Feedforward & Recurrent Neural Networks

Instructor: Lara J. Martin (she/they)

TA: Omkar Kulkarni (he)

https://laramartin.net/NLP-class/

Slides modified from Dr. Frank Ferraro

Learning Objectives

Define the basic architecture of a neural network

Distinguish between count-based, logistic regression, and neural LMs

Define the basic cell architecture of an RNN

Backpropagate loss through an example RNN

Create a simple RNN with PyTorch

For more information, check out Section 3.3 of the SLP textbook
Other resources:
https://en.wikipedia.org/wiki/Perplexity#Perplexity_of_a_probability_model
https://www.geeksforgeeks.org/nlp/perplexity-for-llm-evaluation/

Review: What is perplexity?

$$\text{perplexity} = \exp\left(\frac{-1}{M}\log p(w_1, \ldots, w_M)\right)$$

$$= \exp\left(\frac{-1}{M}\sum_{i=1}^{M}\log p(w_i \mid h_i)\right)$$

$$= \exp\left(\frac{-1}{M}\sum_{i=1}^{M}\log p(w_i \mid w_{i-1}, w_{i-2}, w_{i-3}, \ldots)\right)$$

$$= 2^{H(p)}$$

H(p) is entropy of prediction *p*

Review: Add-λ estimation (Count-based)

Other names: Laplace smoothing, Lidstone smoothing

Pretend we saw each word λ more times than we did

Add λ to all the counts

$$p(\mathrm{z}) \propto count(\mathrm{z}) + \lambda$$

$$= \frac{count(\mathrm{z}) + \lambda}{\sum_v (count(\mathrm{v}) + \lambda)}$$

Review: Adding <UNK> to trigrams

Trigrams	MLE p(trigram)	UNK-ed trigrams	Smoothed p(trigram)
<BOS> <BOS> The	1	<BOS> <BOS> The	2/17
<BOS> The film	1	<BOS> The film	2/17
The film ,	0	The film <UNK>	1/17
film , a	0	film <UNK> a	1/16
, a hit	0	<UNK> a hit	1/16
a hit !	0	a hit <UNK>	1/17
hit ! <EOS>	0	hit <UNK> <EOS>	1/16

Types of Early LMs

Maximum likelihood (MLE): simple counting

Other count-based models

- Laplace smoothing, add- λ
- Interpolation models
- Discounted backoff
- Interpolated (modified) Kneser-Ney
- Good-Turing
- Witten-Bell

Easy to implement

Advanced/ out of scope

Maxent n-gram models

Featureful LMs

Neural n-gram models

Feedforward LMs

Recurrent/autoregressive NNs

Precursor to modern LMs

Ways of using a logistic regression model

Class Prediction

p(Label |Colorless green ideas sleep furiously) = $w_1^T f_1 * \ldots * w_n^T f_n$

Review: LR/Maxent Modeling

$p($ ENTAILED $|$

s: Michael Jordan, coach Phil Jackson and the star cast, including Scottie Pippen, took the Chicago Bulls to six National Basketball Association championships.
h: The Bulls basketball team is based in Chicago.

$) \propto$

$$\frac{1}{Z}\exp\left(\theta_{\text{ENTAILED}}^{T} f(🗎)\right)$$

Ways of using a logistic regression model

Class Prediction

p(Label |Colorless green ideas sleep furiously) = $w_1^T f_1 * \ldots * w_n^T f_n$

Class-specific LM

p(Colorless green ideas sleep furiously | Label) = p(Colorless | Label, <BOS>) $* \ldots *$ p(<EOS> | Label , furiously)

Review: Maxent Models as Featureful n-gram Language Models

p(Colorless green ideas sleep furiously | Label) =
p(Colorless | Label, <BOS>) * … * p(<EOS> | Label , furiously)

The label from our classification problem (e.g., entailed/not entailed) is now on this side of the conditional because we're interested in generating the text, not predicting the label

Model each n-gram term with a maxent model

$$p(x_i \mid y, x_{i-N+1:i-1}) = \text{maxent}(y, x_{i-N+1:i-1}, x_i)$$

generatively trained:
learn to model *(class-specific) language*

Review: Language Model with Maxent n-grams

$$p_n(\text{📄}|y) = \prod_{i=1}^{M} \text{maxent}(\overset{\text{label}}{\downarrow}{y}, \underbrace{x_{i-n+1:i-1}, x_i}_{\text{n-gram}})$$

$$= \prod_{i=1}^{M} \frac{\exp(\theta_{x_i}^{T} f(y, x_{i-n+1:i-1}))}{\sum_{x'} \exp(\theta_{x'}^{T} f(y, x_{i-n+1:i-1}))}$$

Iterate through all possible output vocab types x'---just like in count-based LMs

Review: What Should These Features Do?

$p(x_i \mid y, x_{i-N+1:i-1}) = \text{maxent}(y, x_{i-N+1:i-1}, x_i)$, e.g.,

$$p(\text{sleep} \mid y, \text{green}, \text{ideas}) = \\ \text{maxent}\big(y, x_{i-2,i-1} = (\text{green}, \text{ideas}), x_i = \text{sleep}\big) \\ \propto \exp({\theta_{x_i=\text{sleep}}}^T f(y, x_{i-2,i-1} = (\text{green}, \text{ideas})))$$

New question: If you were given the dense representations for 2 words, how might you represent them together as a single bigram?

Ways of using a logistic regression model

Class Prediction

p(Label |Colorless green ideas sleep furiously) = $w_1^T f_1 * \dots * w_n^T f_n$

Class-specific LM

p(Colorless green ideas sleep furiously | Label) = p(Colorless | Label, <BOS>) $* \dots *$ p(<EOS> | Label , furiously)

"Regular" LM

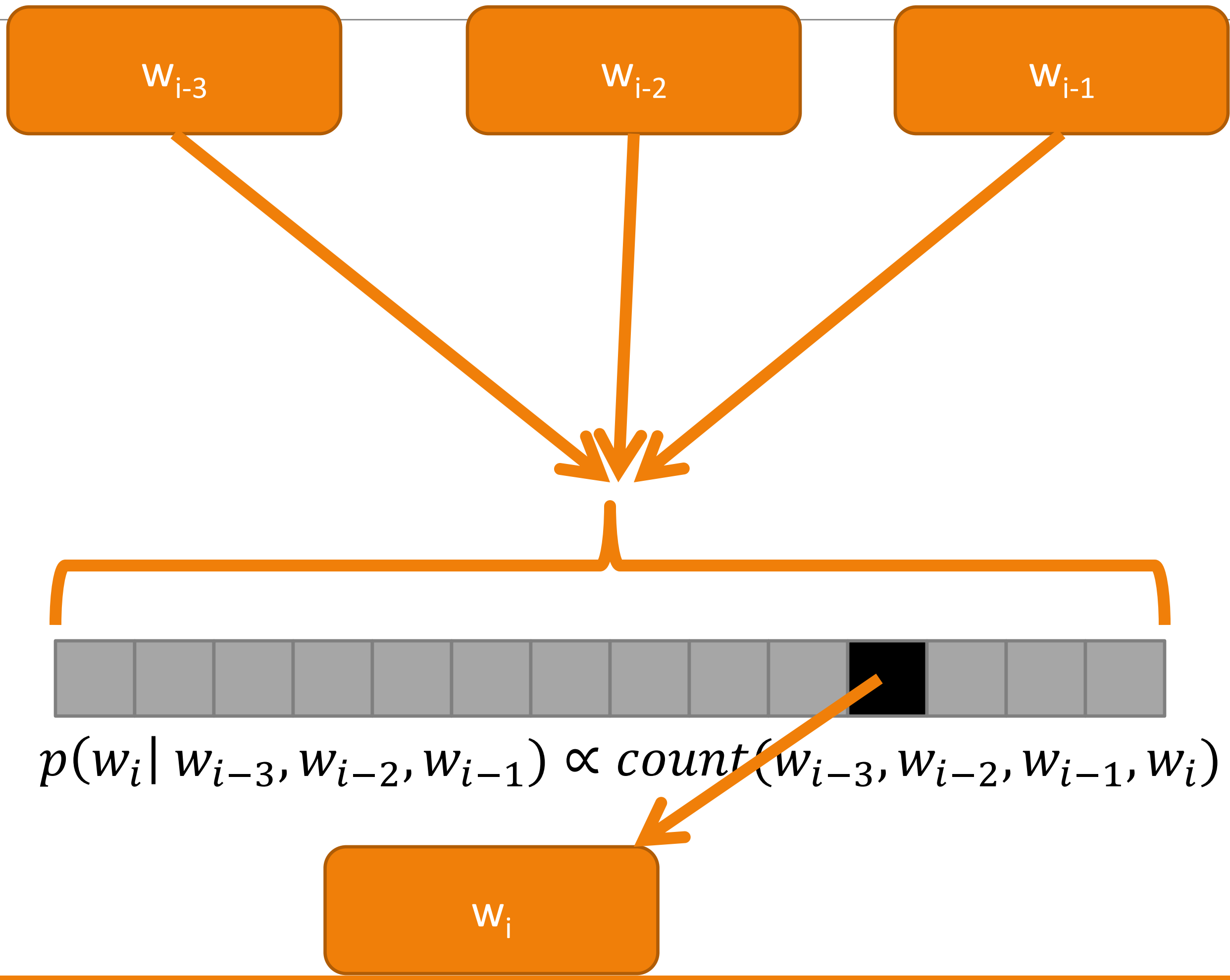
p(Colorless green ideas sleep furiously) = p(Colorless | <BOS> <BOS>) $* \dots *$ p(<EOS> | sleep furiously)

Review: N-gram Language Models

given some context…

compute beliefs about what is likely…

predict the next word



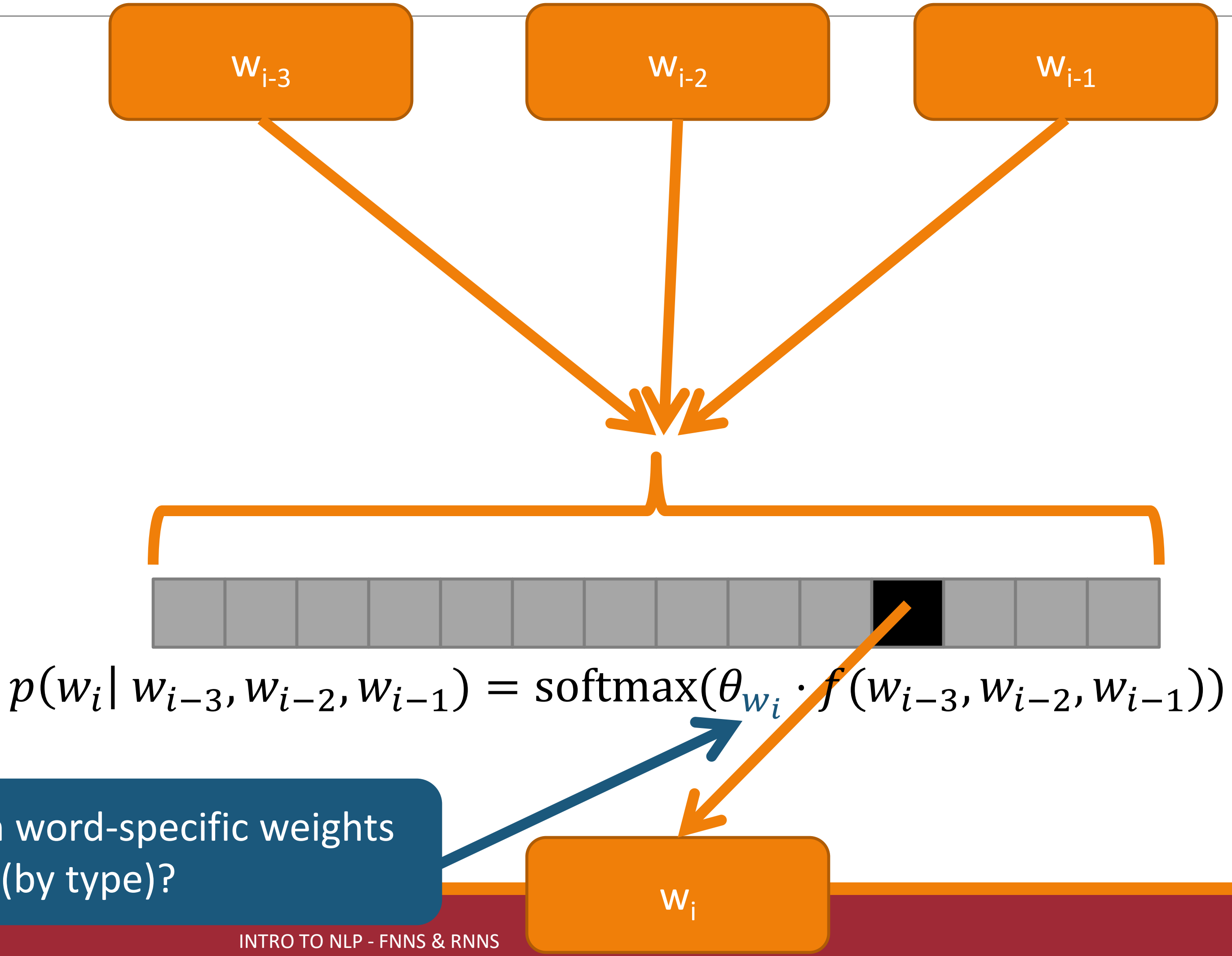
$$p(w_i \mid w_{i-3}, w_{i-2}, w_{i-1}) \propto count(w_{i-3}, w_{i-2}, w_{i-1}, w_i)$$

Review: Maxent/LR Language Models

given some context…

compute beliefs about what is likely…

predict the next word



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

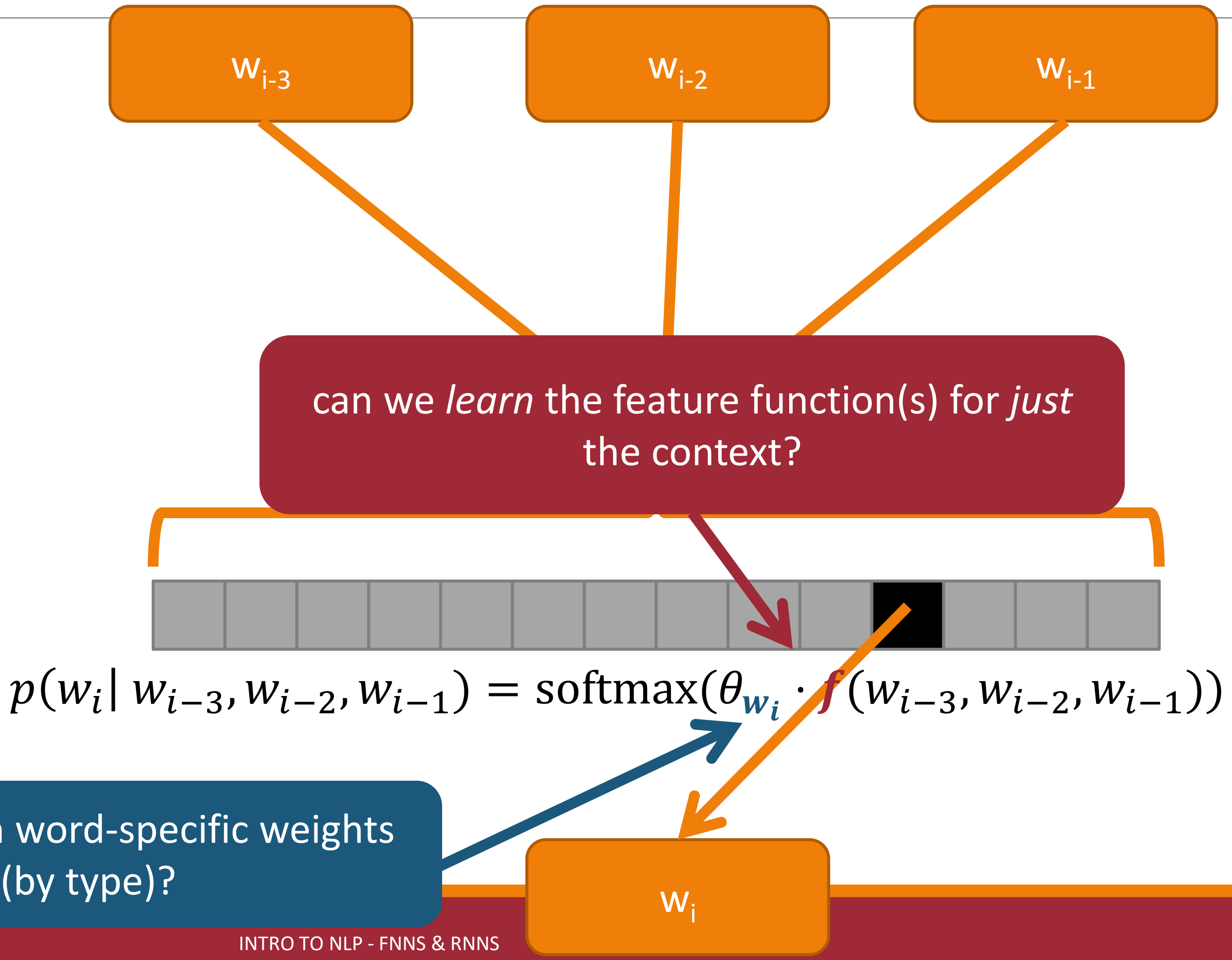
can we learn word-specific weights (by type)?

Review: Neural Language Models

given some context…

compute beliefs about what is likely…

predict the next word



w_{i-3}
w_{i-2}
w_{i-1}
can we *learn* the feature function(s) for *just* the context?
can we learn word-specific weights (by type)?
w_i

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

Review: Neural Language Models

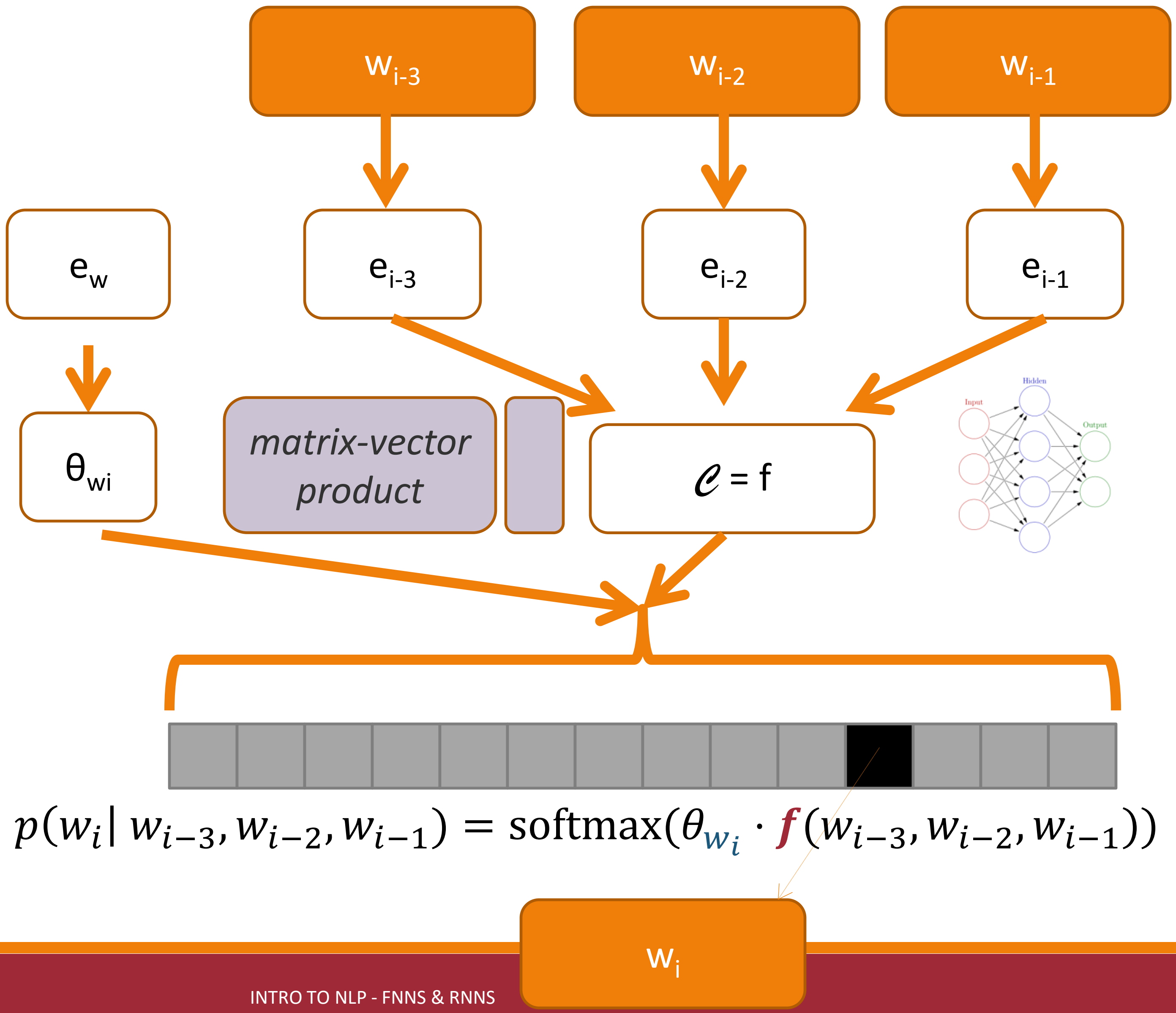
given some context…

create/use "distributed representations"…

combine these representations…

compute beliefs about what is likely…

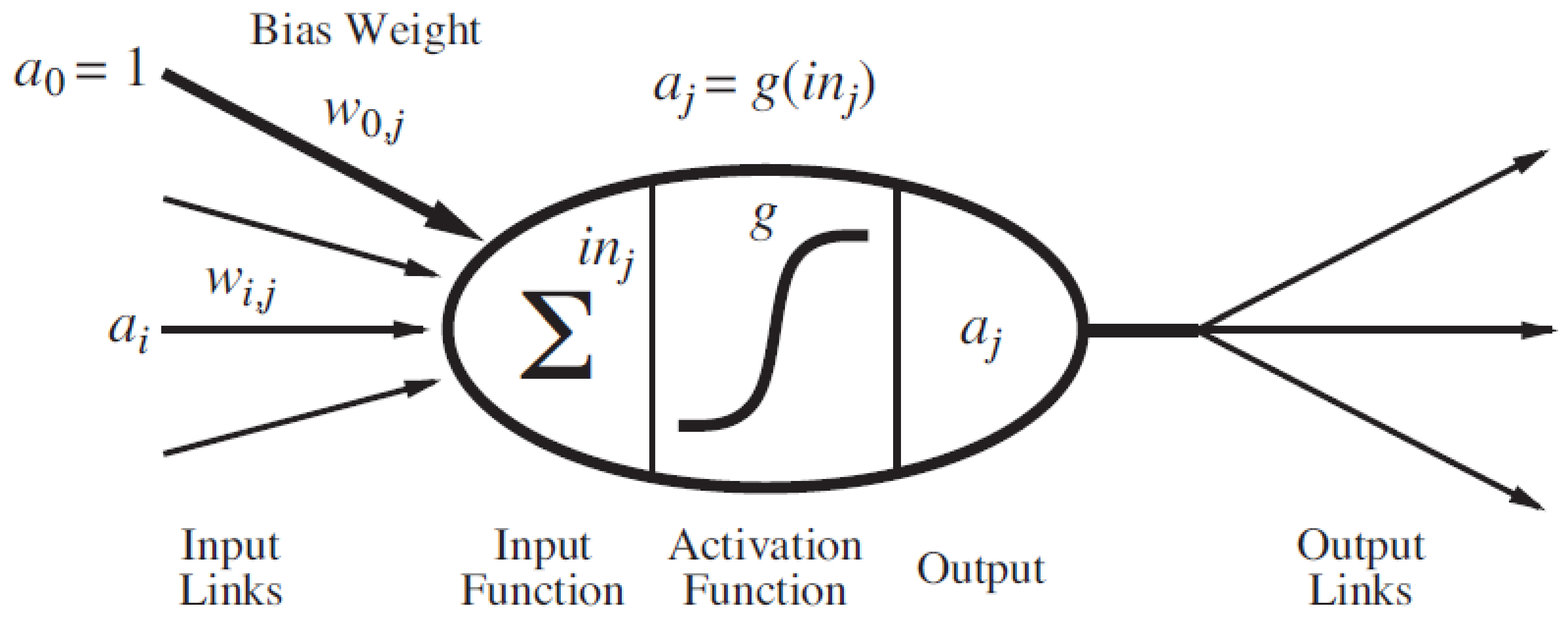
predict the next word



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

Review: Basic Neuron Architecture

$$in_j = w_{0j} + w_{1j}a_1 + w_{2j}a_2 + \cdots + w_{ij}a_i$$



activations
$0 \leq a_i \leq 1$

weights
$-\infty < w_{ij} < \infty$

How are Neural Networks used?

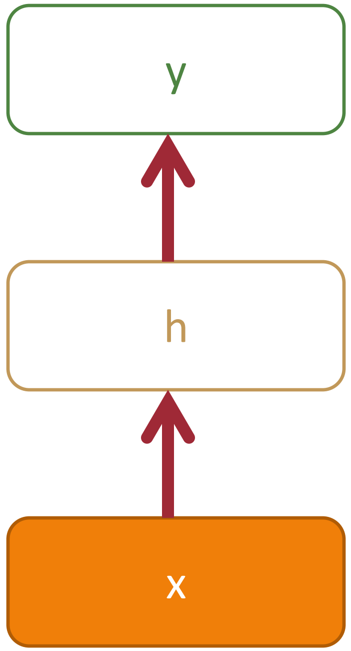
Are neural networks supervised or unsupervised learning?

- Inputs to the network are features of our data set
- Outputs to the network are our labels

Can they be used for classification or regression?

- Either!

Network Types: Flat Input, Flat Output



1. Feed forward

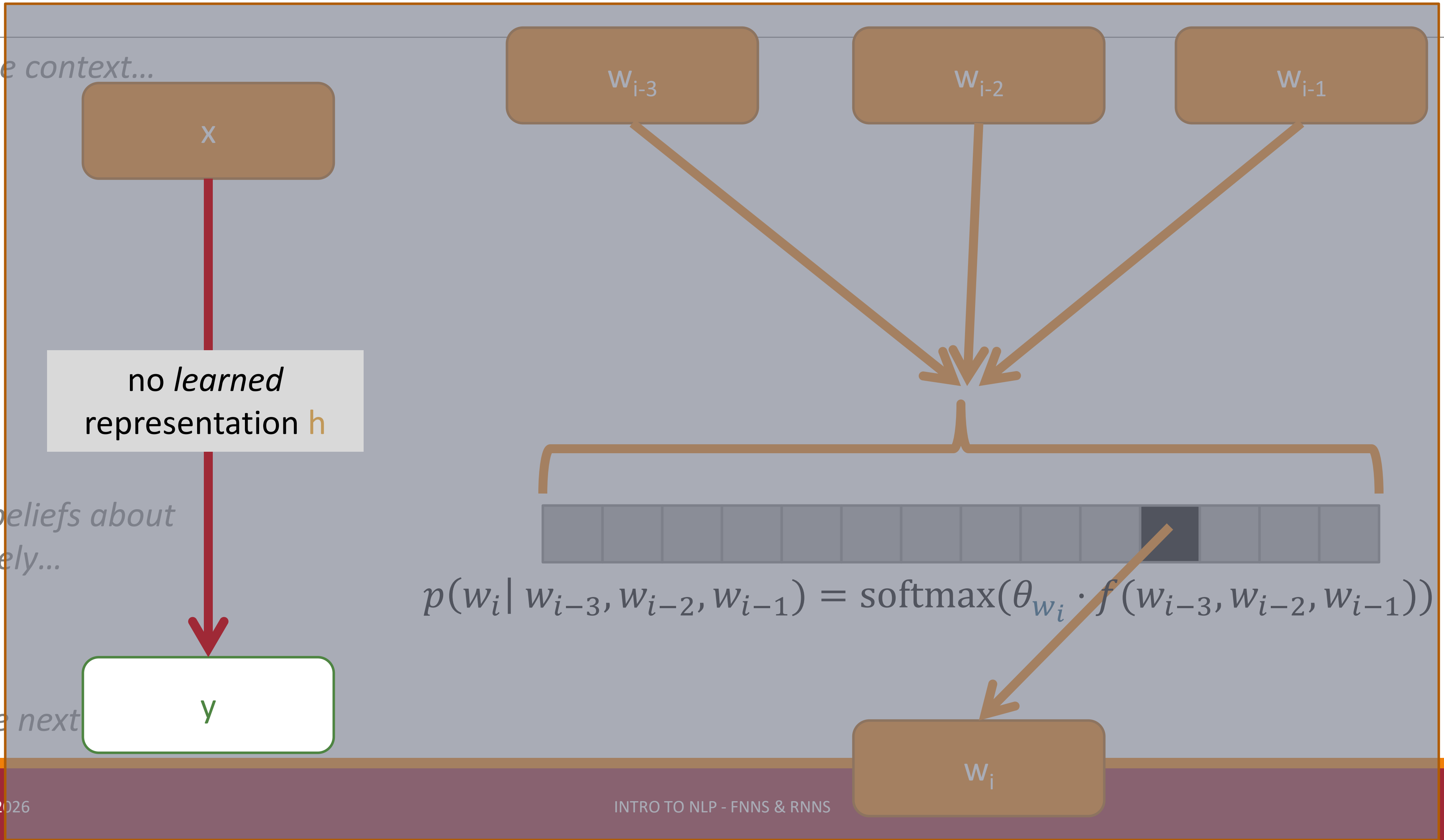
Linearizable feature input
Bag-of-items classification/regression
Basic non-linear model

Maxent Language Models

given some context…

compute beliefs about what is likely…

predict the next



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

Neural Language Models

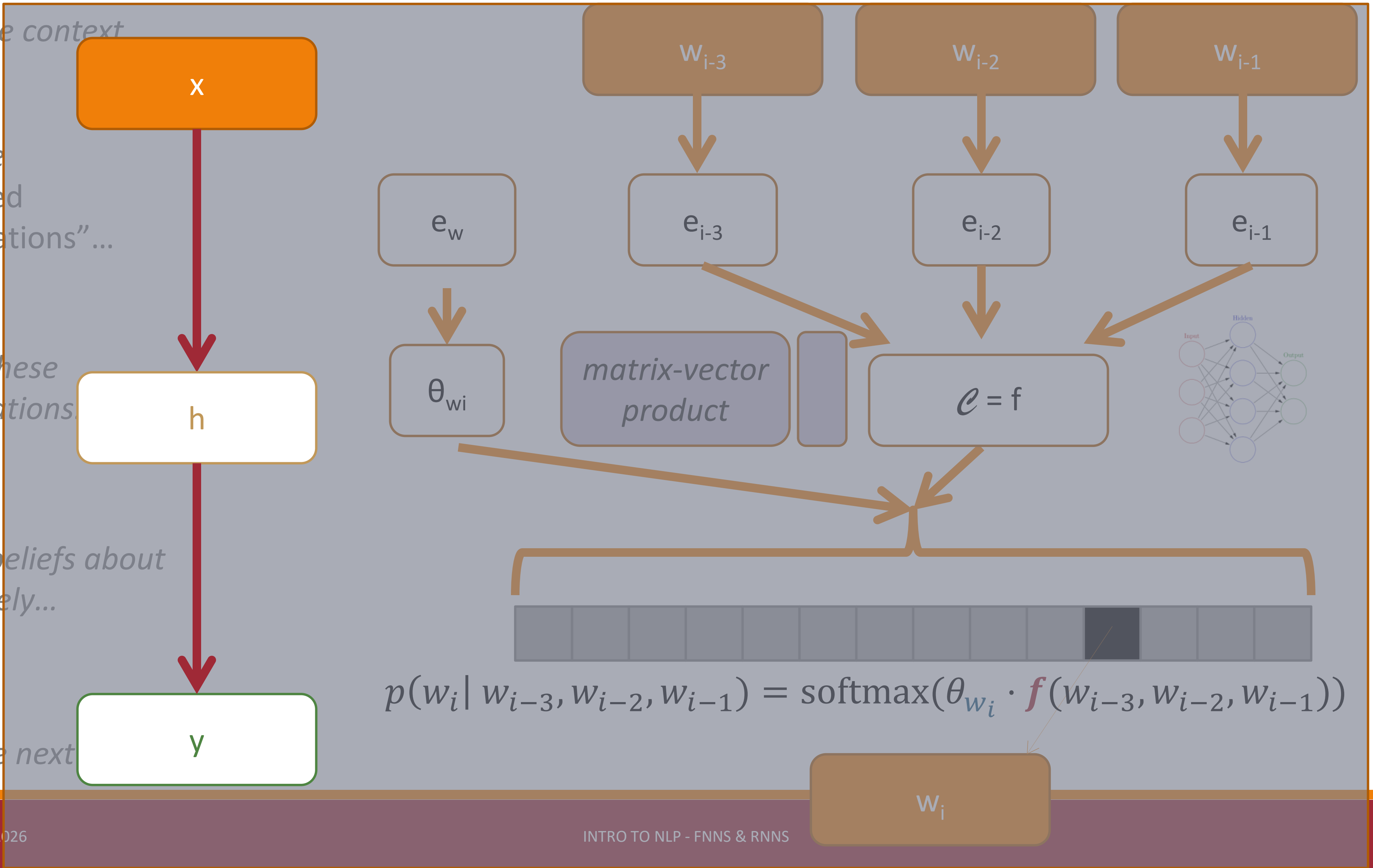
given some context…

create/use "distributed representations"…

combine these representations…

compute beliefs about what is likely…

predict the next…



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

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    (non-linear)
at layer l       activation
                function at l
return argmax_y softmax(θ h_L)
```

$h_0 = x$

$h_l = f_l(W_l h_{l-1} + b_l)$ — $W_l h_{l-1} + b_l$: linear layer; h_l: hidden state at layer l; f_l: (non-linear) activation function at l

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

In Pytorch (torch.nn):

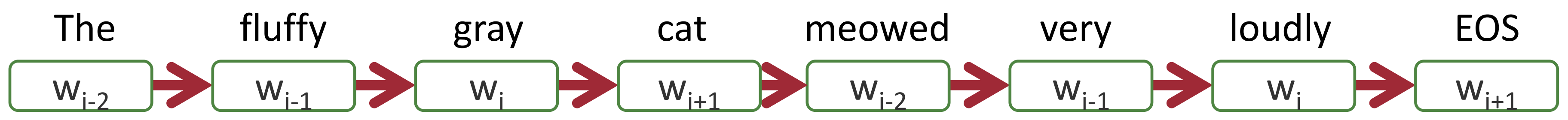
Activation functions:
https://pytorch.org/docs/stable/nn.html?highlight=activation#non-linear-activations-weighted-sum-nonlinearity

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

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

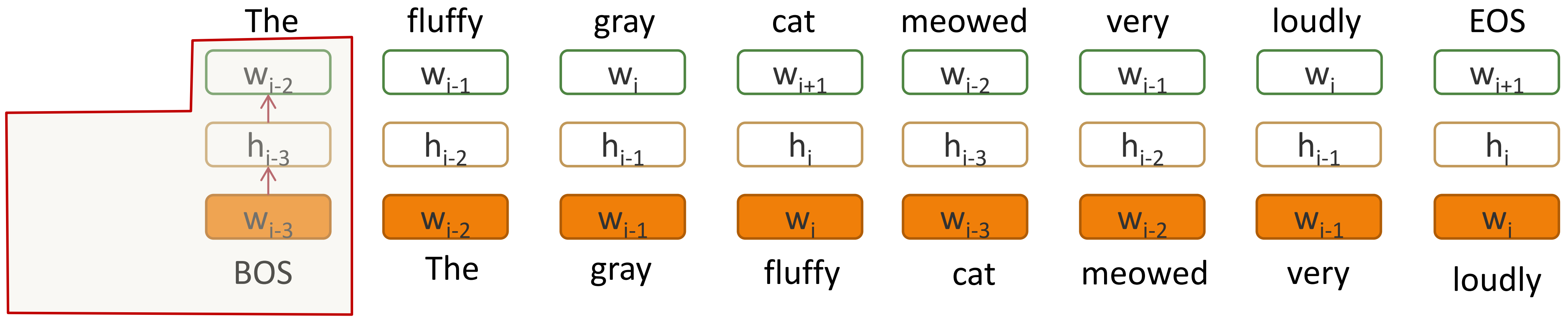
A Neural N-Gram Model

The fluffy gray cat meowed very loudly



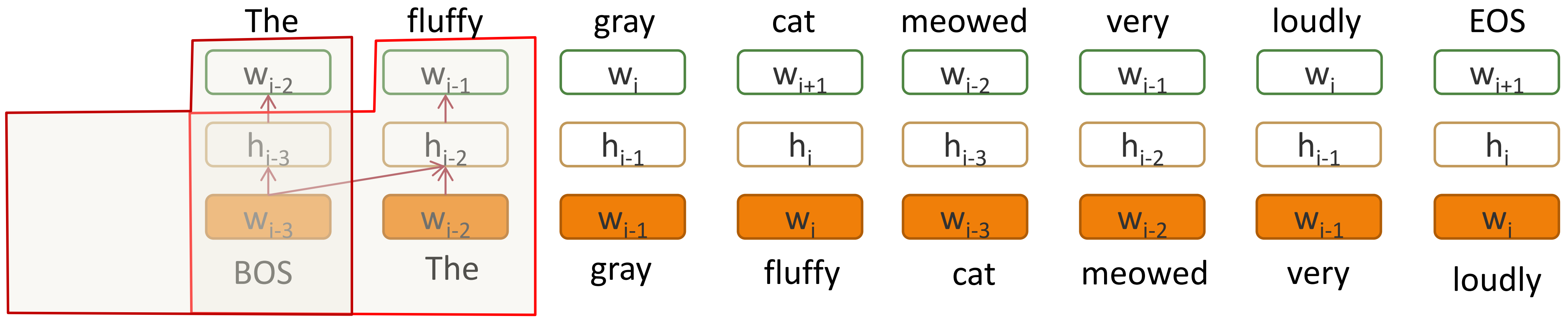
A Neural N-Gram Model (N=3)

The fluffy gray cat meowed very loudly



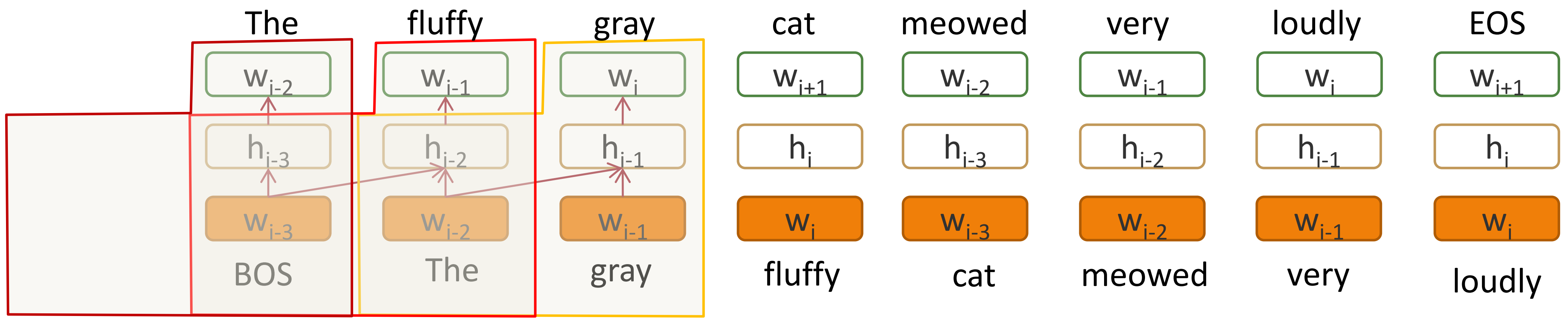
A Neural N-Gram Model (N=3)

The fluffy gray cat meowed very loudly



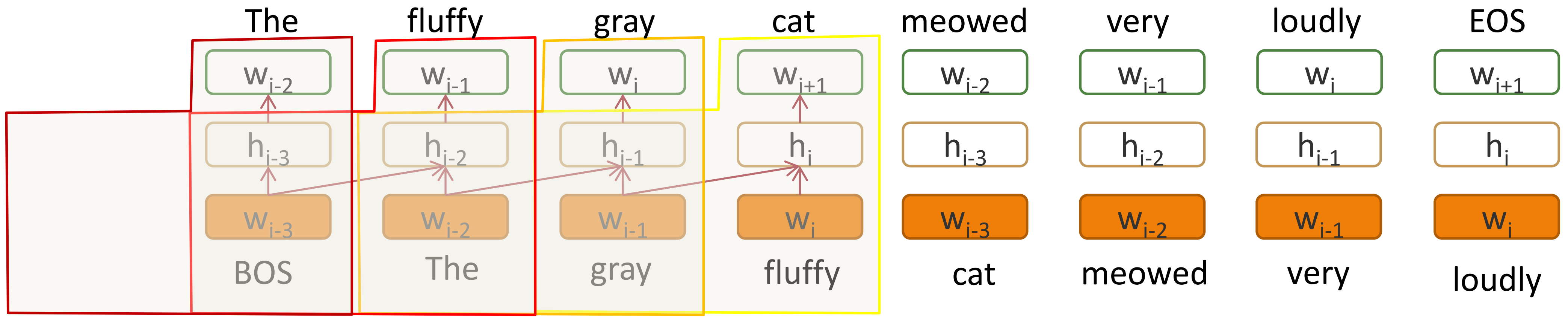
A Neural N-Gram Model (N=3)

The fluffy gray cat meowed very loudly



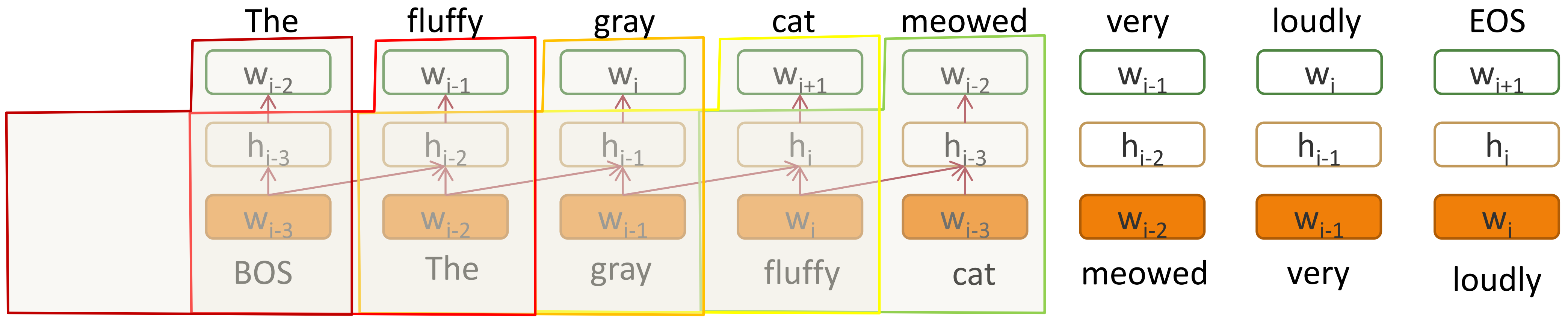
A Neural N-Gram Model (N=3)

The fluffy gray cat meowed very loudly



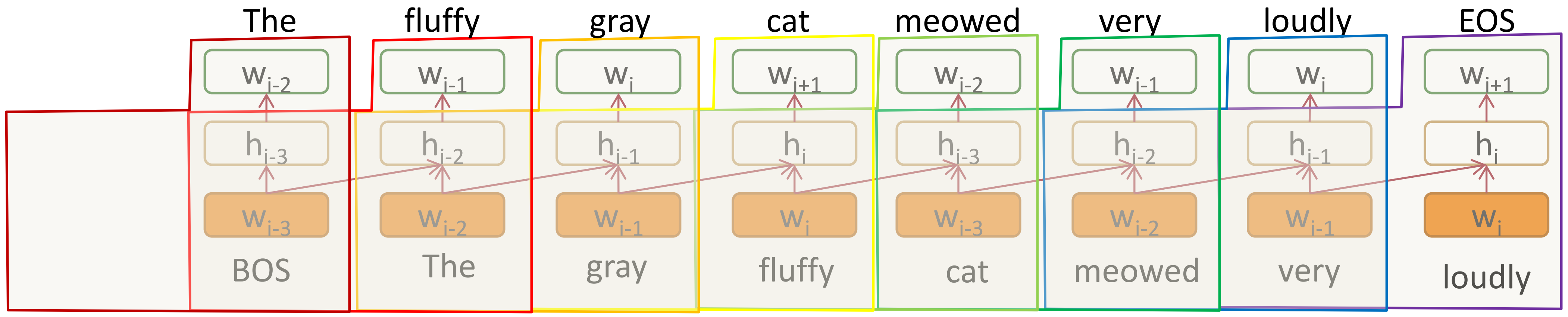
A Neural N-Gram Model (N=3)

The fluffy gray cat meowed very loudly

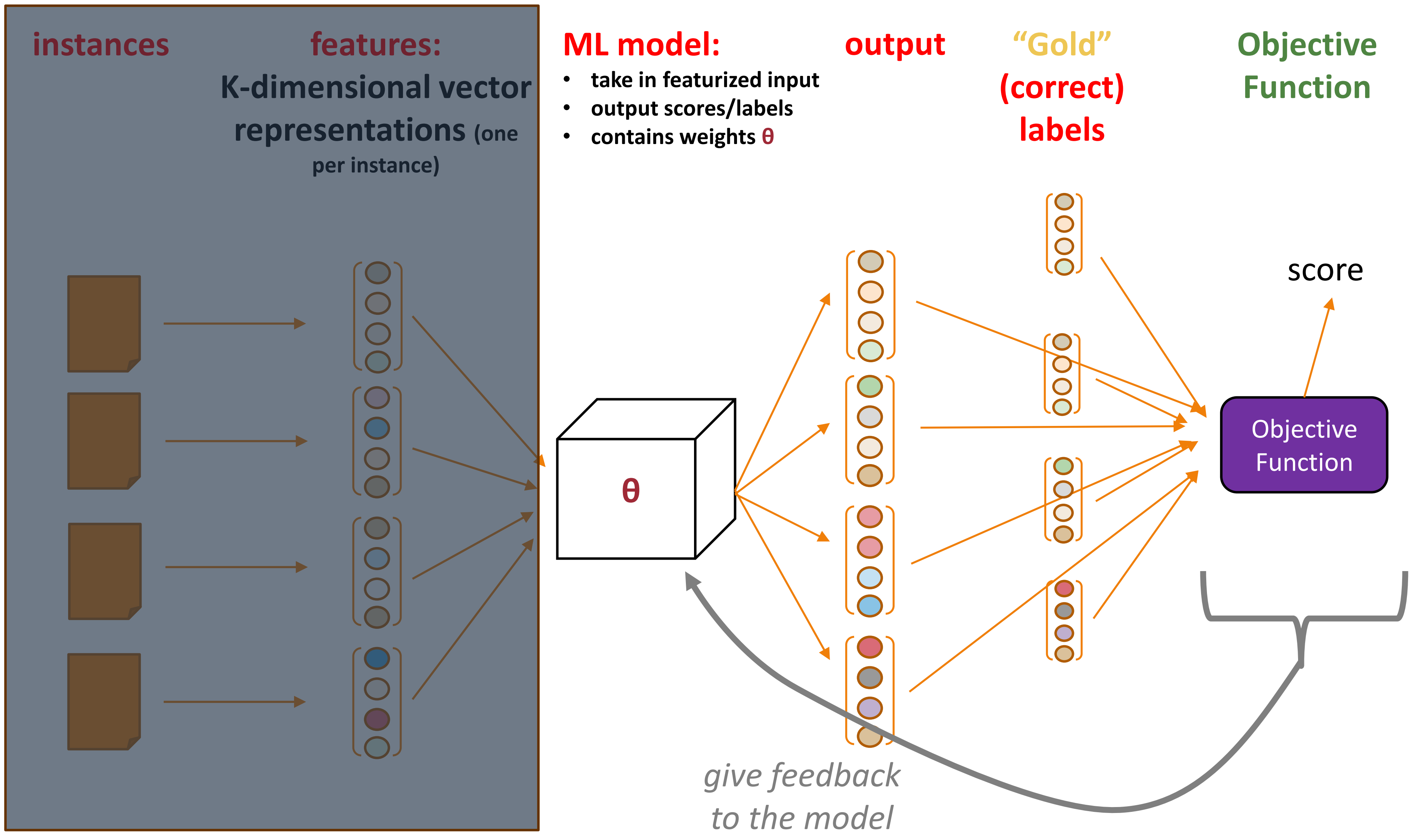


A Neural N-Gram Model (N=3)

The fluffy gray cat meowed very loudly



ML/NLP Framework for Learning



Review:
Maximize Log-Likelihood (Classification)

Original maxent equation

$$\frac{\exp(\theta_y^T f(x))}{\sum_{y'} \exp(\theta_{y'}^T f(x))}$$

$$\log \prod_i p_\theta(y_i|x_i) = \sum_i \log p_\theta(y_i|x_i)$$

Differentiating this becomes nicer (even though Z depends on θ)

Inverse of exp

log(exp(x)) = x

$$= \sum_i \overbrace{\theta_{y_i}^T f(x_i) - \log Z(x_i)}$$

$$= F(\theta)$$

objective is concave

Review: *Minimize* Cross Entropy Loss

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

Model output

True probability (i.e., correct output)

$$L^{\text{xent}}\left(\vec{\hat{y}}, \vec{y}\right) = -\sum_{k=1}^{K} \vec{y}[k] * \log p(y = k|x)$$

Probability distribution from model

index of “1” indicates correct value

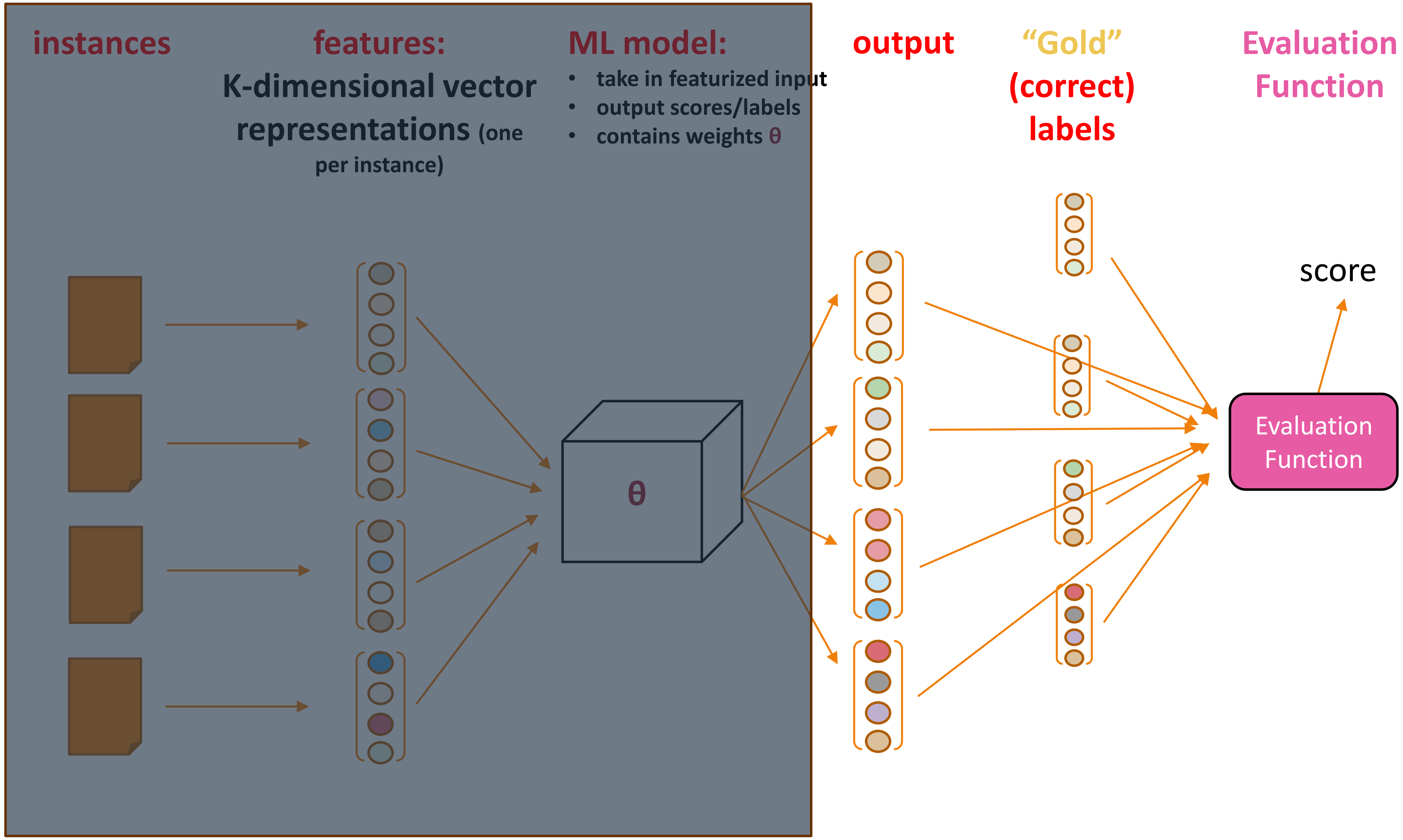
$$\begin{pmatrix} 0 \\ 0 \\ \ldots \\ 1 \\ \ldots \\ 0 \end{pmatrix}$$

one-hot vector

objective is convex

SLP 6.6.1

ML/NLP Framework for Prediction



What are some examples of evaluation functions?

Perplexity in Action

BASELINES

LM Name	N-gram	Params.	Test PPL
Interpolation	3	---	336
Kneser-Ney backoff	3	---	323
Kneser-Ney backoff	5	---	321
Class-based backoff	3	500 classes	312
Class-based backoff	5	500 classes	312

NPLM

N-gram	Word Vector Dim.	Hidden Dim.	Mix with non-neural LM	PPL
5	60	50	No	268
5	60	50	Yes	257
5	30	100	No	276
5	30	100	Yes	252

"we were not able to see signs of over- fitting (on the validation set), possibly because we ran only 5 epochs (over 3 weeks using 40 CPUs)" (Sect. 4.2)

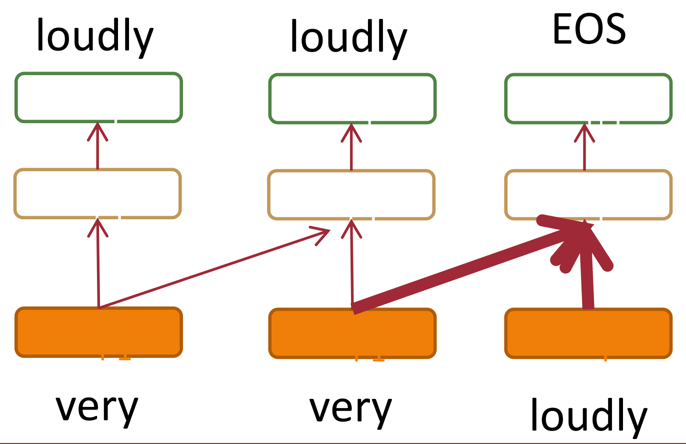
"A Neural Probabilistic Language Model" Bengio et al. (2003)

LM Comparison

N-GRAM/COUNT-BASED	MAXENT/LR	NEURAL
Class-specific	Class-based	Class-based
	Uses features	Uses *embedded* features

A Neural N-Gram Model (N=3)

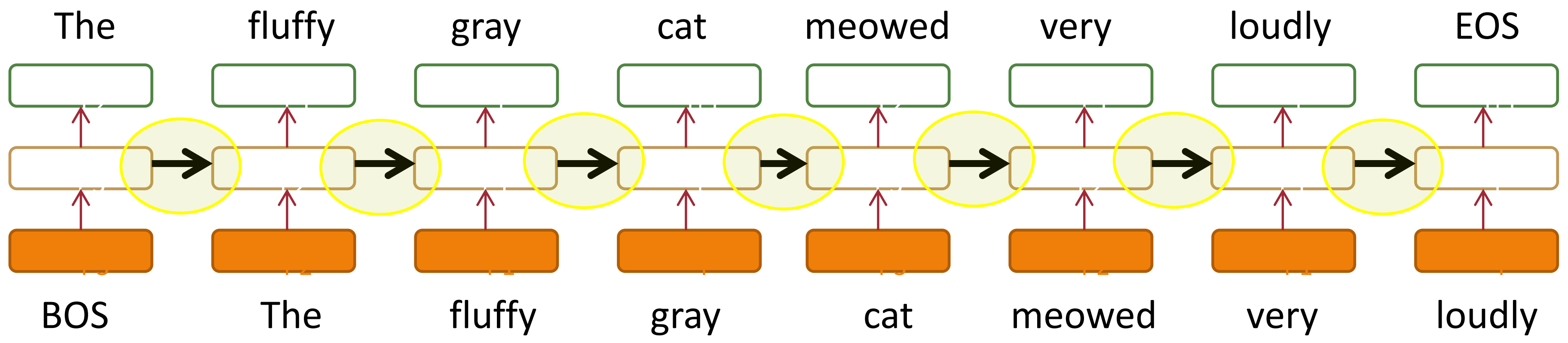
The fluffy gray cat meowed very loudly



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

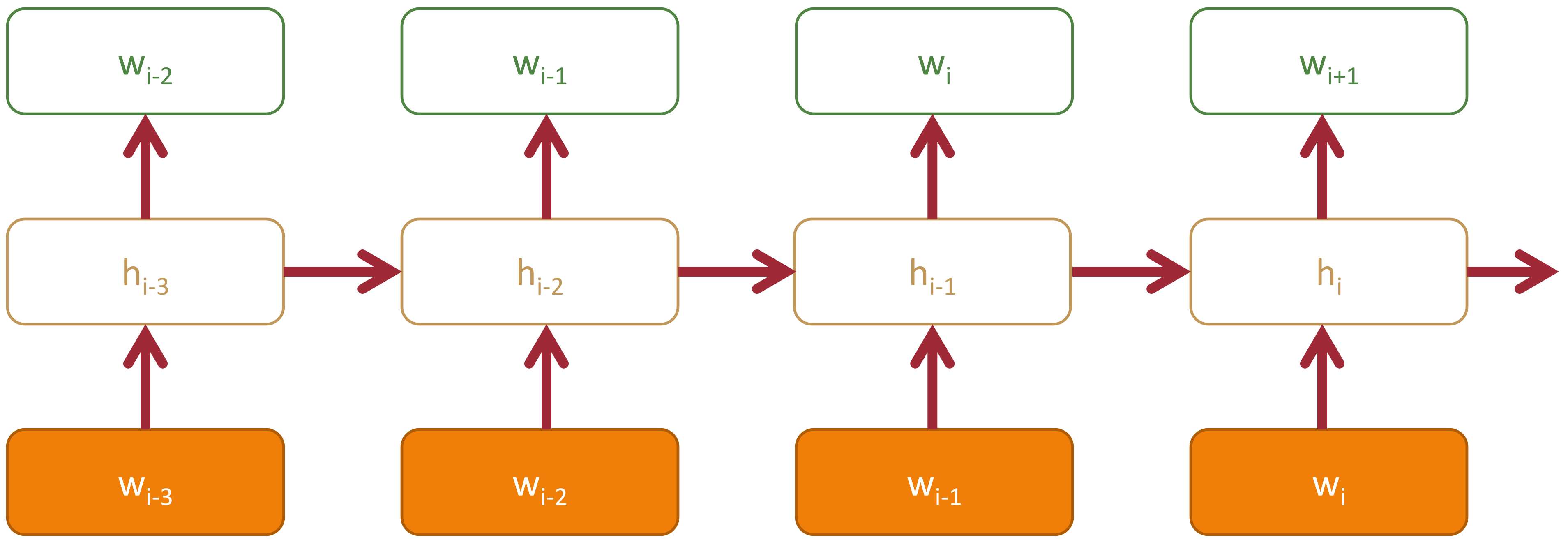
A Recurrent Neural Language Model

The fluffy gray cat meowed very loudly

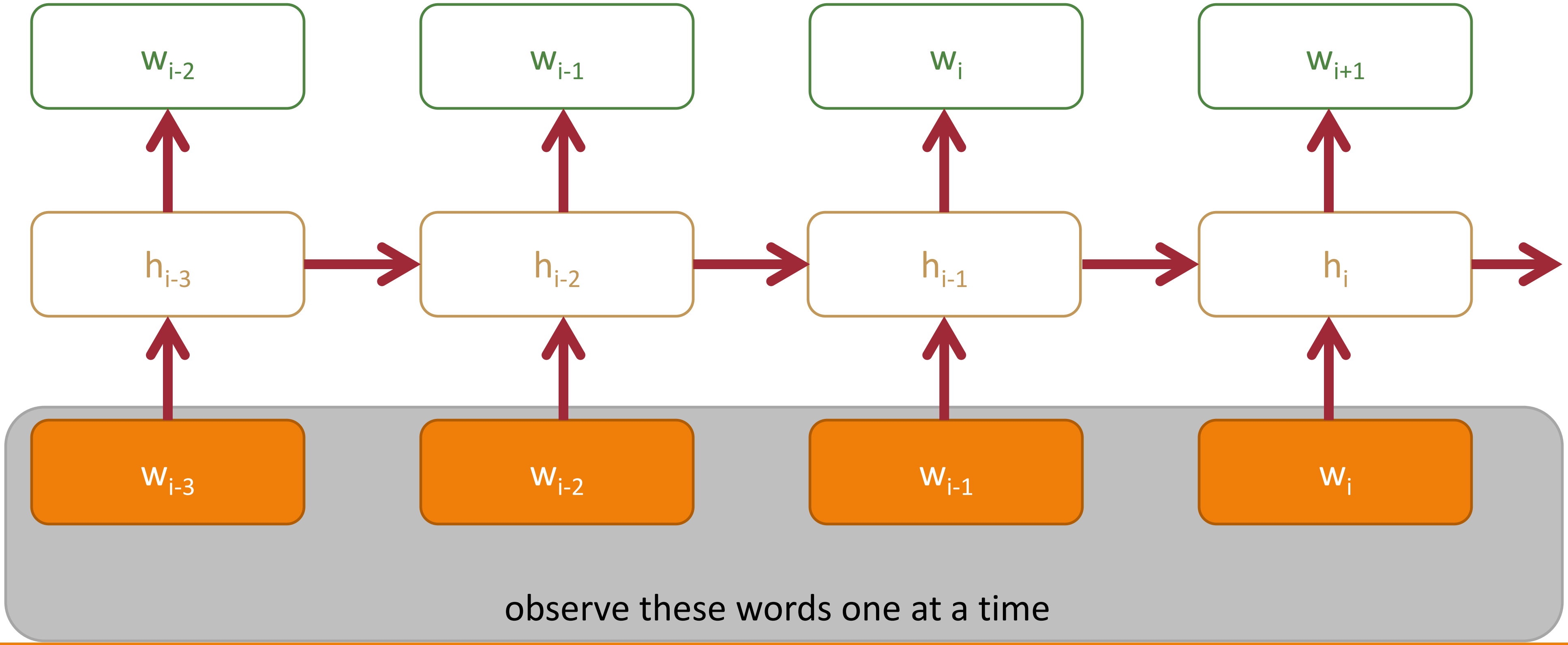


Allowing signal to flow from one hidden state to another could help solve this!

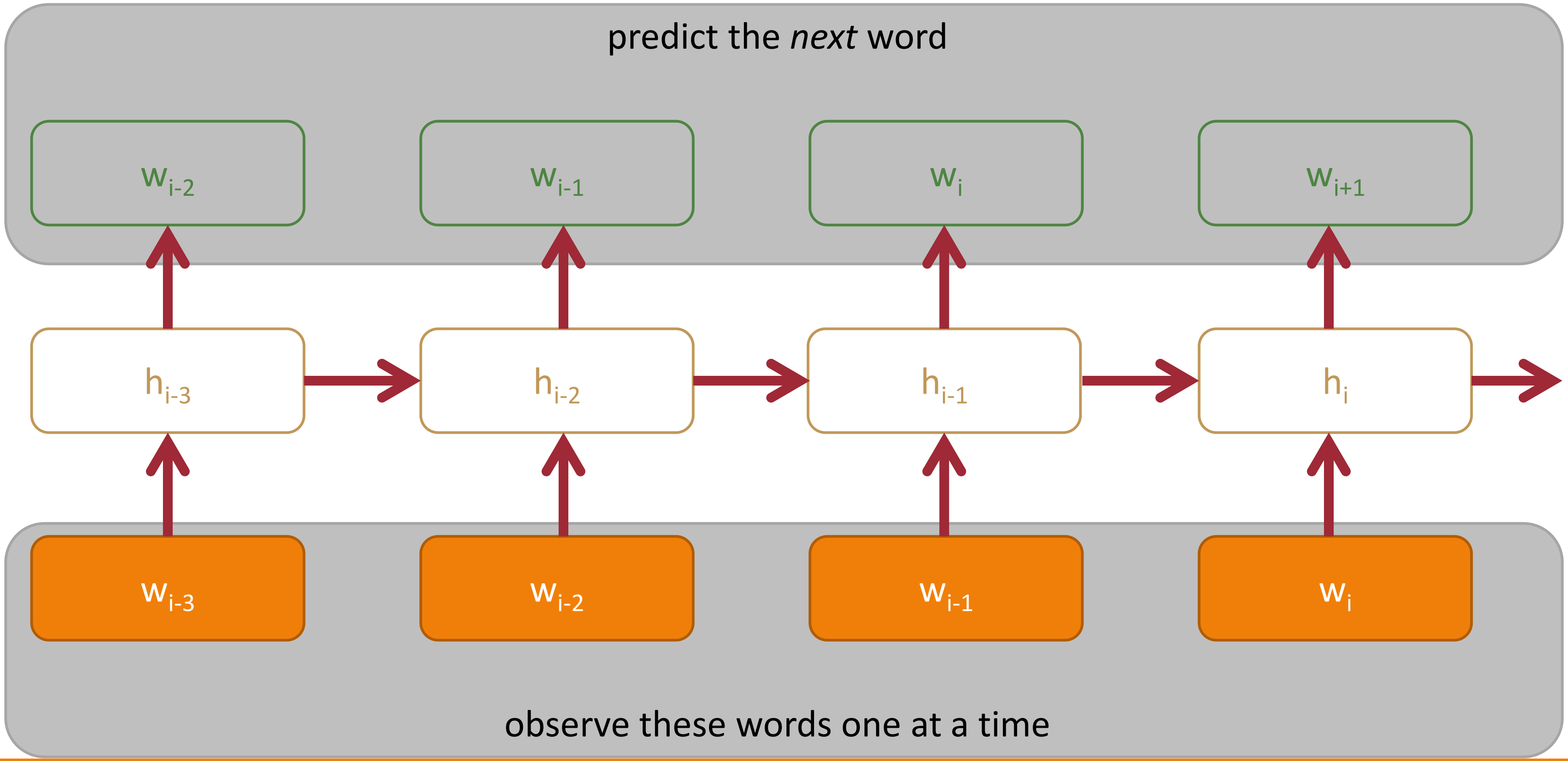
A Classic View of Recurrent Neural Language Modeling



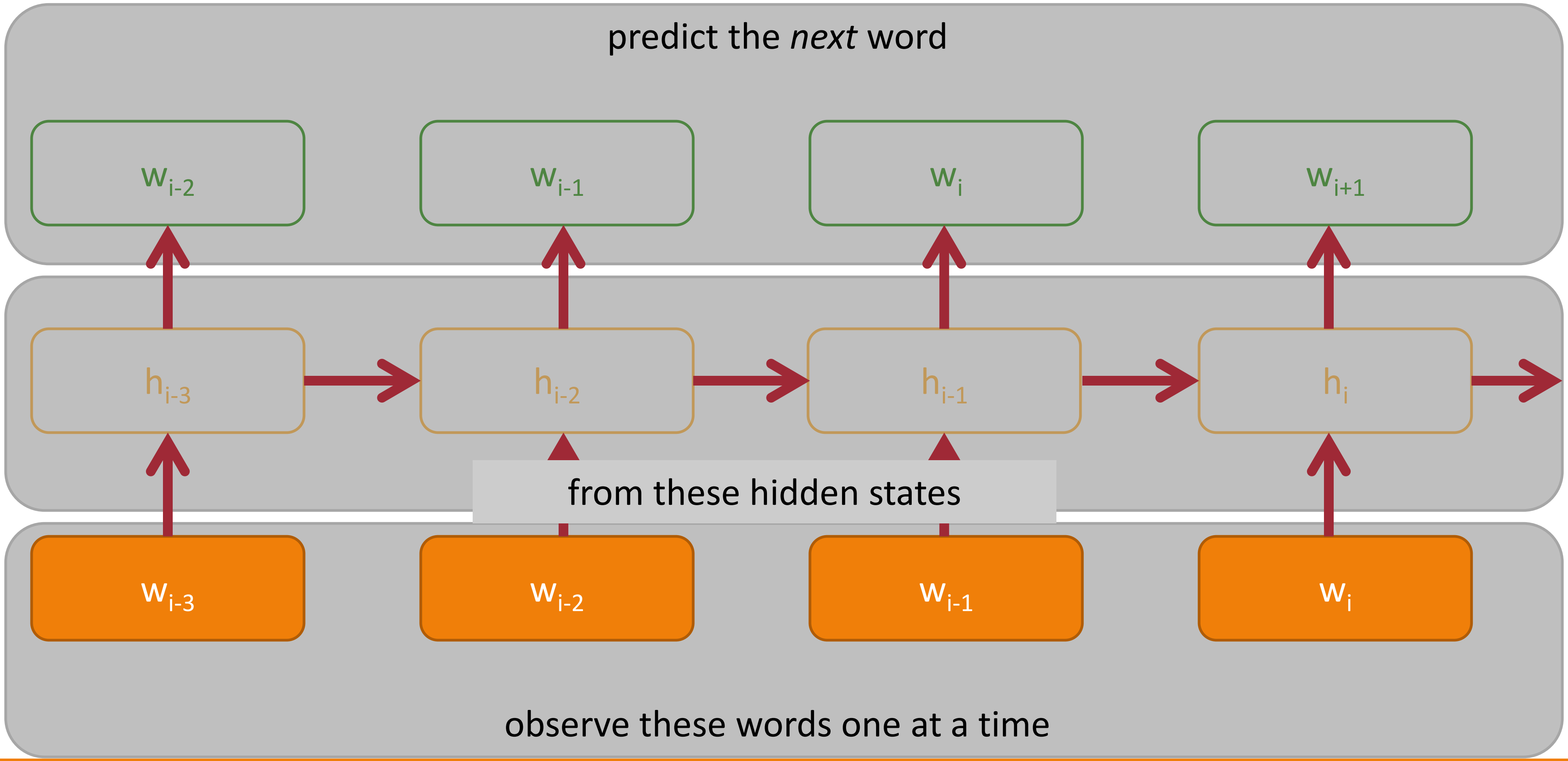
A Classic View of Recurrent Neural Language Modeling



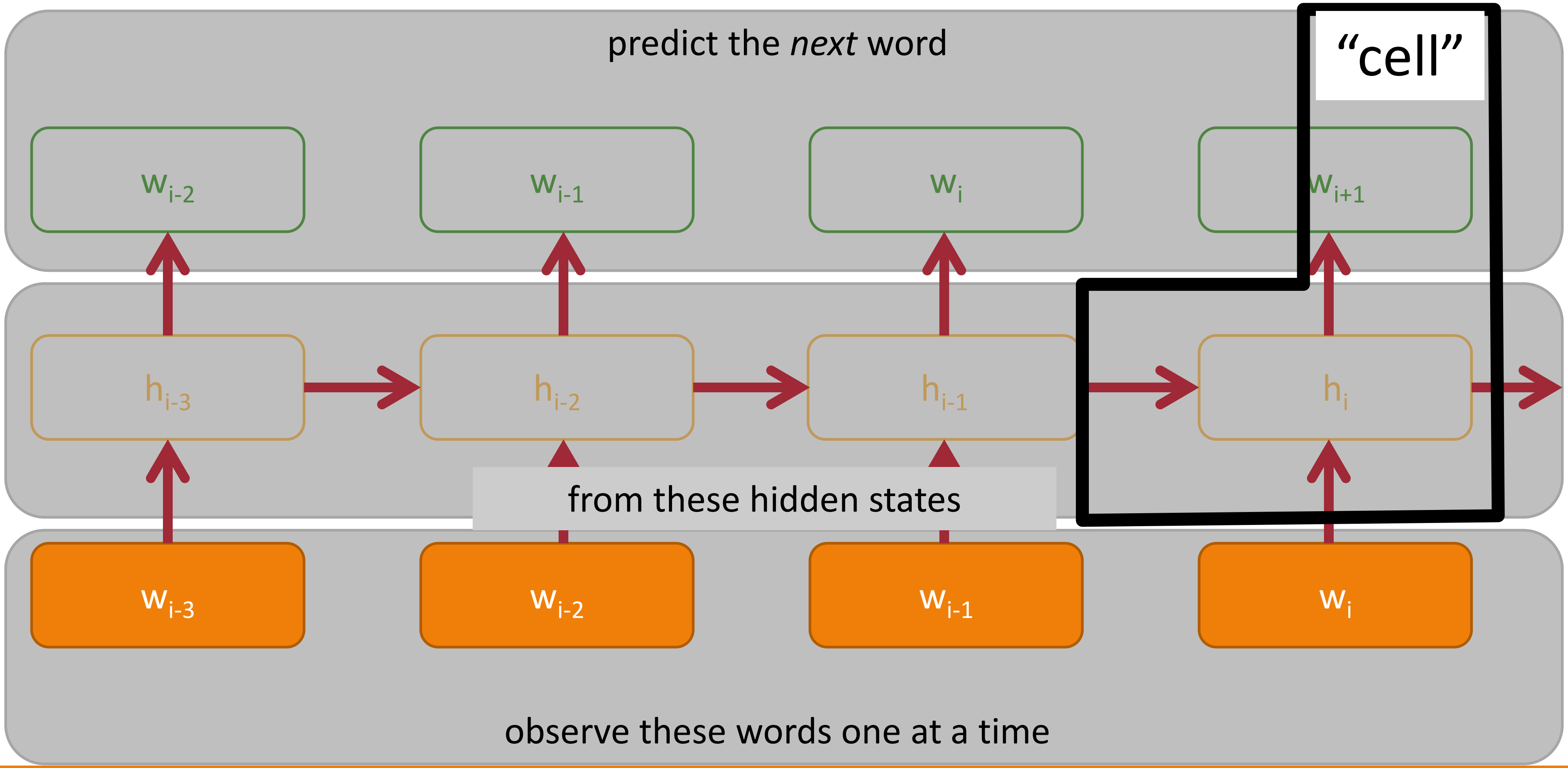
A Classic View of Recurrent Neural Language Modeling



A Classic View of Recurrent Neural Language Modeling



A Classic View of Recurrent Neural Language Modeling



Review: Forward Propagation Example

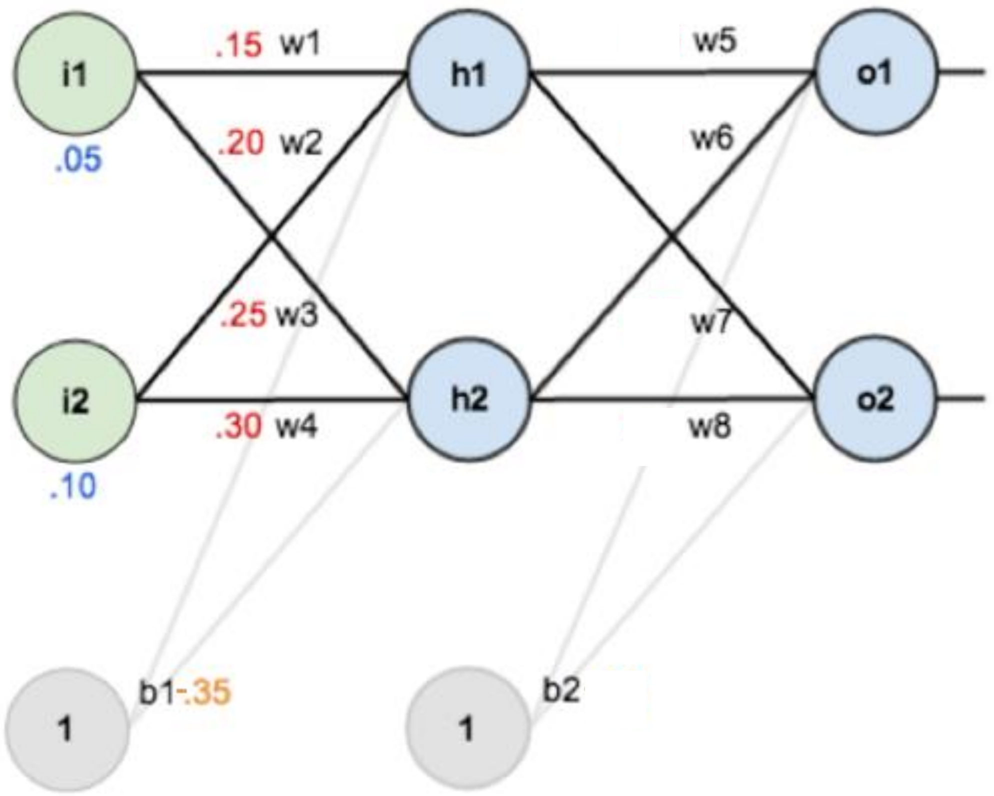
Calculate <u>outputs</u> to the hidden layer (units h1 and h2):

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

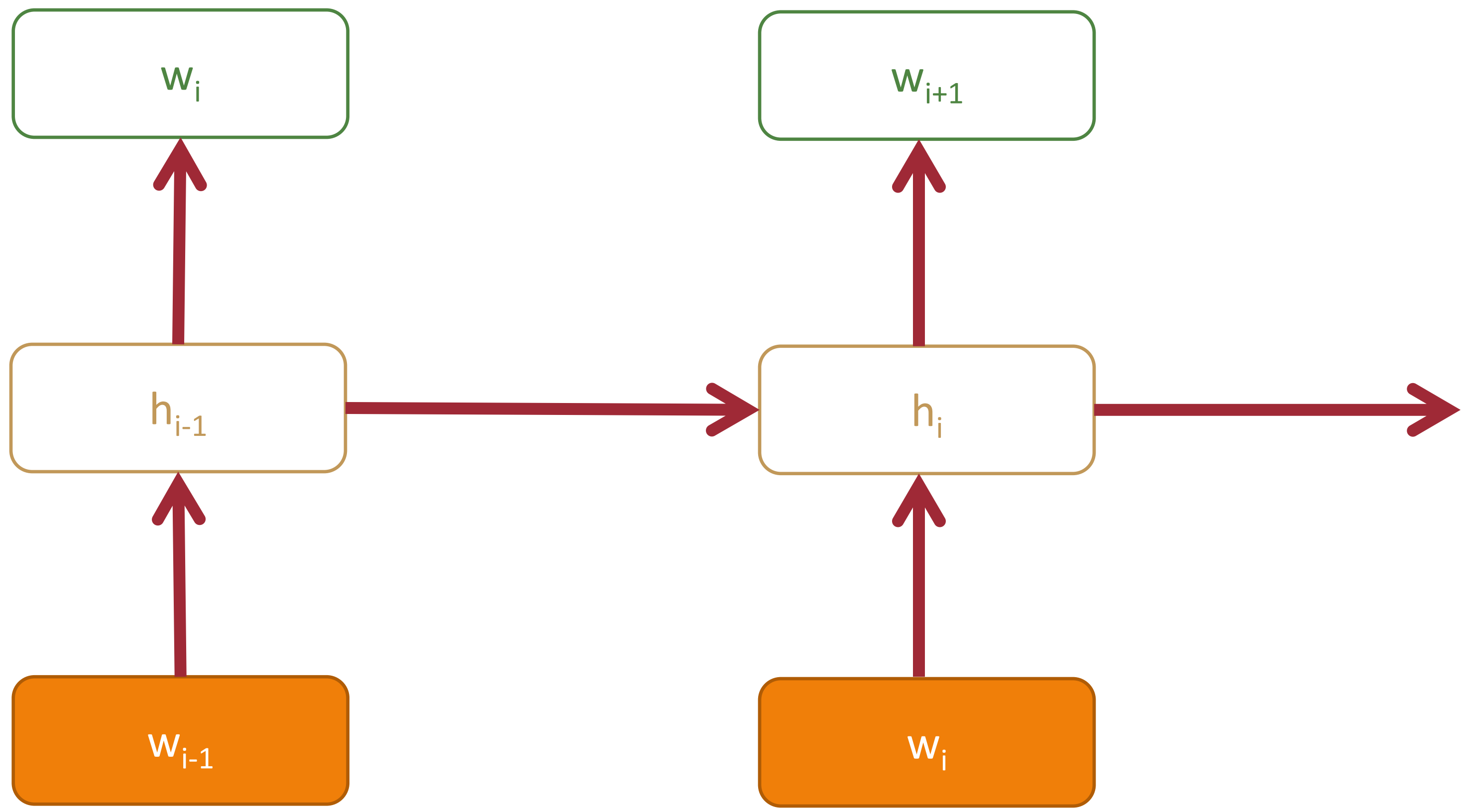


$$out_{h1} = g(in_{h1}) = \frac{1}{1+e^{-in_{h1}}} = \frac{1}{1+e^{-(-.3275)}} = .4188$$

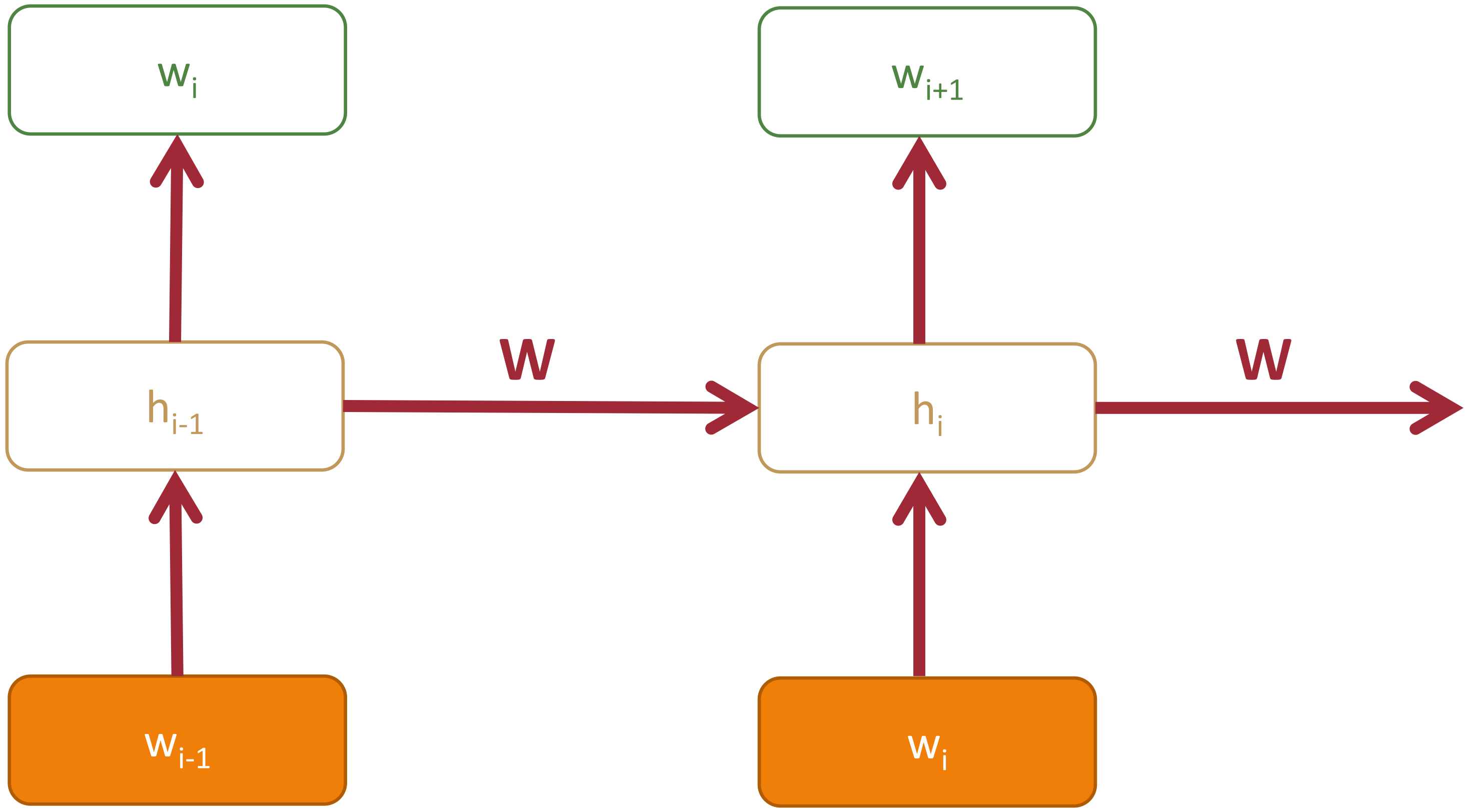
$$out_{h2} = g(in_{h2}) = \frac{1}{1+e^{-in_{h2}}} = \frac{1}{1+e^{-(-.3075)}} = .4237$$

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

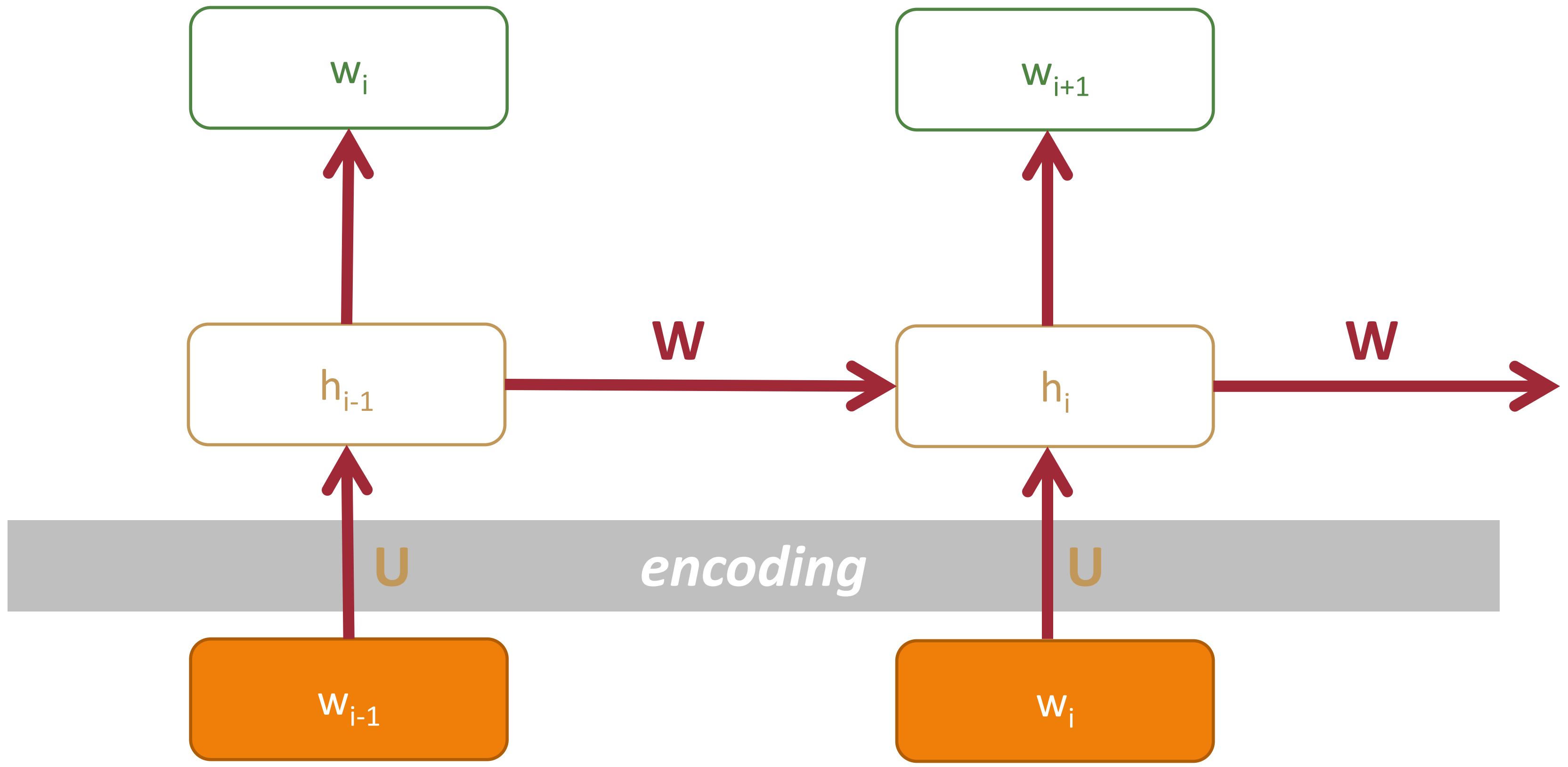
A Recurrent Neural Network Cell



A Recurrent Neural Network Cell

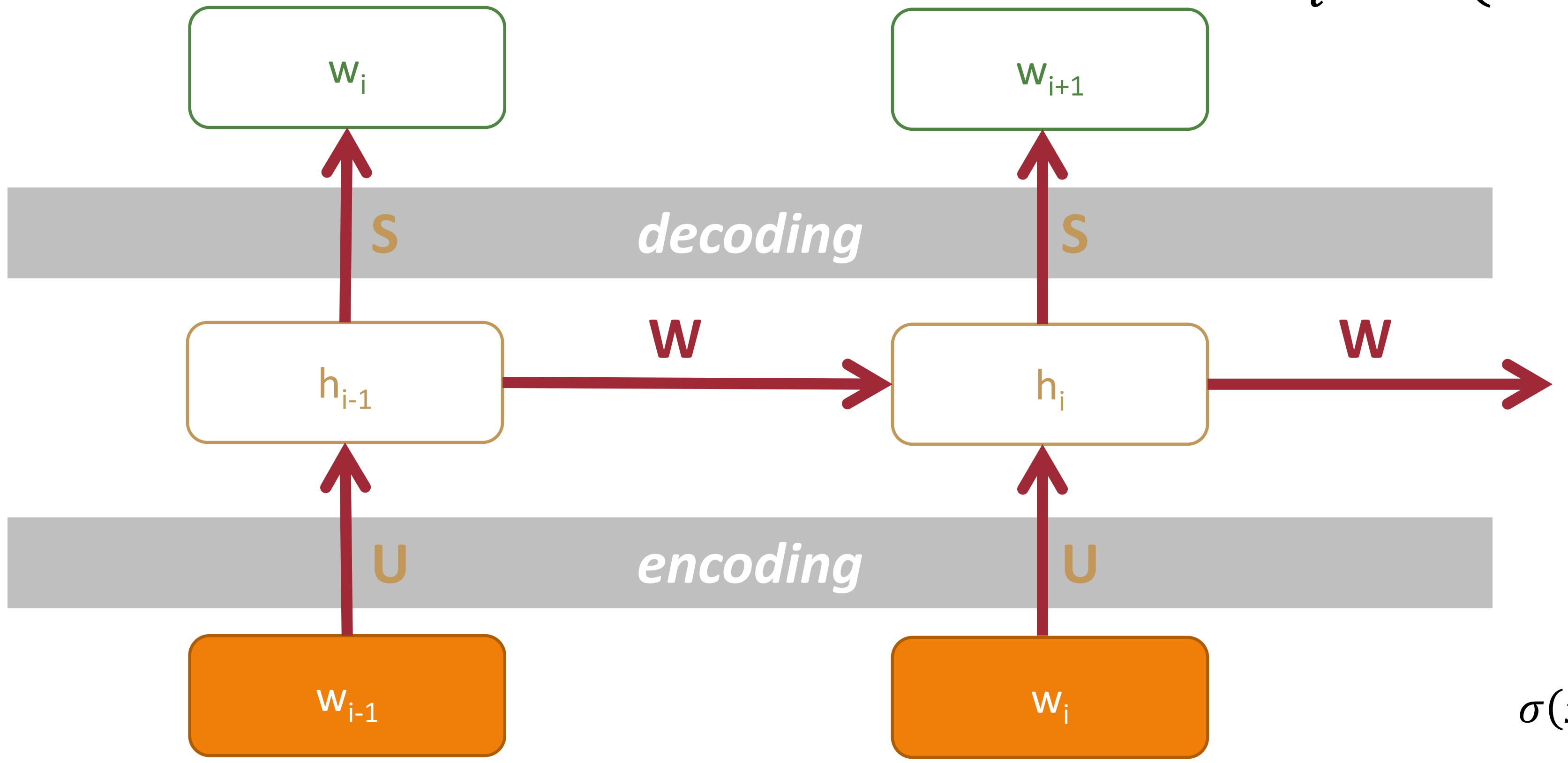


A Recurrent Neural Network Cell



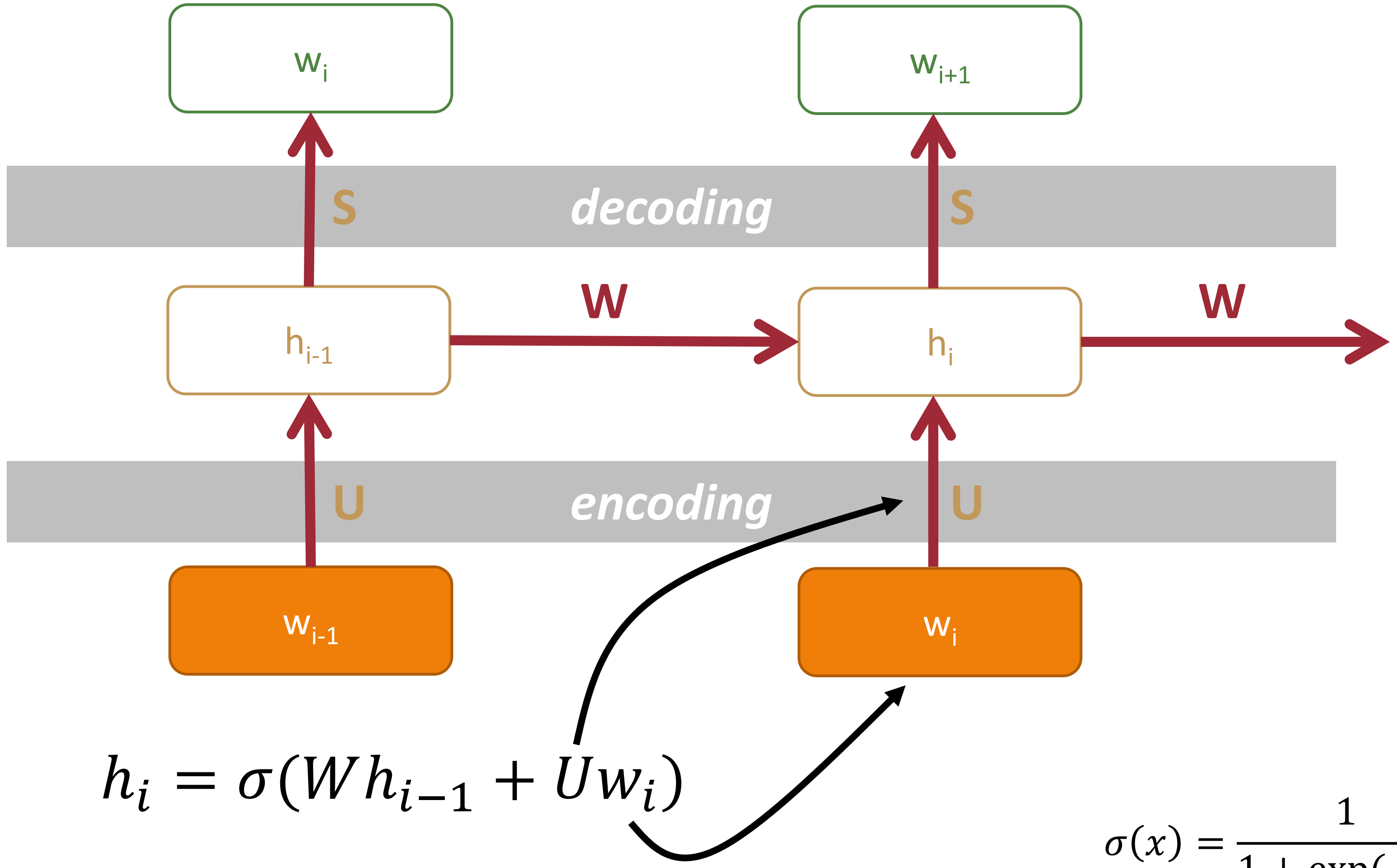
A Recurrent Neural Network Cell

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



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

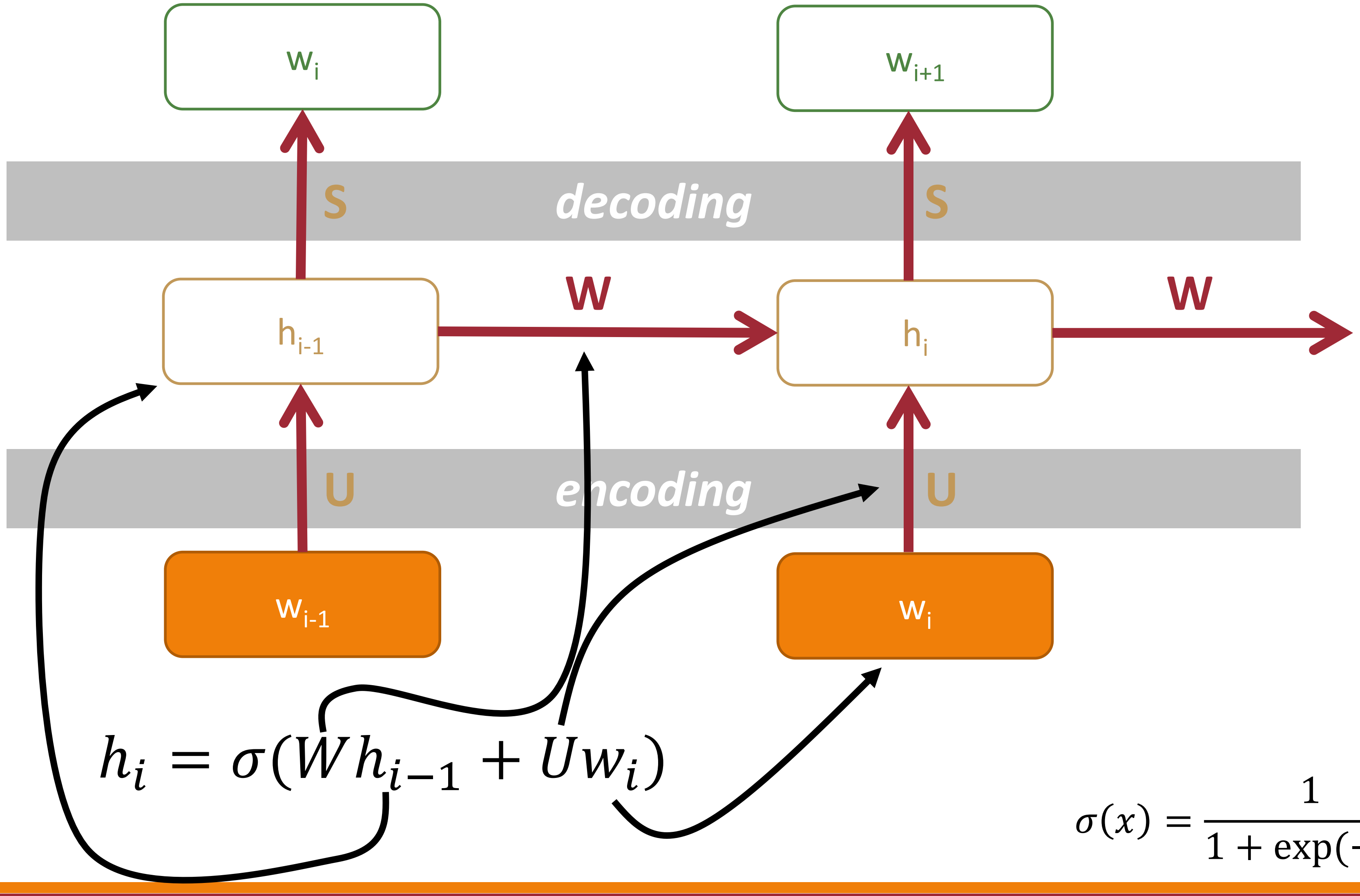
A *Simple* Recurrent Neural Network Cell



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

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

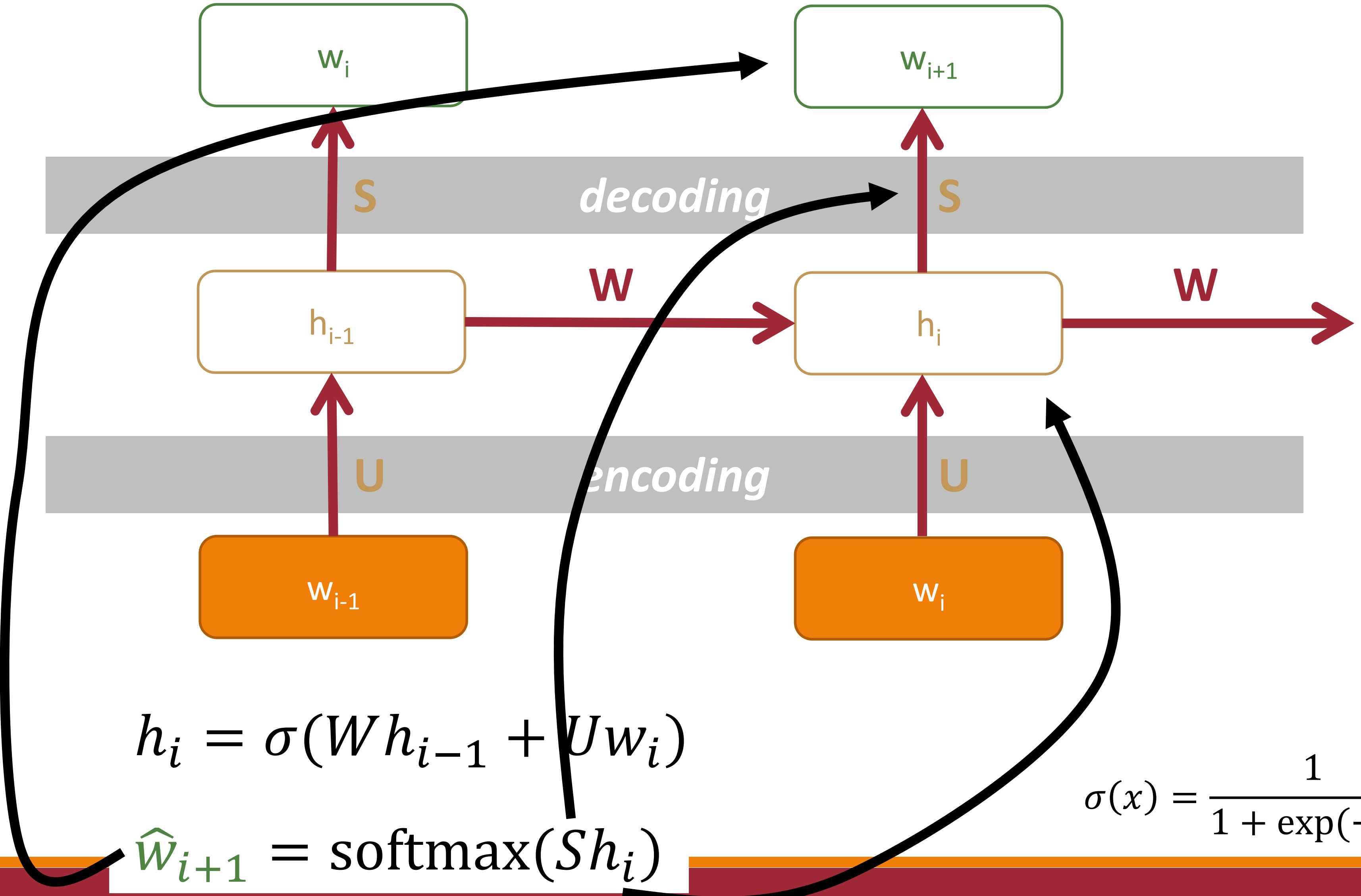
A *Simple* Recurrent Neural Network Cell



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

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

A *Simple* Recurrent Neural Network Cell

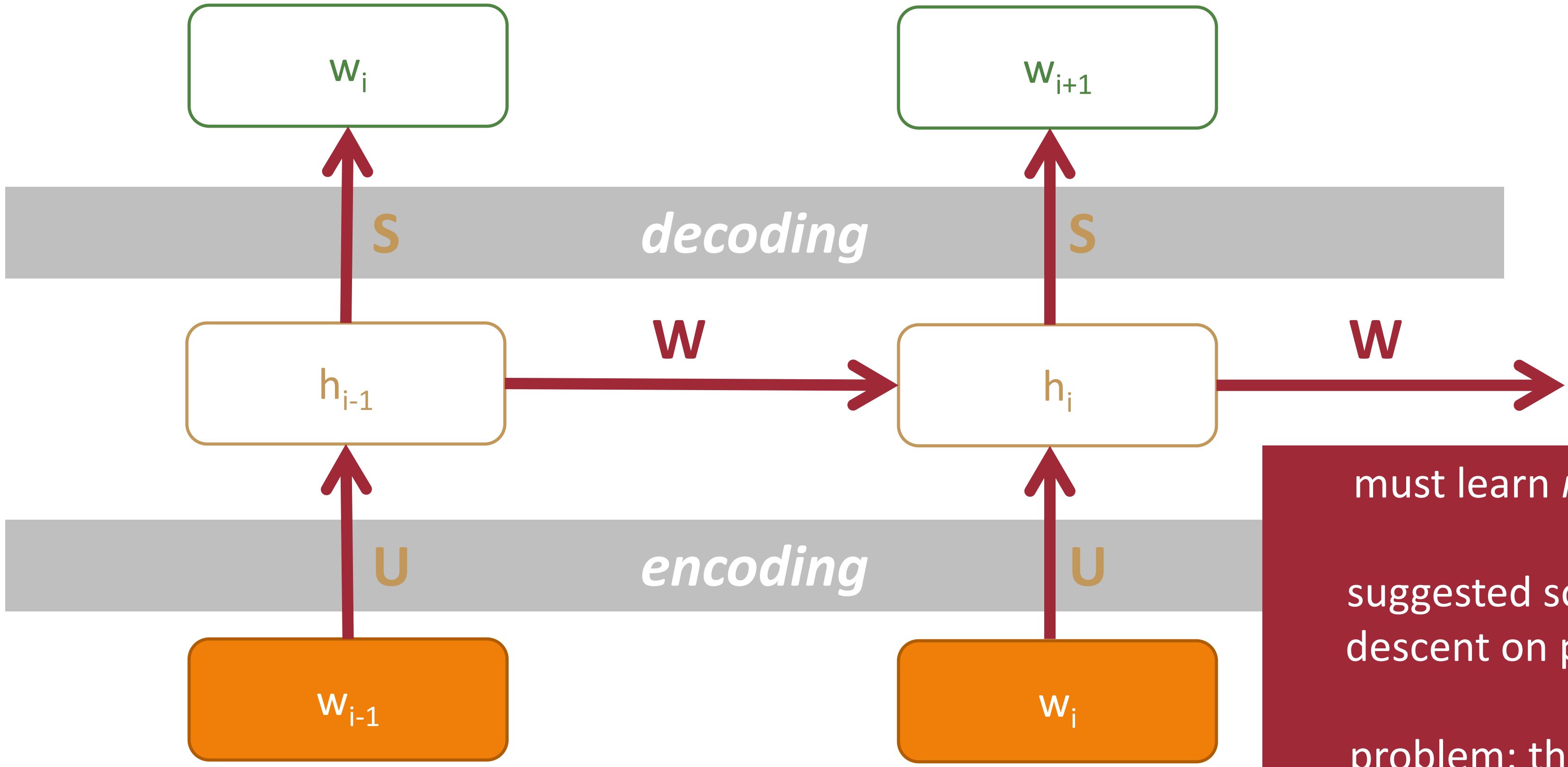


$$h_i = \sigma(W h_{i-1} + U w_i)$$

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

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

A *Simple* Recurrent Neural Network Cell



must learn *matrices* U, S, W

suggested solution: gradient descent on prediction ability

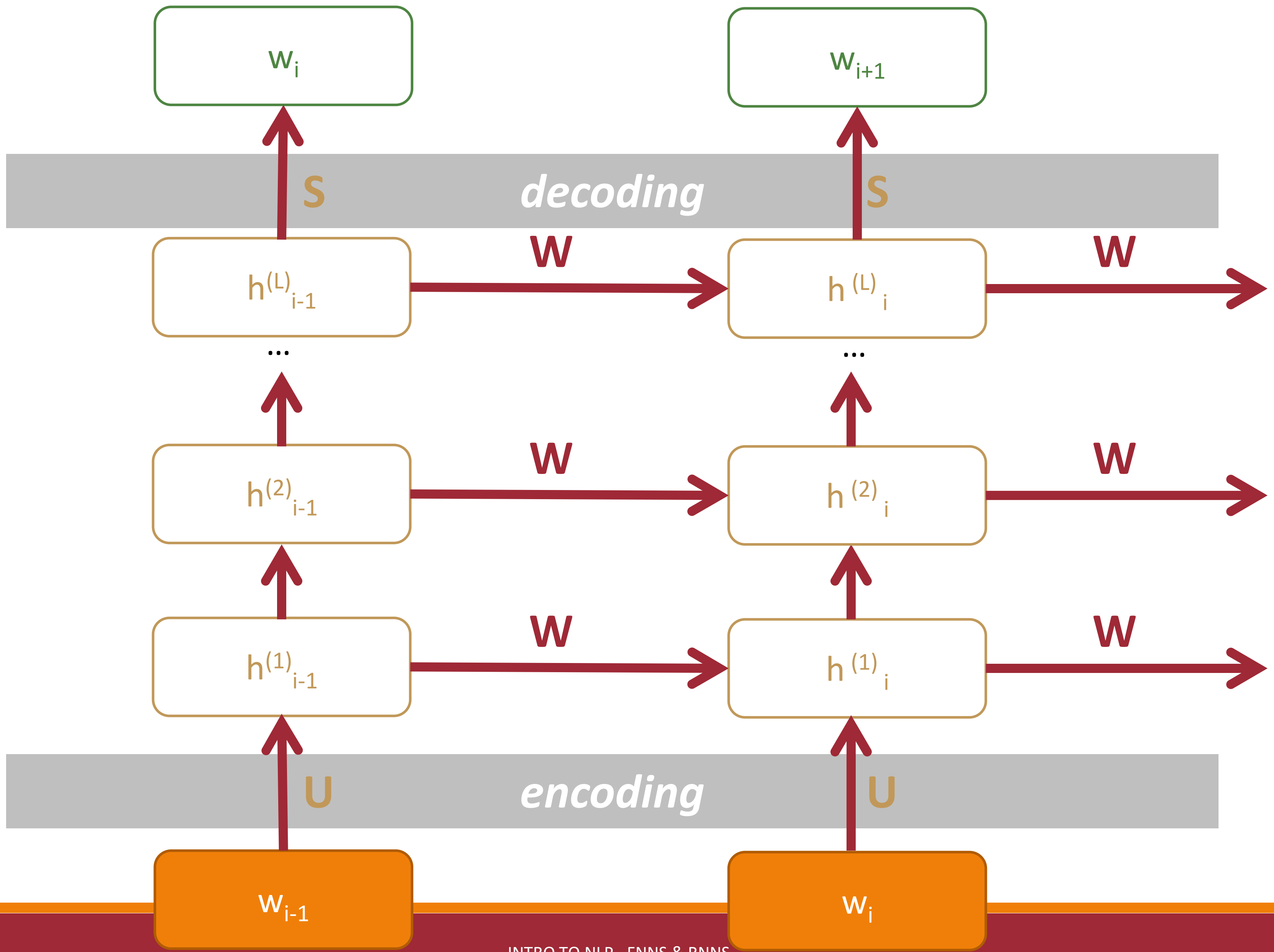
problem: they're *tied* across inputs/timesteps

good news for you: many toolkits do this automatically

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

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

A Multi-Layer *Simple* Recurrent Neural Network Cell



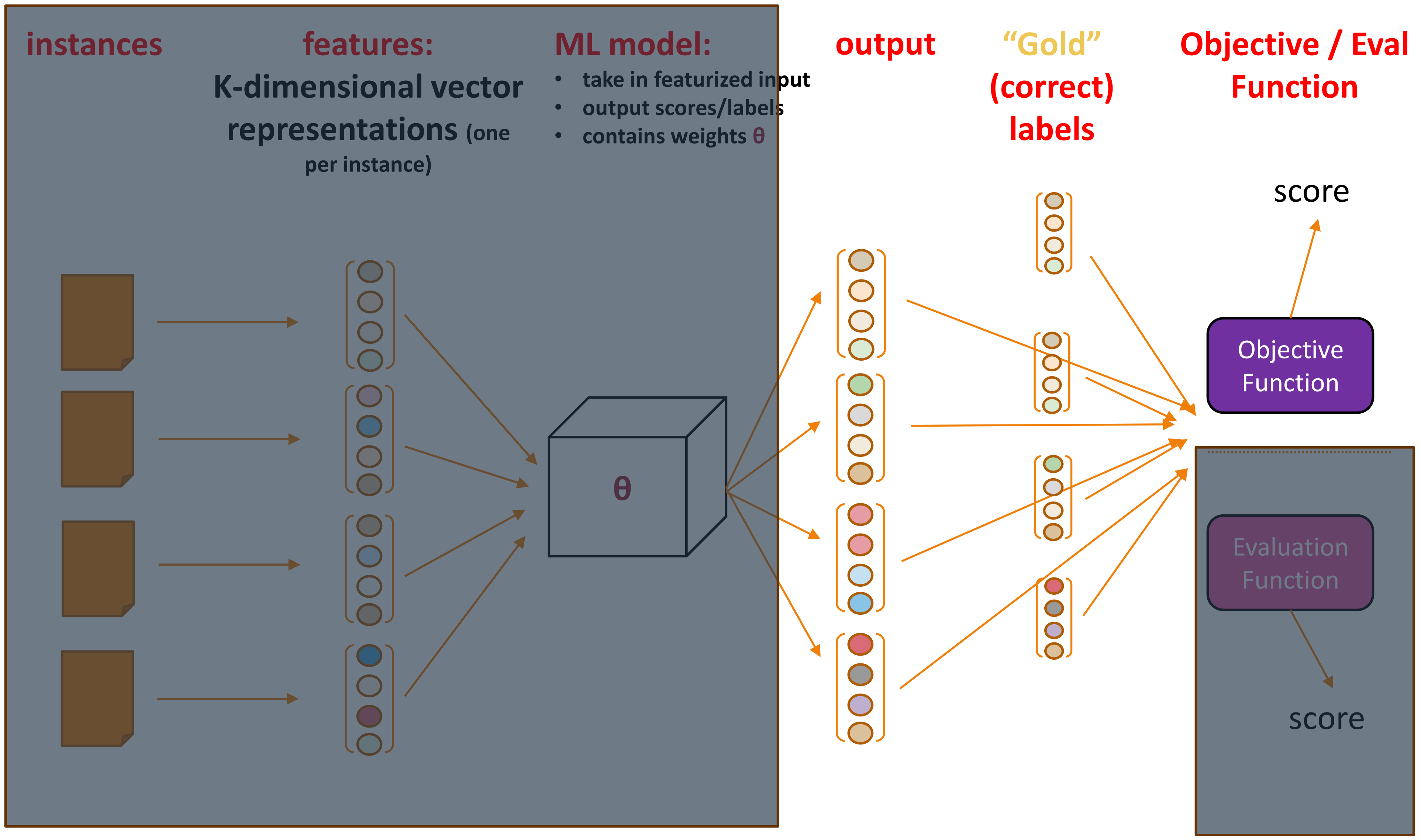
How do you learn an RNN?

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

Model output

True probability (i.e., correct output)

$$L^{\text{xent}}\left(\vec{\hat{y}}, \vec{y}\right) = -\sum_{k=1}^{K} \vec{y}[k] * \log p(y = k|x)$$

index of “1” indicates correct value

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

one-hot vector

Probability distribution from model

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

objective is convex

SLP 6.6.1

Gradient Descent: Backpropagate the Error

```
Initialize model
Set t = 0
Pick a starting value $\theta_t$
Until converged:
  for example(s) sentence i:
    1. Compute loss l on $x_i$
           l = model($x_i$)
    2. Get gradient $g_t = l'(x_i)$
    3. Get scaling factor $\rho_t$
    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 $\theta_t$
Until converged:
  for example(s) sentence i:
    1. Compute loss l on $x_i$
         l = model($x_i$)
    2. Get gradient $g_t = l'(x_i)$
    3. Get scaling factor $\rho_t$
    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

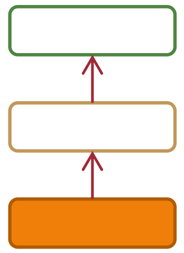
$\log .2$

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

Remember: These probabilities are *computed* as a function of the model parameters!

Gold Label The

Prediction The



BOS

Recurrent NN Loss

(then negate, average)

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

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
…	…

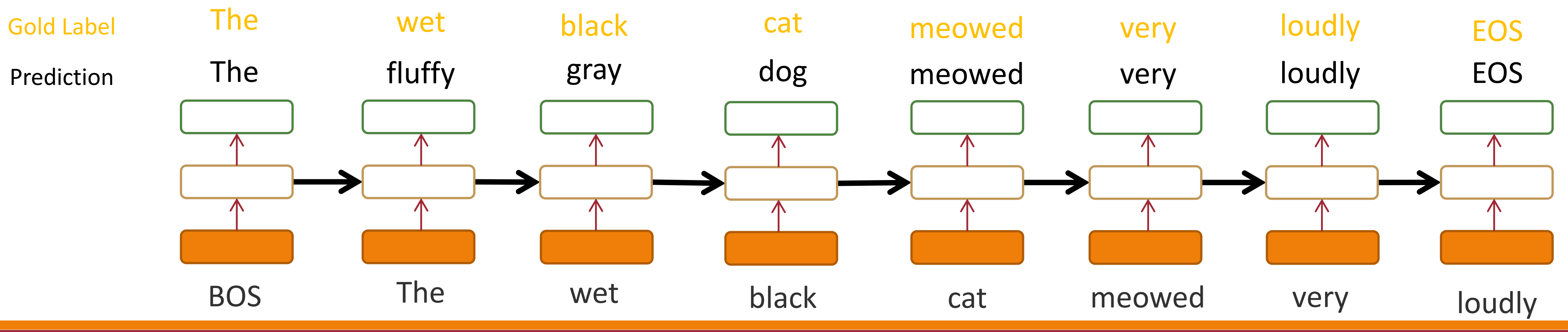
word	prob.
dog	.2
cat	.19
blue	.001
fluffy	.0005
wet	.0005
…	…

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

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
…	…



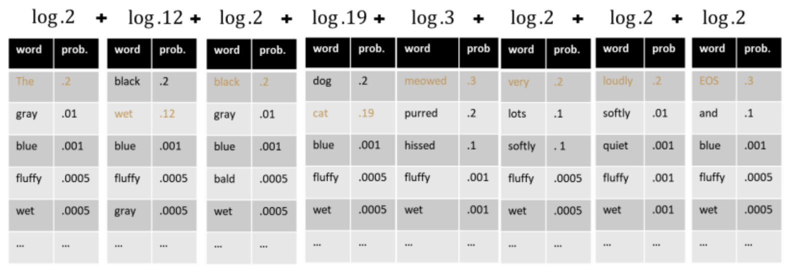
Gradient Descent: Backpropagate the Error

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

Core idea: Train the model to predict what the next word is via maximum likelihood (equivalently, minimizing cross-entropy loss).

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

(then negate, average)



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

word	prob.	word	prob.	word	prob.	word	prob.	word	prob	word	prob.	word	prob	word	prob.
The	.2	black	.2	black	.2	dog	.2	meowed	.3	very	.2	loudly	.2	EOS	.3
gray	.01	wet	.12	gray	.01	cat	.19	purred	.2	lots	.1	softly	.01	and	.1
blue	.001	blue	.001	blue	.001	blue	.001	hissed	.1	softly	. 1	quiet	.001	blue	.001
fluffy	.0005	fluffy	.0005	bald	.0005	fluffy	.0005	fluffy	.001	fluffy	.0005	fluffy	.001	fluffy	.0005
wet	.0005	gray	.0005	wet	.0005	wet	.0005	wet	.001	wet	.0005	wet	.001	wet	.0005
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

Gradient Descent: Backpropagate the Error

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

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

batch

epoch

epoch: a single run over all training data

batch: a run over a subset of the data

Flavors of Gradient Descent

“Online”

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

“Batch”

```
Set $t = 0$
Pick a starting value $\theta_t$
Until converged:
    get batch $B \subset$ full data
    set $g_t = 0$
    for example(s) i in B:
        1. Compute loss l on $x_i$
        2. Accumulate gradient
           $g_t$ += $l'(x_i)$
    done
    Get scaling factor $\rho_t$
    Set $\theta_{t+1} = \theta_t - \rho_t * g_t$
    Set t += 1
```

“Epoch”

```
Set $t = 0$
Pick a starting value $\theta_t$
Until converged:

    set $g_t = 0$
    for example(s) i in full data:
        1. Compute loss l on $x_i$
        2. Accumulate gradient
           $g_t$ += $l'(x_i)$
    done
    Get scaling factor $\rho_t$
    Set $\theta_{t+1} = \theta_t - \rho_t * g_t$
    Set t += 1
```

Why Is Training RNNs Hard?

Conceptually, it can get strange

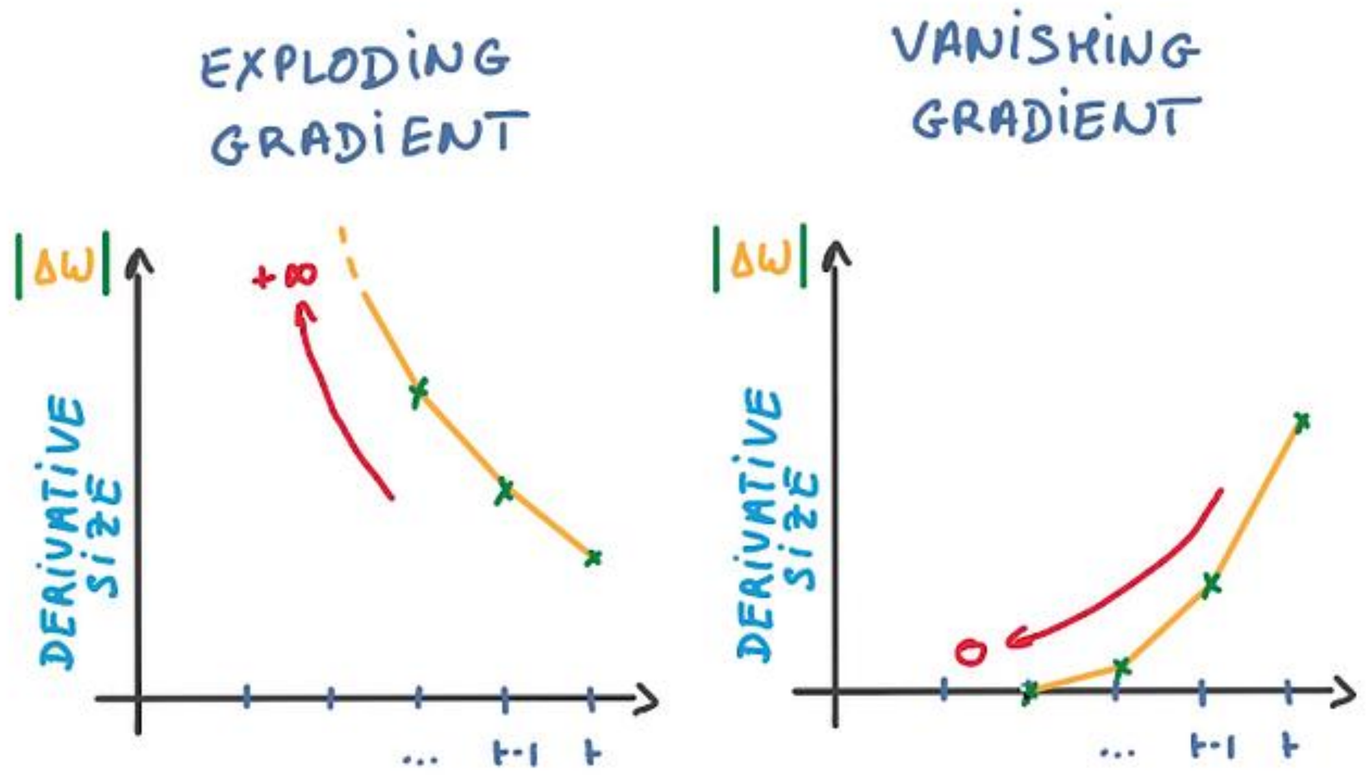
But really getting the gradient just requires many applications of the chain rule for derivatives

Vanishing/Exploding gradients

- Multiply the *same* matrices at *each* timestep ➔ multiply *many* matrices in the gradients
- Causes the network to forget information from many timesteps back

One solution: clip the gradients to a min/max value

Vanishing Gradients



https://miro.medium.com/v2/resize:fit:700/0*pq5wlxZW4zvD9iJH

PyTorch RNN LMs

Pick Your Toolkit

PyTorch

Deeplearning4j

TensorFlow

Caffe

Keras

MXNet

Torch

…

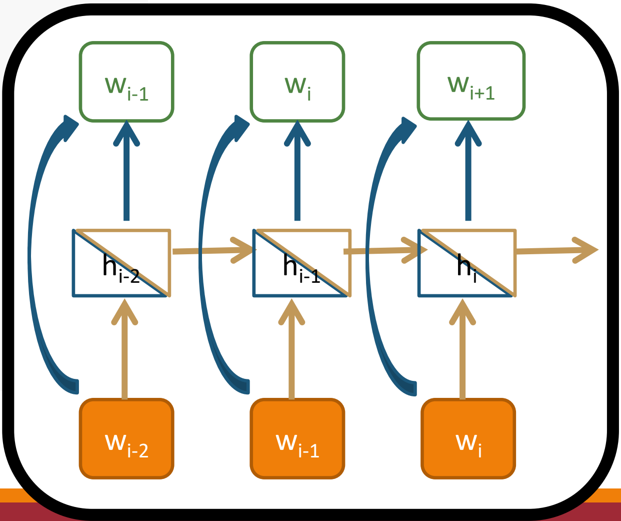
Comparisons:

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

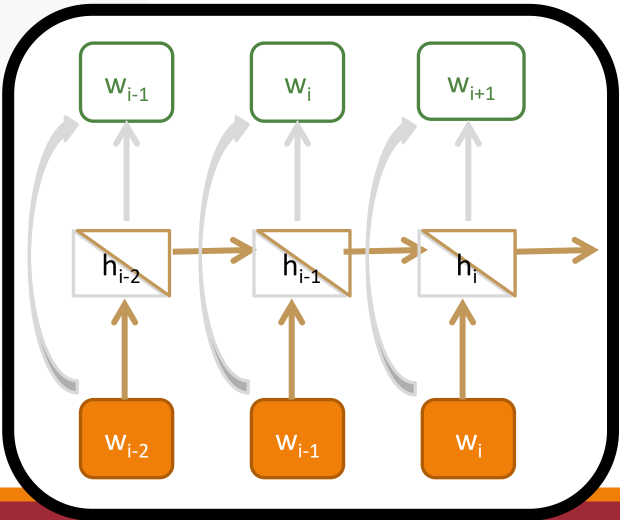
```python
import torch.nn as nn
import torch.nn.functional as F

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
```



Defining A Simple RNN in Python

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

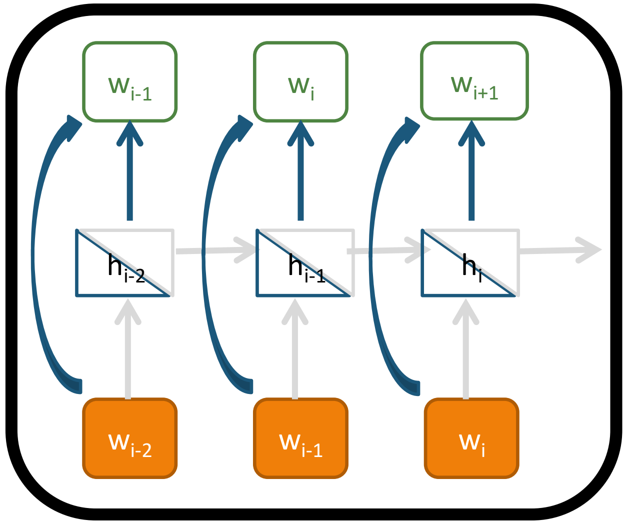
```
import torch.nn as nn
import torch.nn.functional as F

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

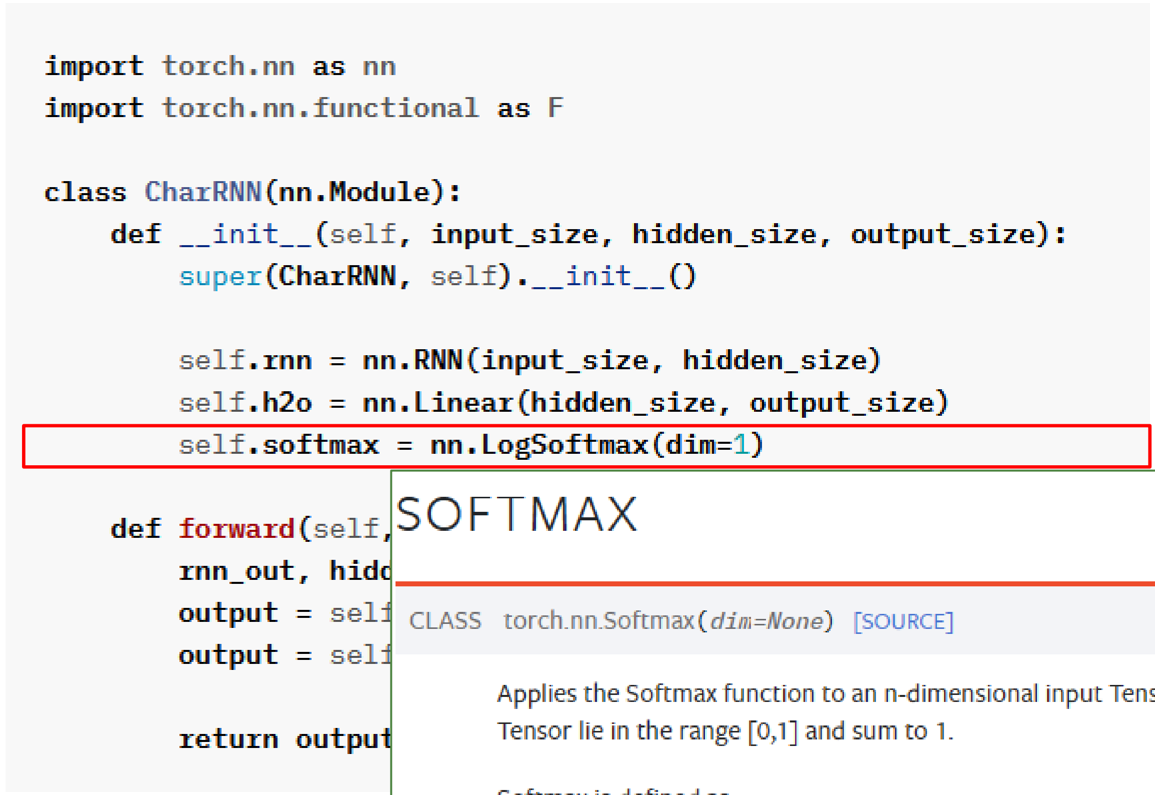
http://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

```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
```

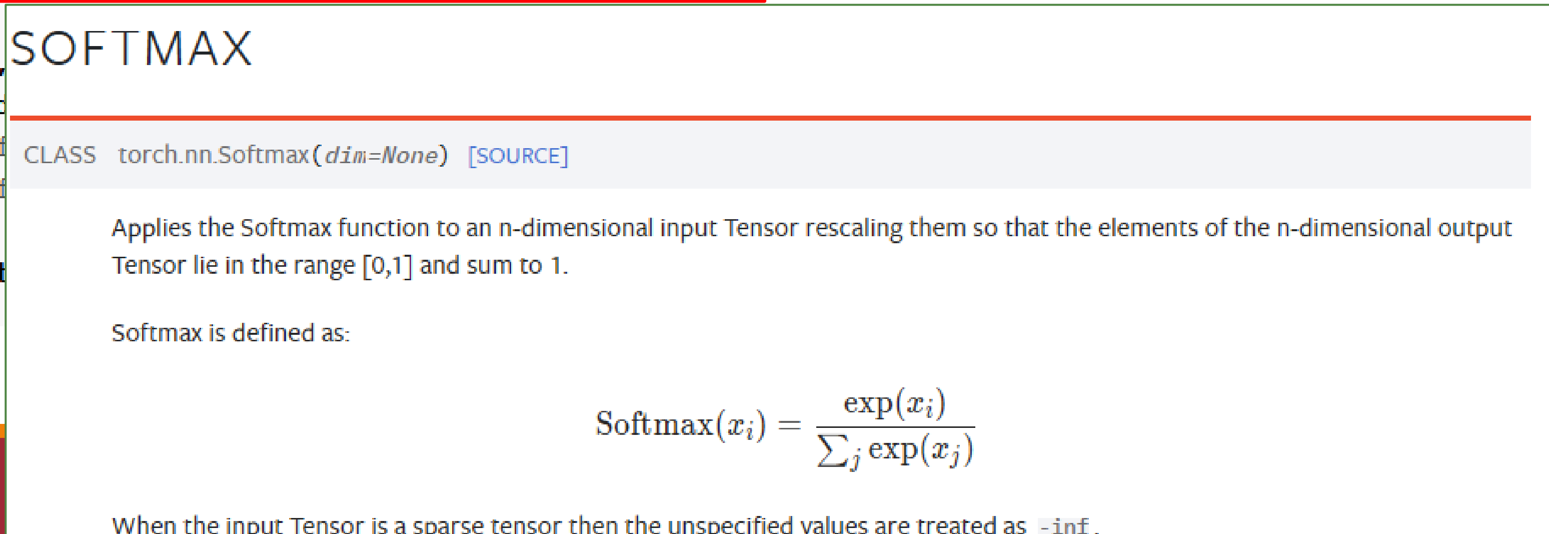


Defining A Simple RNN in Python

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, hid
        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:

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

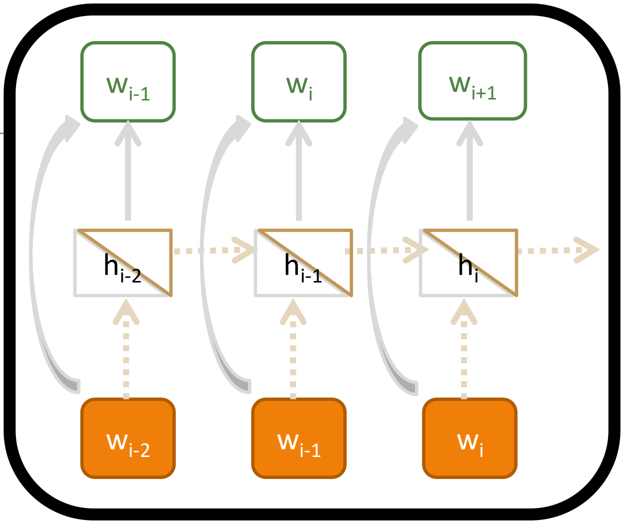
When the input Tensor is a sparse tensor then the unspecified values are treated as `-inf`.

Defining A Simple RNN in Python

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

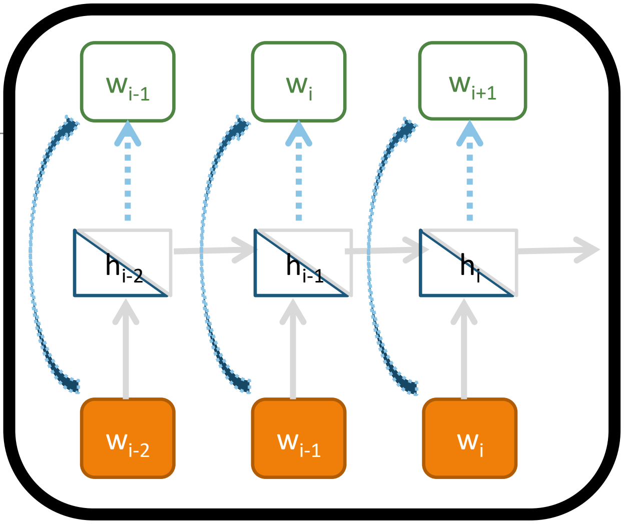
```python
import torch.nn as nn
import torch.nn.functional as F

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
```

encode



Defining A Simple RNN in Python

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



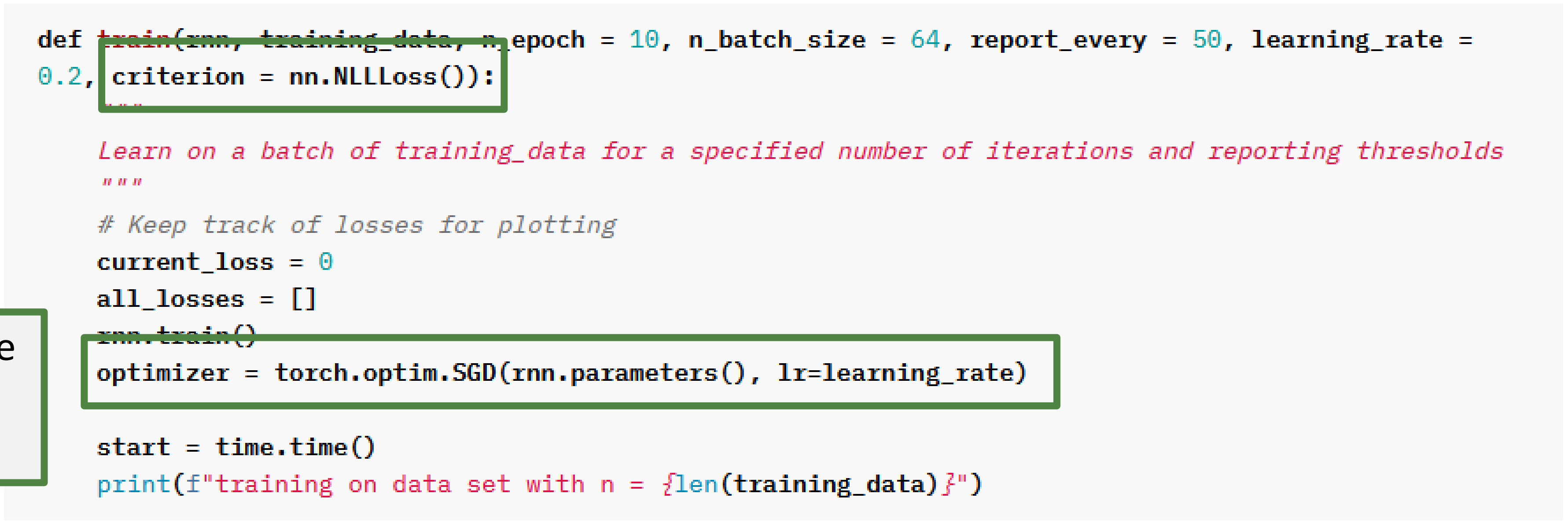
```
import torch.nn as nn
import torch.nn.functional as F

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
```

decode

Training A Simple RNN in Python

Negative log-likelihood

(we'll talk about this)



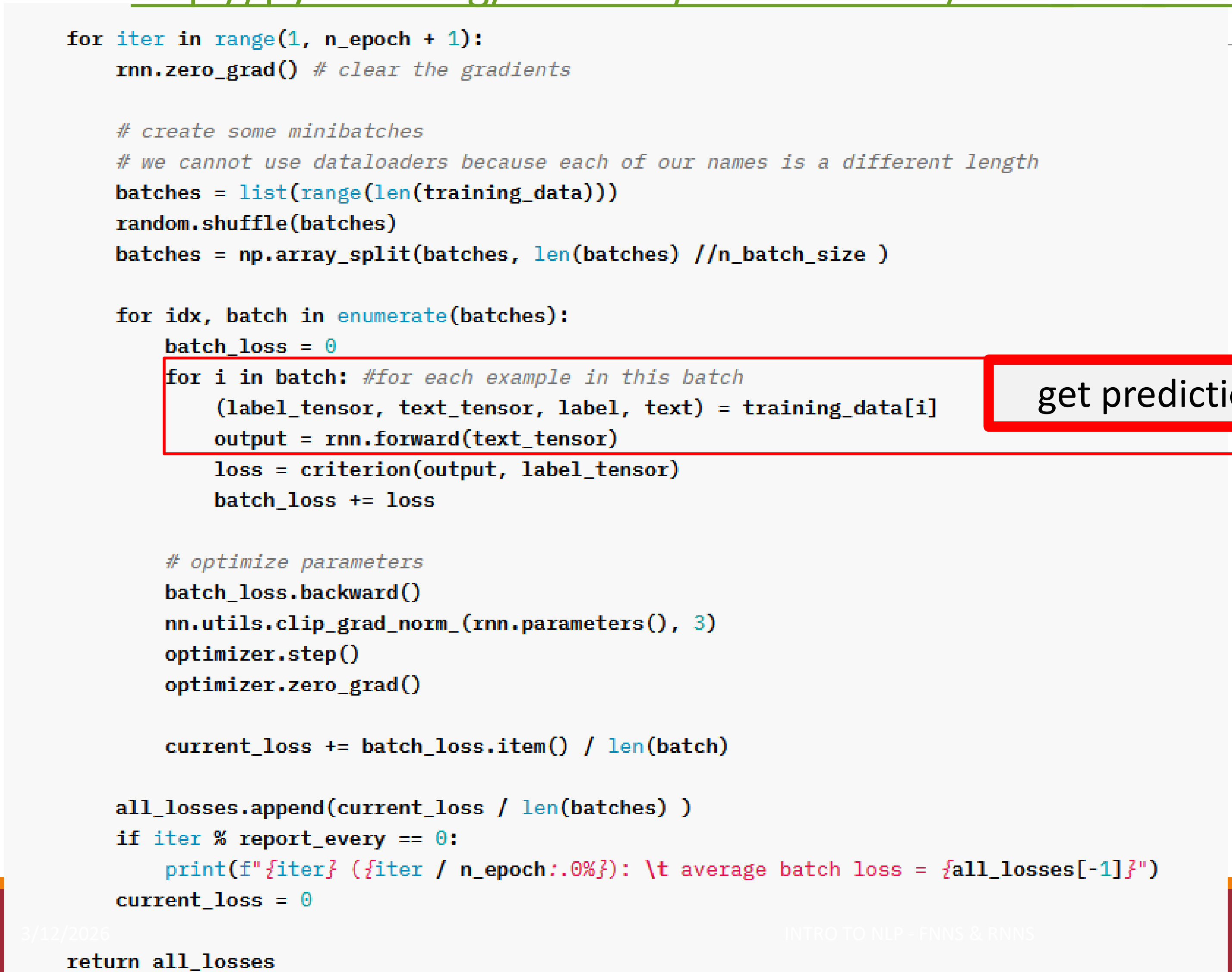
```
def train(rnn, training_data, n_epoch = 10, n_batch_size = 64, report_every = 50, learning_rate =
0.2, criterion = nn.NLLLoss()):
    """
    Learn on a batch of training_data for a specified number of iterations and reporting thresholds
    """
    # Keep track of losses for plotting
    current_loss = 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

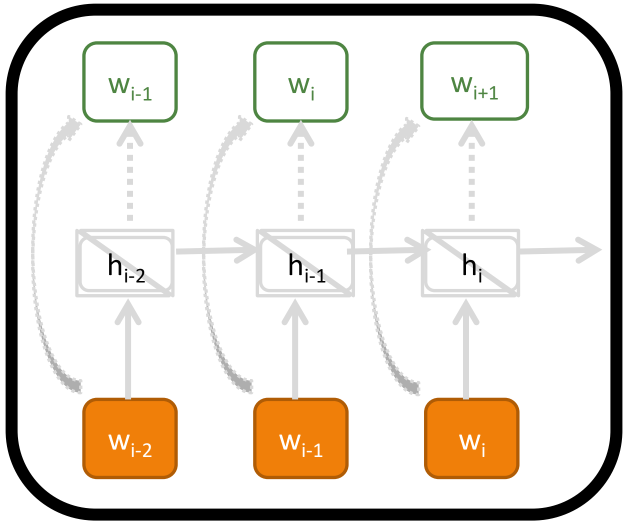
Training A Simple RNN in Python

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



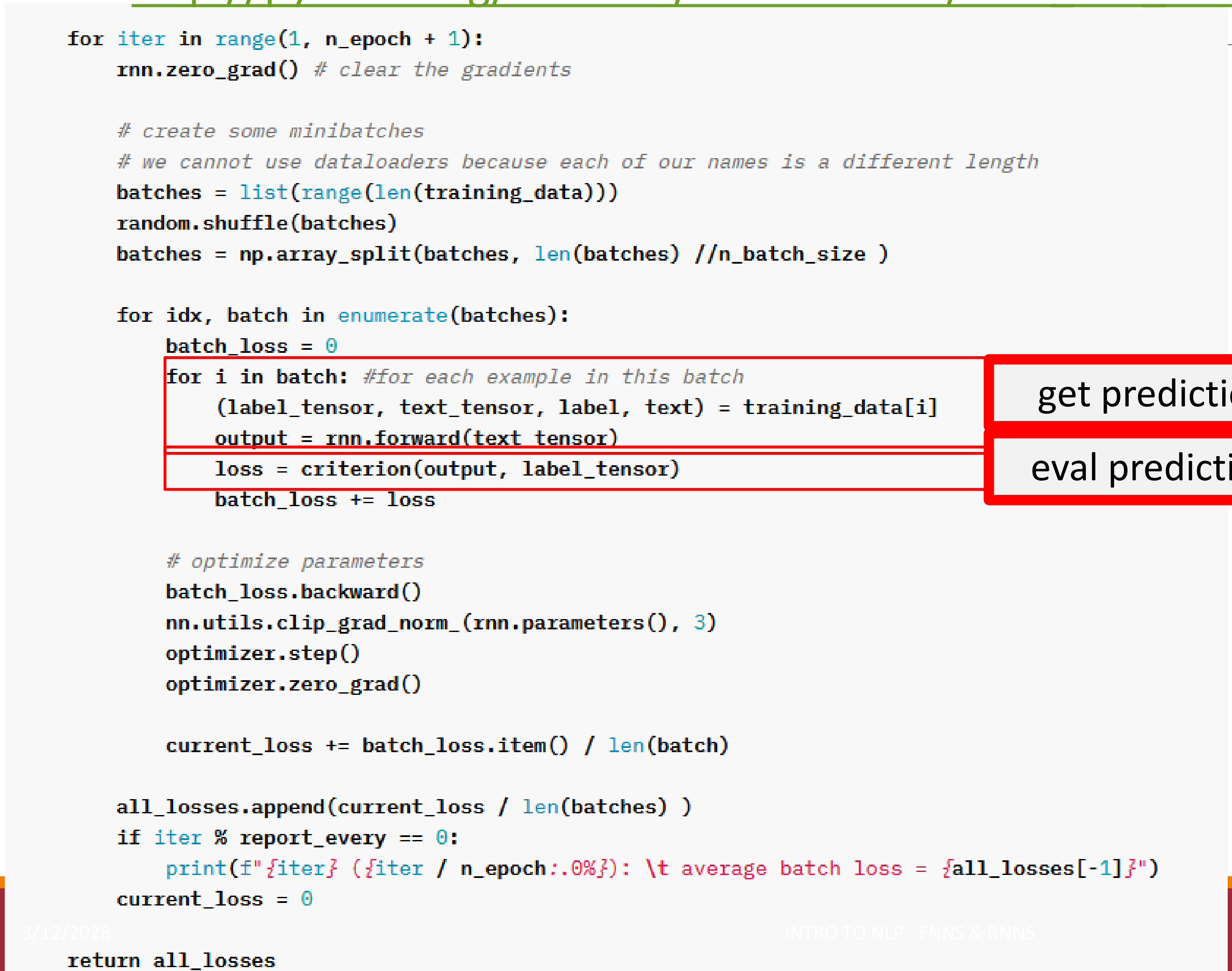
```
for iter in range(1, n_epoch + 1):
    rnn.zero_grad() # clear the gradients

    # 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

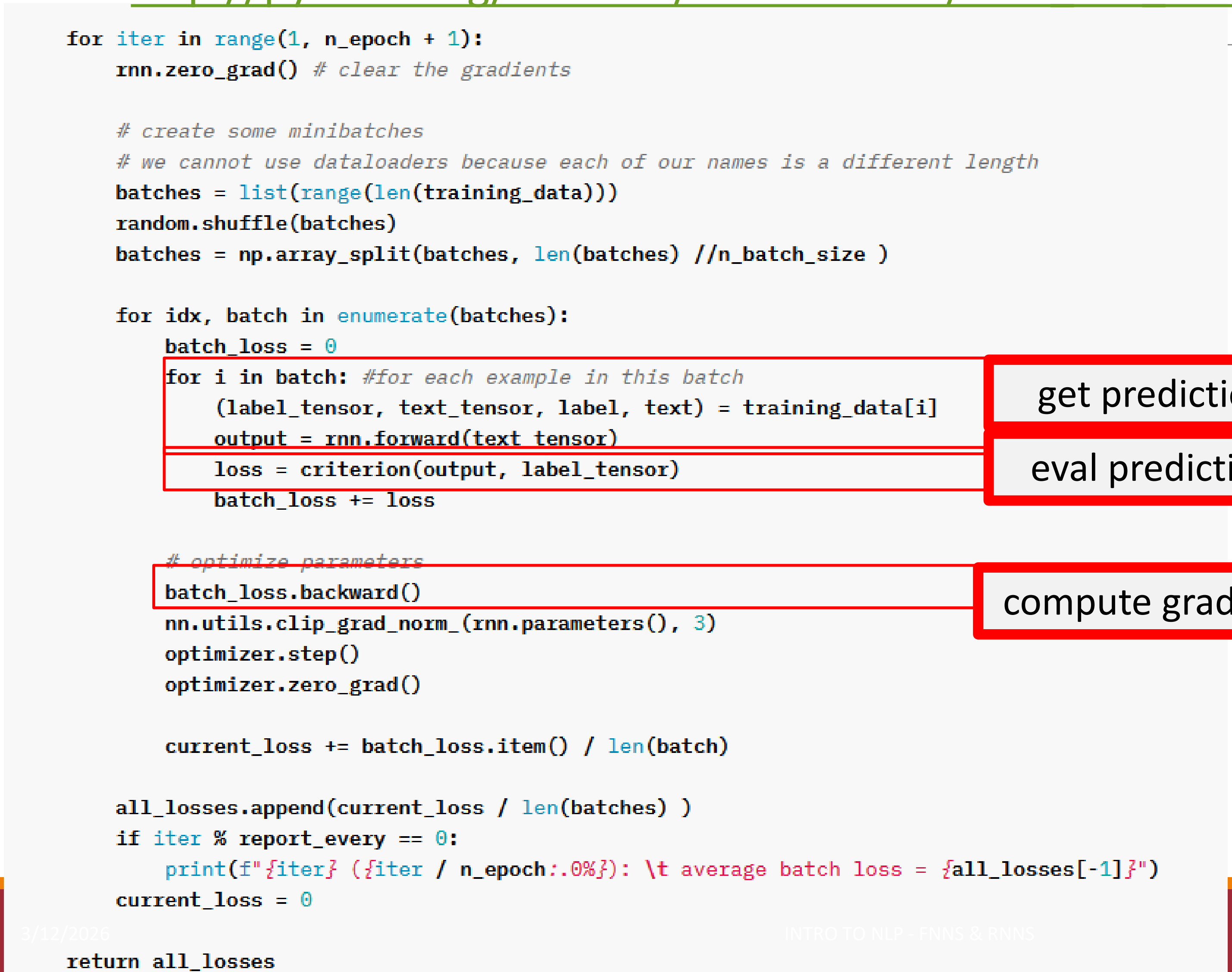
eval predictions

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

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

Training A Simple RNN in Python

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



```python
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

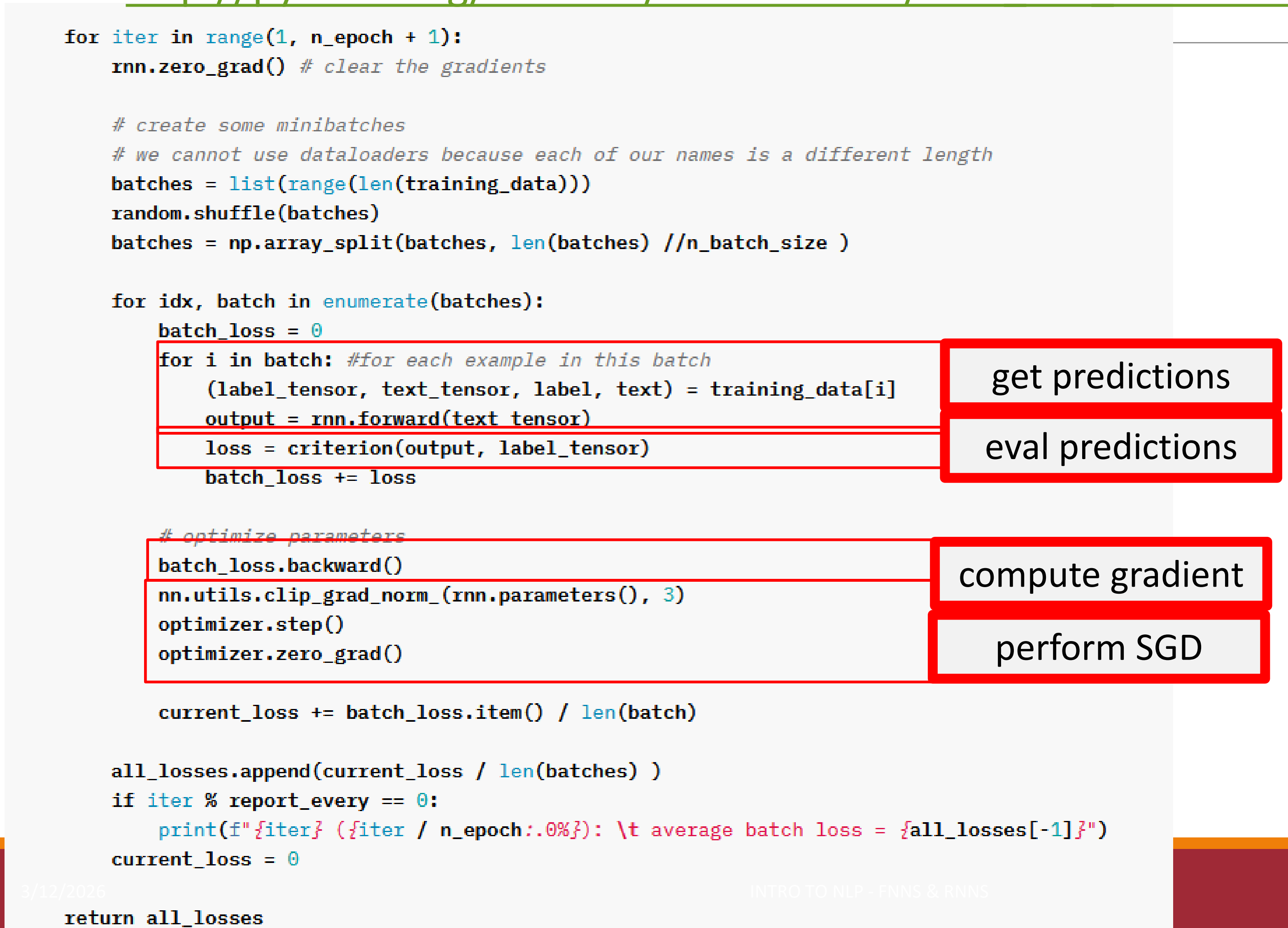
eval predictions

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 θ_{t+1} = θ_t - ρ_t * g_t
        5. Set t += 1
```

Training A Simple RNN in Python

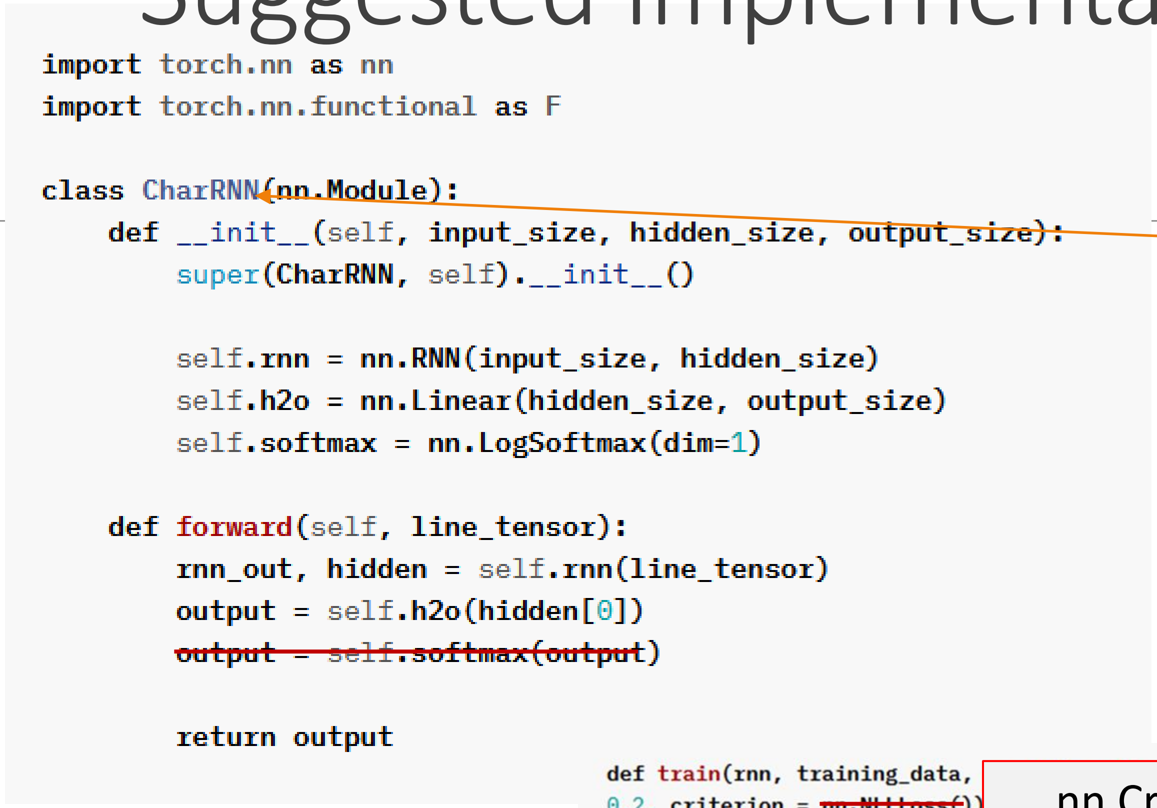
http://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html



```python
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
```

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

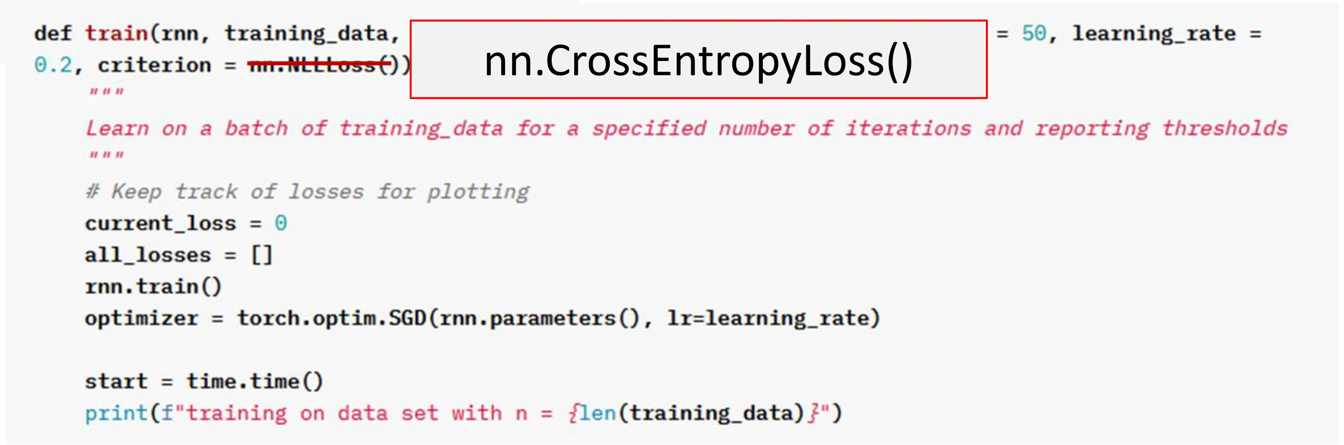
Suggested Implementation Changes



```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,                    = 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()

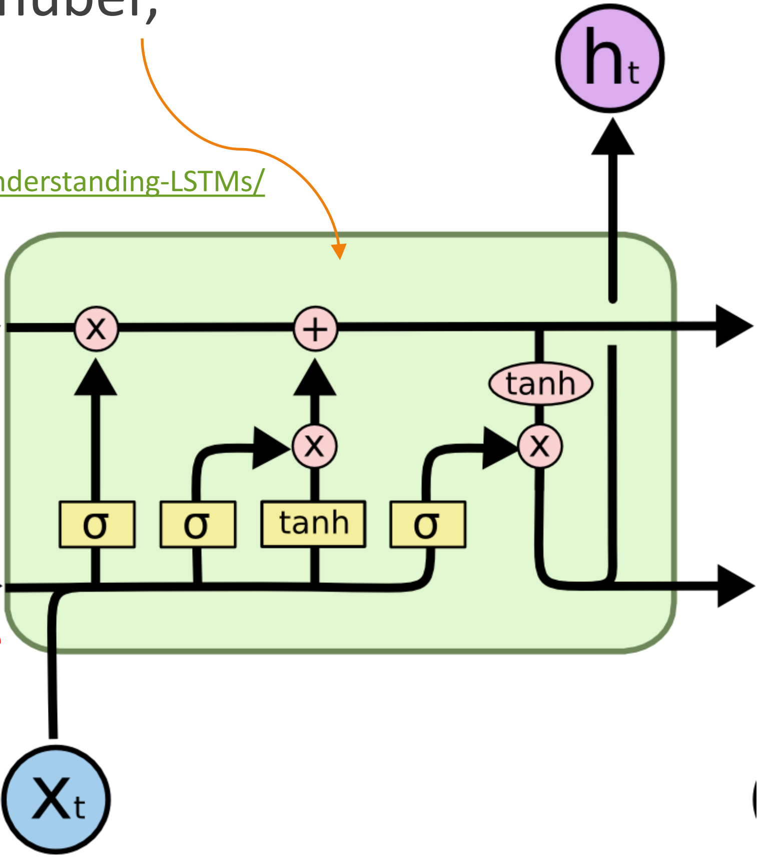
Another Solution: LSTMs/GRUs

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

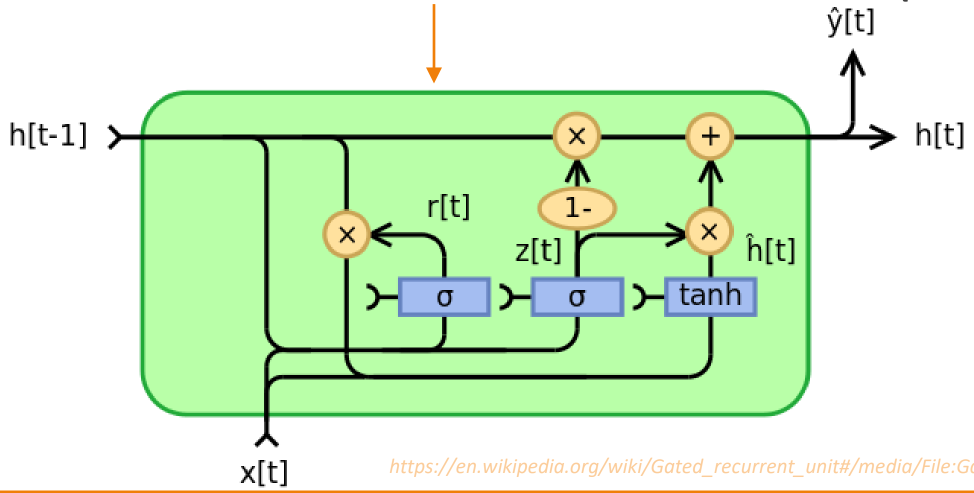
Basic Ideas: *learn to forget* http://colah.github.io/posts/2015-08-Understanding-LSTMs/

forget line

representation line



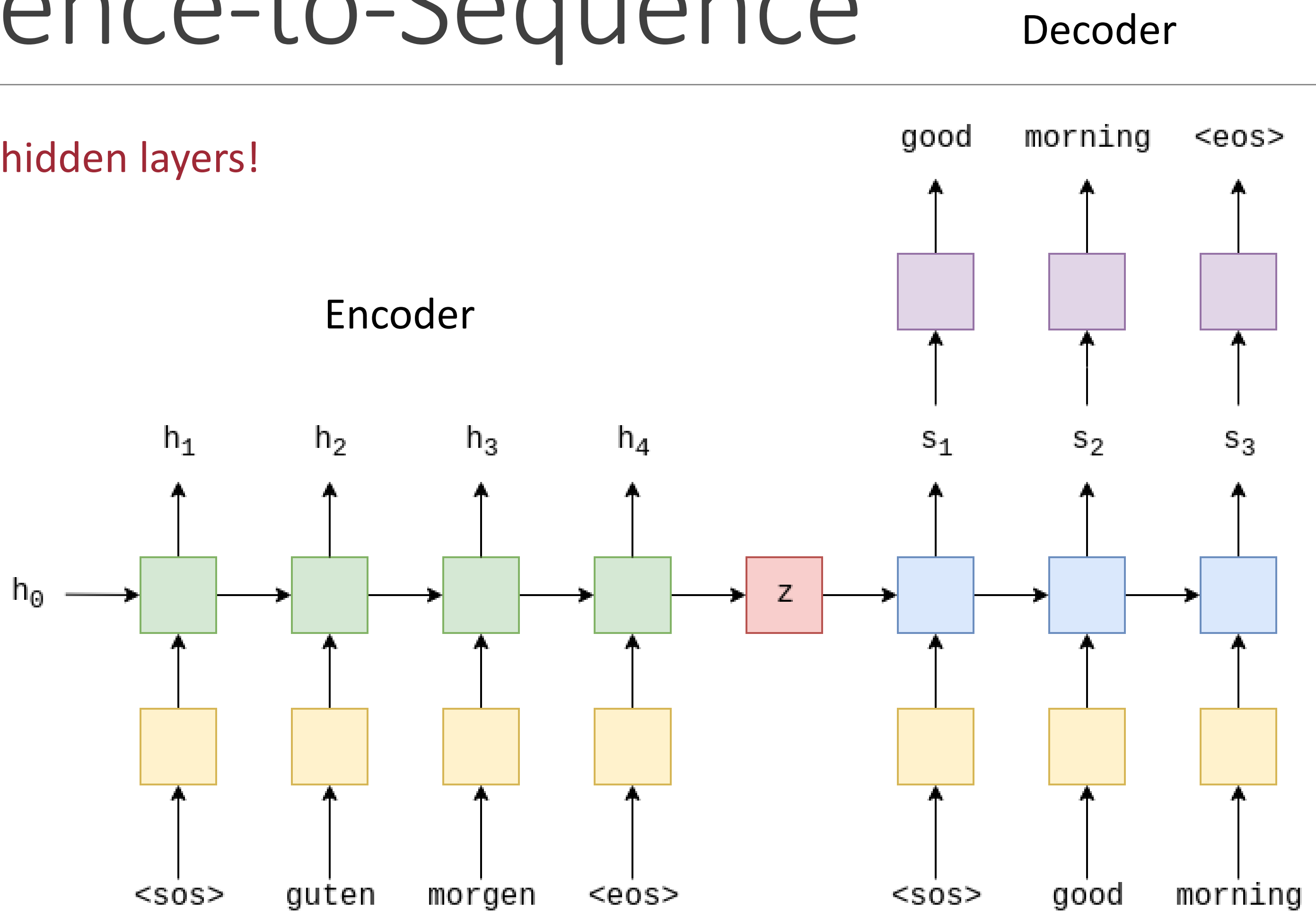
GRU: Gated Recurrent Unit (Cho et al., 2014)



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

Sequence-to-Sequence

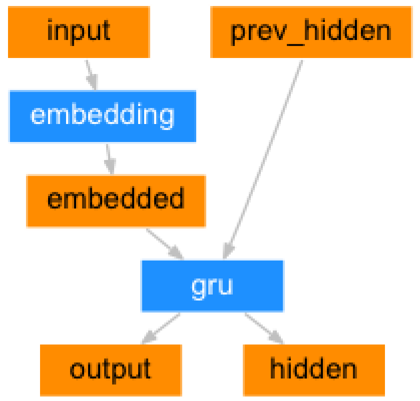
Note that this still has hidden layers!



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

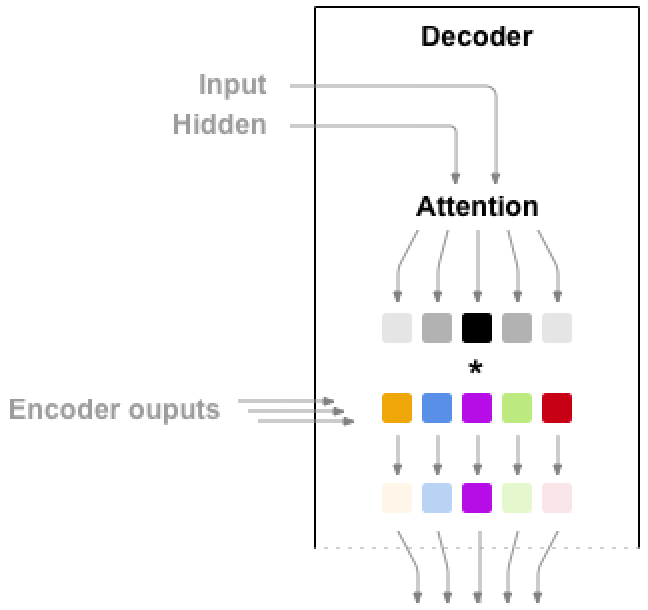
Encoder



```python
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

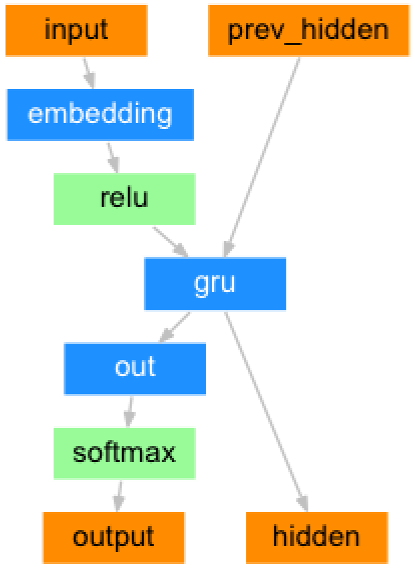
Attention



```
class BahdanauAttention(nn.Module):
    def __init__(self, hidden_size):
        super(BahdanauAttention, self).__init__()
        self.Wa = nn.Linear(hidden_size, hidden_size)
        self.Ua = nn.Linear(hidden_size, hidden_size)
        self.Va = nn.Linear(hidden_size, 1)

    def forward(self, query, keys):
        scores = self.Va(torch.tanh(self.Wa(query) + self.Ua(keys)))
        scores = scores.squeeze(2).unsqueeze(1)

        weights = F.softmax(scores, dim=-1)
        context = torch.bmm(weights, keys)

        return context, weights
```

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

Decoder



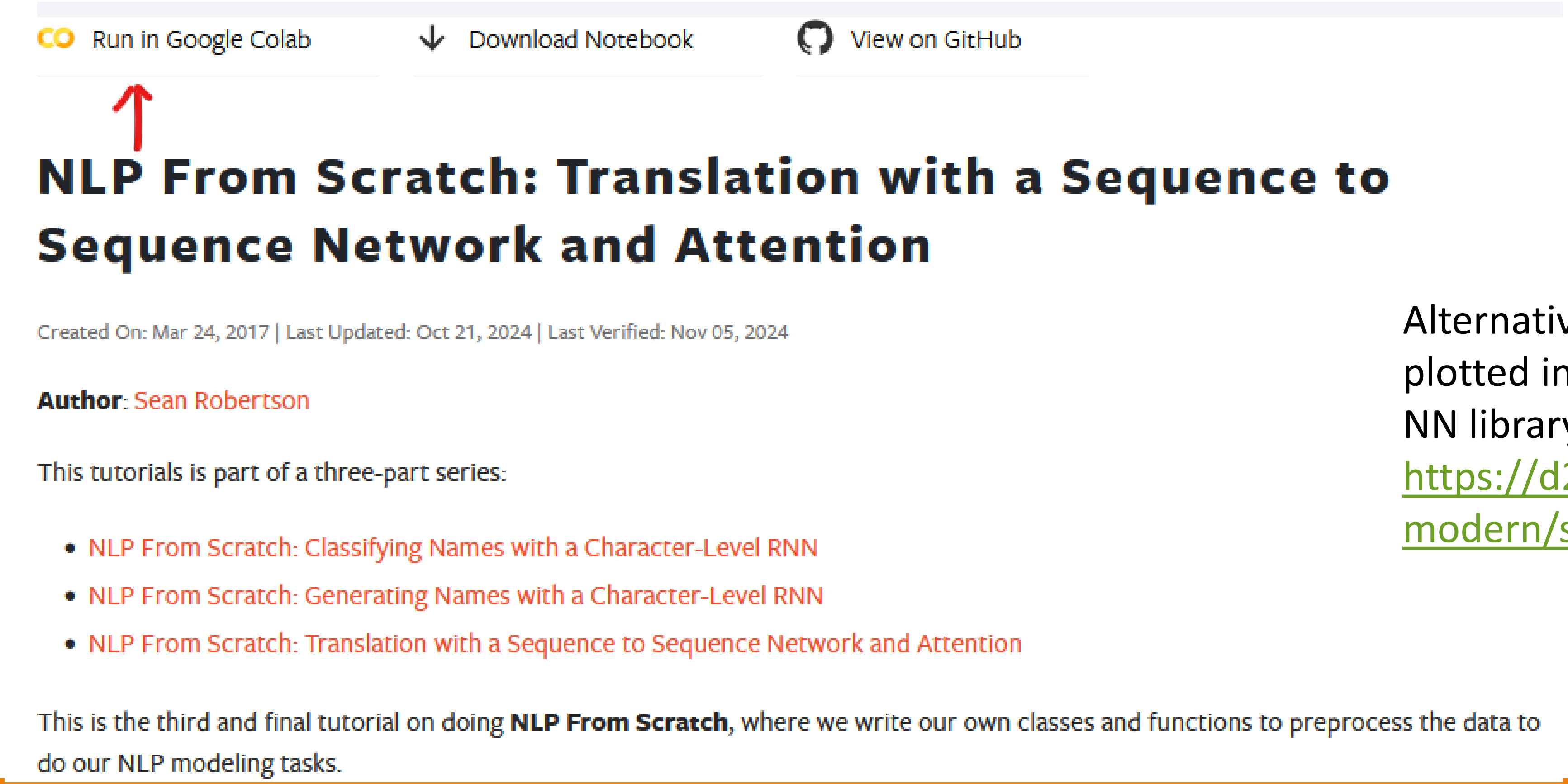
https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html

```
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

You will need to download the data. You can run
!wget https://download.pytorch.org/tutorial/data.zip
!unzip data.zip

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



NLP From Scratch: Translation with a Sequence to Sequence Network and Attention

Created On: Mar 24, 2017 | Last Updated: Oct 21, 2024 | Last Verified: Nov 05, 2024

Author: Sean Robertson

This tutorials is part of a three-part series:

- NLP From Scratch: Classifying Names with a Character-Level RNN
- NLP From Scratch: Generating Names with a Character-Level RNN
- NLP From Scratch: Translation with a Sequence to Sequence Network and Attention

This is the third and final tutorial on doing **NLP From Scratch**, where we write our own classes and functions to preprocess the data to do our NLP modeling tasks.

Alternative tutorial that shows loss plotted in real time but uses a different NN library: https://d2l.ai/chapter_recurrent-modern/seq2seq.html