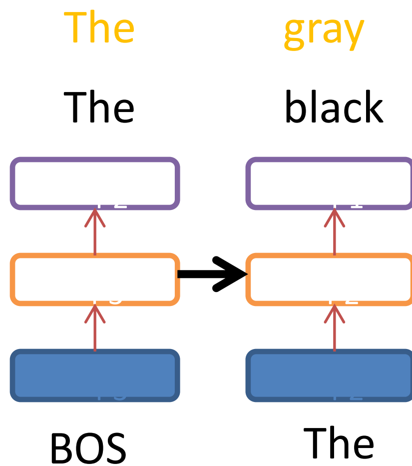
Recurrent NN Loss

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

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

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

The gray

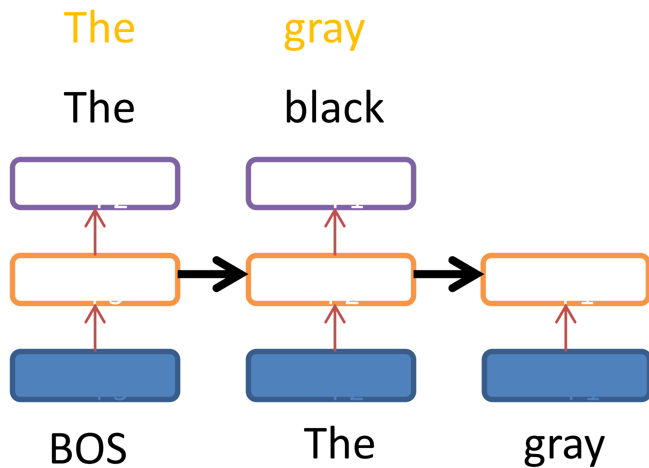


Recurrent NN Loss

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

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

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



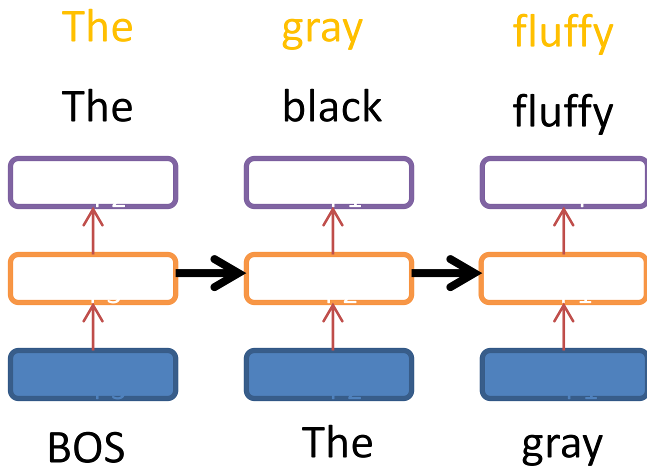
Recurrent NN Loss

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

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

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

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



Recurrent NN Loss

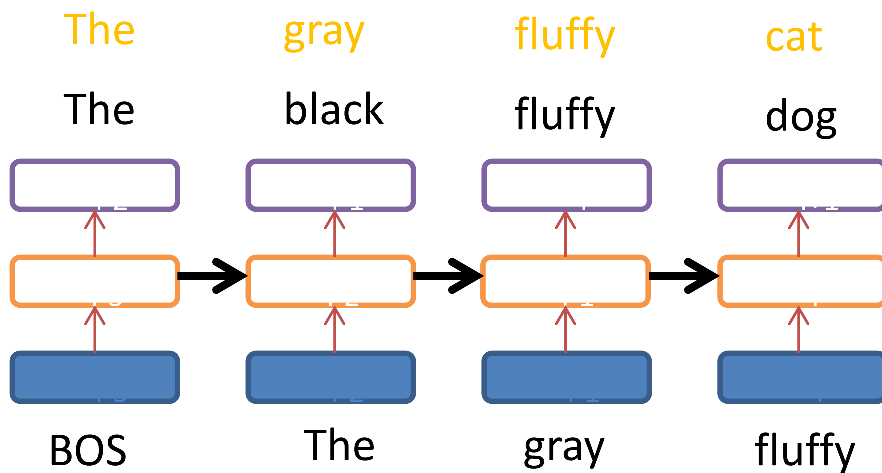
$\log .2 + \log .12 + \log .2 + \log .19$

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

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

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

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



Recurrent NN Loss

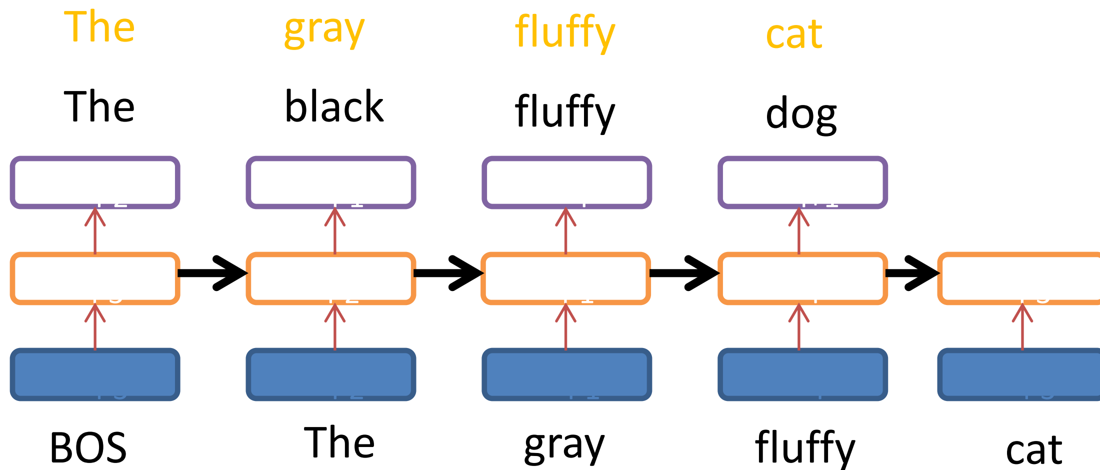
$\log .2 + \log .12 + \log .2 + \log .19$

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

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

word	prob.
fluffy	.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

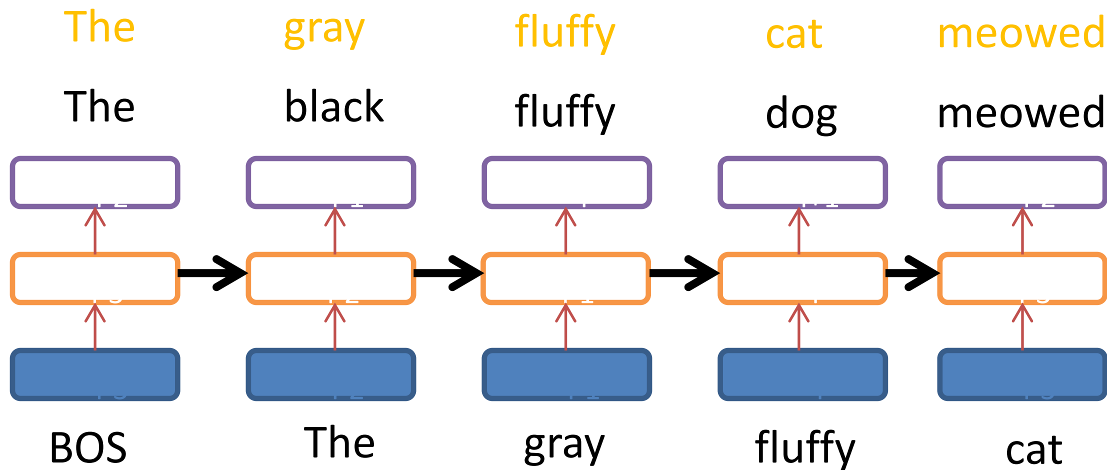
word	prob.
The	.2
gray	.01
blue	.001
fluffy	.0005
wet	.0005
…	…

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

word	prob.
fluffy	.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

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

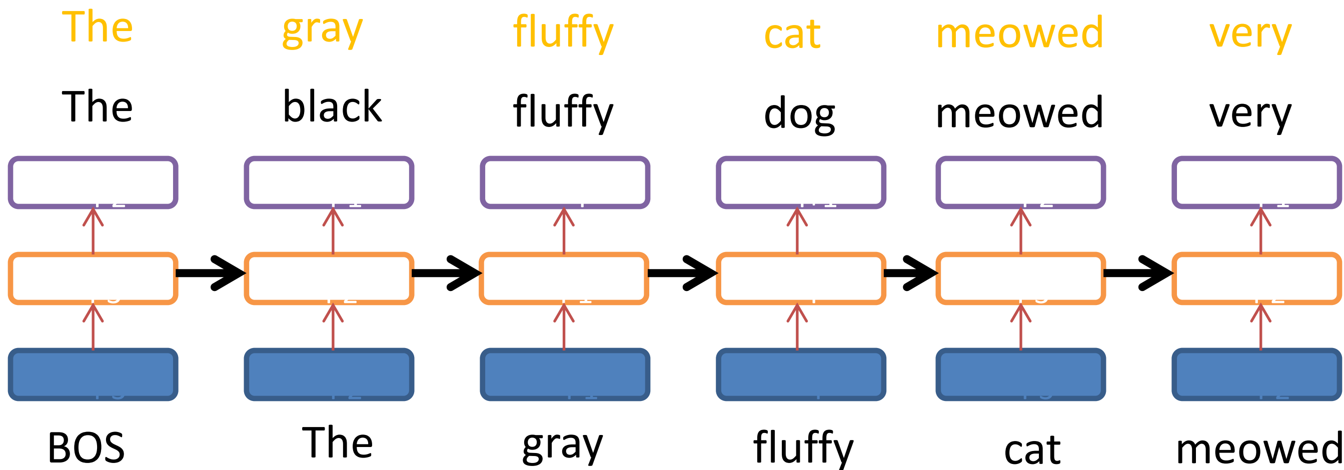
word	prob.
The	.2
gray	.01
blue	.001
fluffy	.0005
wet	.0005
…	…

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

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

word	prob.	word	prob
dog	.2	meowed	.3
cat	.19	purred	.2
blue	.001	hissed	.1
fluffy	.0005	fluffy	.001
wet	.0005	wet	.001
…	…	…	…

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



Recurrent NN Loss

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

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

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

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

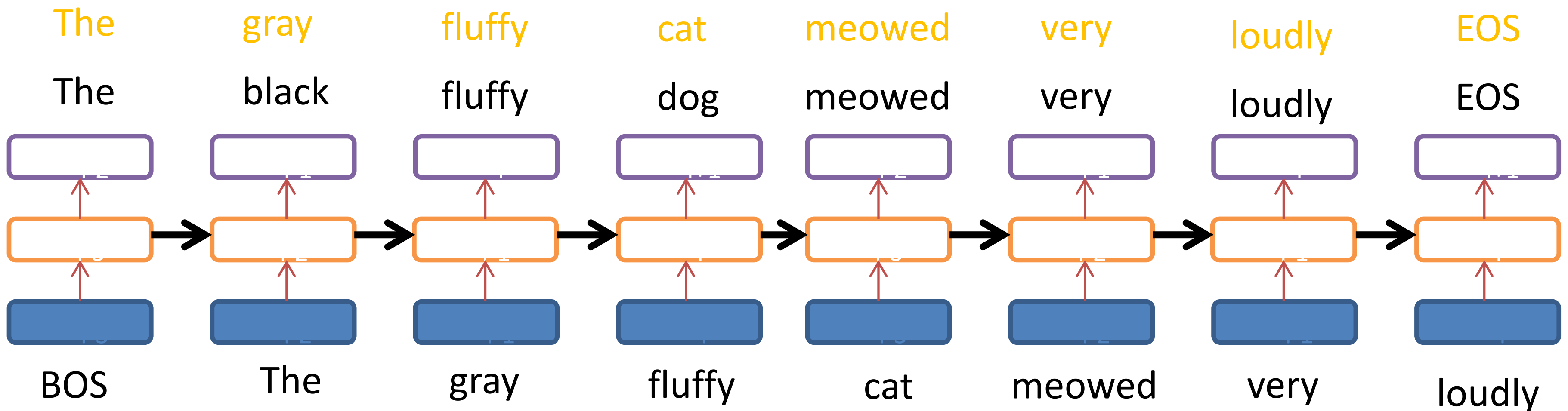
word	prob.
dog	.2
cat	.19
blue	.001
fluffy	.0005
wet	.0005
…	…

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

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

word	prob
loudly	.2
softly	.01
quiet	.001
fluffy	.001
wet	.001
…	…

word	prob.
EOS	.3
and	.1
blue	.001
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.
The	.2
gray	.01
blue	.001
fluffy	.0005
wet	.0005
…	…

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

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

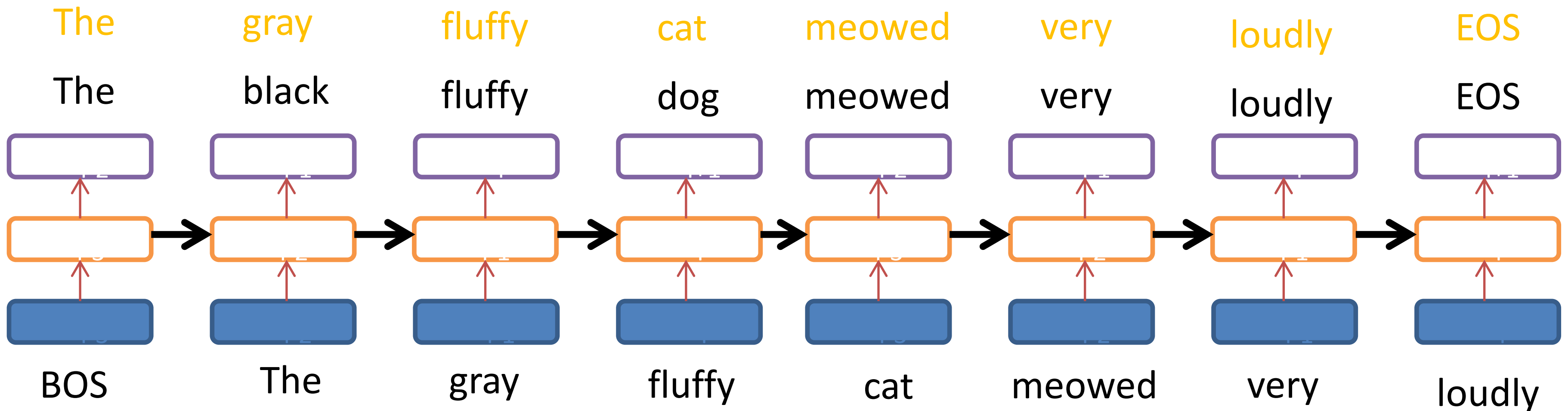
word	prob.
dog	.2
cat	.19
blue	.001
fluffy	.0005
wet	.0005
…	…

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

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

word	prob
loudly	.2
softly	.01
quiet	.001
fluffy	.001
wet	.001
…	…

word	prob.
EOS	.3
and	.1
blue	.001
fluffy	.0005
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, and exp)

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	fluffy	.2	dog	.2	meowed	.3	very	.2	loudly	.2	EOS	.3
gray	.01	gray	.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	wet	.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

Outline

Core Problem

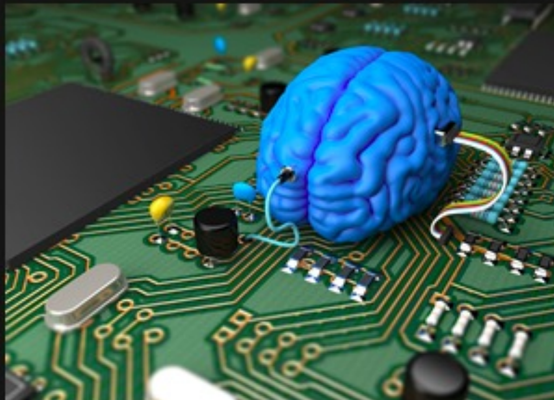
Basic cell definition

Example in PyTorch

~~Deep Learning~~

Natural Language Processing

What society thinks I do



What my friends think I do

What other computer scientists think I do

What mathematicians think I do

What I think I do

What I actually do

Pick Your Toolkit

PyTorch
Deeplearning4j
TensorFlow
DyNet
Caffe
Keras
MxNet
Gluon
CNTK
…

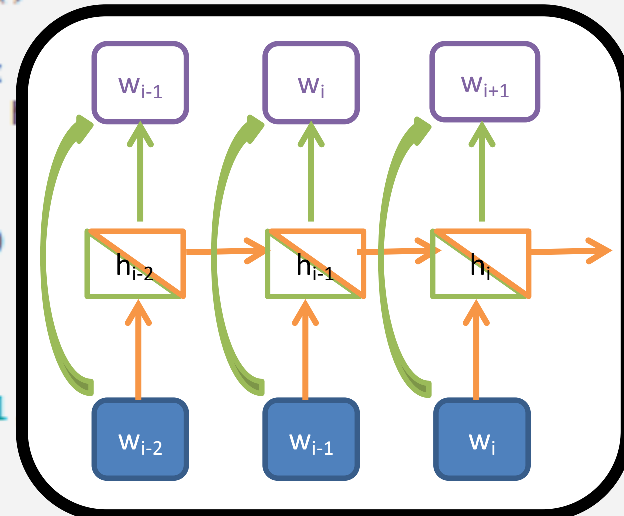
Comparisons:
https://en.wikipedia.org/wiki/Comparison_of_deep_learning_software
https://deeplearning4j.org/compare-dl4j-tensorflow-pytorch
https://github.com/zer0n/deepframeworks (older---2015)

Defining A Simple RNN in Python

(Modified Very Slightly)

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

```
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 = self.i2h(combined)
        output = self.i2o(combined)
        output = self.softmax(output)
        return output, hidden

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

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

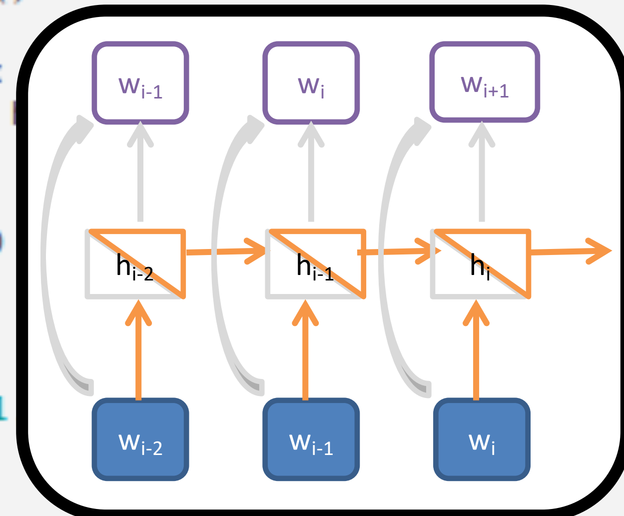


Defining A Simple RNN in Python

(Modified Very Slightly)

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

```
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 = self.i2h(combined)
        output = self.i2o(combined)
        output = self.softmax(output)
        return output, hidden

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

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

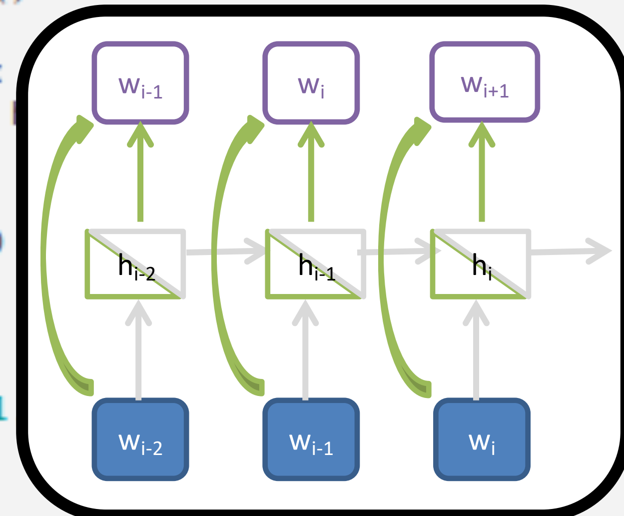


Defining A Simple RNN in Python

(Modified Very Slightly)

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

```
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 = self.i2h(combined)
        output = self.i2o(combined)
        output = self.softmax(output)
        return output, hidden

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

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



Defining A Simple RNN in Python

(Modified Very Slightly)

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

```
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(
        combined
        hidden =
        output =
        output =
        return o

    def initHidd
        return V

n_hidden = 128
rnn = RNN(n_lett
```

https://pytorch.org/docs/stable/nn.html

0.4.1
version selector ▼

Search docs

NOTES
- Autograd mechanics
- Broadcasting semantics
- CUDA semantics
- Extending PyTorch
- Frequently Asked Questions
- Multiprocessing best practices
- Serialization semantics
- Windows FAQ

PACKAGE REFERENCE
- torch
- torch.Tensor

class **torch.nn.LogSoftmax**(*dim=None*) [source]

Applies the *Log(Softmax(x))* function to an n-dimensional input Tensor. The LogSoftmax formulation can be simplified as

$$\text{LogSoftmax}(x_i) = \log\left(\frac{\exp(x_i)}{\sum_j \exp(x_j)}\right)$$

Shape:
- Input: any shape
- Output: same as input

Parameters: **dim** (*int*) – A dimension along which Softmax will be computed (so every slice along dim will sum to 1).

Returns: a Tensor of the same dimension and shape as the input with values in the range [-inf, 0)

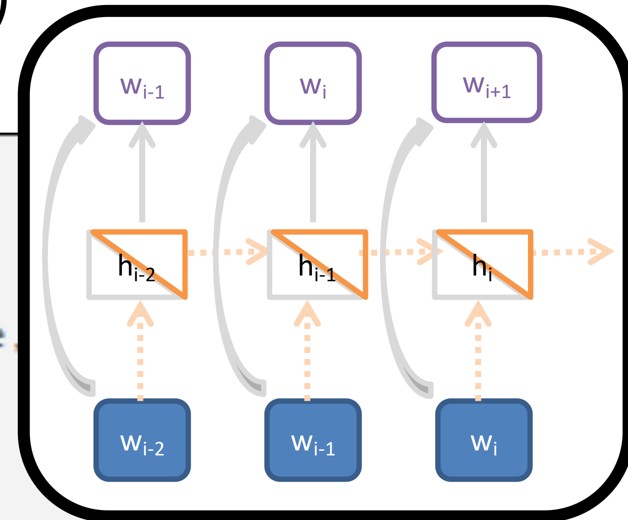
Examples:

```
>>> m = nn.LogSoftmax()
>>> input = torch.randn(2, 3)
>>> output = m(input)
```

Defining A Simple RNN in Python

(Modified Very Slightly)

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



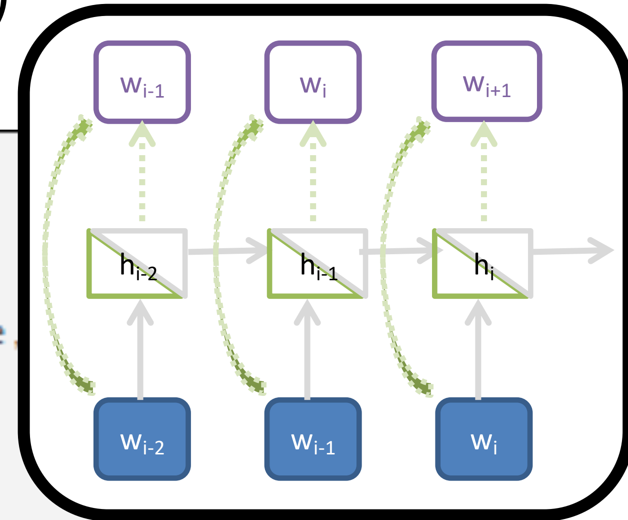

```python
import torch.nn as nn
from torch.autograd import Variable

class RNN(nn.Module):
    def __init__(self, input_size, hidden_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_categories)
```

encode

Defining A Simple RNN in Python

(Modified Very Slightly)

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




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

class RNN(nn.Module):
    def __init__(self, input_size, hidden_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_categories)
```

(we'll talk about this)

decode

Training A Simple RNN in Python

(Modified Very Slightly)

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

```
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

(Modified Very Slightly)

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

Negative log-likelihood

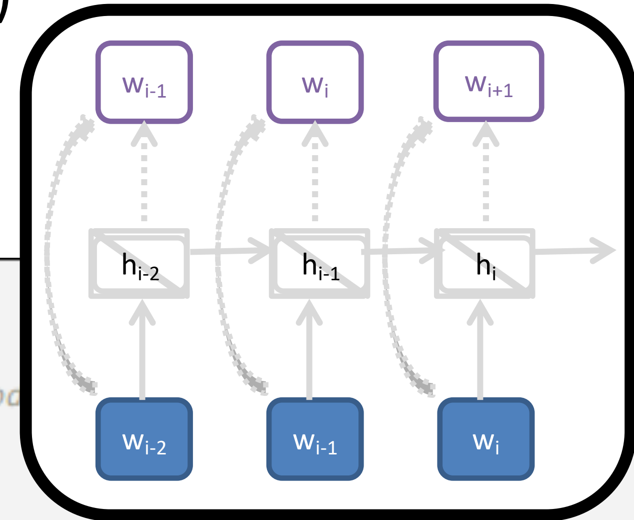
(we'll talk about this)

```
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

(Modified Very Slightly)

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 explod

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

(Modified Very Slightly)

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

Training A Simple RNN in Python

(Modified Very Slightly)

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

Training A Simple RNN in Python

(Modified Very Slightly)

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

Suggested Implementation Changes

```python
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_ca
```

current Pytorch refers to this a "cell"

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

nn.CrossEntropyLoss()

```python
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]
```

Comparing this to `train` in A2

```
criterion = nn.NLLLoss()

learning_rate = 0.005 # If you set this too high, it

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, mult
    for p in rnn.parameters():
        p.data.add_(-learning_rate, p.grad.data)

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

```
def train(model, data_inst, data_labels, weight_adjuster, loss_fn,
          batch_size=100, num_epochs=5, epoch_callback=None):
  for epoch in range(num_epochs):
    print("Epoch %d" % epoch)
    batch_start = 0
    batch_end = min(batch_size, len(data_inst))
    num_batches = int(numpy.ceil(len(data_inst) / batch_size))
    for batch_idx, batch in enumerate(range(num_batches)):
      batch = data_inst[batch_start:batch_end]
      weight_adjuster.zero_grad()
      logits = model(batch)

      labels = data_labels[batch_start:batch_end]

      loss = loss_fn(input=logits, target=labels)
      print("Epoch %d, Batch %d: %d --> %d, Batch loss %f" % (epoch, batch_idx,
                                                              batch_start,
                                                              batch_end,
                                                              loss.item()))
      loss.backward()
      weight_adjuster.step()

      batch_start = batch_end
      batch_end = batch_end+batch_size
      batch_end = min(batch_end, len(data_inst))

    if epoch_callback is not None:
      epoch_callback()
```

`batch_size` = ??
`weight_adjuster` = ??
`num_epochs` = ??
`epoch_callback` = ??

Comparing this to `train` in A2

```python
criterion = nn.NLLLoss()

learning_rate = 0.005 # If you set this too high, it

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, mult
    for p in rnn.parameters():
        p.data.add_(-learning_rate, p.grad.data)

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

```python
def train(model, data_inst, data_labels, weight_adjuster, loss_fn,
          batch_size=100, num_epochs=5, epoch_callback=None):
  for epoch in range(num_epochs):
    print("Epoch %d" % epoch)
    batch_start = 0
    batch_end = min(batch_size, len(data_inst))
    num_batches = int(numpy.ceil(len(data_inst) / batch_size))
    for batch_idx, batch in enumerate(range(num_batches)):
      batch = data_inst[batch_start:batch_end]
      weight_adjuster.zero_grad()
      logits = model(batch)

      labels = data_labels[batch_start:batch_end]

      loss = loss_fn(input=logits, target=labels)
      print("Epoch %d, Batch %d: %d --> %d, Batch loss %f" % (epoch, batch_idx,
                                                              batch_start,
                                                              batch_end,
                                                              loss.item()))

      loss.backward()
      weight_adjuster.step()

      batch_start = batch_end
      batch_end = batch_end+batch_size
      batch_end = min(batch_end, len(data_inst))

    if epoch_callback is not None:
      epoch_callback()
```

`batch_size` = 1
`weight_adjuster` = optim.SGD
`num_epochs` = N/A (call the tutorial train separately)
`epoch_callback` = None

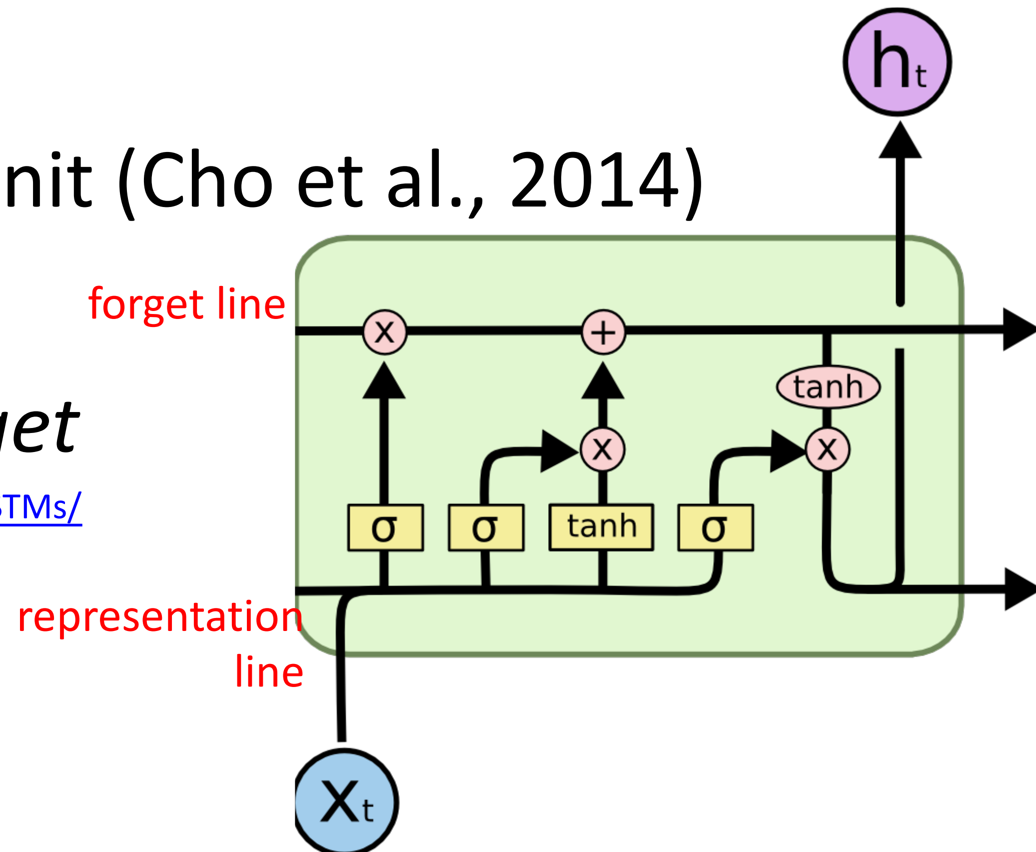
Another Solution: LSTMs/GRUs

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

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

Basic Ideas: *learn to forget*

http://colah.github.io/posts/2015-08-Understanding-LSTMs/



X vs. XCell (pytorch)

- “X”Cell (RNNCell, LSTMCell, GRUCell)
  - The core cell definition
- “X” (RNN, LSTM, GRU)
  - The sequential application of “X”Cell to your input data

Outline

Core Problem

Basic cell definition

Example in PyTorch