CMSC 473/673
Natural Language Processing

Instructor: Lara J. Martin (she/they)

TA: Duong Ta (he)

Slides modified from Dr. Frank Ferraro

Learning Objectives

Analyze code of a RNN language model

Review: Neural Language Models

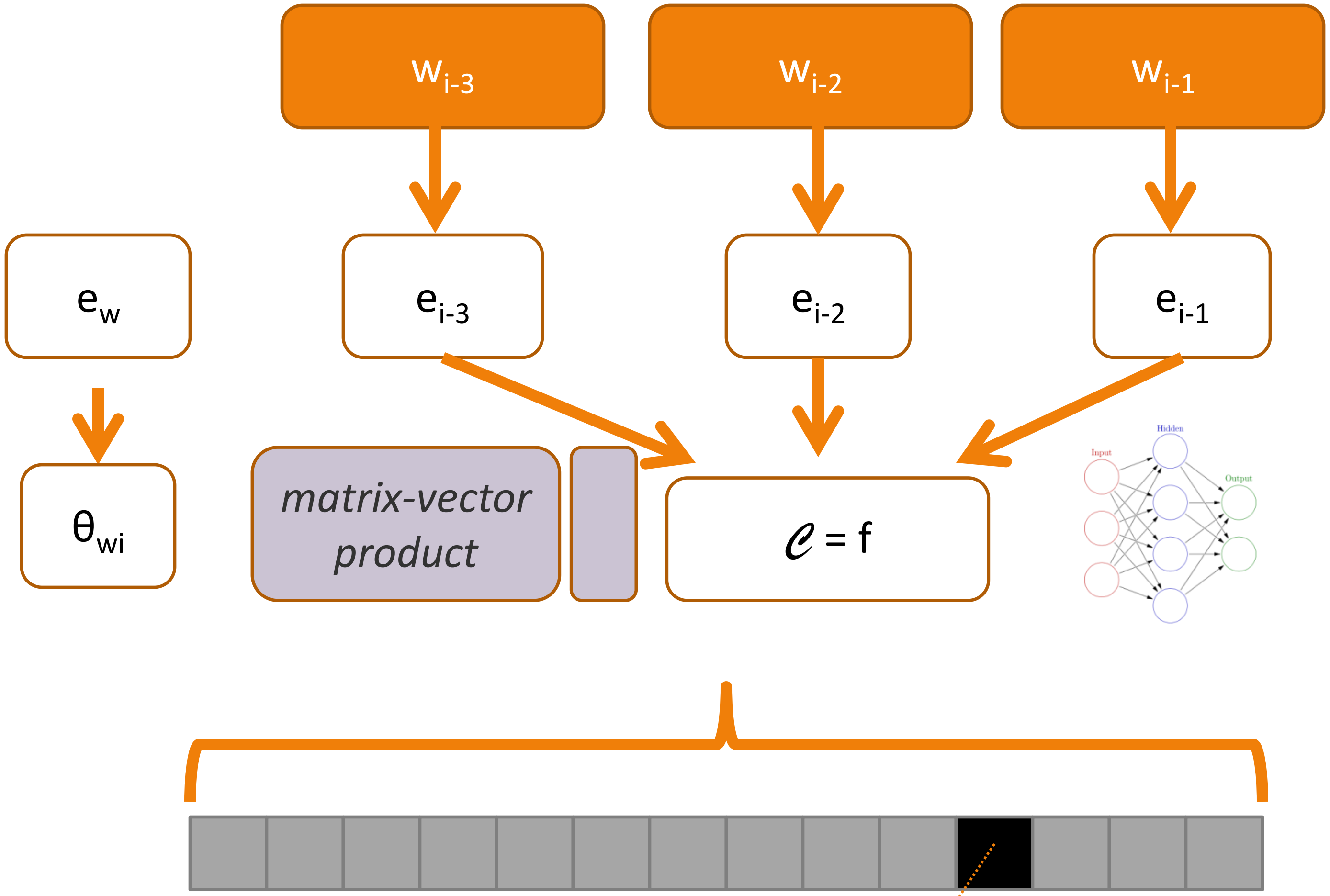
given some context…

create/use “distributed representations”…

combine these representations…

compute beliefs about what is likely…

predict the next word

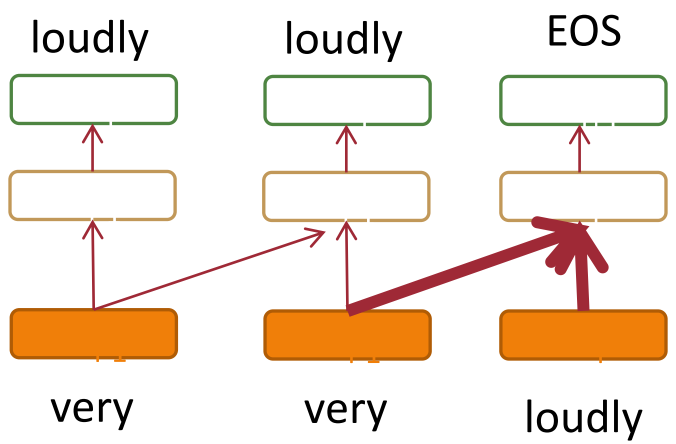


$$p(w_i |\, w_{i-3}, w_{i-2}, w_{i-1}) = \mathrm{softmax}(\theta_{w_i} \cdot \boldsymbol{f}(w_{i-3}, w_{i-2}, w_{i-1}))$$

w_i

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

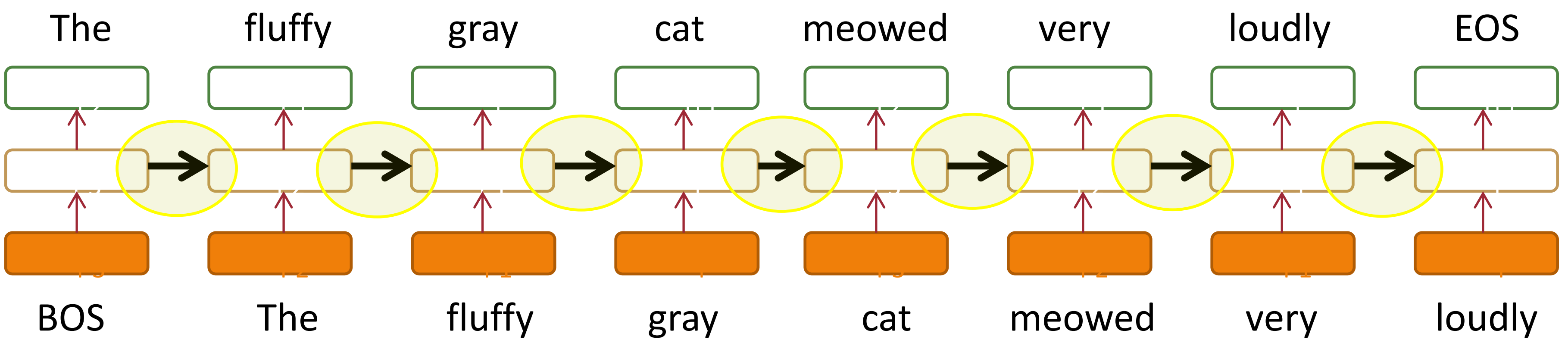
The fluffy gray cat meowed very loudly



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

Review:
A Recurrent Neural Language Model

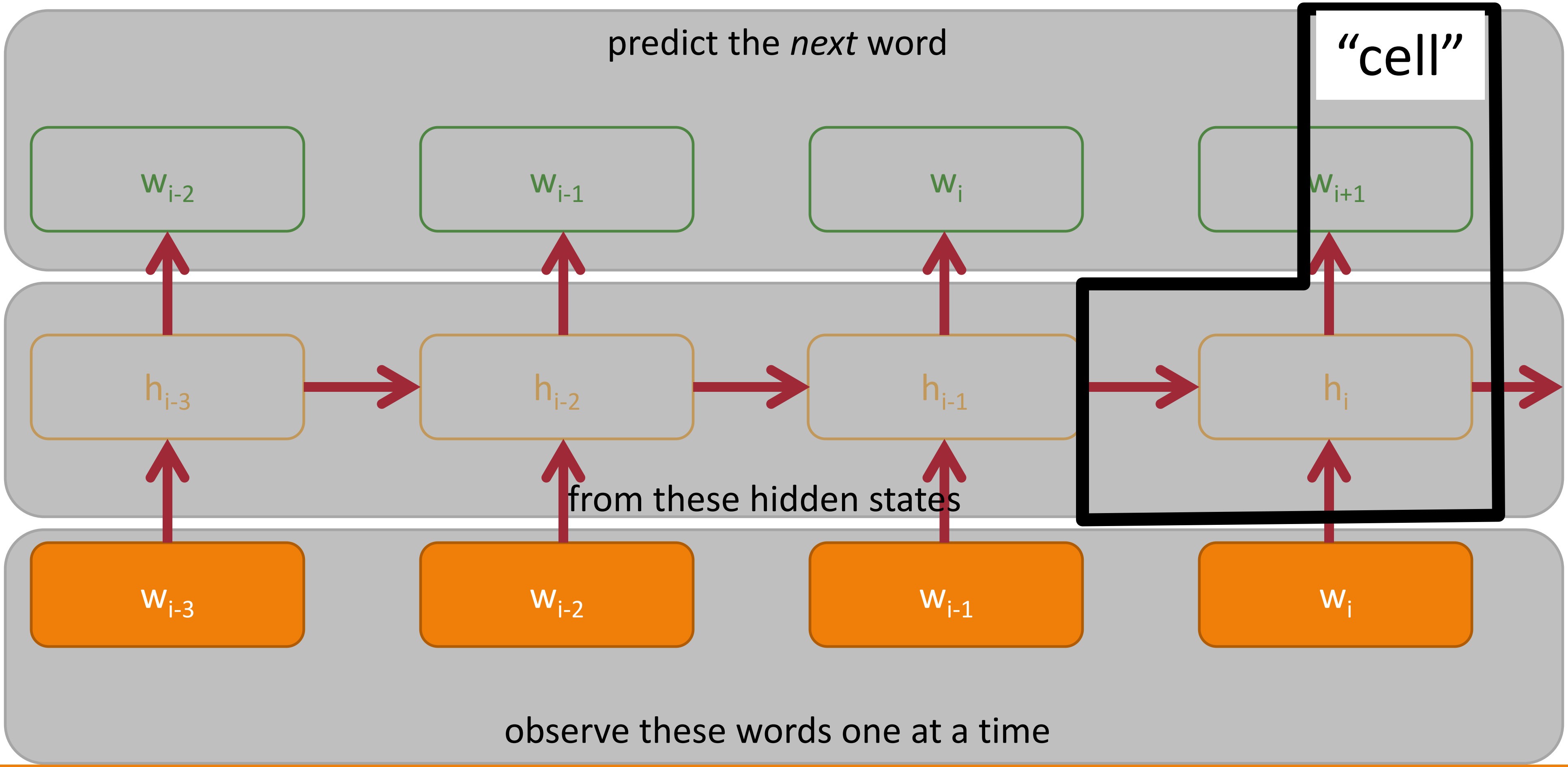
The fluffy gray cat meowed very loudly



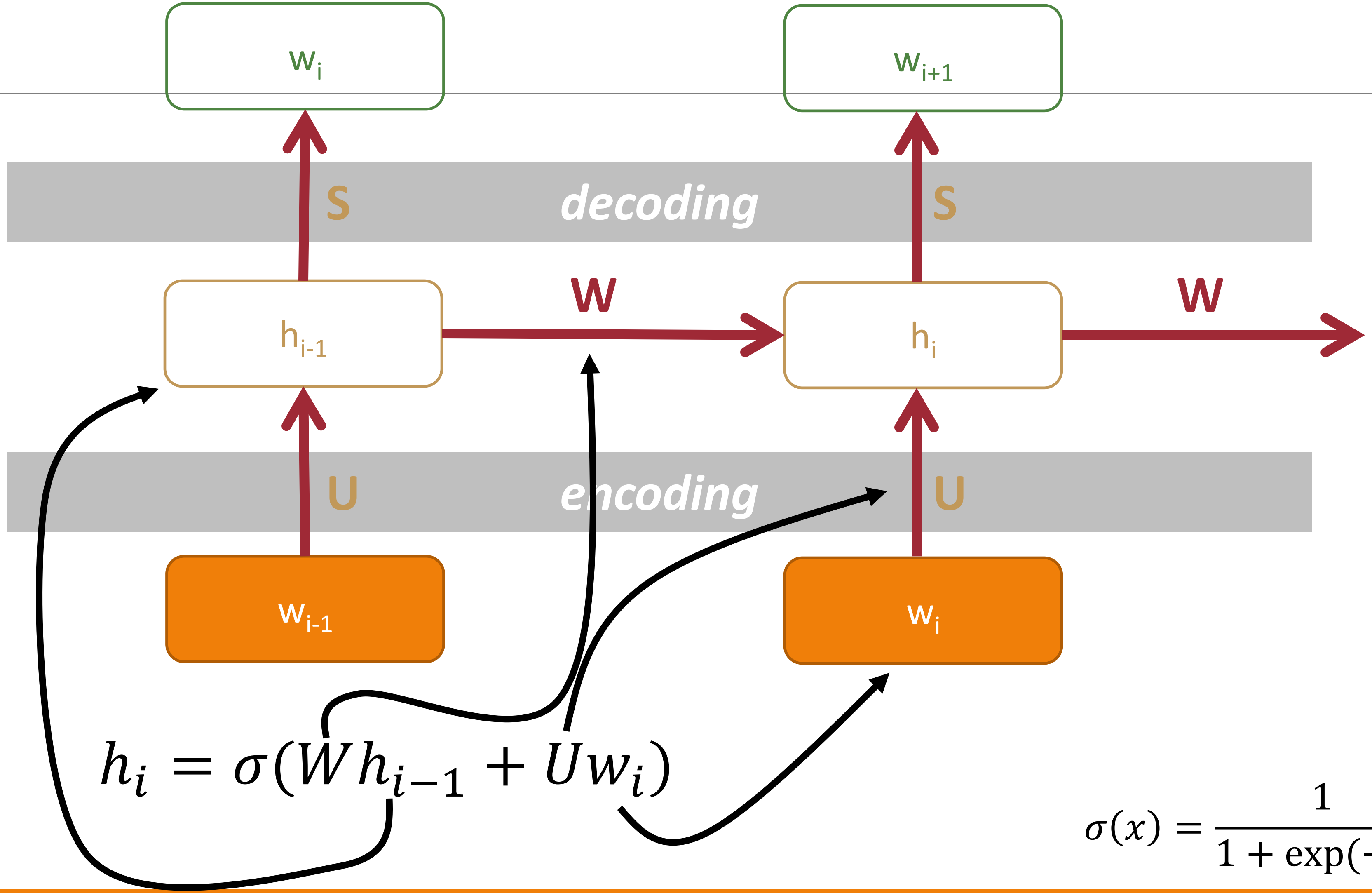
Critical issue: the amount of information flow is fundamentally restricted!!!

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

Review: A Classic View of Recurrent Neural Language Modeling



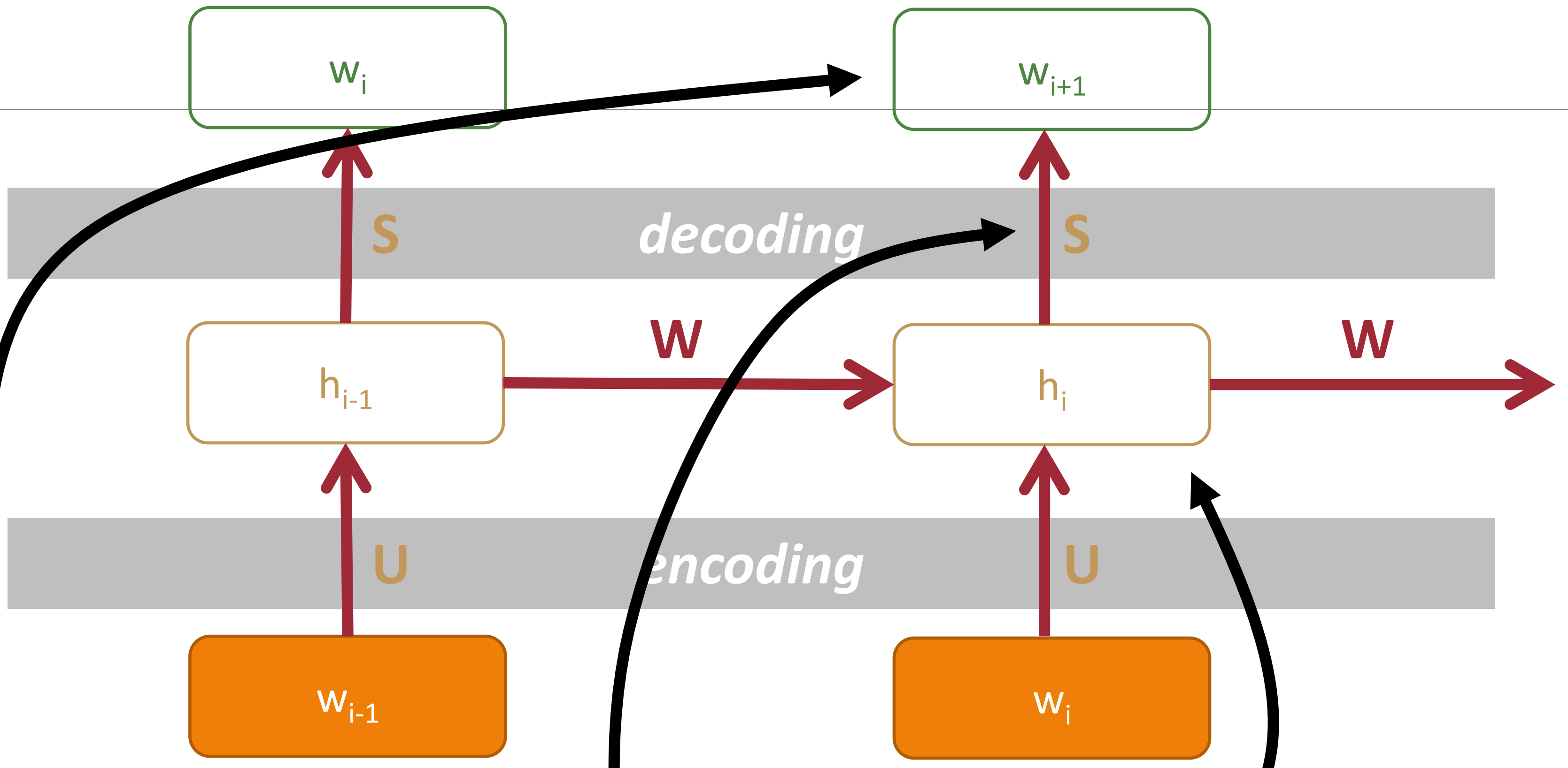
Review:
A *Simple* Recurrent Neural Network Cell



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

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

Review: A *Simple* Recurrent Neural Network Cell

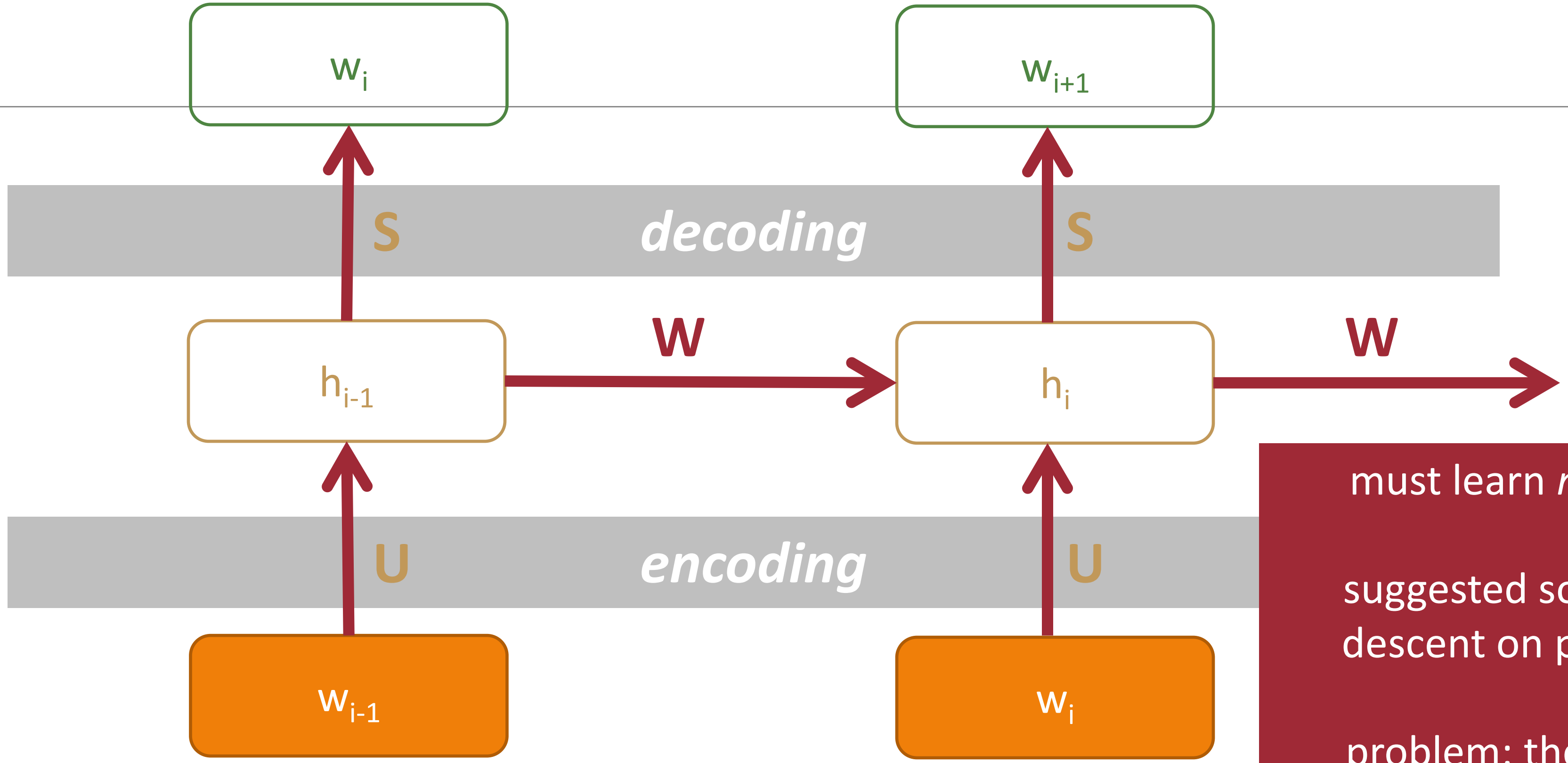


$$h_i = \sigma(W h_{i-1} + U w_i)$$

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

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

Review:
A *Simple* Recurrent Neural Network Cell



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

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

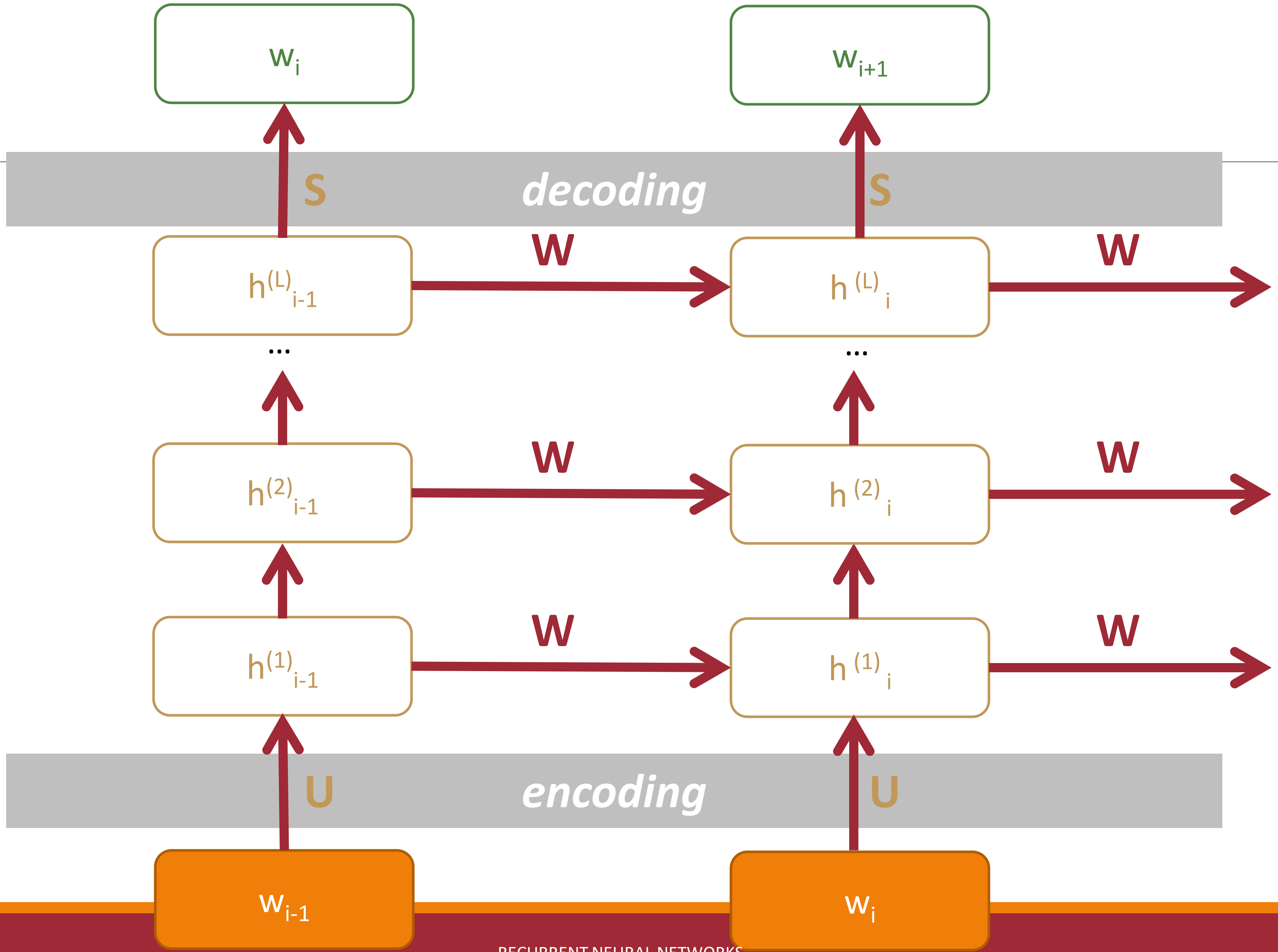
must learn *matrices* U, S, W

suggested solution: gradient descent on prediction ability

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

good news for you: many toolkits do this automatically

Review: A Multi-Layer *Simple* Recurrent Neural Network Cell



Review: 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).

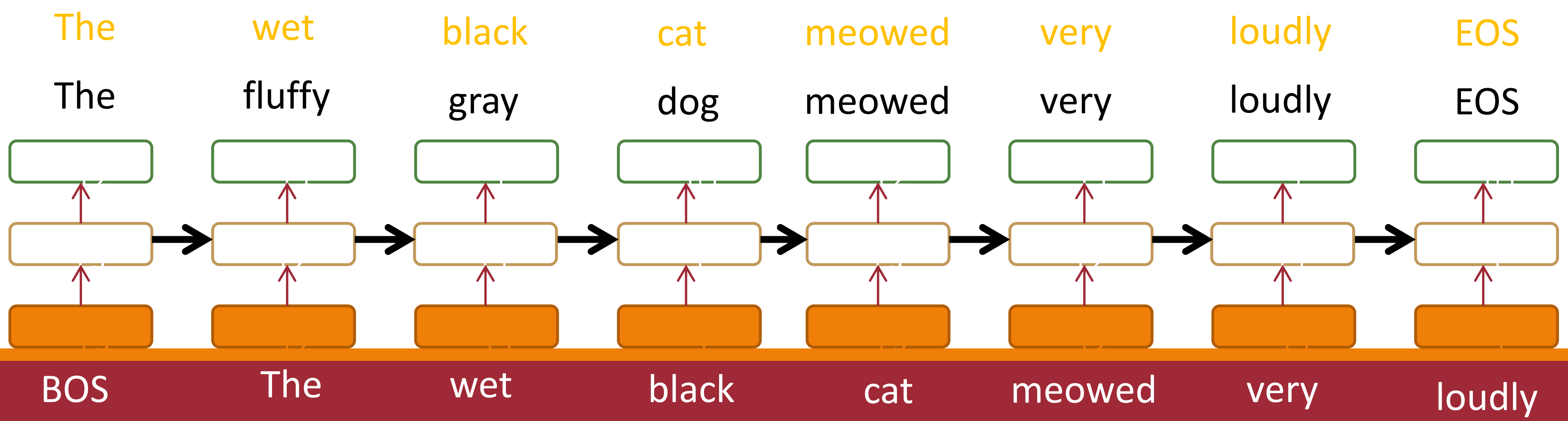
This **loss** is the sum of the per-token cross-entropy loss

Review: 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
…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…



Review: 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

PyTorch RNN LMs

Pick Your Toolkit

PyTorch	Keras
Deeplearning4j	MxNet
TensorFlow	Gluon
DyNet	CNTK
Caffe	…

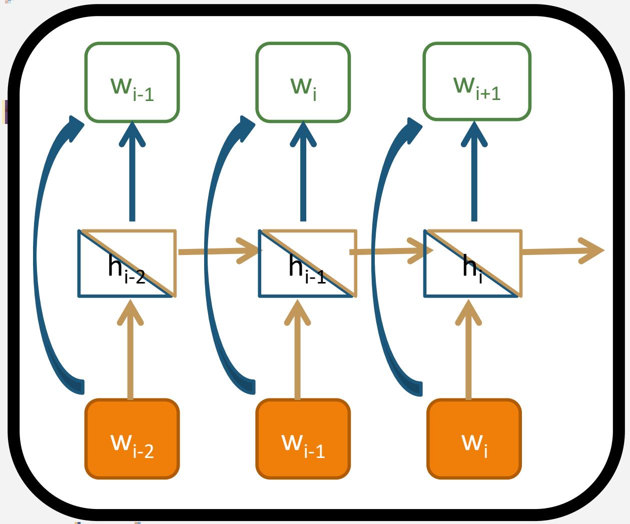
Comparisons:
https://en.wikipedia.org/wiki/Comparison_of_deep_learning_software

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)
```

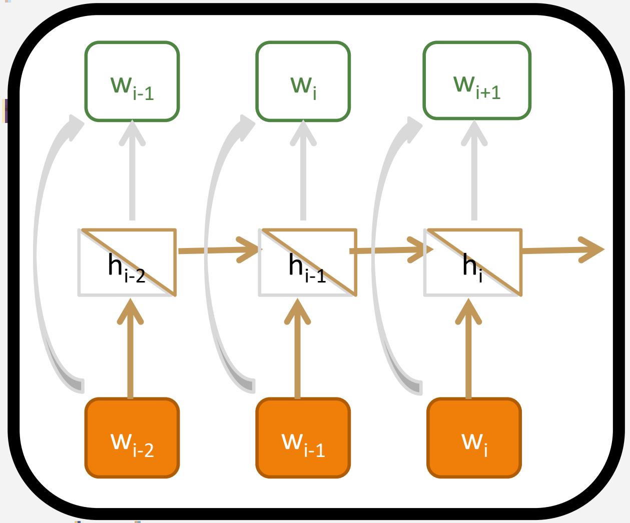


Defining A Simple RNN in Python

(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)
```

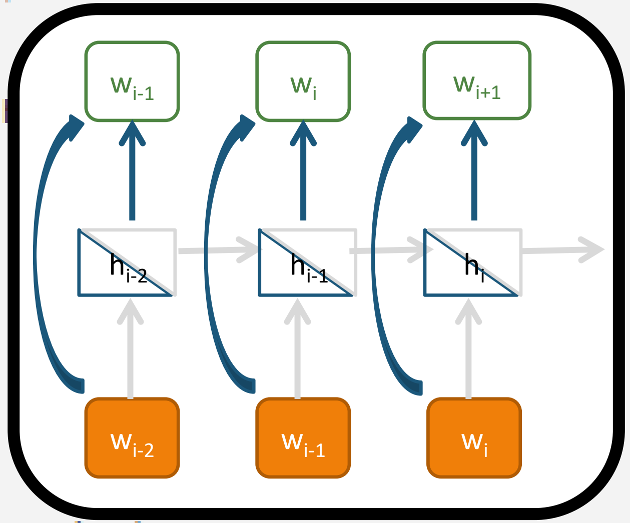


Defining A Simple RNN in Python

(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)
```

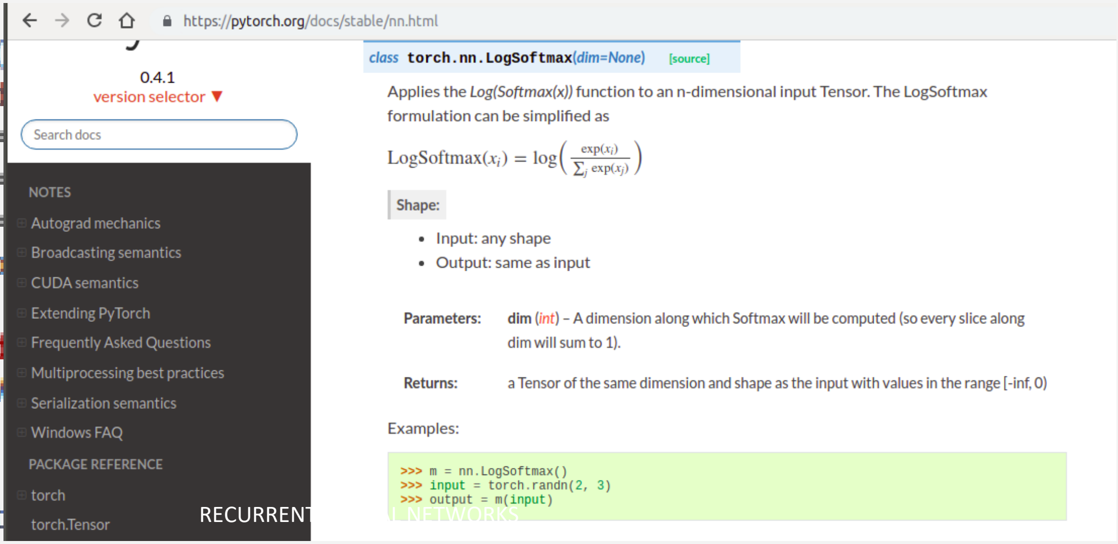


Defining A Simple RNN in Python

(Modified Very Slightly)

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

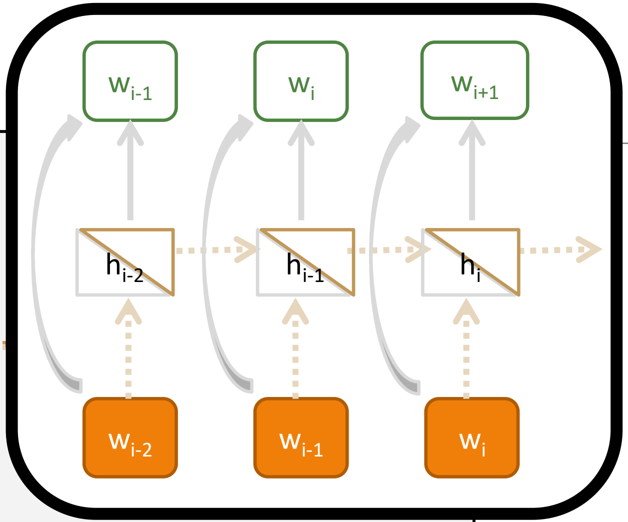
```
import torch.nn as nn
from torch.autograd import Variable

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
```



Defining A Simple RNN in Python

(Modified Very Slightly)

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



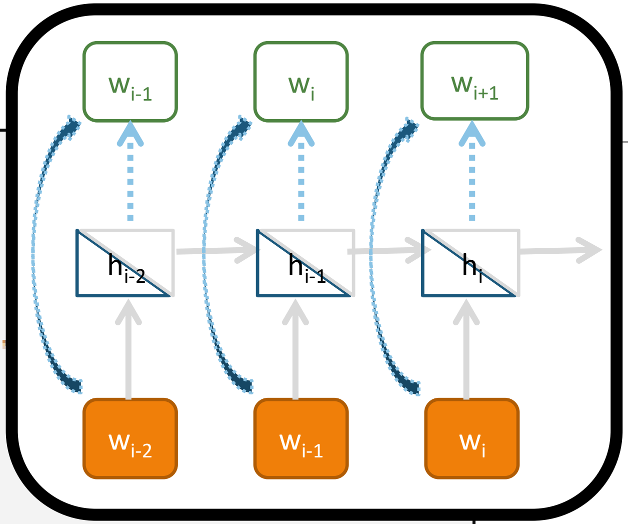

```python
import torch.nn as nn
from torch.autograd import Variable

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)
```

decode

(we'll talk about this)

Training A Simple RNN in Python

(Modified Very Slightly)

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

Negative log-likelihood

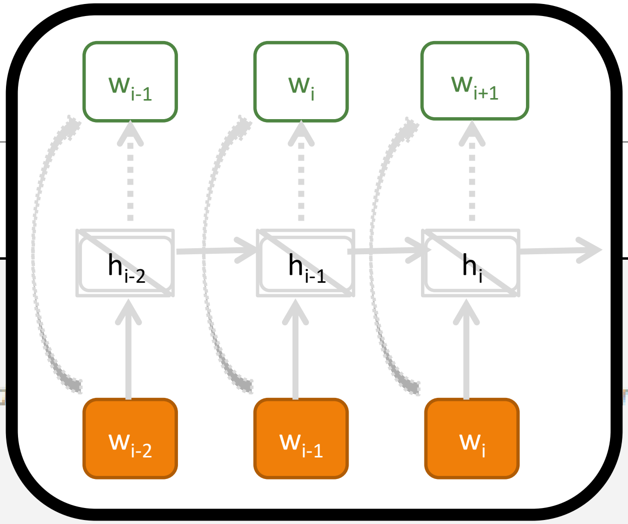
(we'll talk about this)

```
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 exploc

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

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

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

(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

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

(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

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

Suggested Implementation Changes

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

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

        self.hidden_size = hidden_size

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

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

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

n_hidden = 128
rnn = RNN(n_letters, n_hidden, n_ca
```

current Pytorch refers to this a "cell"

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

nn.CrossEntropyLoss()

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

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

    rnn.zero_grad()

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

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

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

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

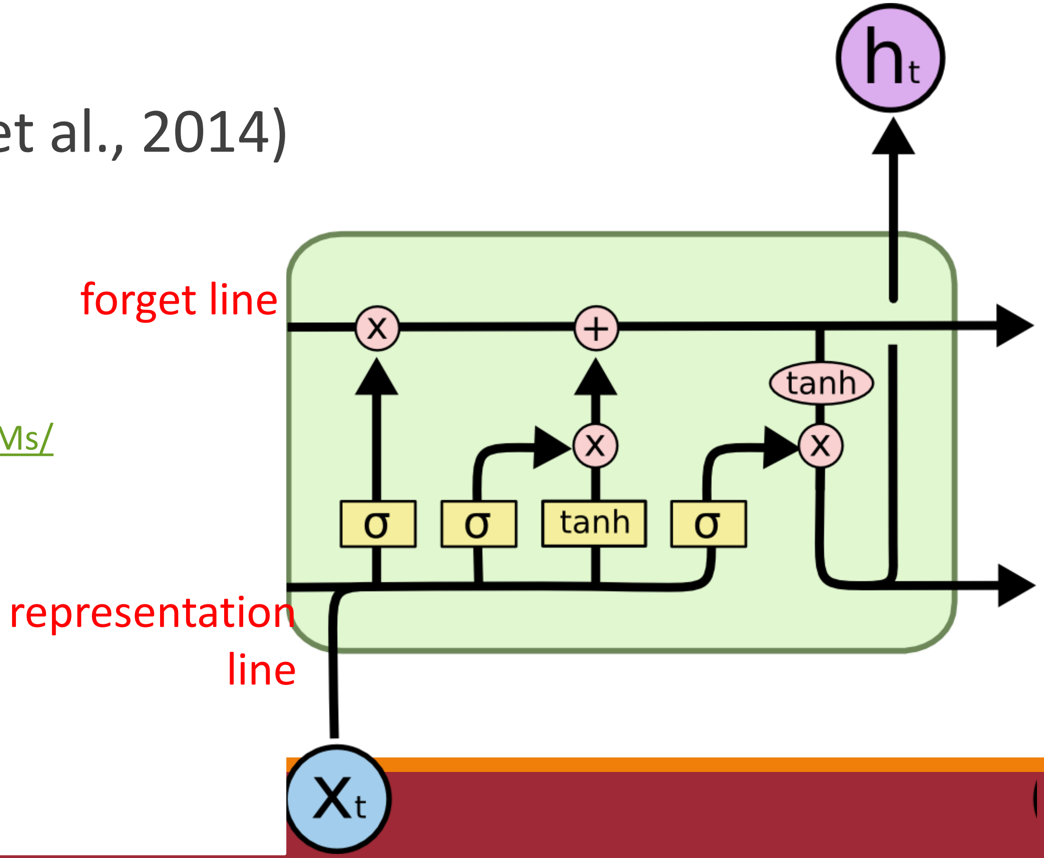
Another Solution: LSTMs/GRUs

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

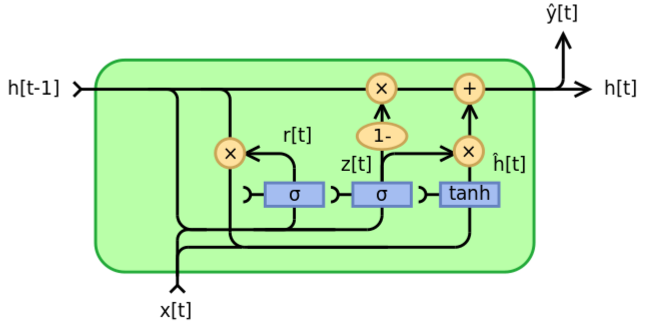
GRU: Gated Recurrent Unit (Cho et al., 2014)

Basic Ideas: *learn to forget*

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

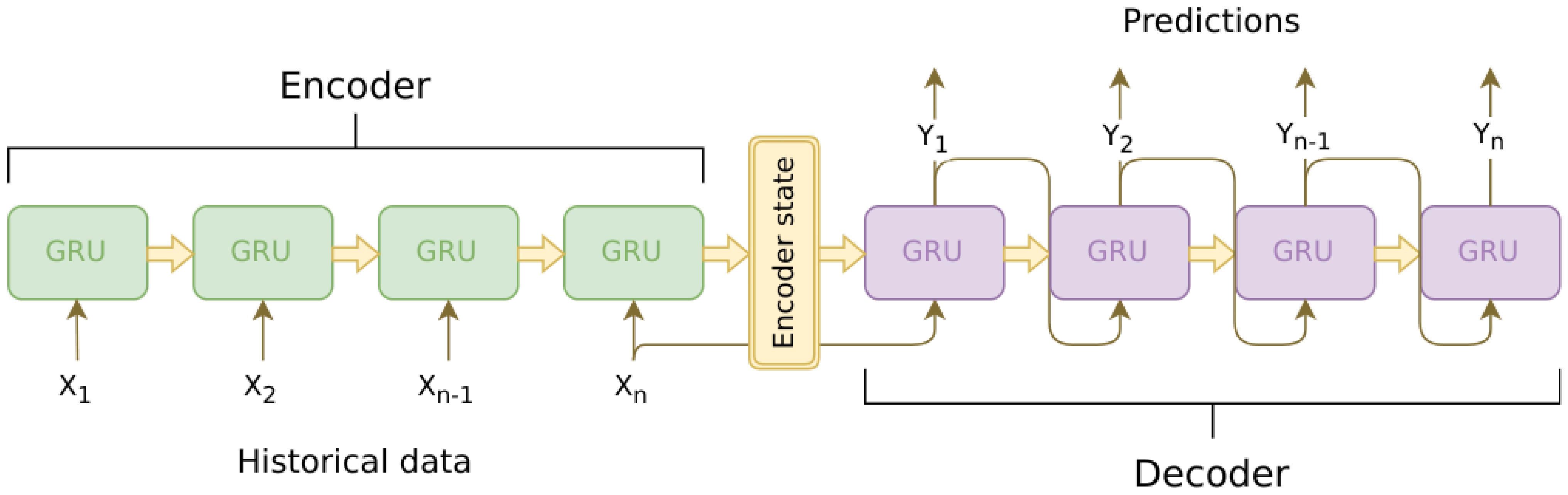


Gated Recurrent Unit



Sequence-to-Sequence

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



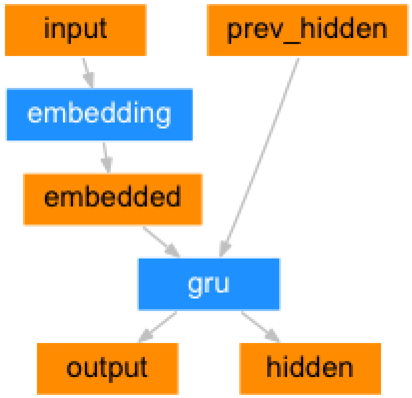
https://jeddy92.github.io/JEddy92.github.io/ts_seq2seq_intro/

Note that this still has hidden layers!

Seq2Seq Tutorial

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

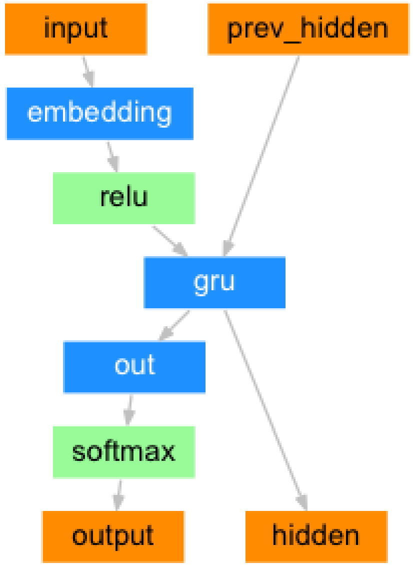
Encoder




```
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
```