Outline

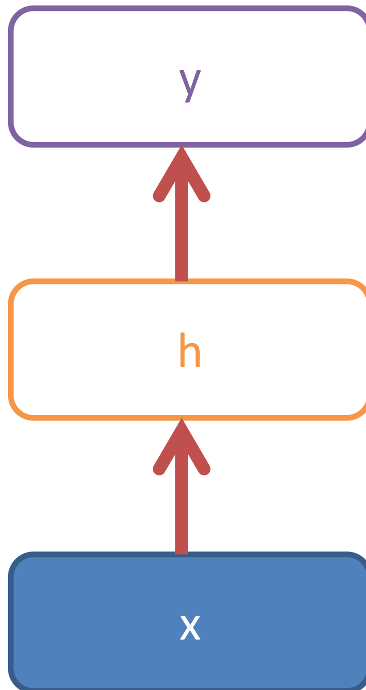
Convolutional Neural Networks

- What *is* a convolution?
- Multidimensional Convolutions
- Typical Convnet Operations
- Deep convnets

Recurrent Neural Networks

- Types of recurrence
- A basic recurrent cell
- BPTT: Backpropagation through time

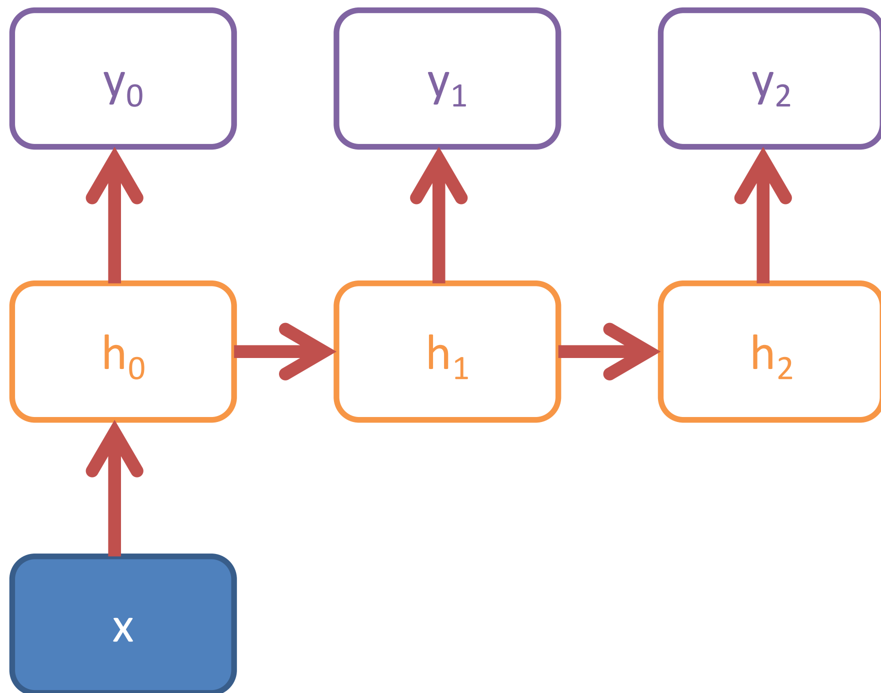
Network Types



Feed forward

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

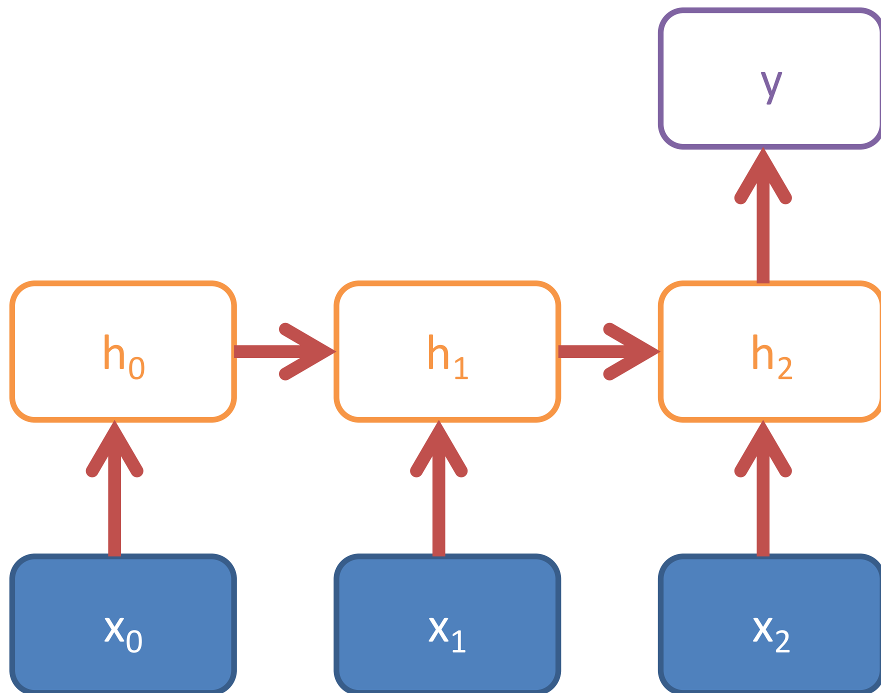
Network Types



Recursive: One input, Sequence output

Automated caption generation

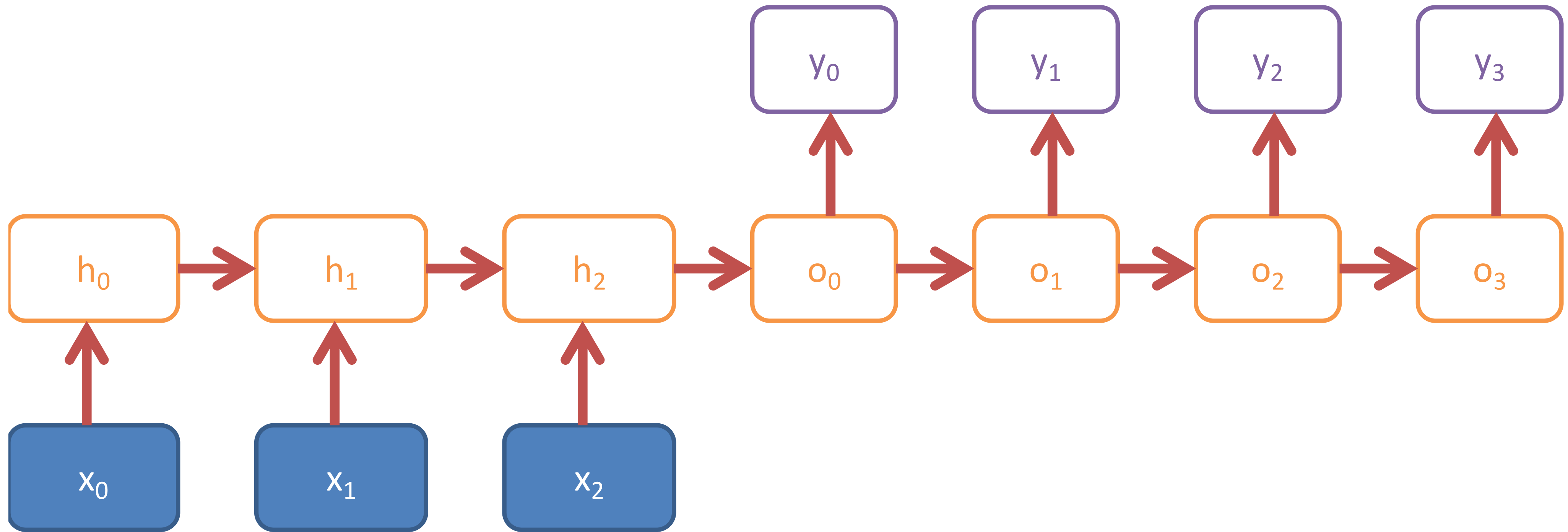
Network Types



Recursive: Sequence input, one output

Document classification
Action recognition in video (high-level)

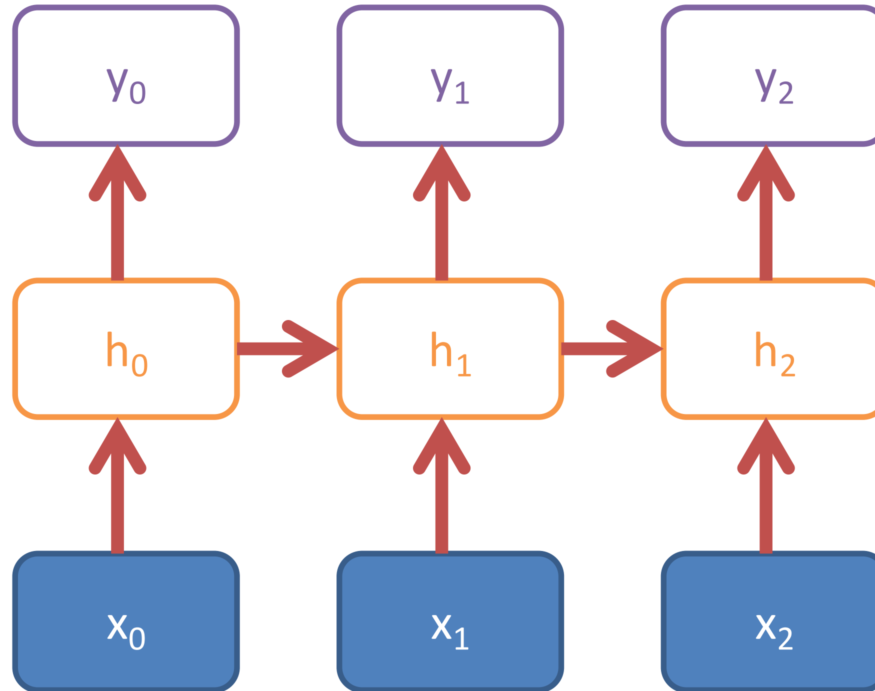
Network Types



Recursive: Sequence input, Sequence output (time delay)

Machine translation
Sequential description
Summarization

Network Types



Recursive: Sequence input, Sequence output

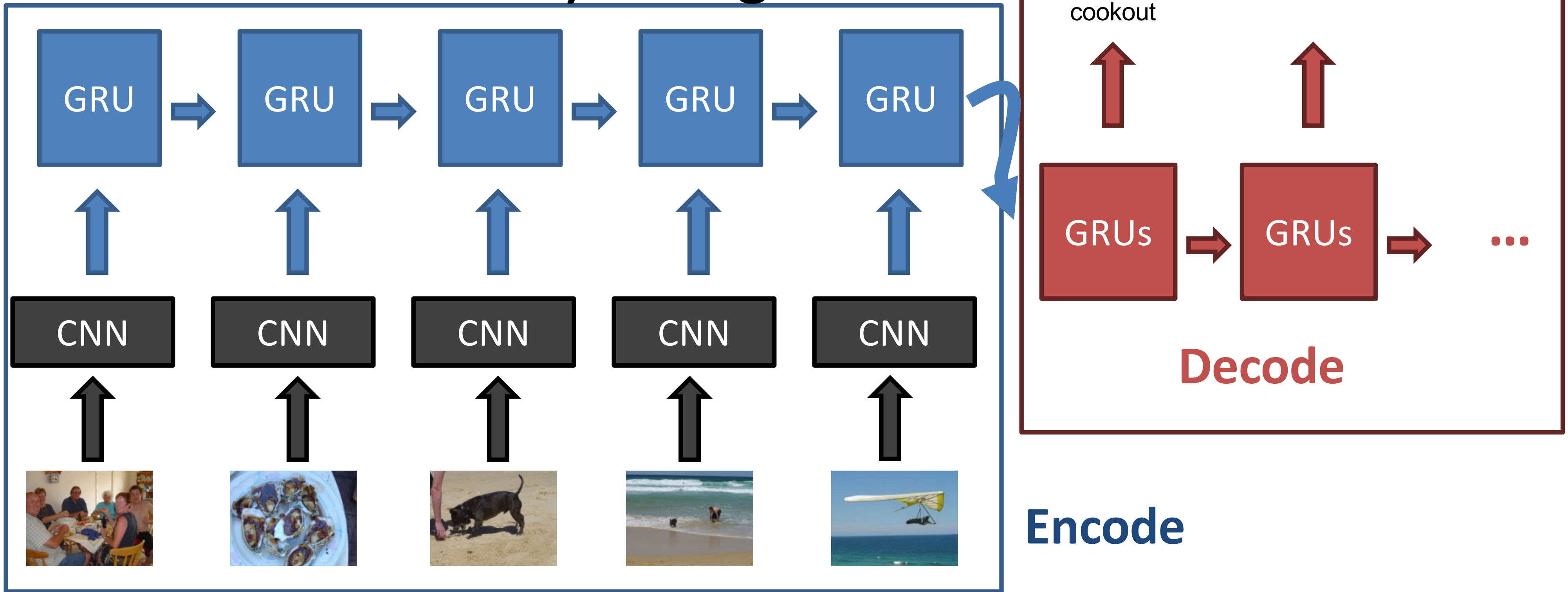
Part of speech tagging
Action recognition (fine grained)

RNN Outputs: Image Captions

Show and Tell: A Neural Image Caption Generator, CVPR 15

Slide credit: Arun Mallya

RNN Output: Visual Storytelling



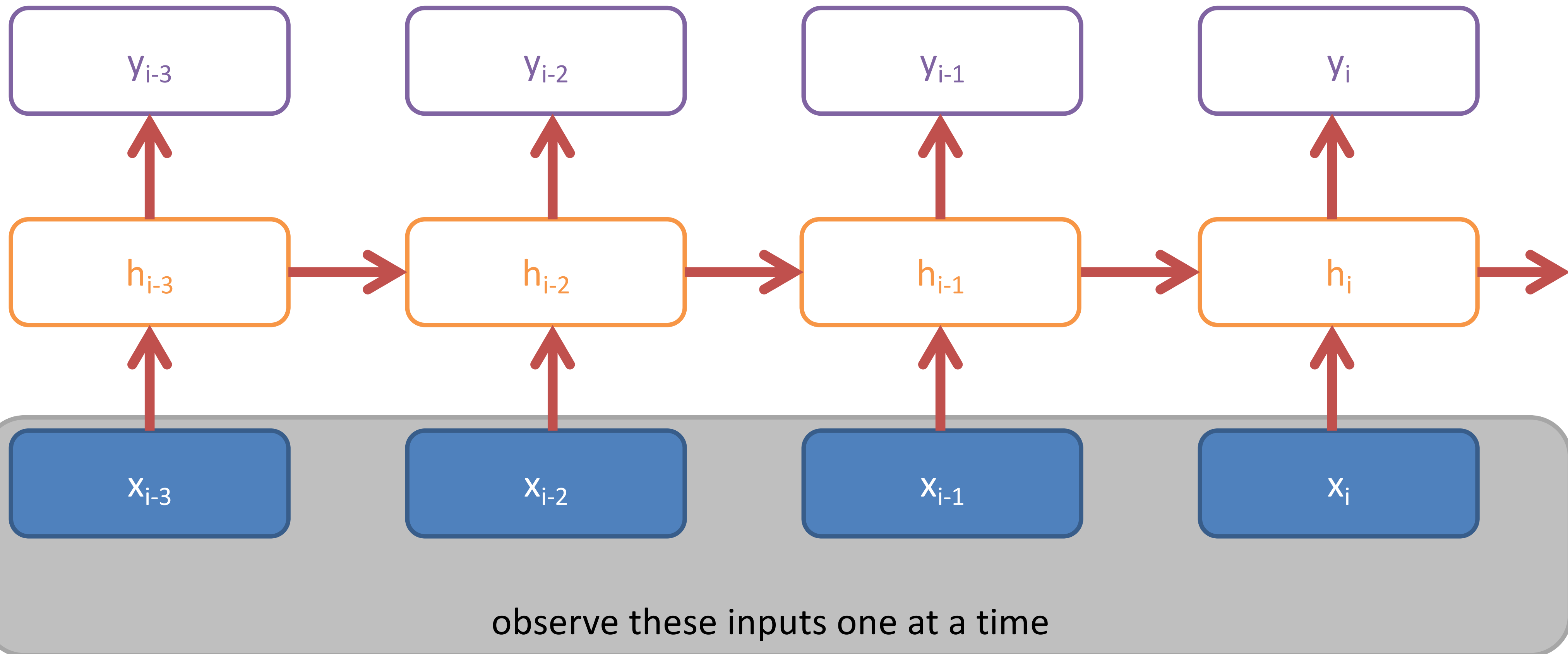
Human Reference

The family has gathered around the dinner table to share a meal together. They all pitched in to help cook the seafood to perfection. Afterwards they took the family dog to the beach to get some exercise. The waves were cool and refreshing! The dog had so much fun in the water. One family member decided to get a better view of the waves!

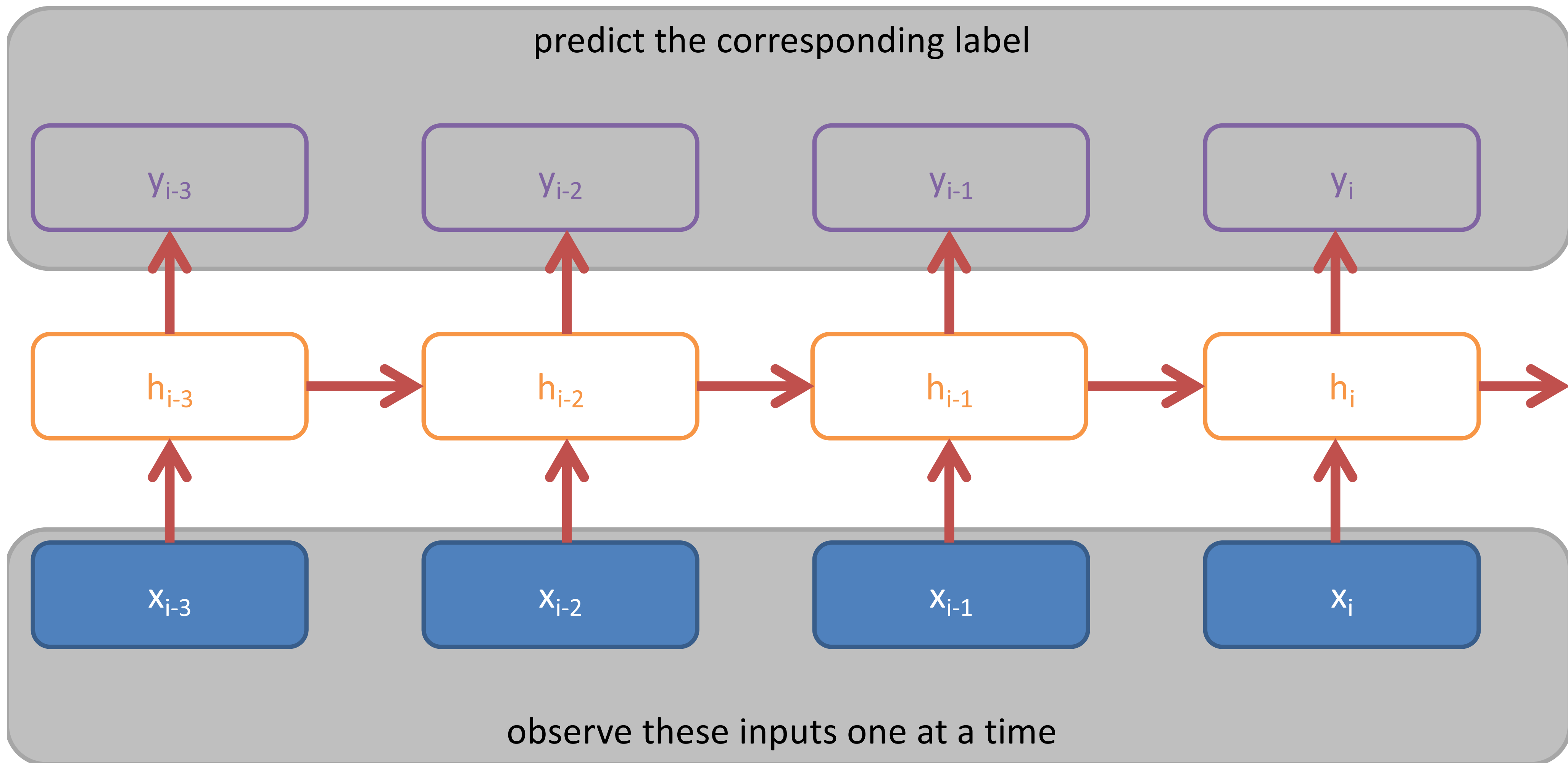
Huang et al. (2016)

The family got together for a cookout. They had a lot of delicious food. The dog was happy to be there. They had a great time on the beach. They even had a swim in the water.

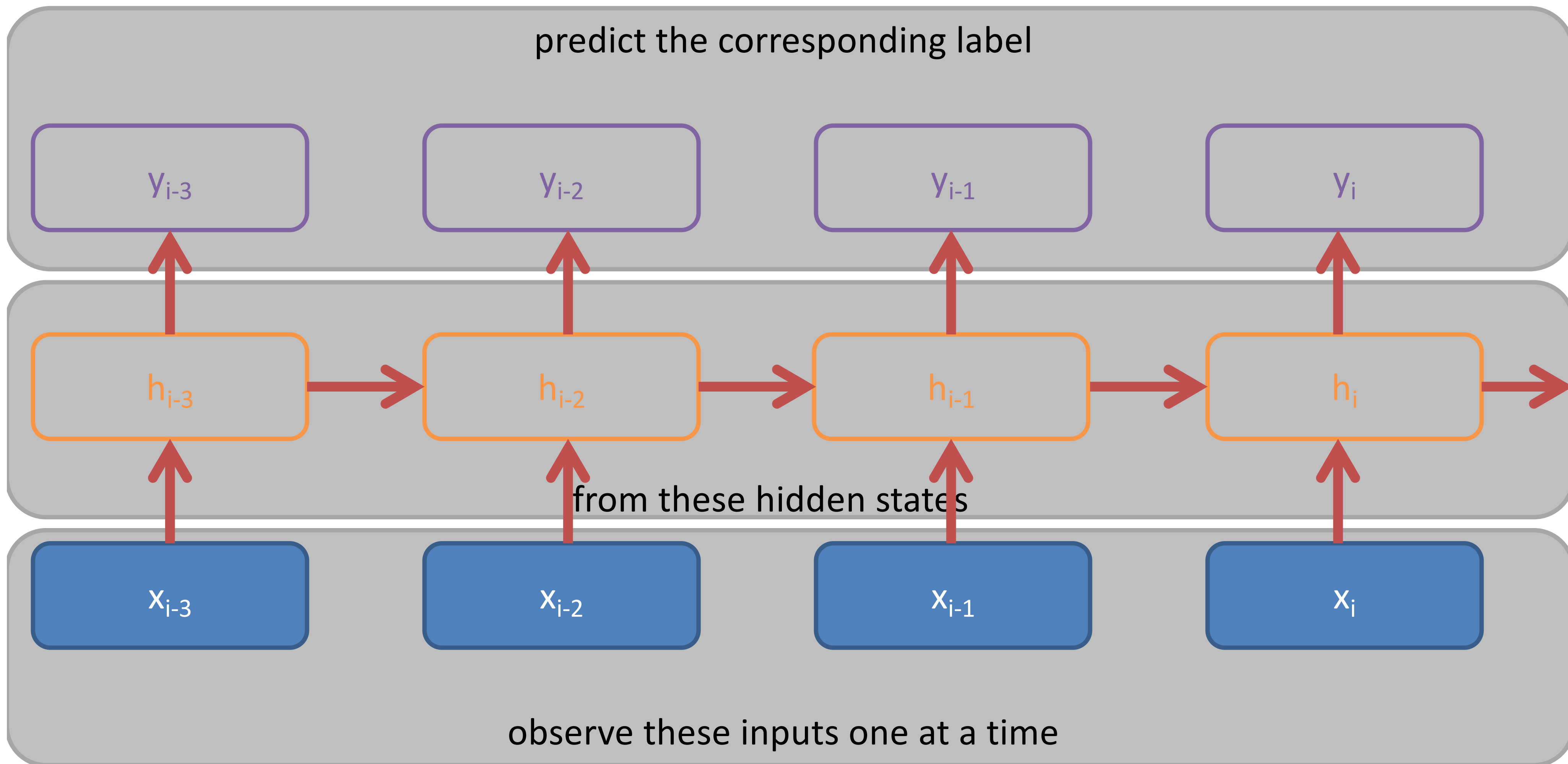
Recurrent Networks



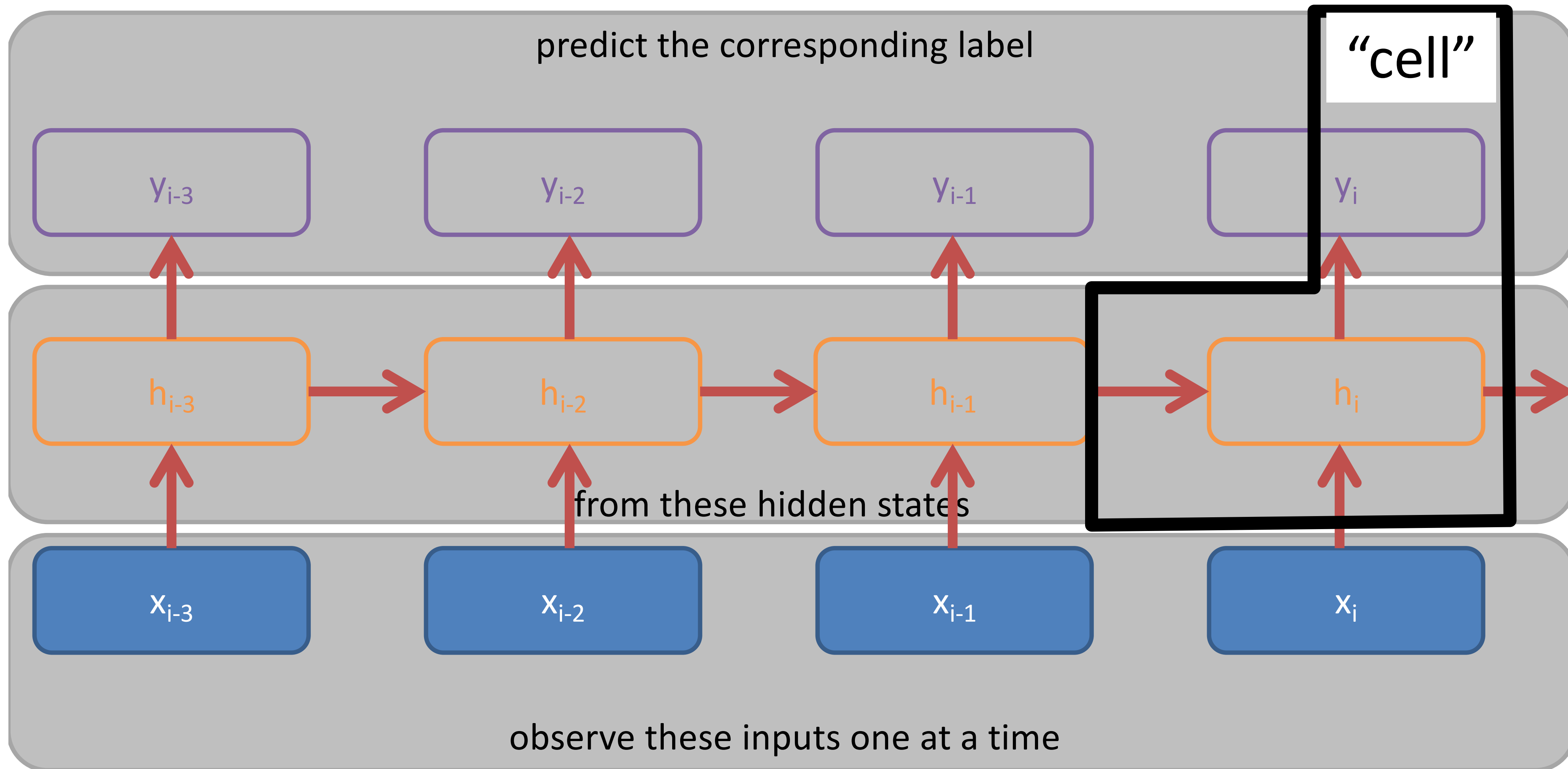
Recurrent Networks



Recurrent Networks



Recurrent Networks



Outline

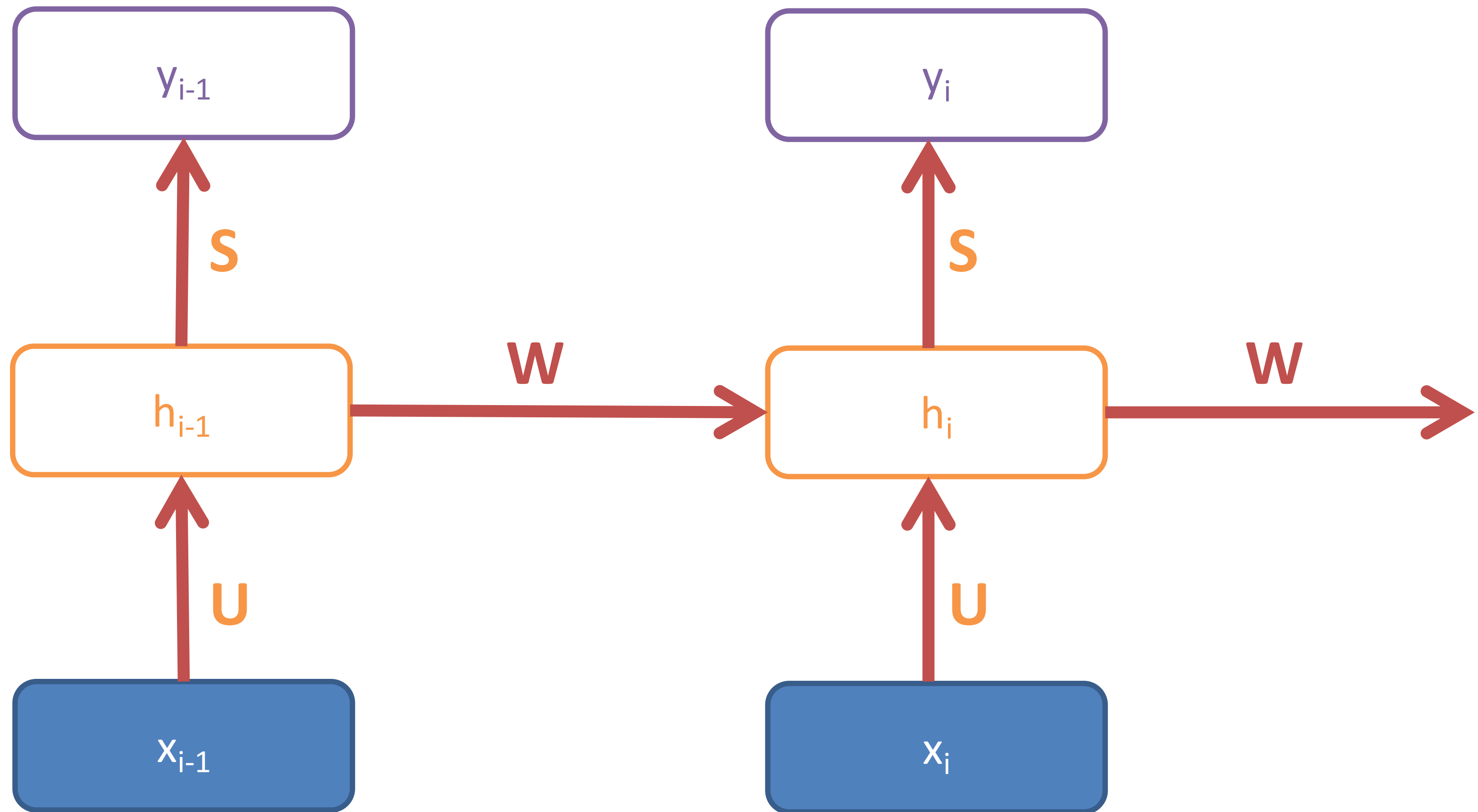
Convolutional Neural Networks

- What *is* a convolution?
- Multidimensional Convolutions
- Typical Convnet Operations
- Deep convnets

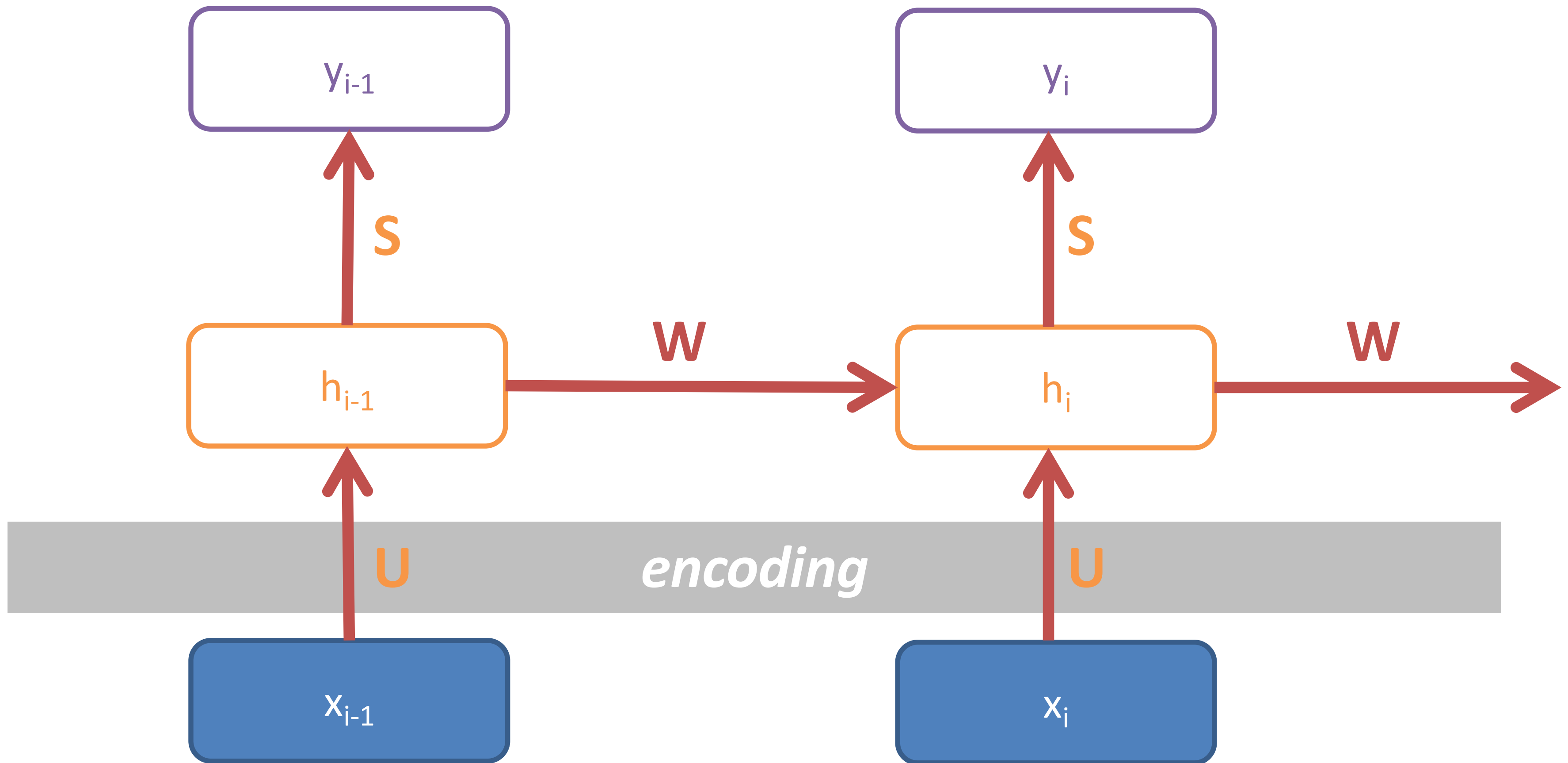
Recurrent Neural Networks

- Types of recurrence
- A basic recurrent cell
- BPTT: Backpropagation through time

A *Simple* Recurrent Neural Network Cell

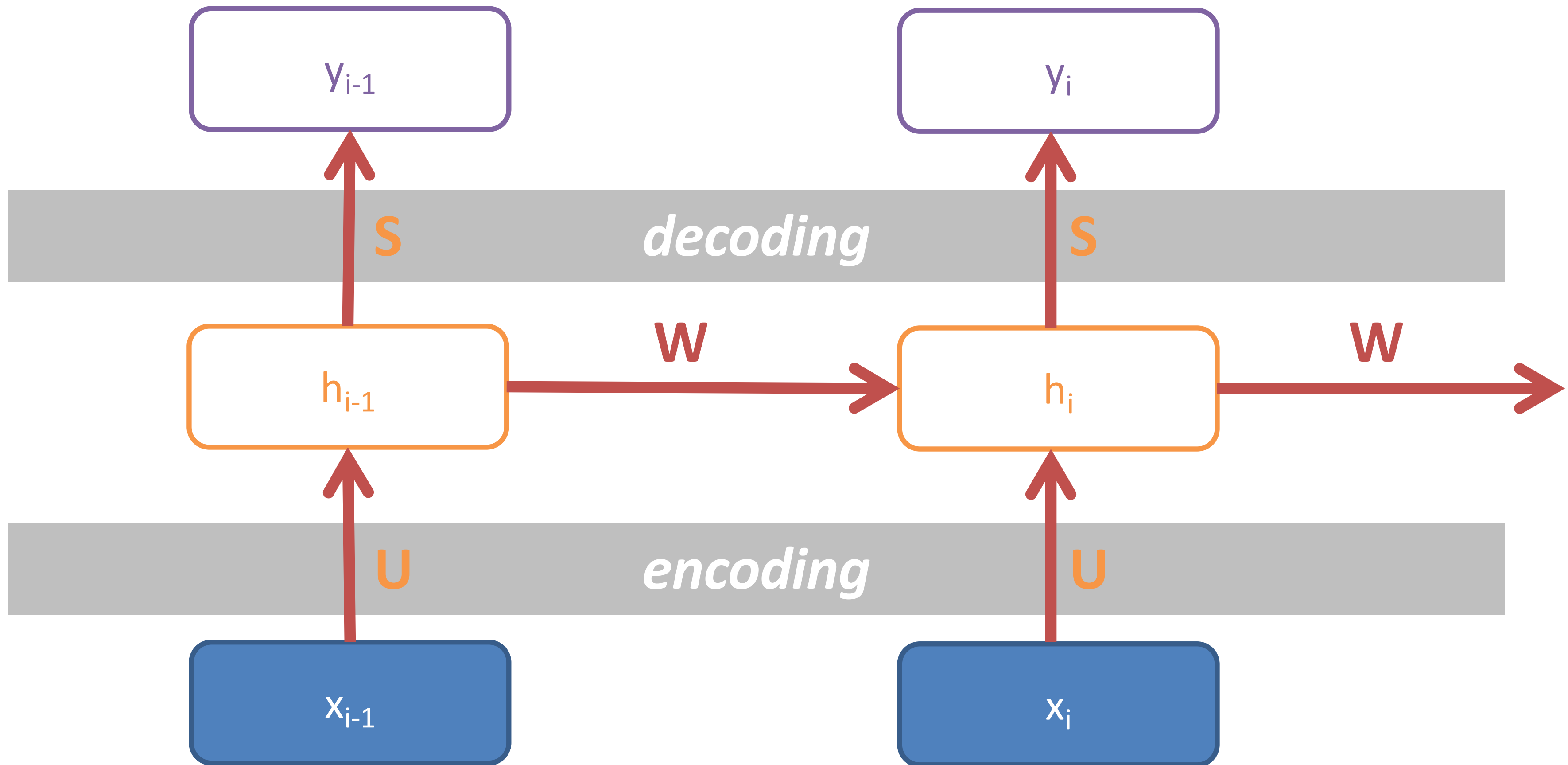


A *Simple* Recurrent Neural Network Cell



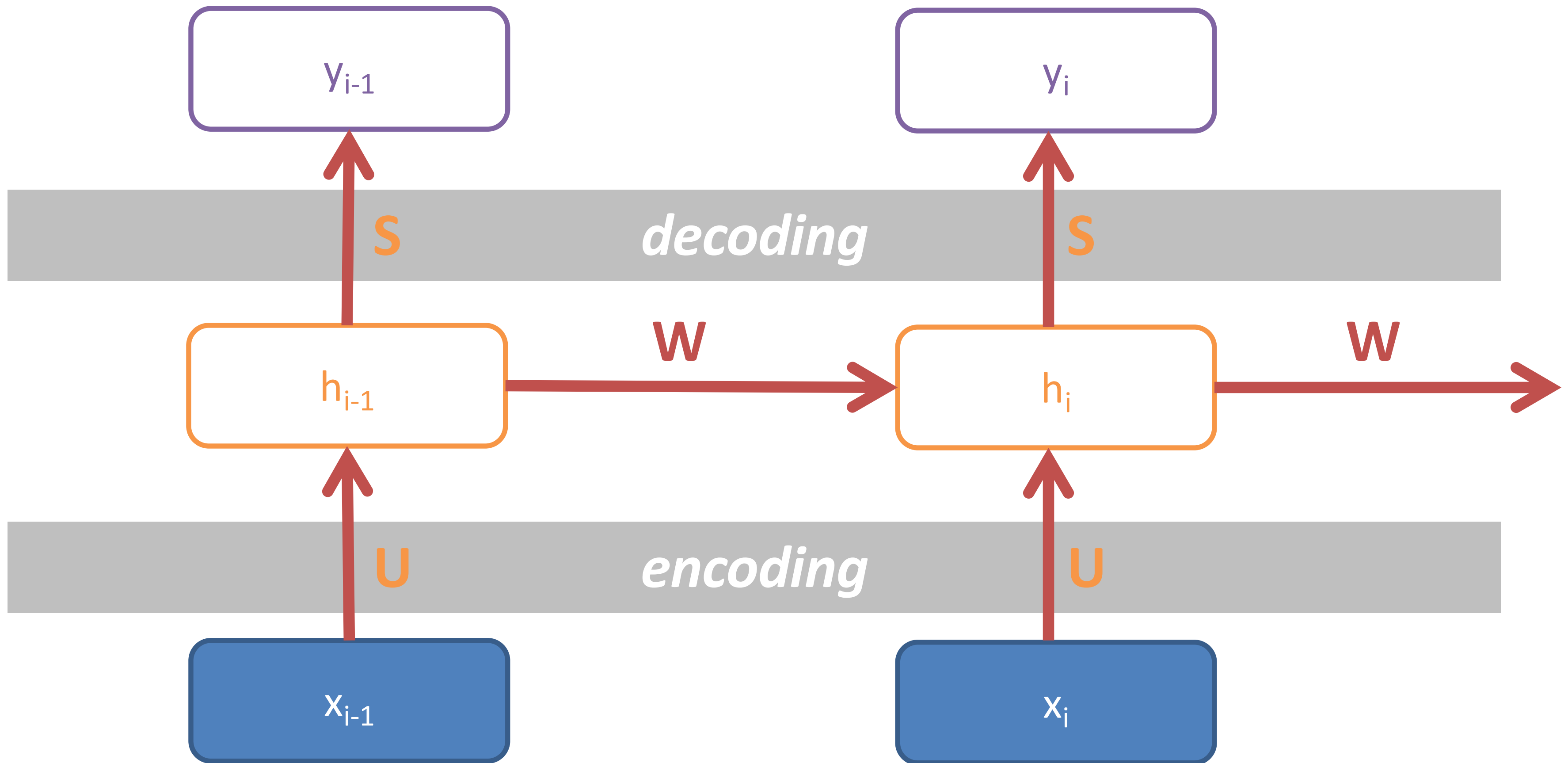
$$h_i = \tanh(W h_{i-1} + U x_i)$$

A *Simple* Recurrent Neural Network Cell



$$h_i = \tanh(Wh_{i-1} + Ux_i) \qquad y_i = \text{softmax}(Sh_i)$$

A *Simple* Recurrent Neural Network Cell



$$h_i = \tanh(W h_{i-1} + U x_i)$$

$$y_i = \text{softmax}(S h_i)$$

Weights are shared over time

unrolling/unfolding: copy the RNN cell across time (inputs)

Outline

Convolutional Neural Networks

- What *is* a convolution?
- Multidimensional Convolutions
- Typical Convnet Operations
- Deep convnets

Recurrent Neural Networks

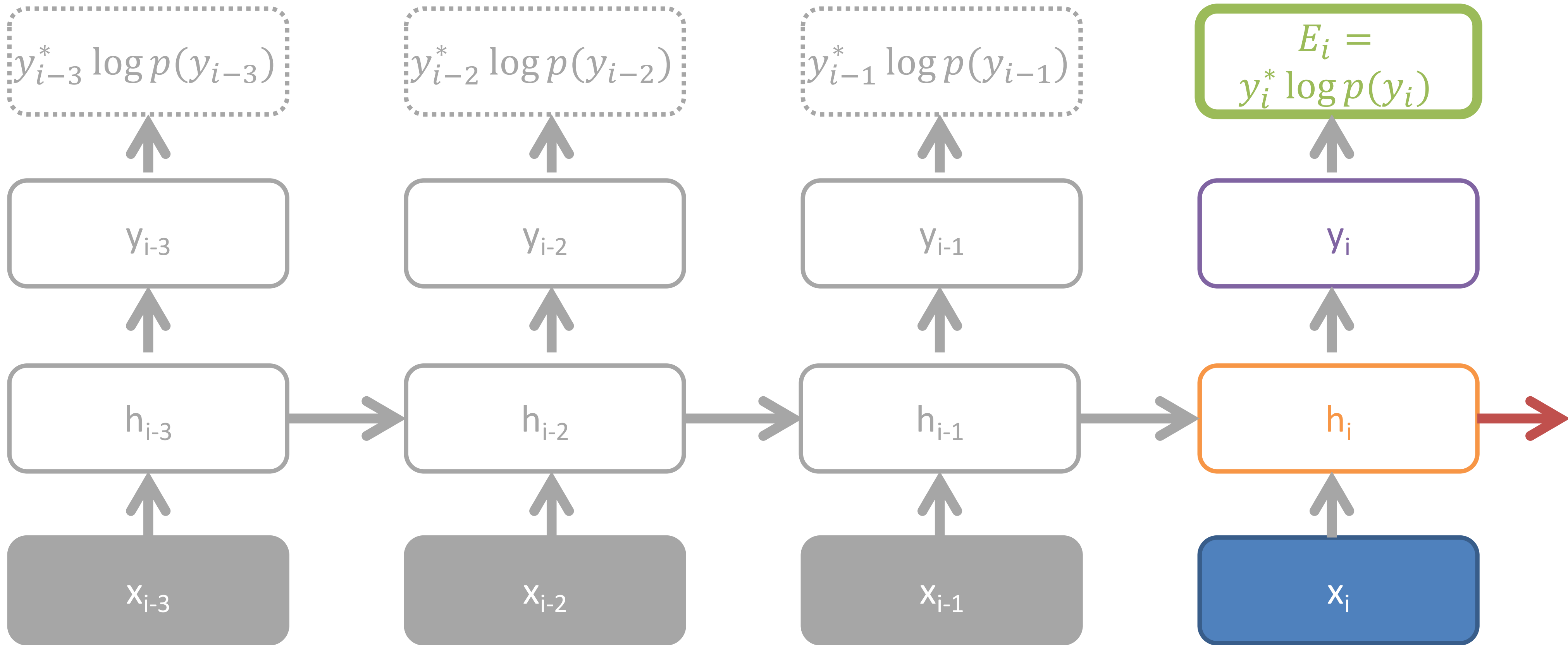
- Types of recurrence
- A basic recurrent cell
- BPTT: Backpropagation through time

BackPropagation Through Time (BPTT)

"Unfold" the network to create a single, large, feed-forward network

1. Weights are copied (W → $W^{(t)}$)
2. Gradients computed ($\eth W^{(t)}$), and
3. Summed ($\sum_t \eth W^{(t)}$)

BPTT



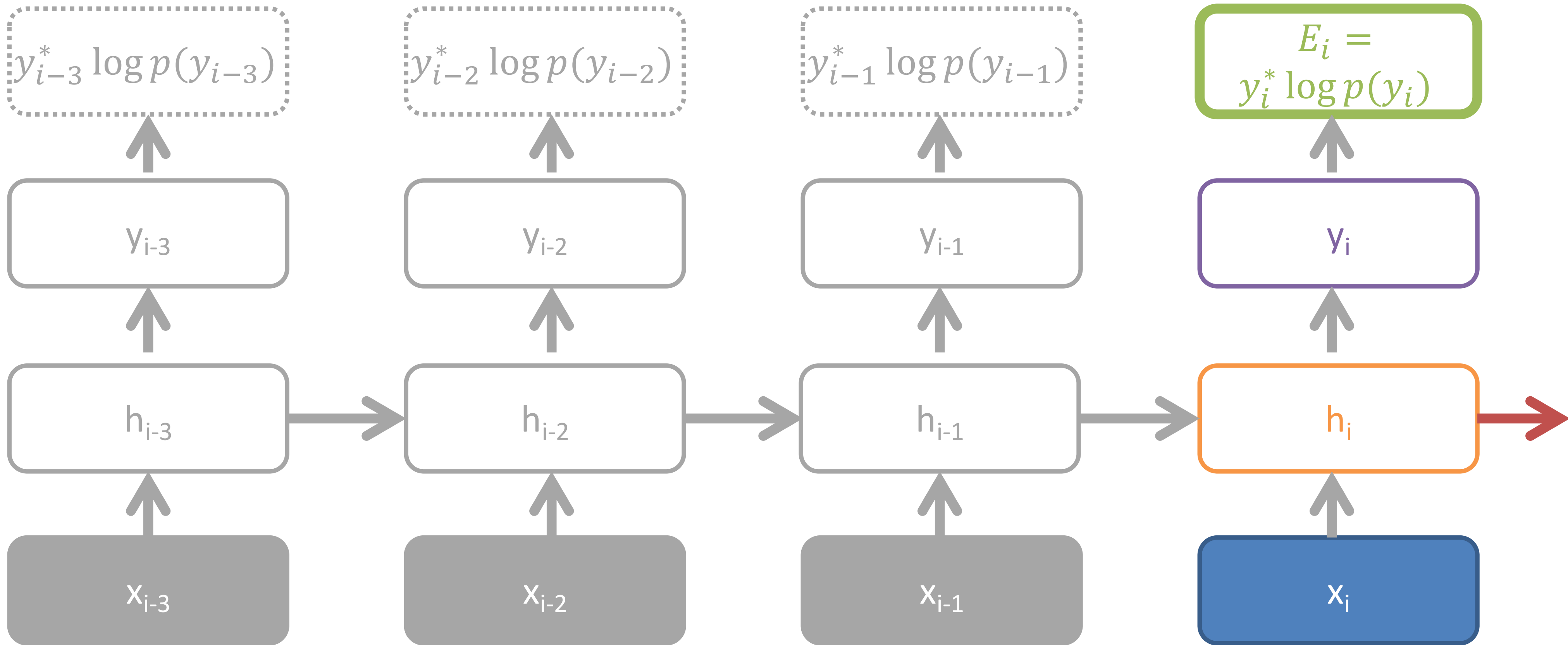
$$\frac{\partial E_i}{\partial W} = \frac{\partial E_i}{\partial y_i}\frac{\partial y_i}{\partial W} = \frac{\partial E_i}{\partial y_i}\frac{\partial y_i}{\partial h_i}\frac{\partial h_i}{\partial W}$$

$$y_i = \text{softmax}(Sh_i)$$

$$h_i = \tanh(Wh_{i-1} + Ux_i)$$

per-step loss: cross entropy

BPTT



$$y_i = \text{softmax}(Sh_i)$$

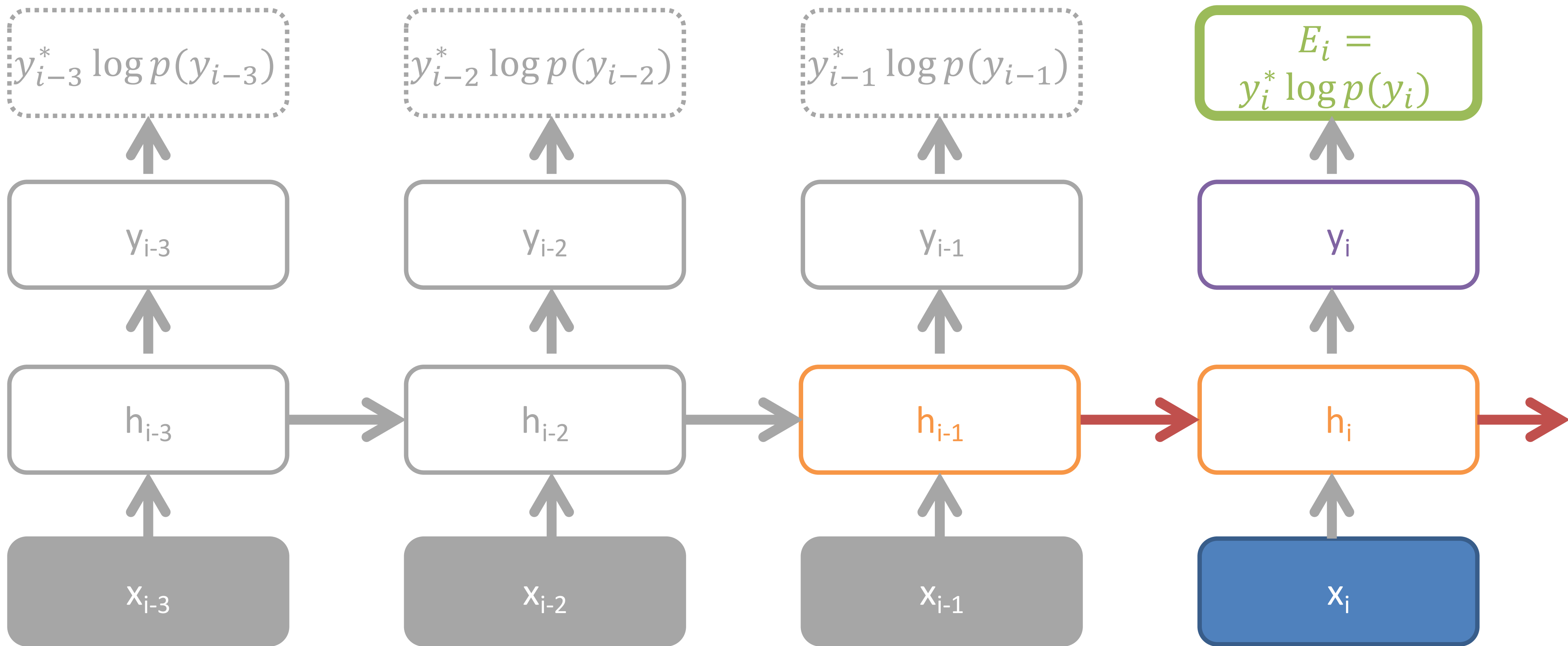
$$h_i = \tanh(Wh_{i-1} + Ux_i)$$

per-step loss: cross entropy

$$\frac{\partial E_i}{\partial W} = \frac{\partial E_i}{\partial y_i}\frac{\partial y_i}{\partial W} = \frac{\partial E_i}{\partial y_i}\frac{\partial y_i}{\partial h_i}\frac{\partial h_i}{\partial W}$$

$$\frac{\partial h_i}{\partial W} = \tanh'(Wh_{i-1} + Ux_i)\frac{\partial Wh_{i-1}}{\partial W}$$

BPTT



$$y_i = \mathrm{softmax}(Sh_i)$$

$$h_i = \tanh(Wh_{i-1} + Ux_i)$$

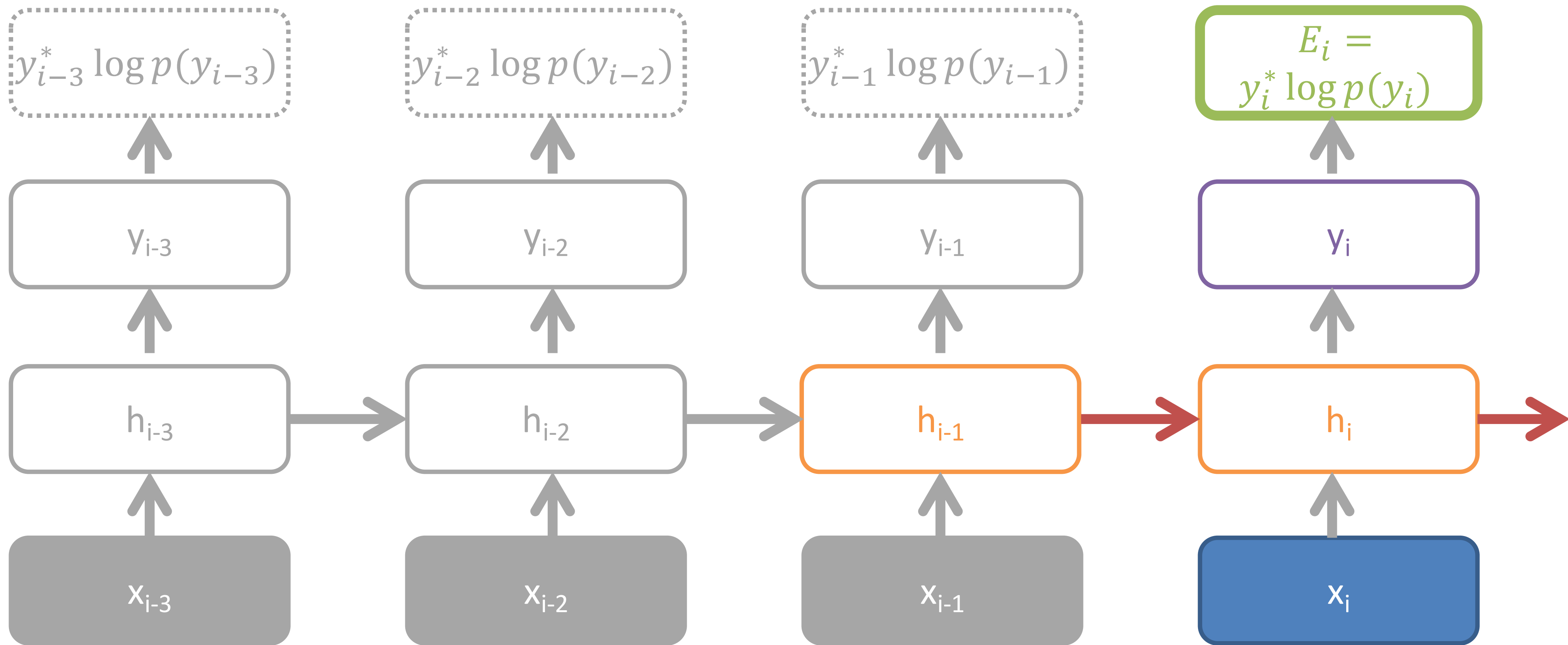
per-step loss: cross entropy

$$\frac{\partial E_i}{\partial W} = \frac{\partial E_i}{\partial y_i}\frac{\partial y_i}{\partial W} = \frac{\partial E_i}{\partial y_i}\frac{\partial y_i}{\partial h_i}\frac{\partial h_i}{\partial W}$$

$$\frac{\partial h_i}{\partial W} = \tanh'(Wh_{i-1} + Ux_i)\frac{\partial Wh_{i-1}}{\partial W}$$

$$= \tanh'(Wh_{i-1} + Ux_i)\left(h_{i-1} + W\frac{\partial h_{i-1}}{\partial W}\right)$$

BPTT



$$y_i = \text{softmax}(Sh_i)$$

$$h_i = \tanh(Wh_{i-1} + Ux_i)$$

per-step loss: cross entropy

$$\frac{\partial E_i}{\partial W} = \frac{\partial E_i}{\partial y_i}\frac{\partial y_i}{\partial h_i}\frac{\partial h_i}{\partial W} = \eth h_i \frac{\partial h_i}{\partial W} = \delta_l^{(i)}$$

$$\delta_l^{(i)} = \frac{\partial E_i}{\partial y_i}\frac{\partial y_i}{\partial h_i}\frac{\partial h_i}{\partial h_l}\frac{\partial h_l}{\partial W}$$

$$\frac{\partial h_i}{\partial W} = \tanh'(Wh_{i-1} + Ux_i)\left(h_{i-1} + W\frac{\partial h_{i-1}}{\partial W}\right) = \delta_i h_{i-1} + \delta_i W \eth h_{i-1}\left(h_{i-2} + W\frac{\partial h_{i-2}}{\partial W}\right)$$

BPTT

$$\frac{\partial h_i}{\partial W} = \tanh'(W h_{i-1} + U x_i)\left(h_{i-1} + W\,\frac{\partial h_{i-1}}{\partial W}\right)$$

$$= \tanh'(W h_{i-1} + U x_i) h_{i-1} + \tanh'(W h_{i-1} + U x_i) W \tanh'(W h_{i-2} + U x_{i-1})\left(h_{i-2} + W\,\frac{\partial h_{i-2}}{\partial W}\right)$$

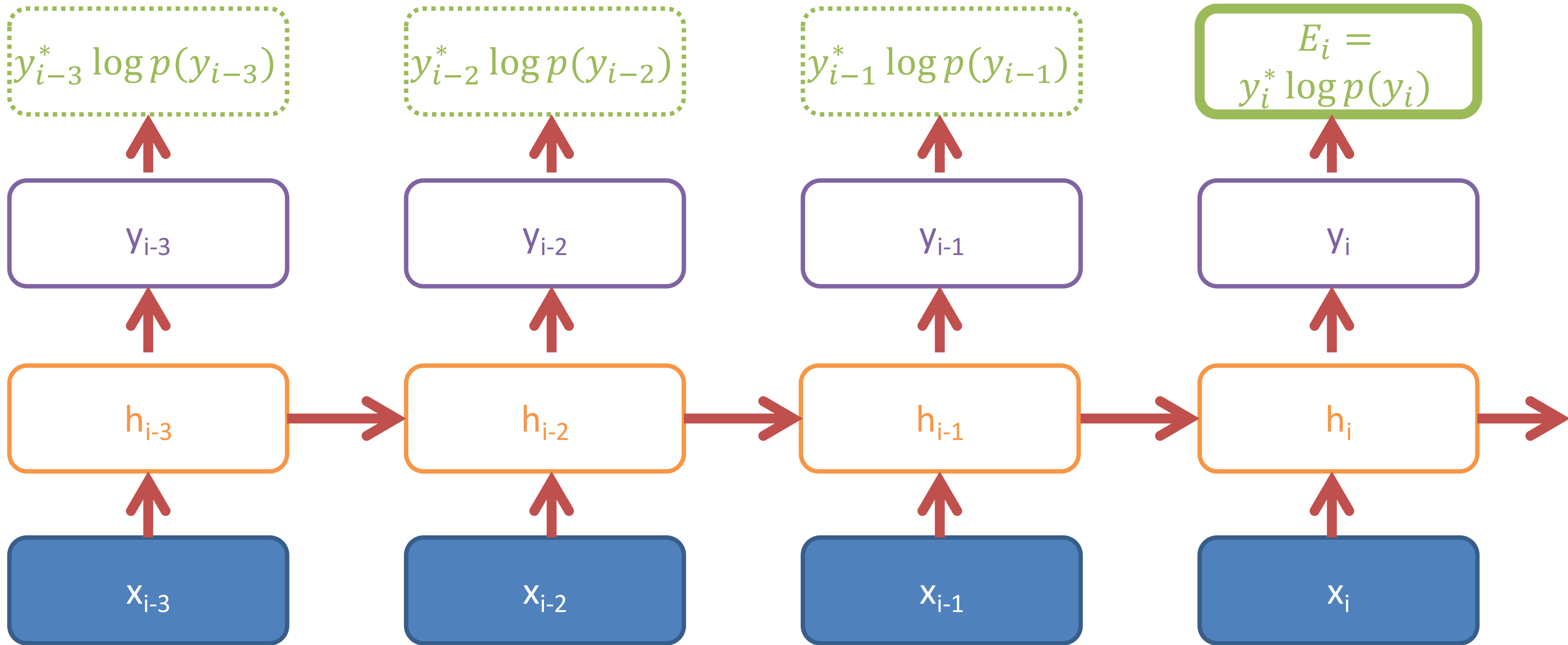
$$= \sum_j \frac{\partial E_i}{\partial y_i}\frac{\partial y_i}{\partial h_i}\frac{\partial h_i}{\partial h_l}\frac{\partial h_l}{\partial W^{(l)}}$$

$$= \sum_j \delta_j^{(i)}\,\frac{\partial h_l}{\partial W^{(l)}}$$

$$\delta_l^{(i)} = \frac{\partial E_i}{\partial y_i}\frac{\partial y_i}{\partial h_i}\frac{\partial h_i}{\partial h_l}$$

per-loss, per-step backpropagation error

BPTT



$$y_i = \mathrm{softmax}(Sh_i)$$

$$h_i = \tanh(Wh_{i-1} + Ux_i)$$

per-step loss: cross entropy

$$\frac{\partial E_i}{\partial W} = \sum_j \frac{\partial E_i}{\partial y_i} \frac{\partial y_i}{\partial h_i} \frac{\partial h_i}{\partial W^{(j)}}$$

compact form

hidden chain rule

Why Is Training RNNs Hard?

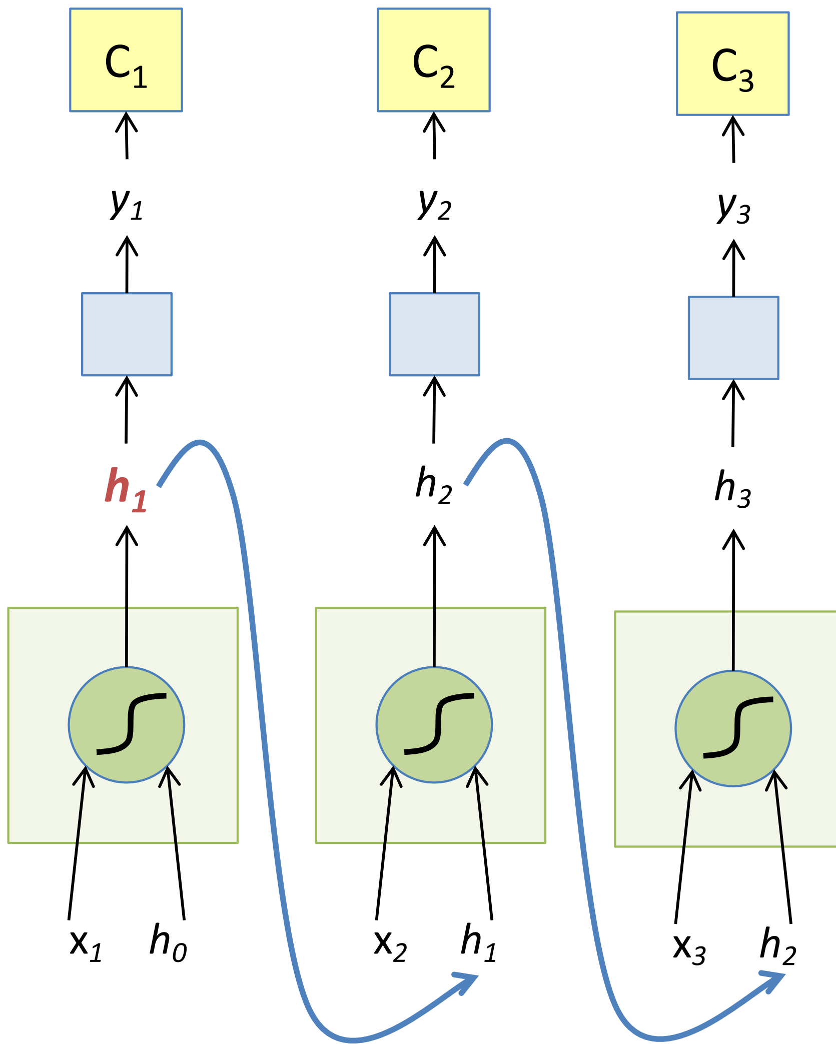
$$\frac{\partial C_t}{\partial h_1} = \left(\frac{\partial C_t}{\partial y_t}\right)\left(\frac{\partial y_t}{\partial h_1}\right)$$

$$= \left(\frac{\partial C_t}{\partial y_t}\right)\left(\frac{\partial y_t}{\partial h_t}\right)\left(\frac{\partial h_t}{\partial h_{t-1}}\right)\cdots\left(\frac{\partial h_2}{\partial h_1}\right)$$

Vanishing gradients

Multiply the *same* matrices at *each* timestep ➔ multiply *many* matrices in the gradients

The Vanilla RNN Backward



$$h_t = \tanh W \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix}$$

$$y_t = \mathrm{F}(h_t)$$

$$C_t = \mathrm{Loss}(y_t, \mathrm{GT}_t)$$

$$\frac{\partial C_t}{\partial h_1} = \left(\frac{\partial C_t}{\partial y_t}\right)\left(\frac{\partial y_t}{\partial h_1}\right)$$

$$= \left(\frac{\partial C_t}{\partial y_t}\right)\left(\frac{\partial y_t}{\partial h_t}\right)\left(\frac{\partial h_t}{\partial h_{t-1}}\right)\cdots\left(\frac{\partial h_2}{\partial h_1}\right)$$

Slide credit: Arun Mallya

Vanishing Gradient Solution: Motivation

$$\frac{\partial C_t}{\partial h_1} = \left(\frac{\partial C_t}{\partial y_t}\right)\left(\frac{\partial y_t}{\partial h_1}\right)$$

$$= \left(\frac{\partial C_t}{\partial y_t}\right)\left(\frac{\partial y_t}{\partial h_t}\right)\left(\frac{\partial h_t}{\partial h_{t-1}}\right)\cdots\left(\frac{\partial h_2}{\partial h_1}\right)$$

$$h_t = \tanh W \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix}$$

$$y_t = \mathrm{F}(h_t)$$

$$C_t = \mathrm{Loss}(y_t, \mathrm{GT}_t)$$

Identity

$$h_t = h_{t-1} + F(x_t)$$

$$\Rightarrow \left(\frac{\partial h_t}{\partial h_{t-1}}\right) = 1$$

The gradient does not decay as the error is propagated all the way back aka "Constant Error Flow"

Slide credit: Arun Mallya

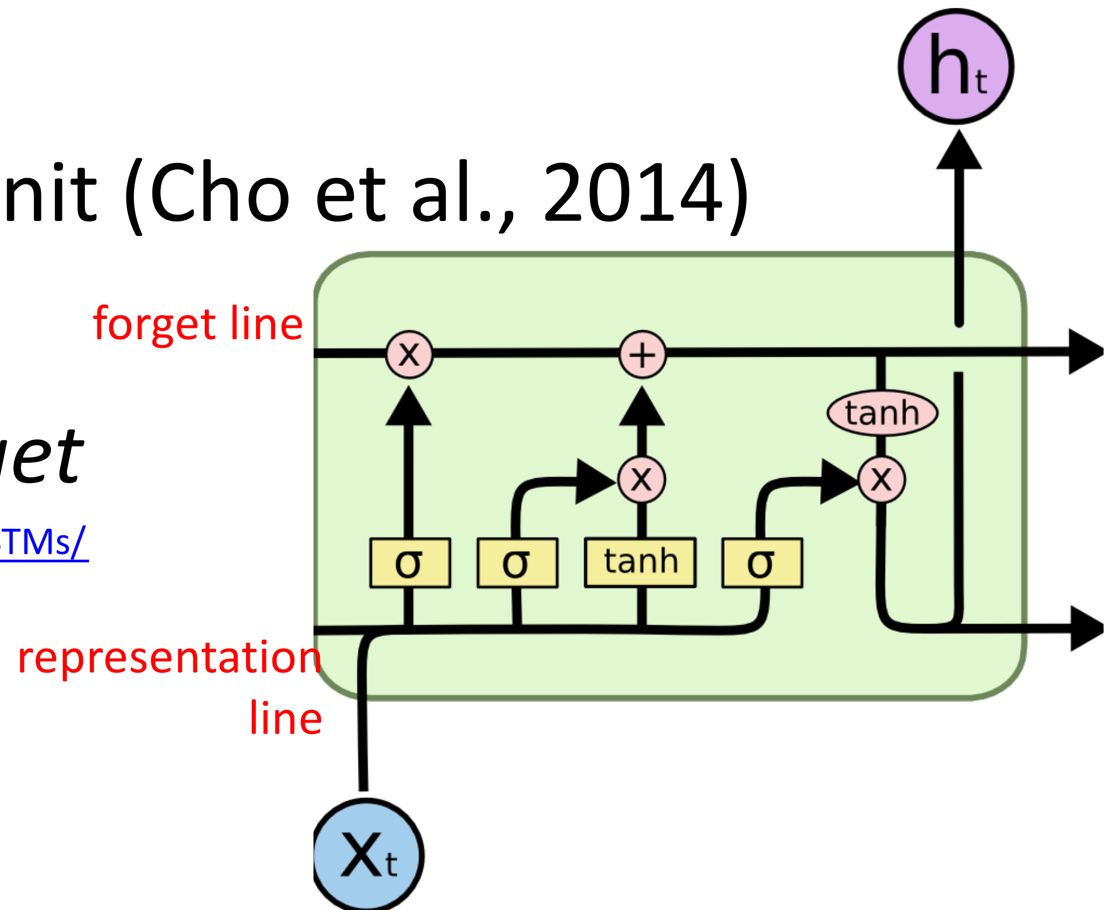
Vanishing Gradient Solution: Model Implementations

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

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

Basic Ideas: *learn to forget*

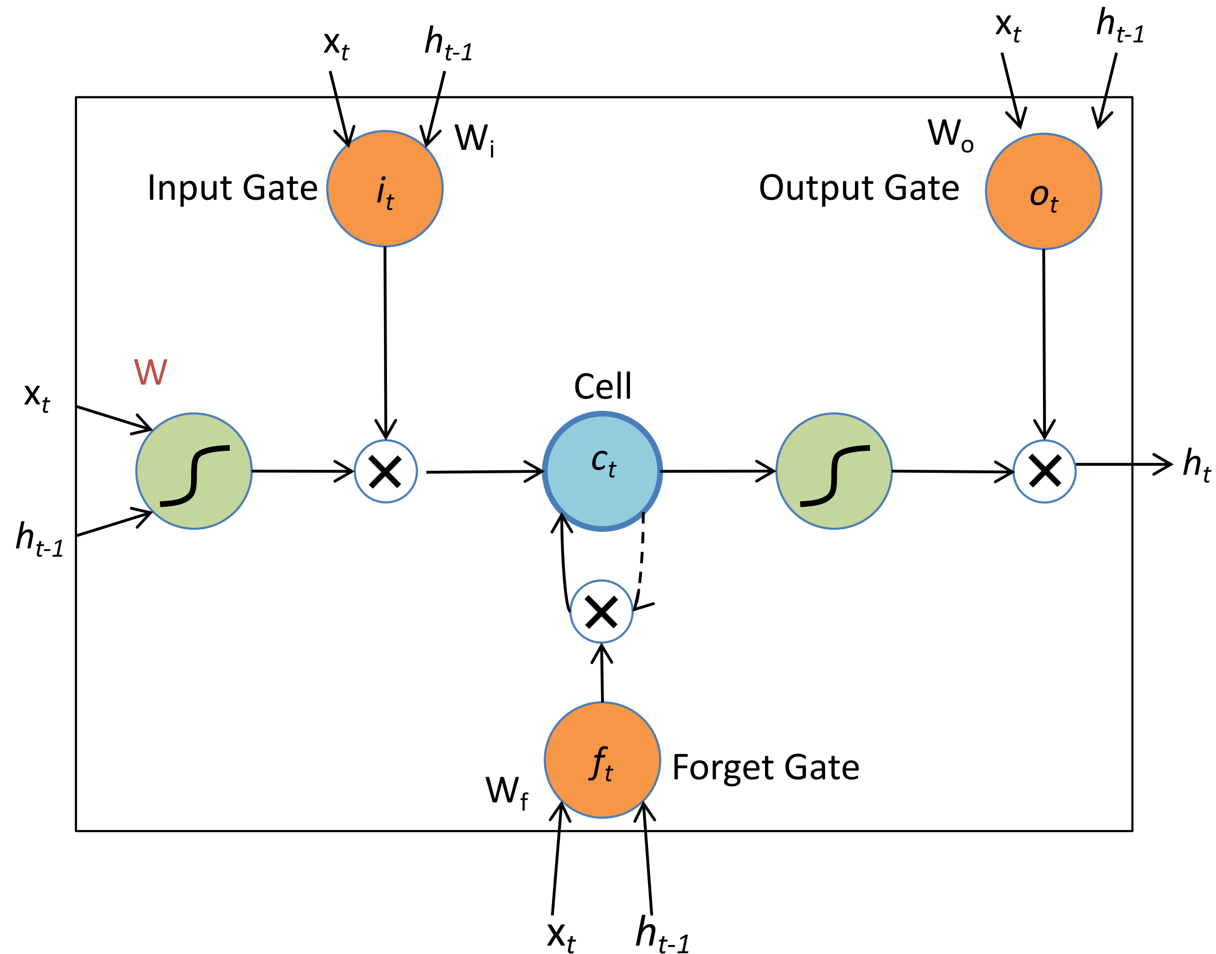
http://colah.github.io/posts/2015-08-Understanding-LSTMs/



Long Short-Term Memory (LSTM): Hochreiter et al., (1997)

Create a "Constant Error Carousel" (CEC) which ensures that gradients don't decay

A memory cell that acts like an accumulator (contains the identity relationship) over time



$$c_t = f_t \otimes c_{t-1} + i_t \otimes \tanh\left(W \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix}\right)$$

$$f_t = \sigma\left(W_f \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} + b_f\right)$$

Slide credit: Arun Mallya

I want to use CNNs/RNNs/Deep Learning in my project. I don’t want to do this all by hand.

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), 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)
```

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), 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)
```

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), 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, 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_categories)
```

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

```
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 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

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

Slide Credit

http://slazebni.cs.illinois.edu/spring17/lec01_cnn_architectures.pdf

http://slazebni.cs.illinois.edu/spring17/lec02_rnn.pdf