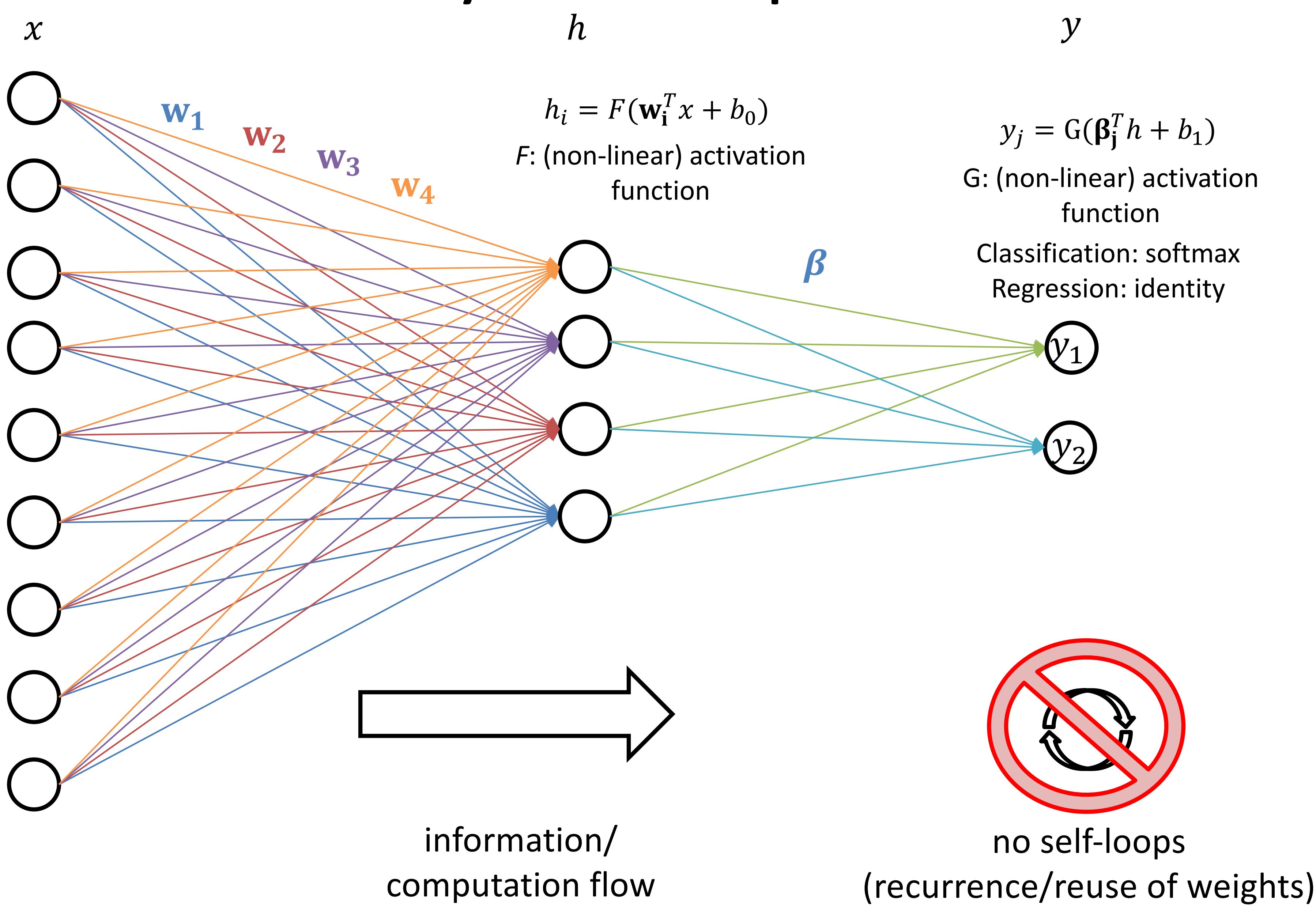
Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs)

CMSC 678

UMBC

Recap from last time…

Feed-Forward Neural Network: Multilayer Perceptron



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
   $g_t += l'(x_i)$

done
Get scaling factor ρ_t
Set $\theta_{t+1} = \theta_t - \rho_t * g_t$
Set $t += 1$

"Batch"

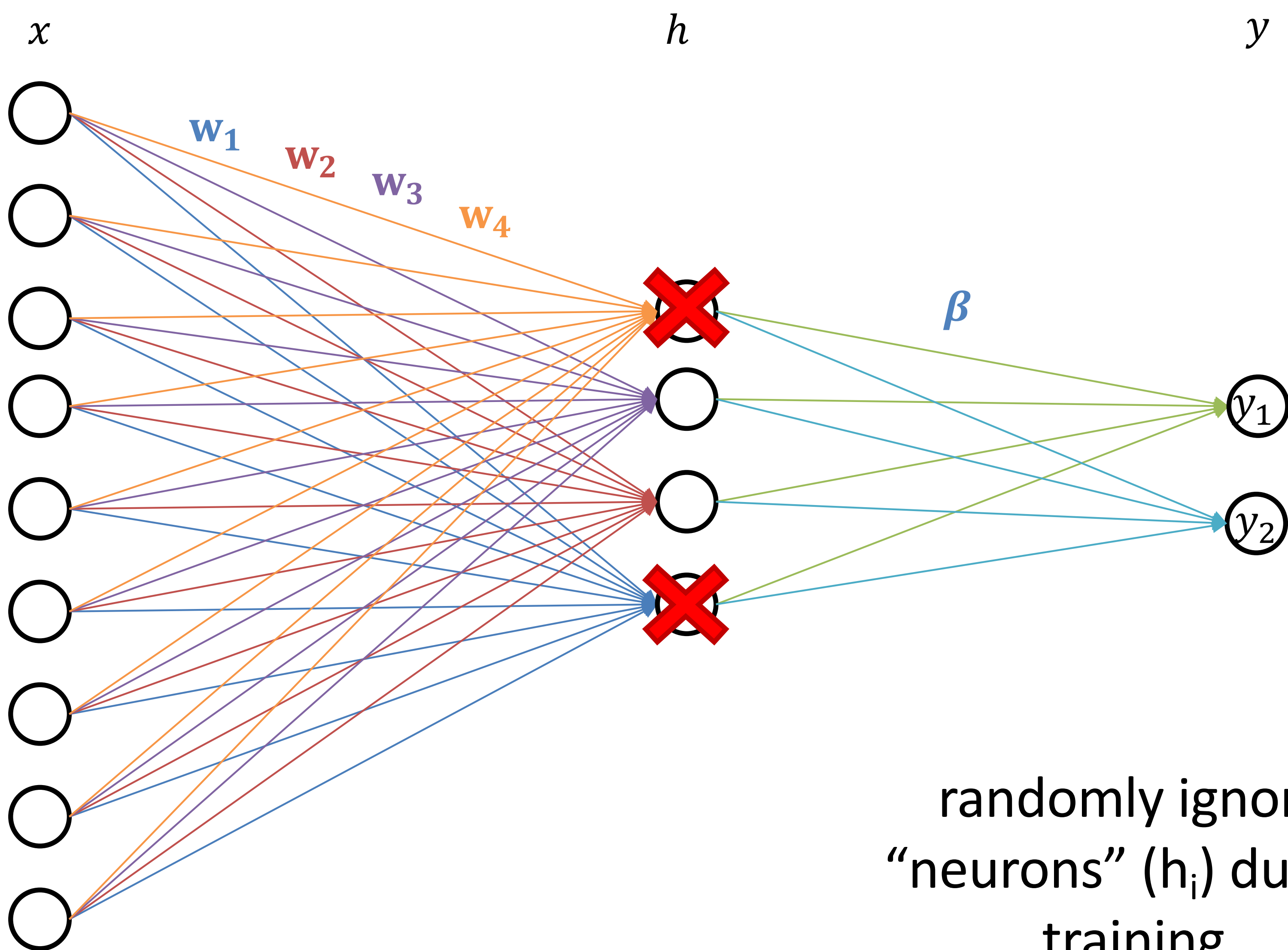
Set $t = 0$
Pick a starting value θ_t
Until converged:

set $g_t = 0$
for example(s) i in full data:

1. Compute loss l on x_i
2. Accumulate gradient
   $g_t += l'(x_i)$

done
Get scaling factor ρ_t
Set $\theta_{t+1} = \theta_t - \rho_t * g_t$
Set $t += 1$

Dropout: Regularization in Neural Networks

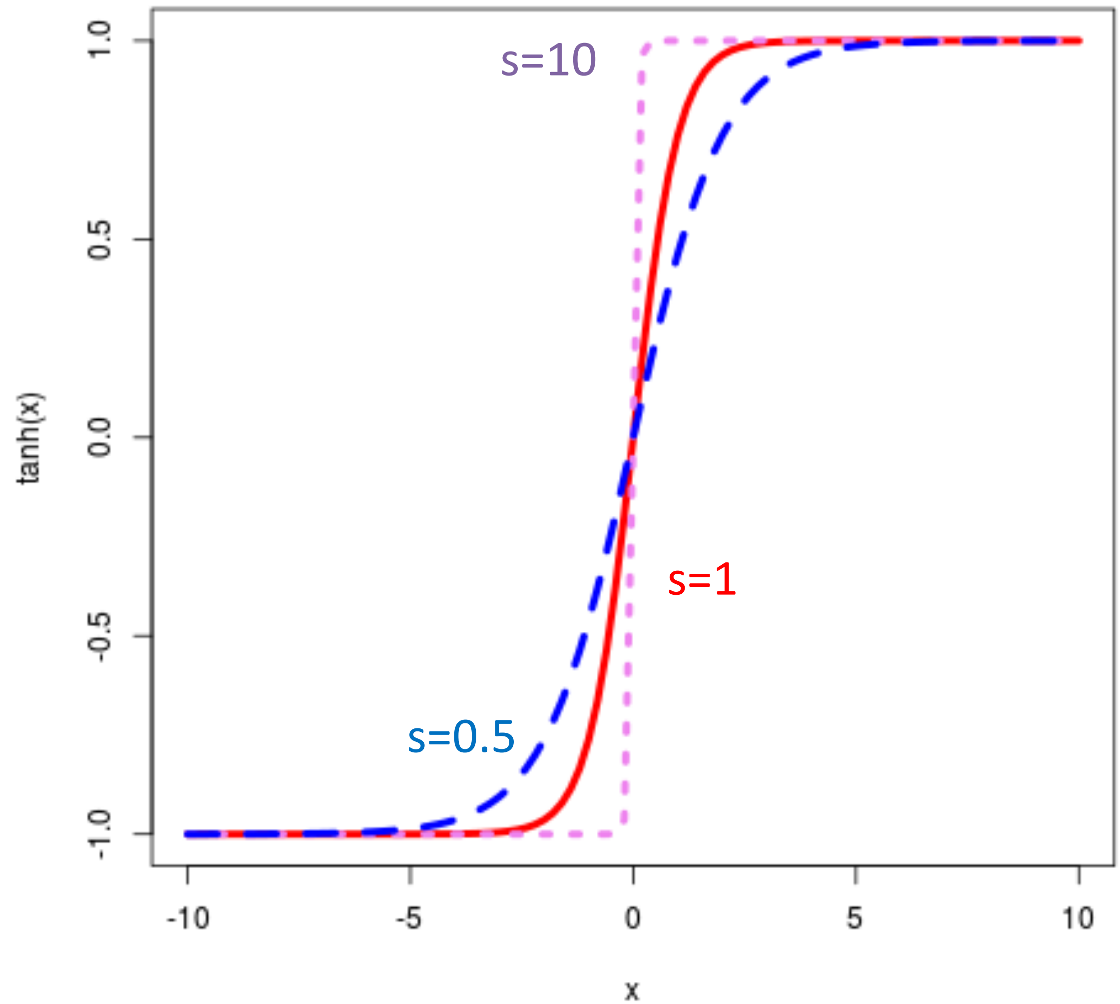


randomly ignore "neurons" (h_i) during training

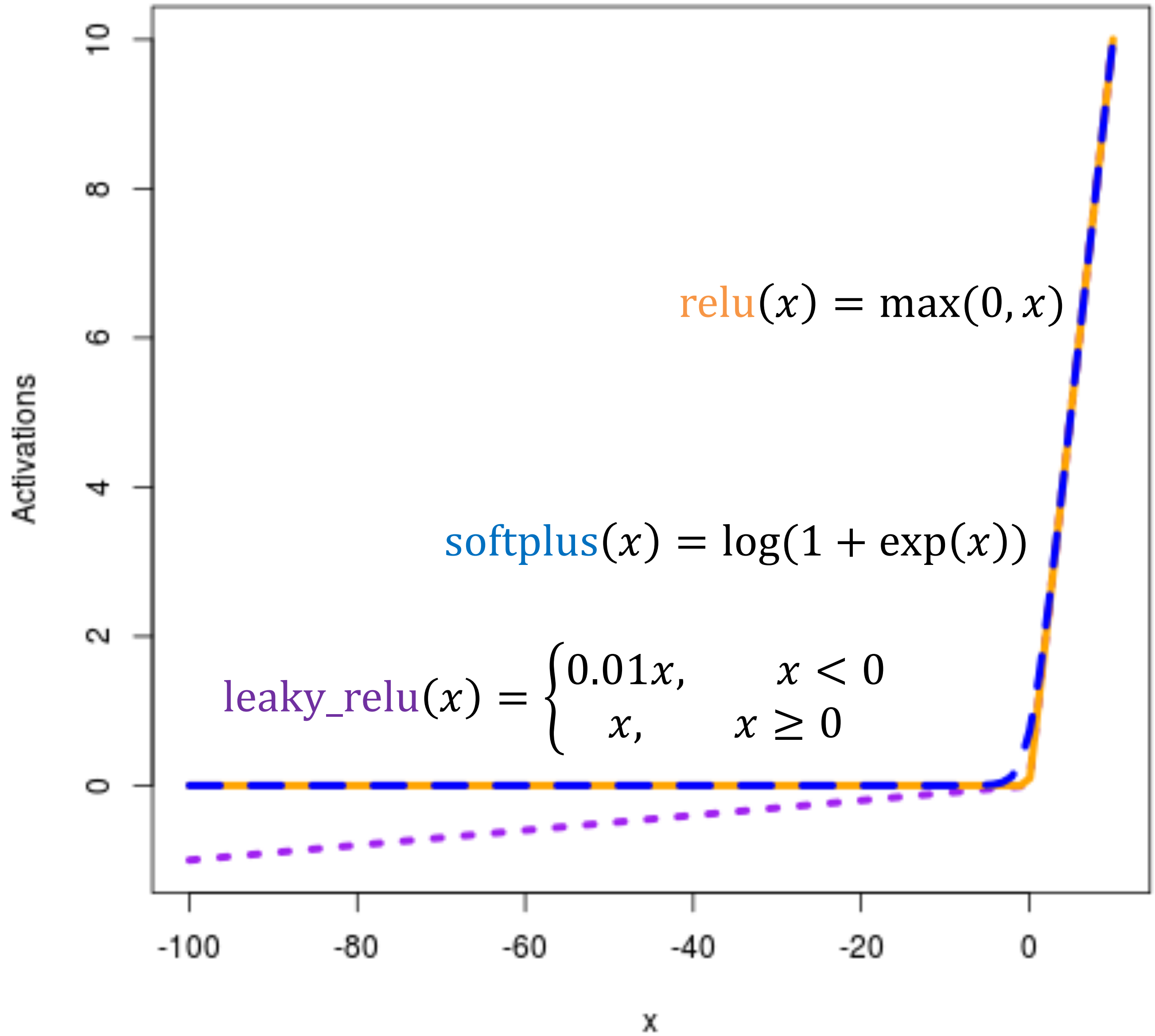
tanh Activation

$$\tanh_s(x) = \frac{2}{1 + \exp(-2 * s * x)} - 1$$

$$= 2\sigma_s(x) - 1$$



Rectifiers Activations



Outline

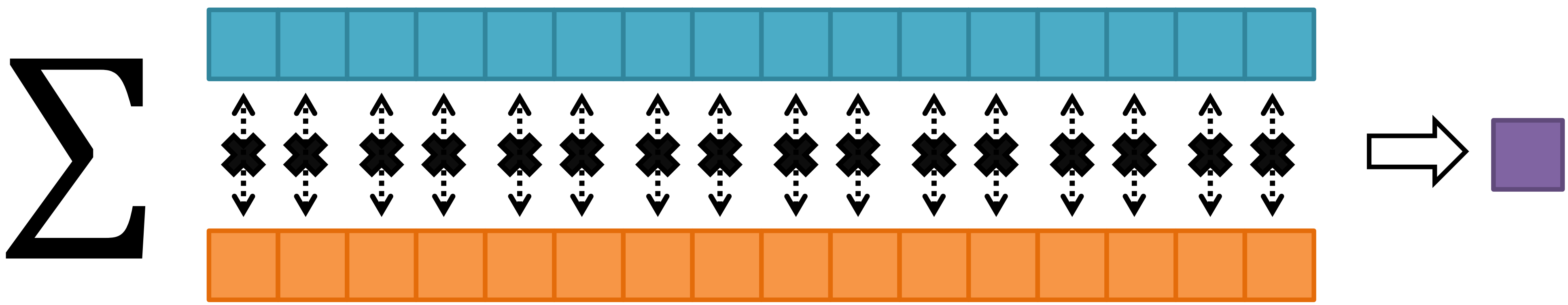
Convolutional Neural Networks

- 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

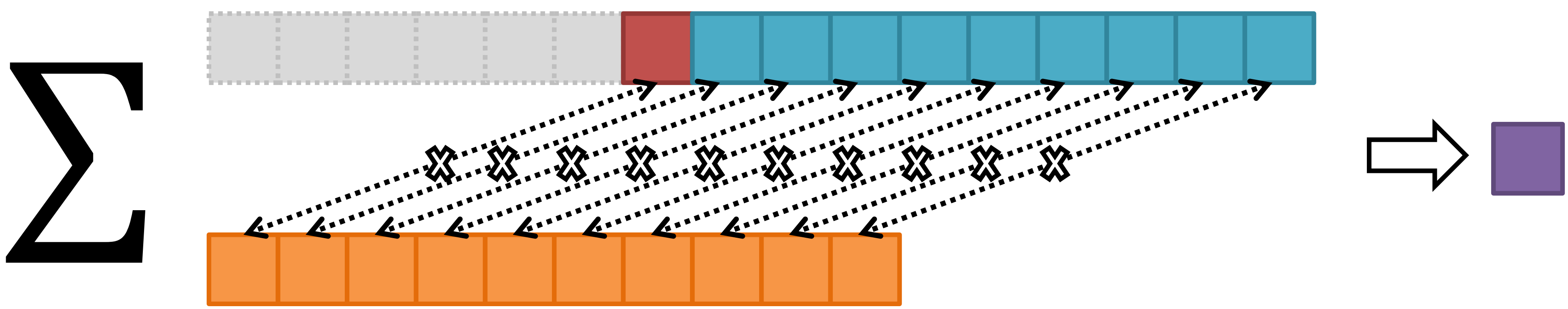
Dot Product



$$x^T y = \sum_k x_k y_k$$

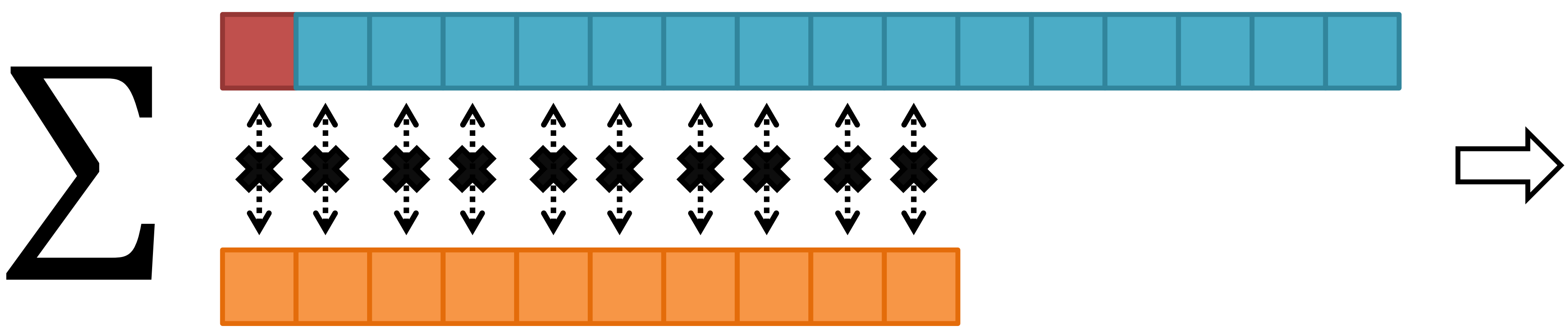
Convolution: Modified Dot Product Around a Point

$$(x^T y)_i = \sum_{k<K} x_{k+i} y_k$$



Convolution: Modified Dot Product Around a Point

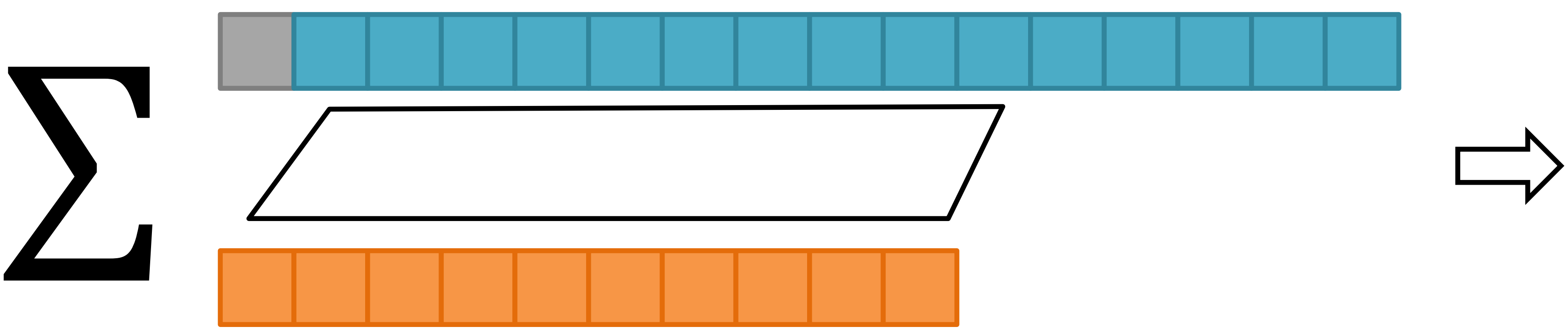
$$(x^T y)_i = \sum_k x_{k+i} y_k$$

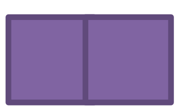


$$(x \star y)[i] =$$

Convolution: Modified Dot Product Around a Point

$$(x^T y)_i = \sum_k x_{k+i} y_k$$

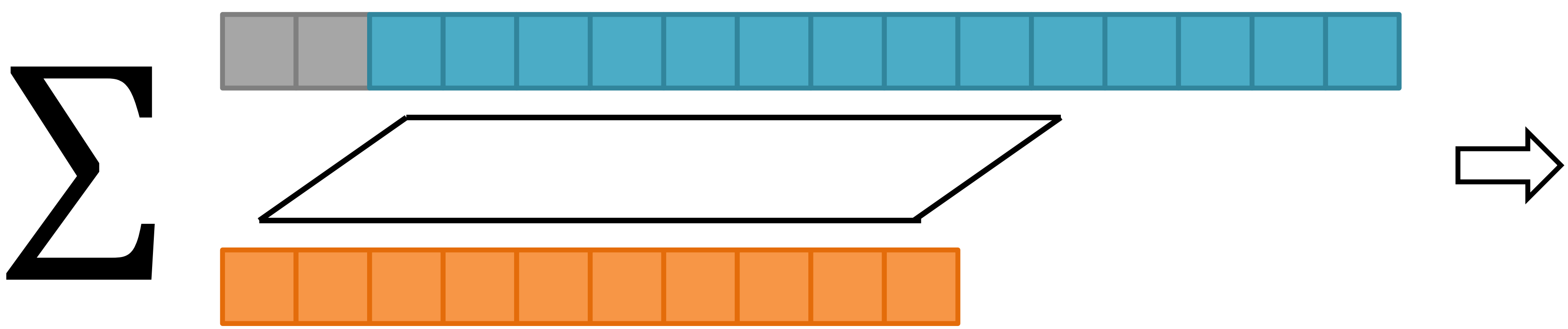


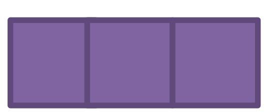
$(x \star y)[i] =$ 

Convolution/cross-correlation

Convolution: Modified Dot Product Around a Point

$$(x^T y)_i = \sum_k x_{k+i} y_k$$

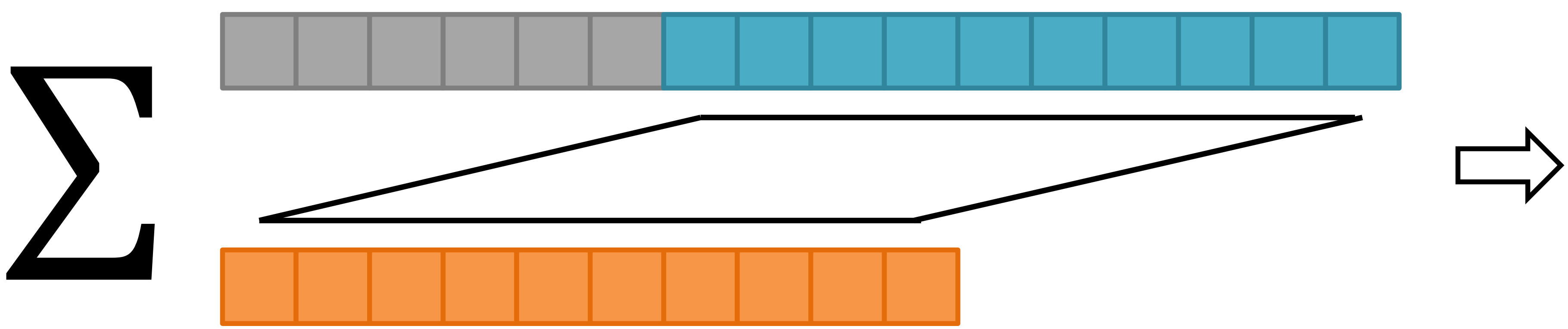


$(x \star y)[i] =$ 

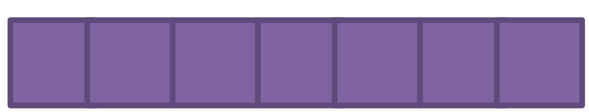
Convolution/cross-correlation

Convolution: Modified Dot Product Around a Point

$$(x^T y)_i = \sum_k x_{k+i} y_k$$



$(x \star y)[i] =$

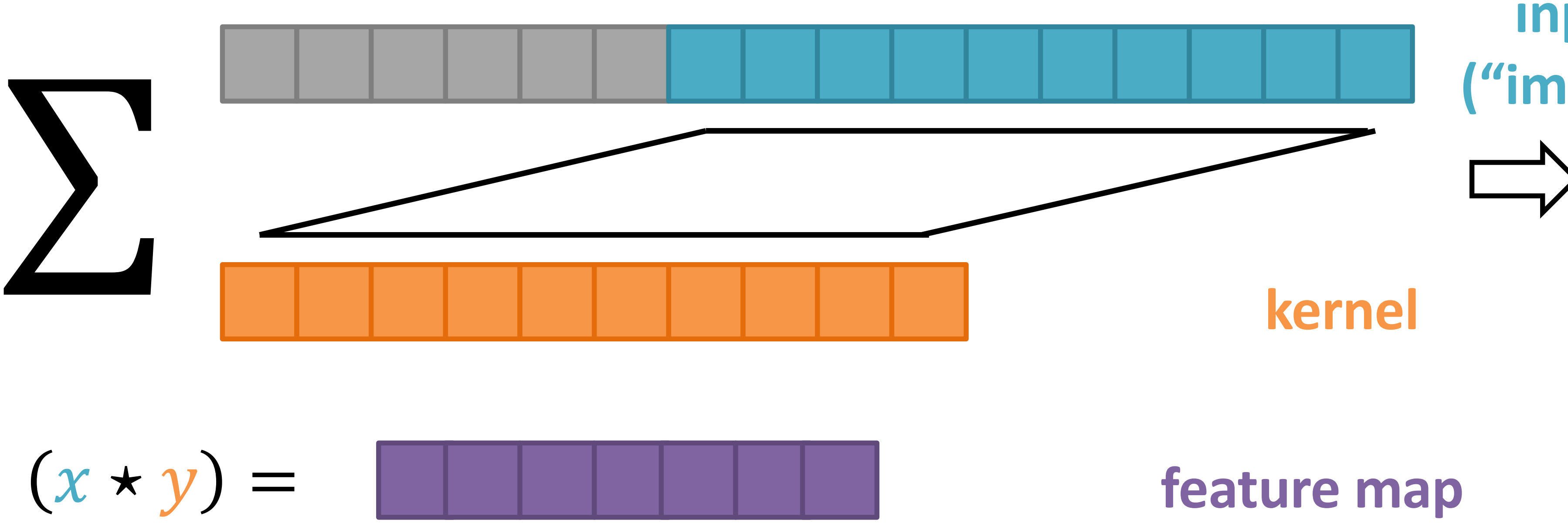


Convolution/cross-correlation

Convolution: Modified Dot Product Around a Point

$$(x^T y)_i = \sum_k x_{k+i} y_k$$

1-D convolution



$(x \star y) =$

feature map

Convolution/cross-correlation

Outline

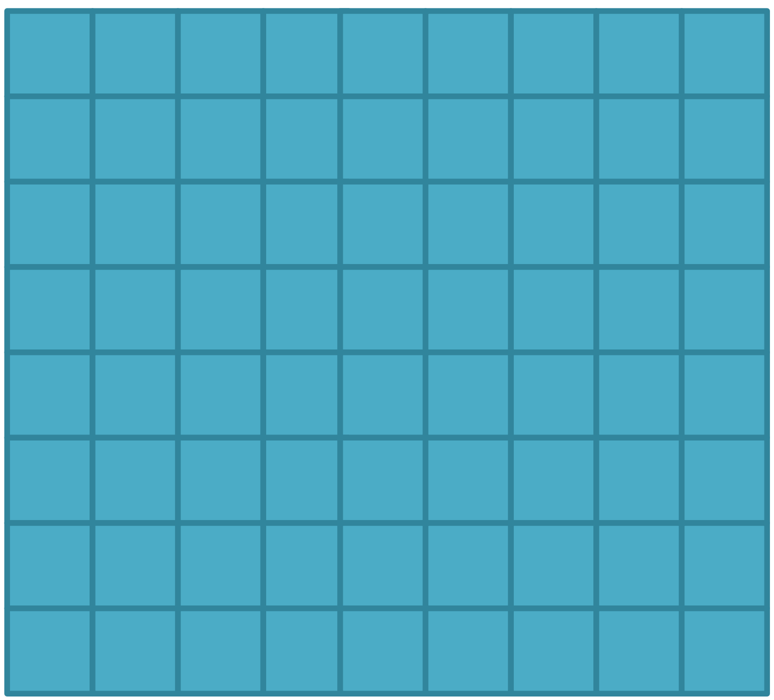
Convolutional Neural Networks

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

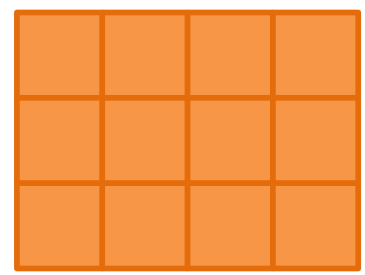
Recurrent Neural Networks

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

2-D Convolution



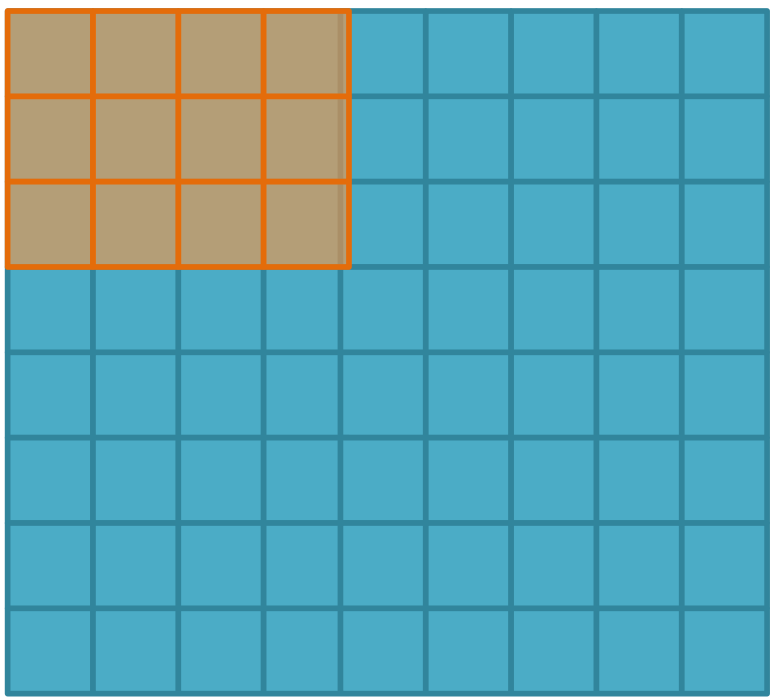
input
(“image”)



kernel

width: shape of the kernel
(often square)

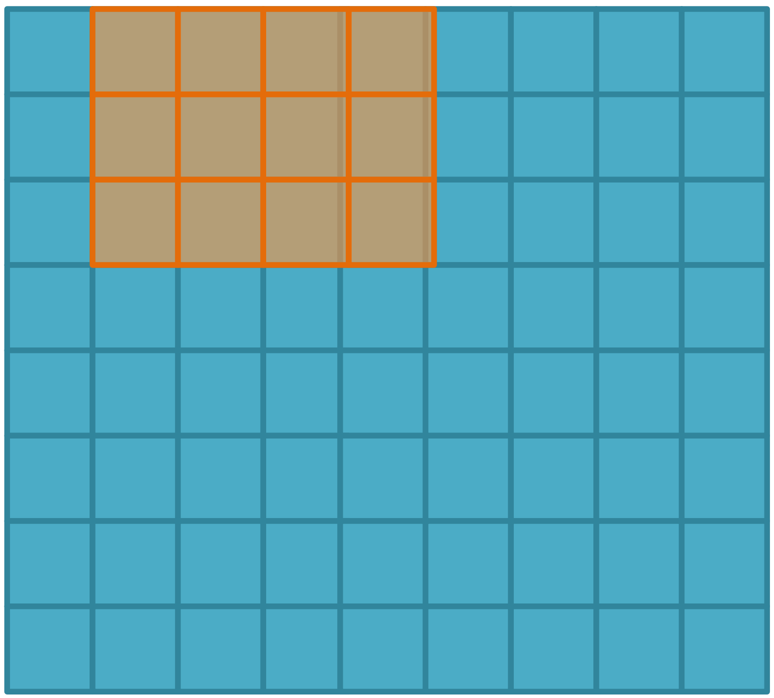
2-D Convolution



stride(s): how many spaces to move the kernel

width: shape of the kernel (often square)

2-D Convolution



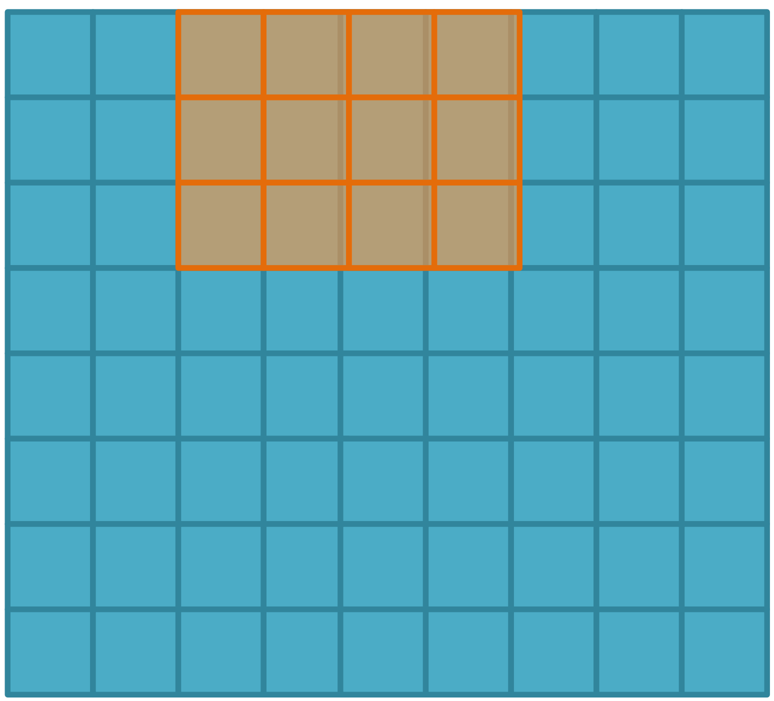
input
("image")

stride(s): how many spaces to move the kernel

stride=1

width: shape of the kernel (often square)

2-D Convolution



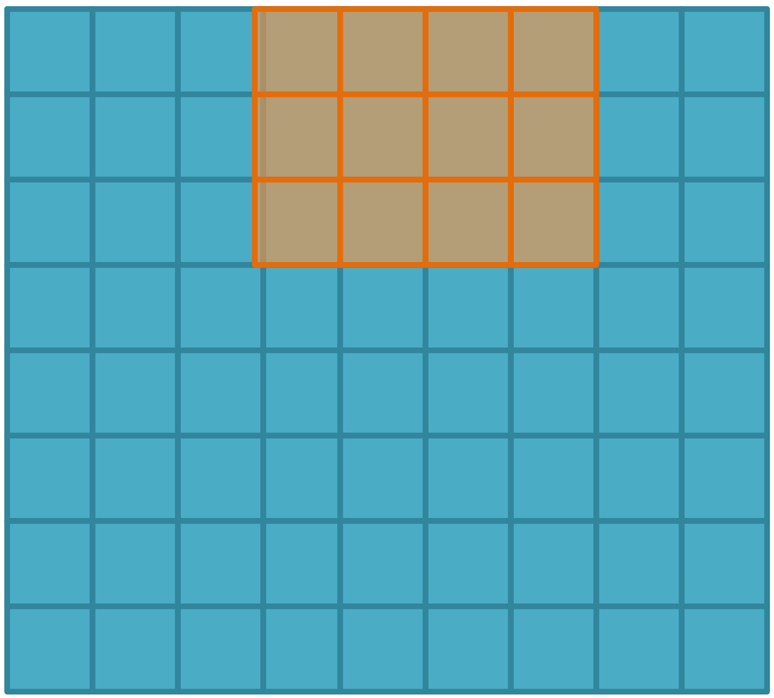
input
(“image”)

stride(s): how many spaces to move the kernel

stride=1

width: shape of the kernel (often square)

2-D Convolution

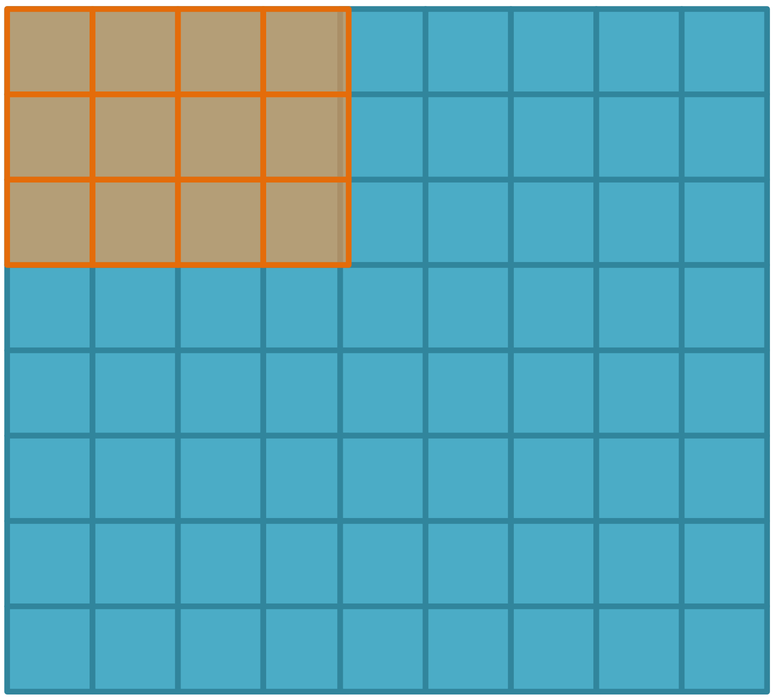


stride(s): how many spaces to move the kernel

stride=1

width: shape of the kernel (often square)

2-D Convolution



input
("image")

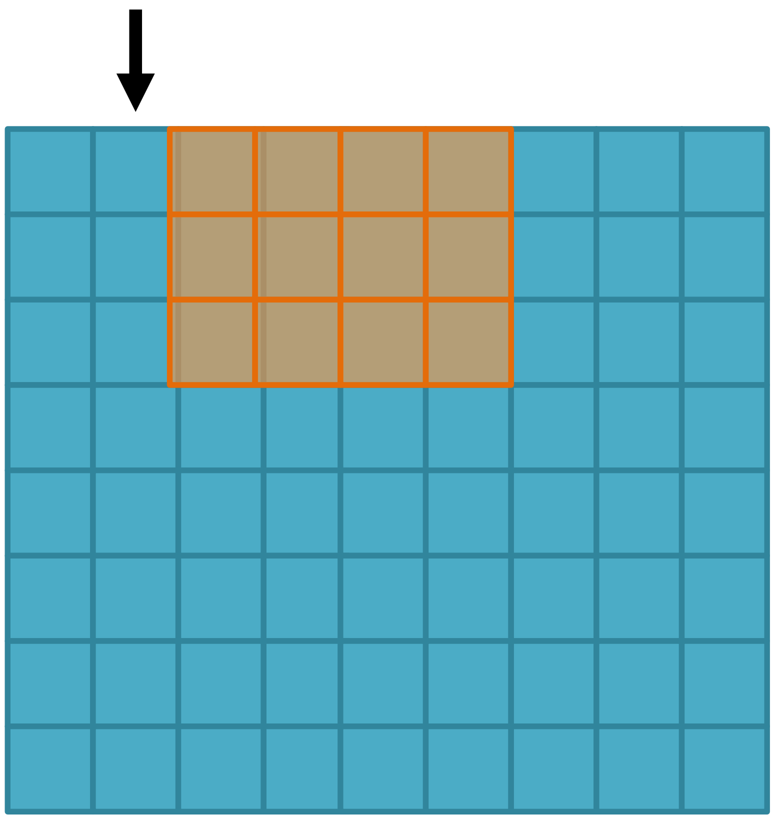
stride(s): how many spaces to move the kernel

stride=2

width: shape of the kernel (often square)

2-D Convolution

skip starting here



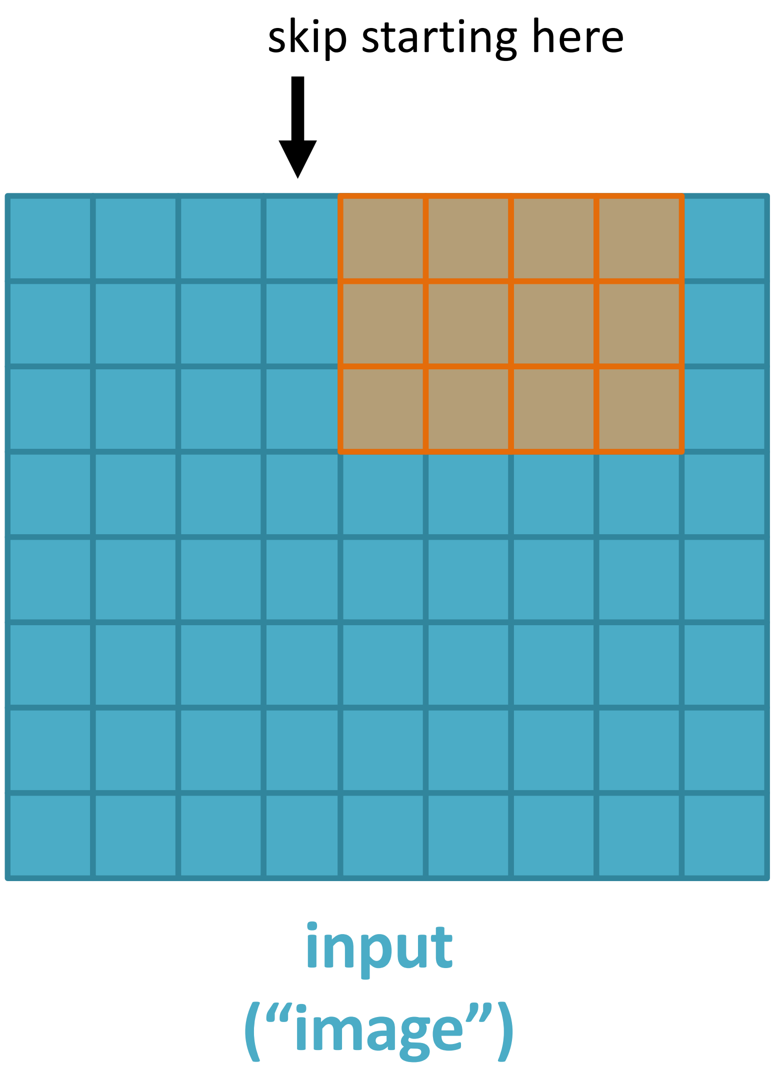
input
("image")

stride(s): how many spaces to move the kernel

stride=2

width: shape of the kernel (often square)

2-D Convolution



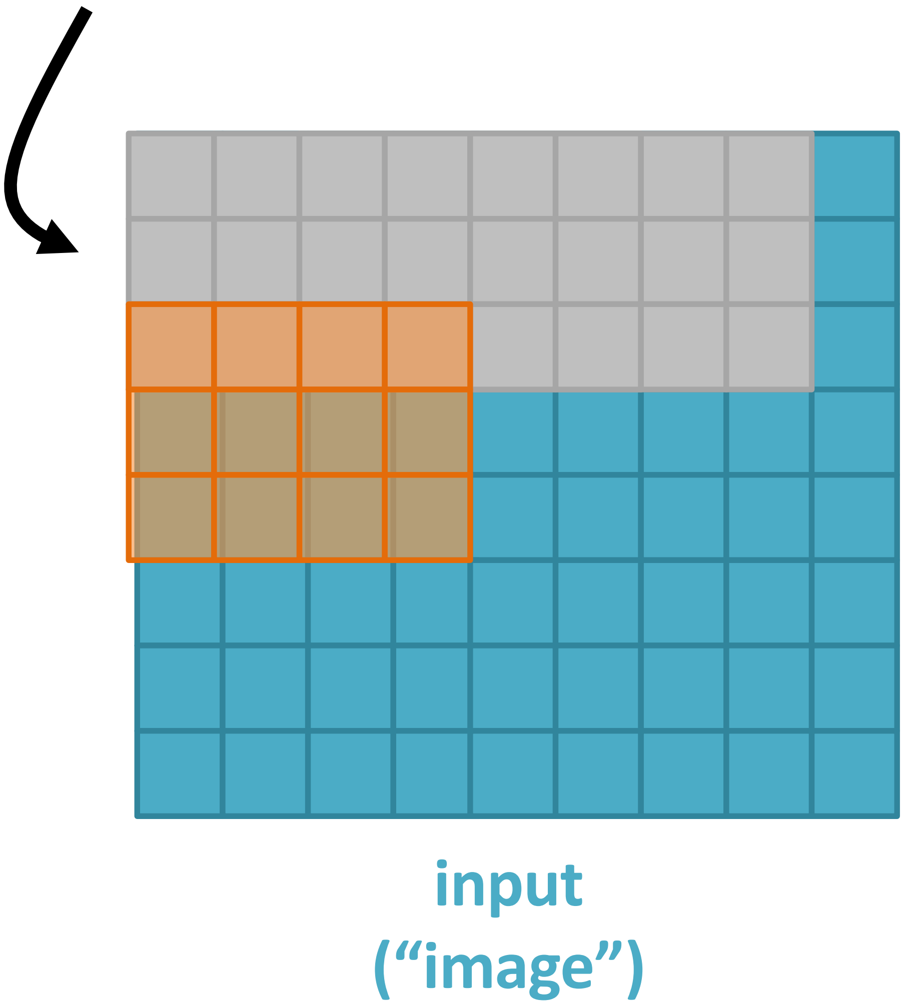
stride(s): how many spaces to move the kernel

stride=2

width: shape of the kernel (often square)

2-D Convolution

skip starting here



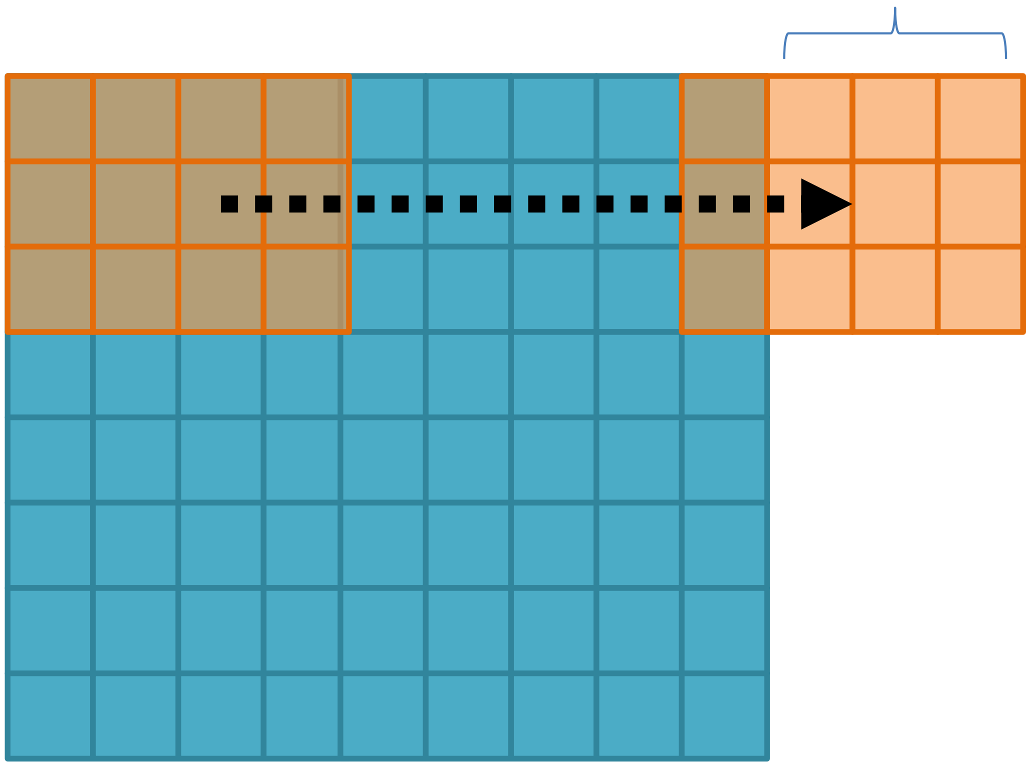
stride(s): how many spaces to move the kernel

stride=2

width: shape of the kernel (often square)

2-D Convolution

pad with 0s
(one option)



input
("image")

stride(s): how many spaces to move the kernel

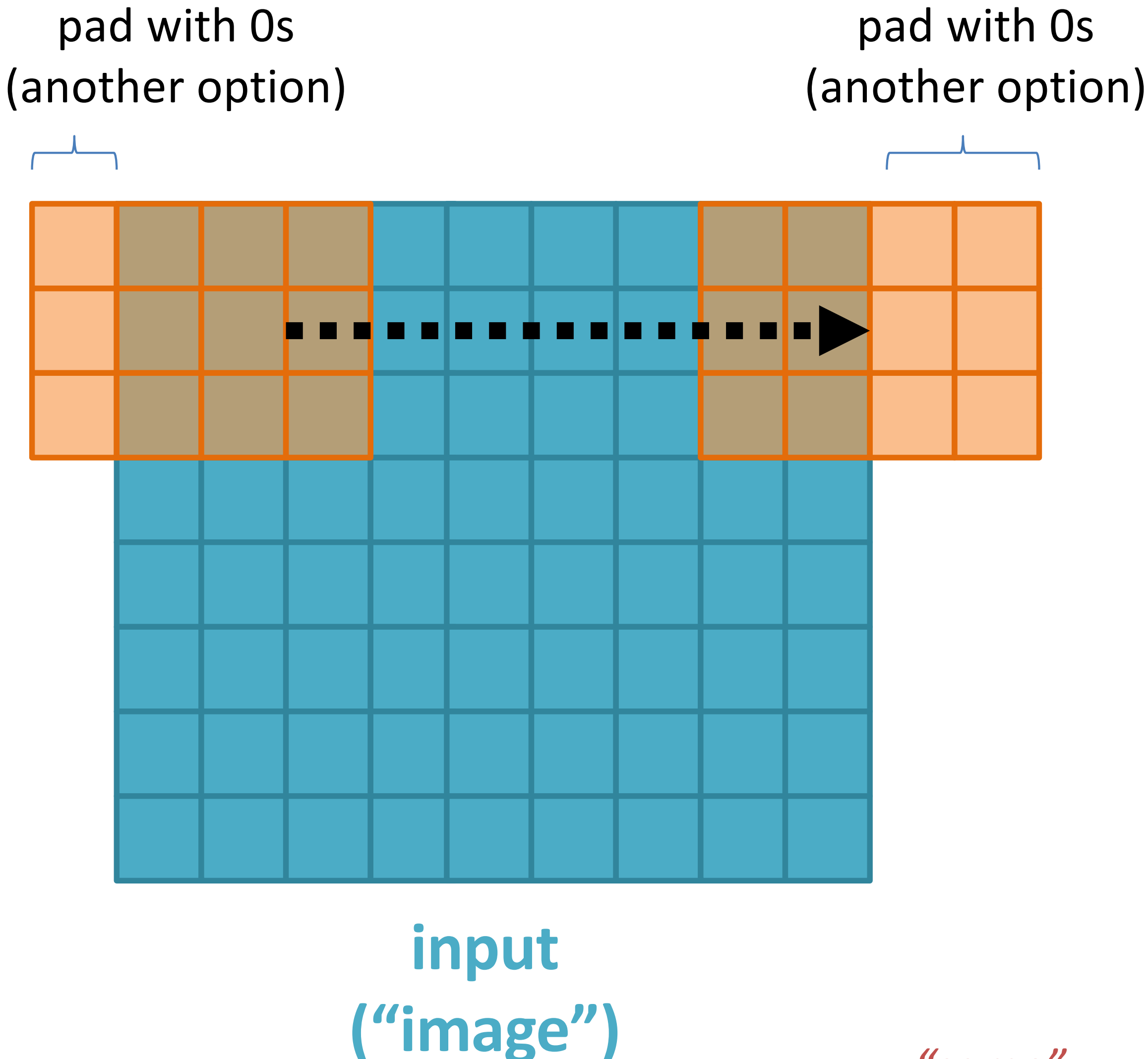
padding: how to handle input/kernel shape mismatches

width: shape of the kernel (often square)

"same":
input.shape == output.shape

"different":
input.shape ≠ output.shape

2-D Convolution



stride(s): how many spaces to move the kernel

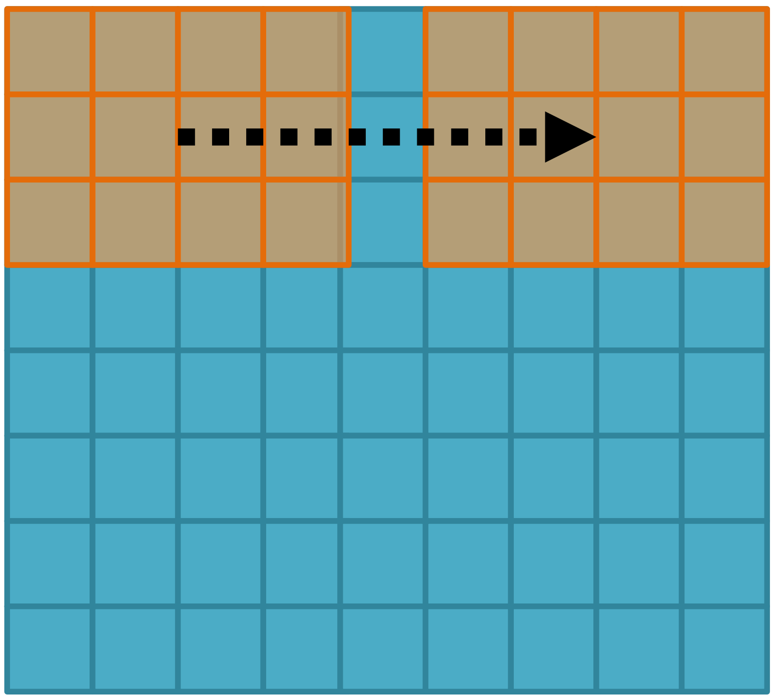
padding: how to handle input/kernel shape mismatches

width: shape of the kernel (often square)

"same": input.shape == output.shape

"different": input.shape ≠ output.shape

2-D Convolution



input
("image")

stride(s): how many spaces to move the kernel

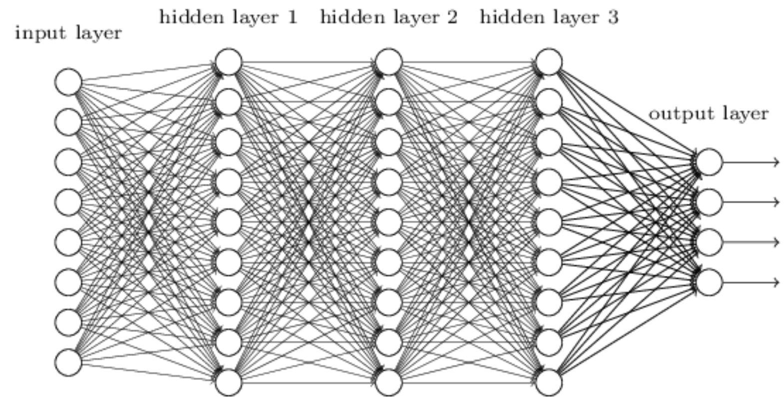
padding: how to handle input/kernel shape mismatches

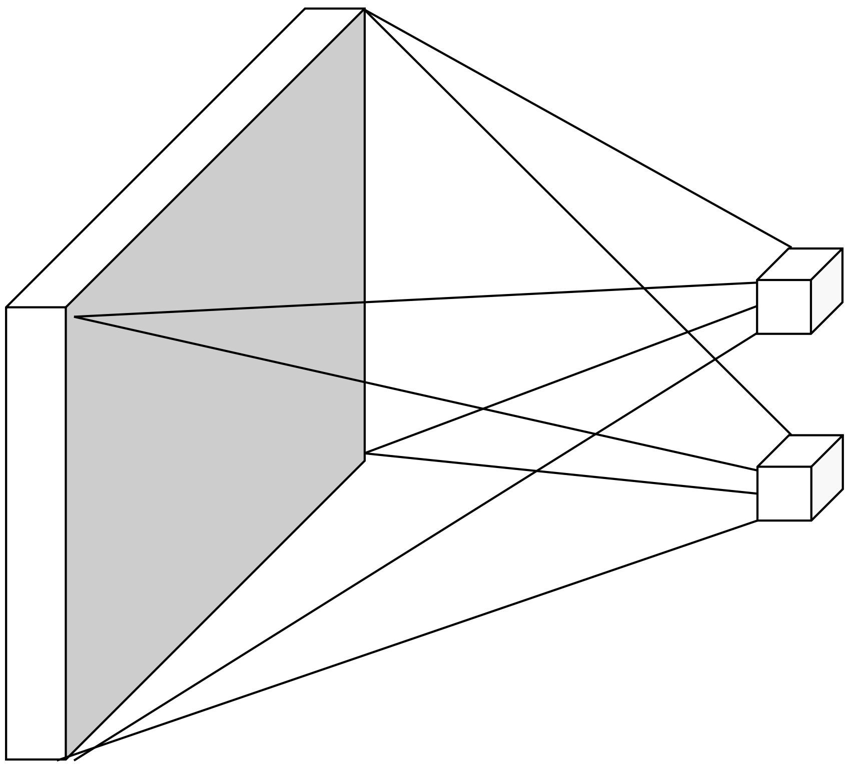
width: shape of the kernel (often square)

"same":
input.shape == output.shape

"different":
input.shape ≠ output.shape

From fully connected to convolutional networks



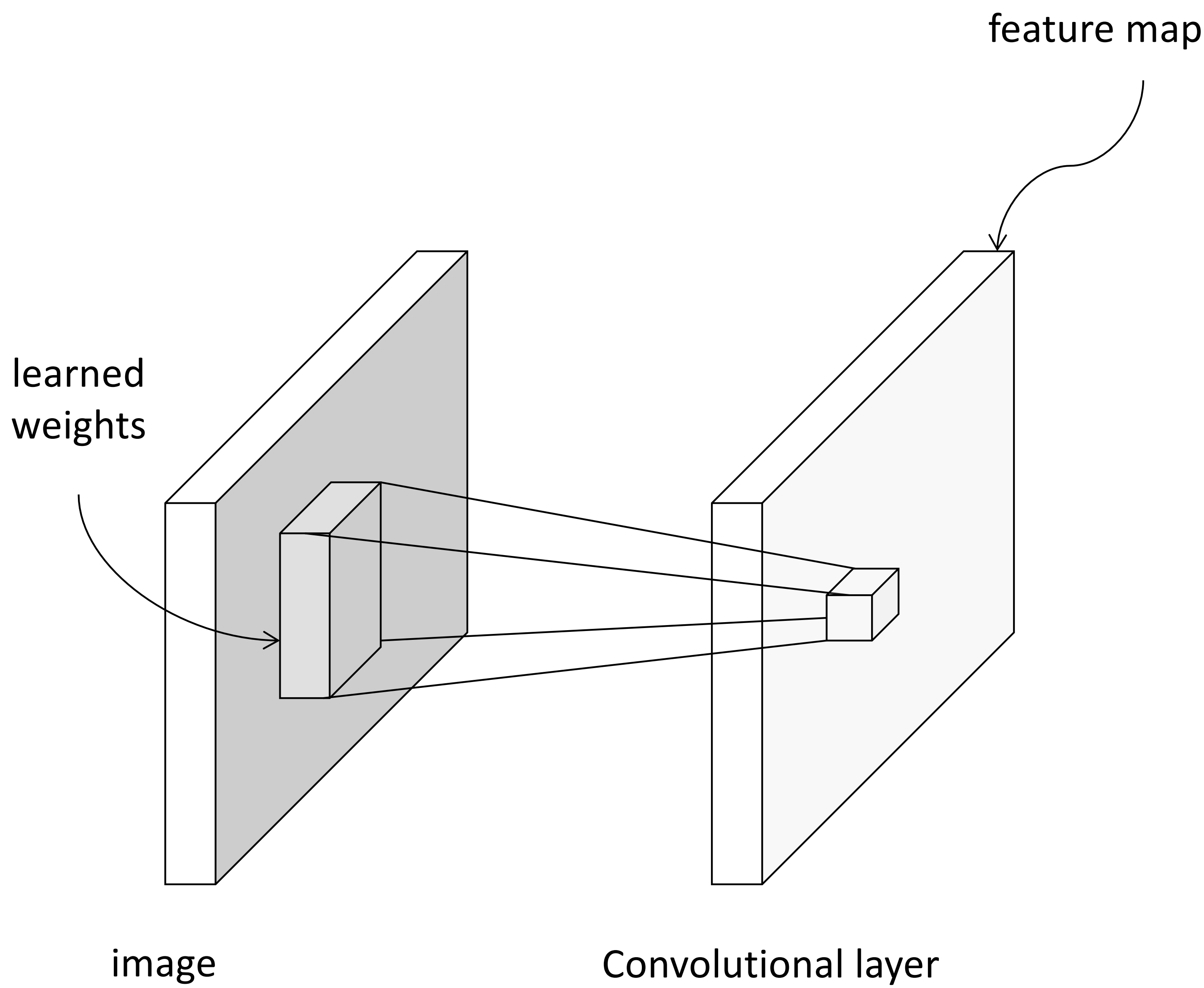


image

Fully connected layer

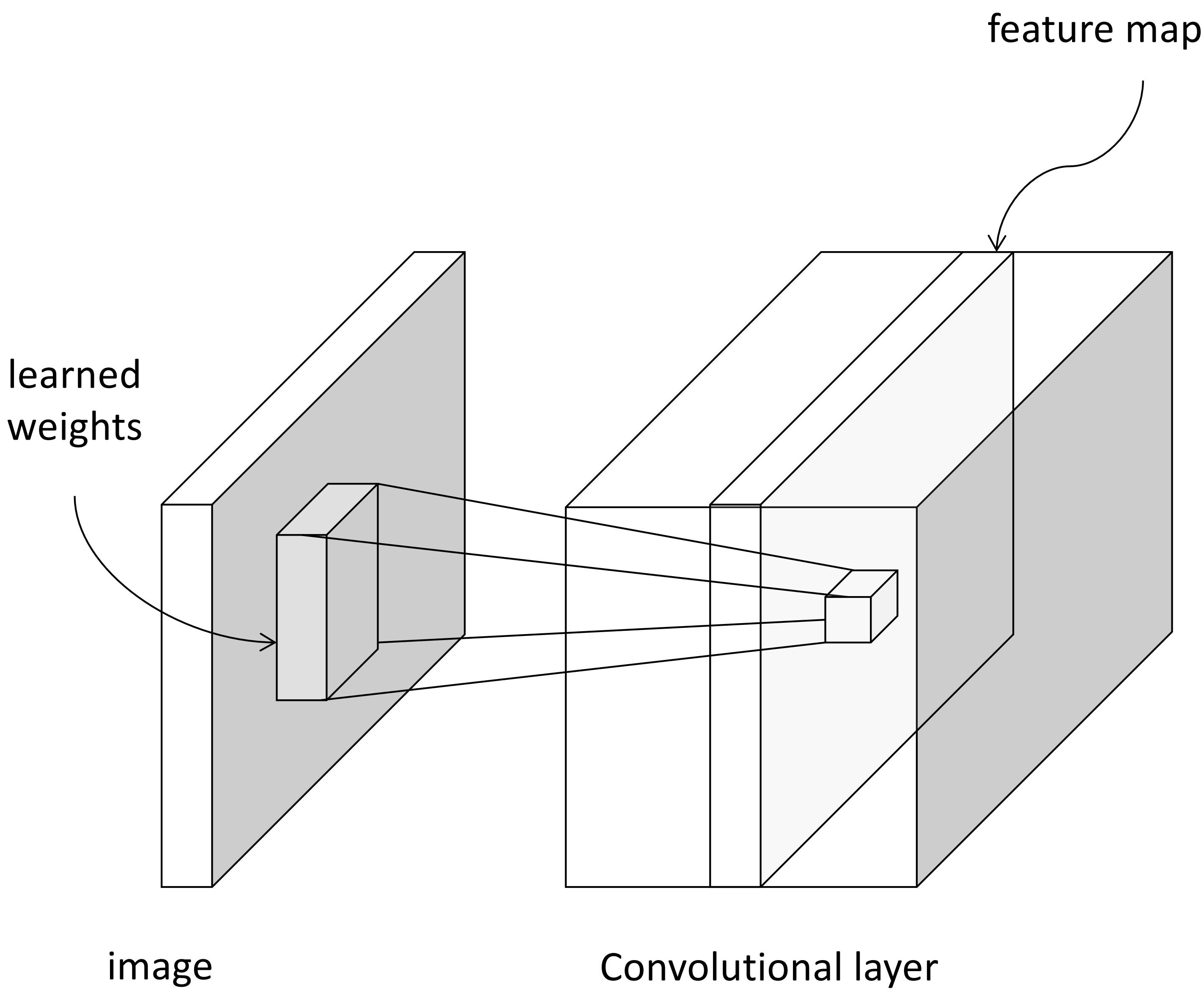
Slide credit: Svetlana Lazebnik

From fully connected to convolutional networks



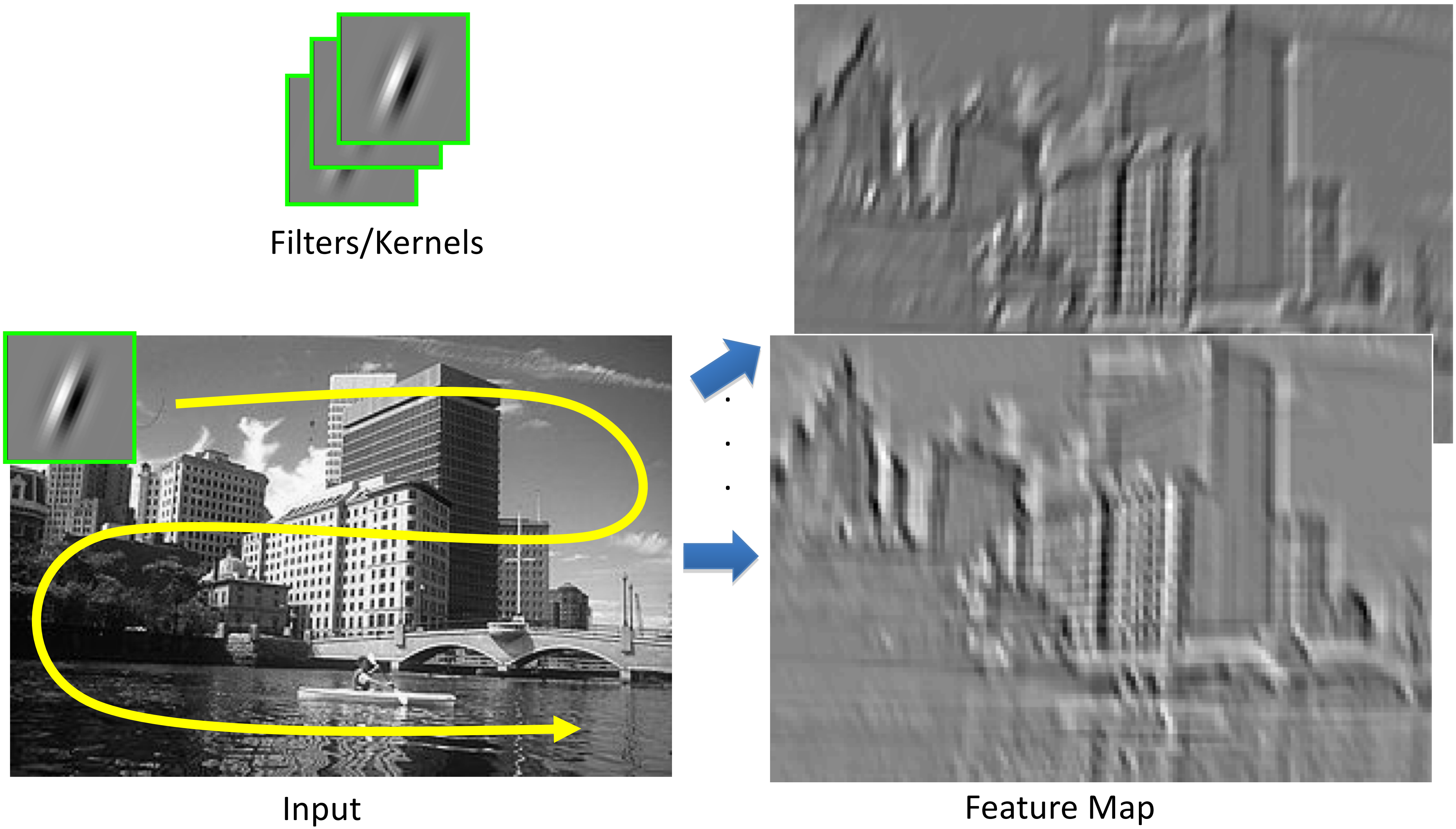
Slide credit: Svetlana Lazebnik

From fully connected to convolutional networks



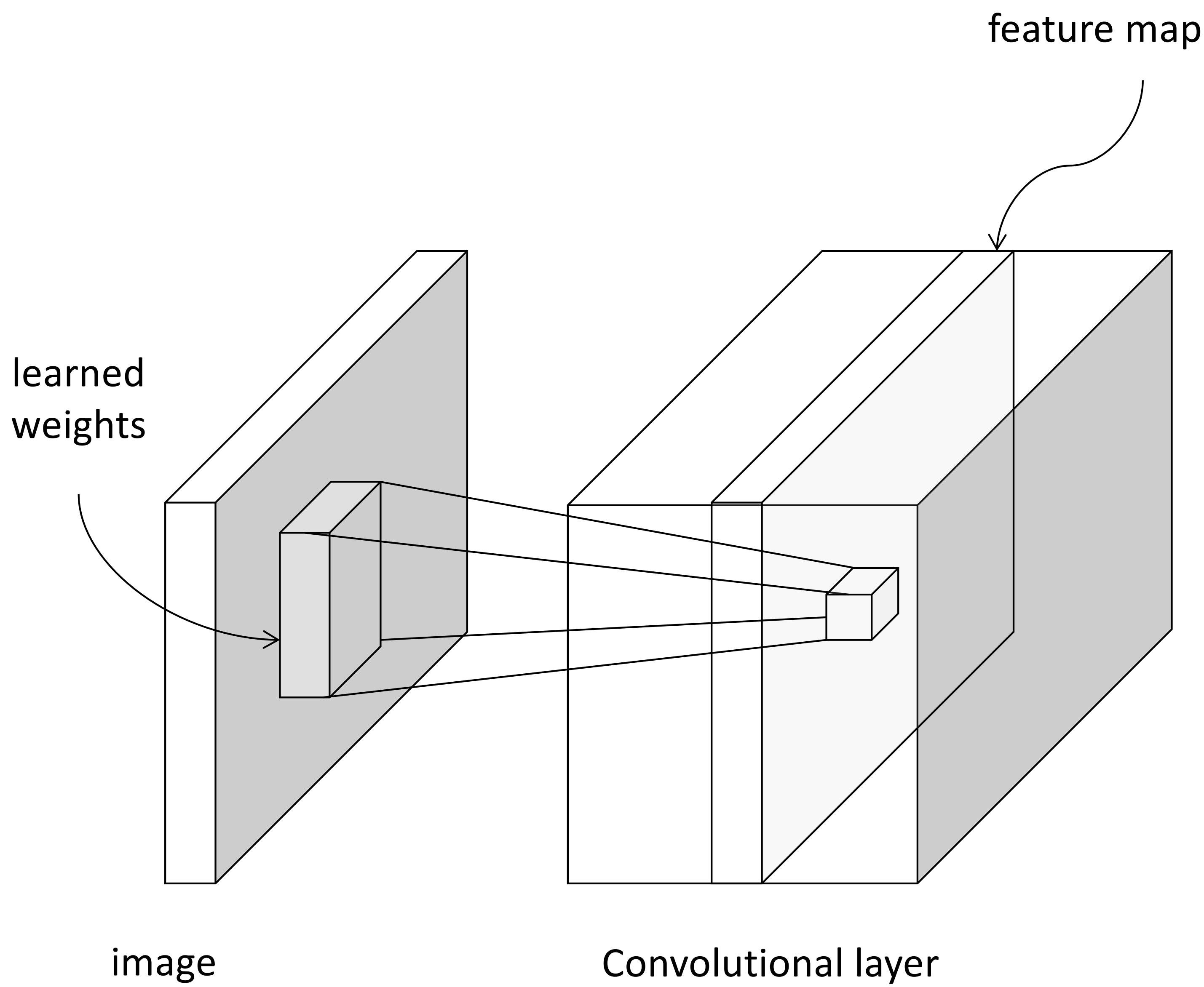
Slide credit: Svetlana Lazebnik

Convolution as feature extraction



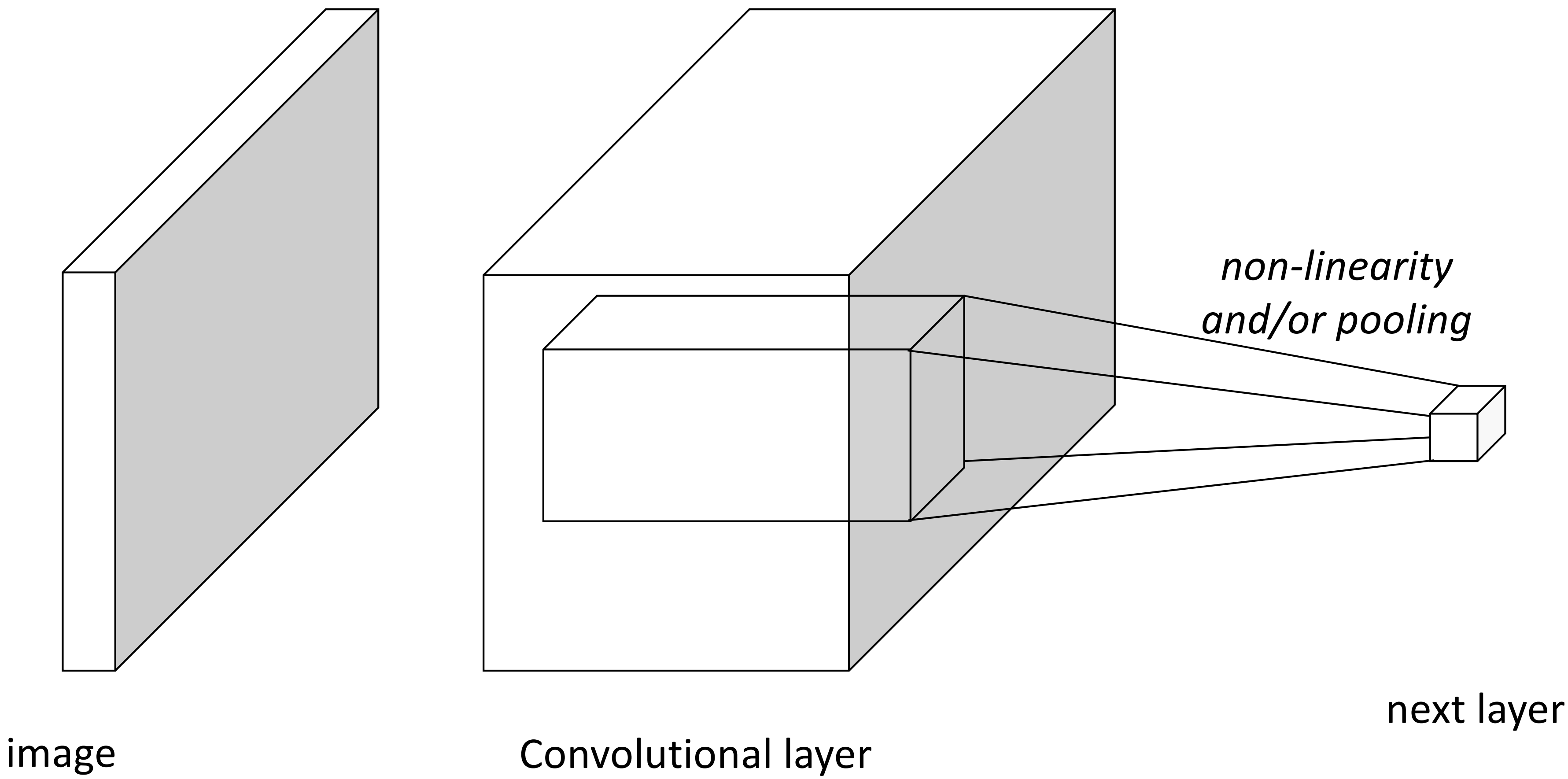
Slide credit: Svetlana Lazebnik

From fully connected to convolutional networks



Slide credit: Svetlana Lazebnik

From fully connected to convolutional networks



Slide adapted: Svetlana Lazebnik

Outline

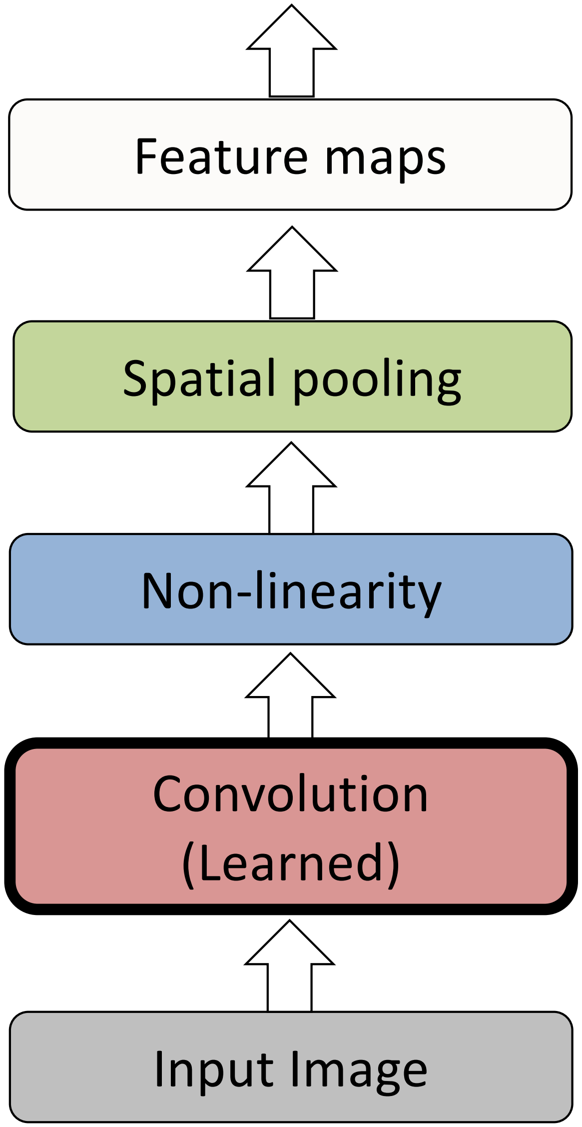
Convolutional Neural Networks

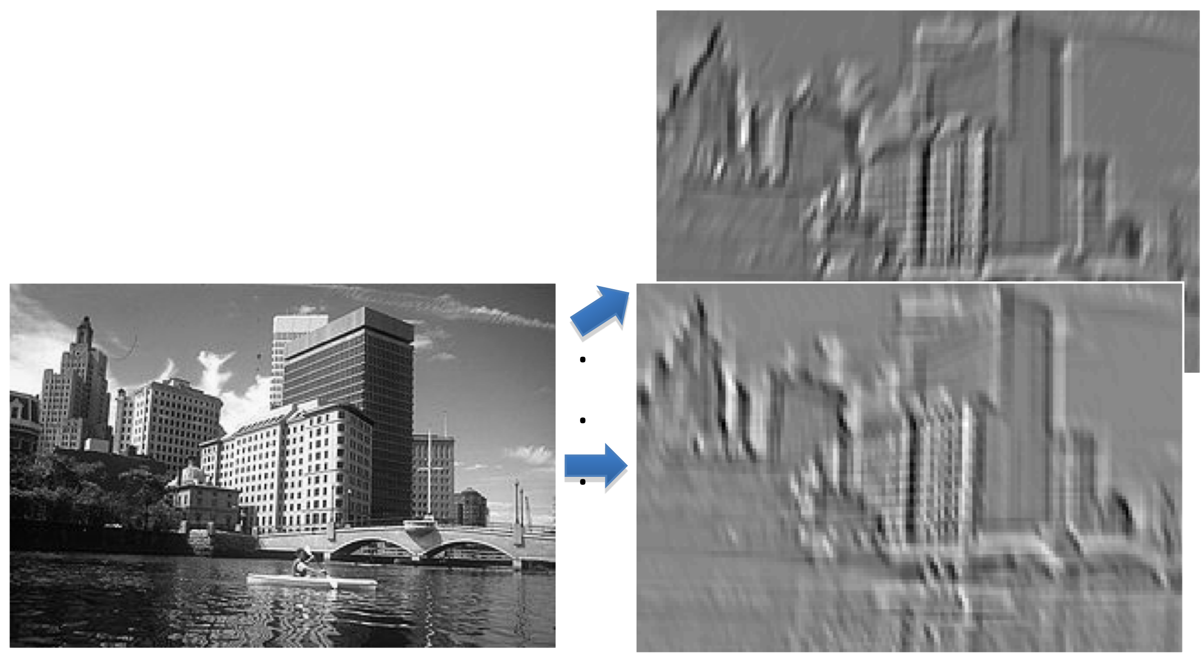
- What *is* a convolution?
- Multidimensional Convolutions
- Typical Convnet Operations
- Deep convnets

Recurrent Neural Networks

- Types of recurrence
- A basic recurrent cell
- BPTT: Backpropagation through time
- Solving vanishing gradients problem

Key operations in a CNN



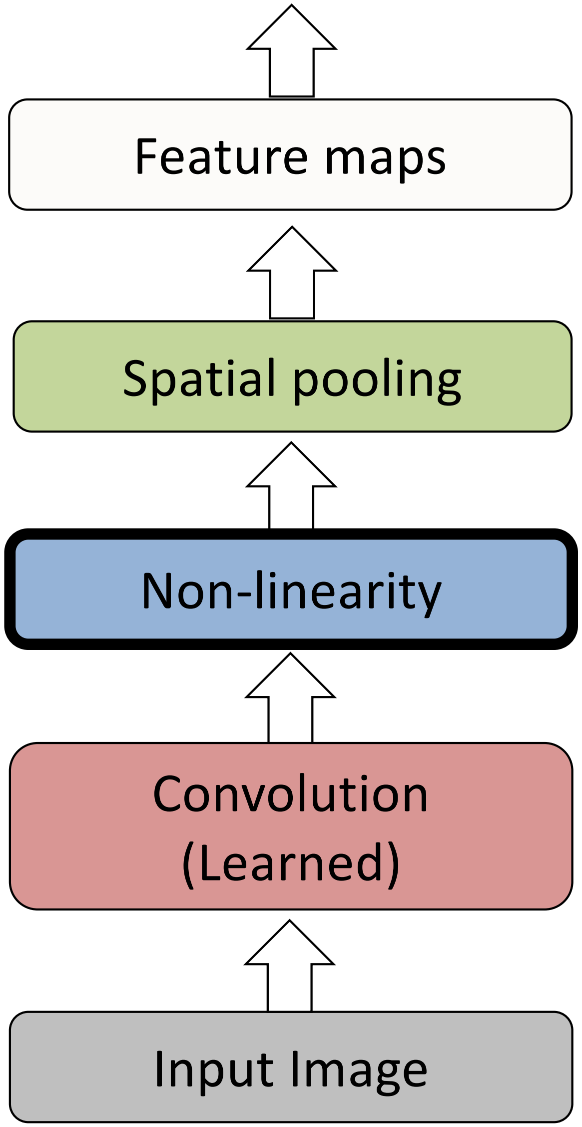


Input

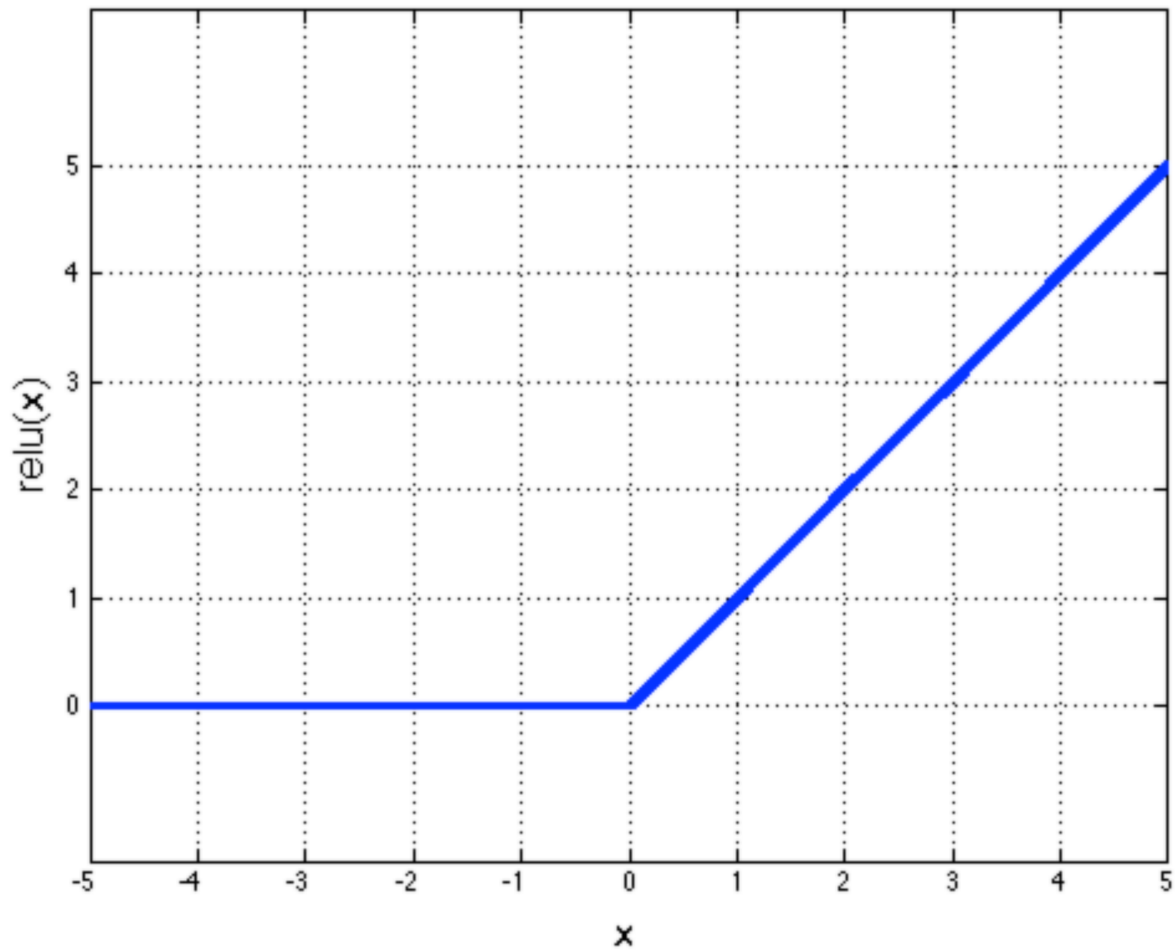
Feature Map

Slide credit: Svetlana Lazebnik, R. Fergus, Y. LeCun

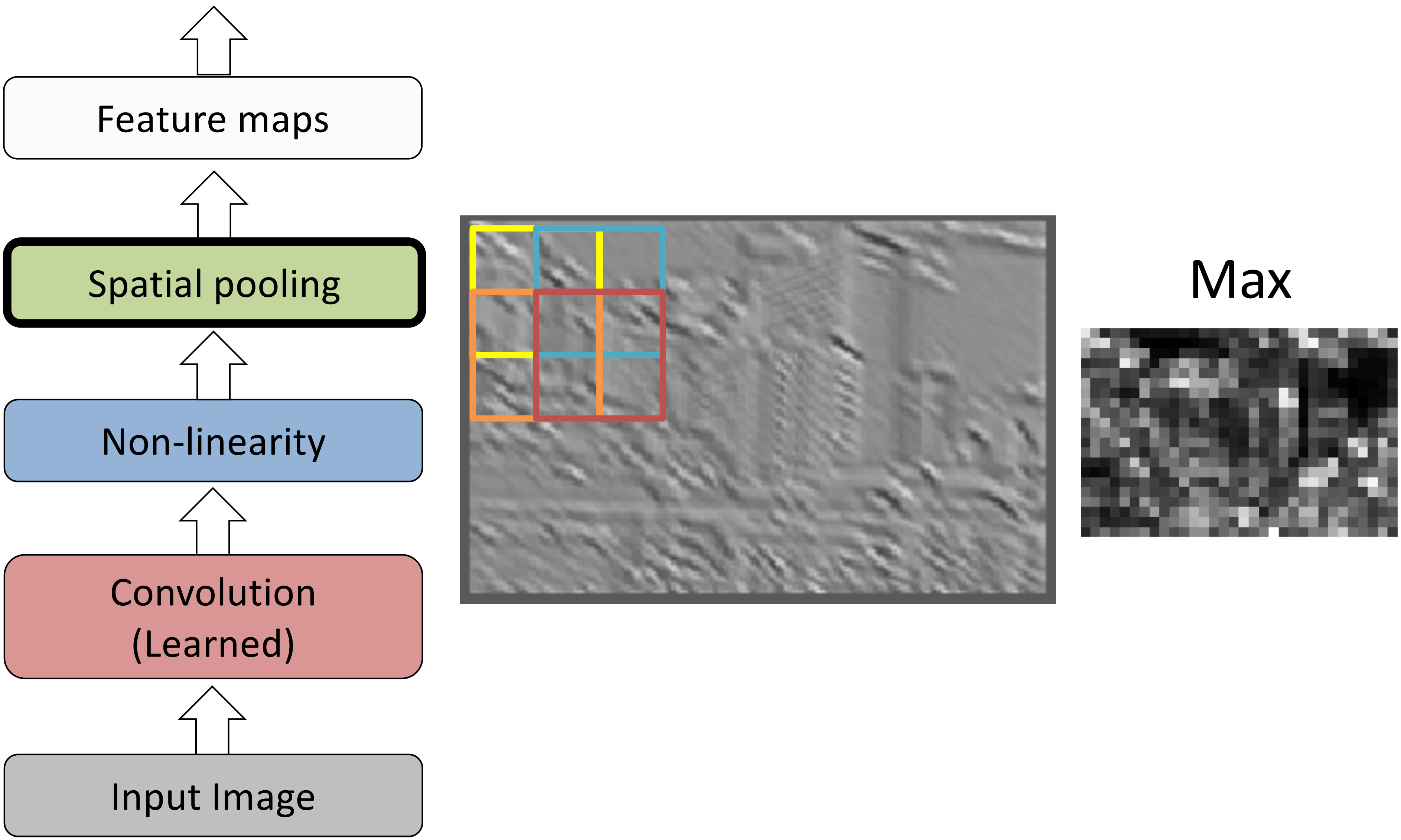
Key operations



Example: Rectified Linear Unit (ReLU)



Key operations



Slide credit: Svetlana Lazebnik, R. Fergus, Y. LeCun

Design principles

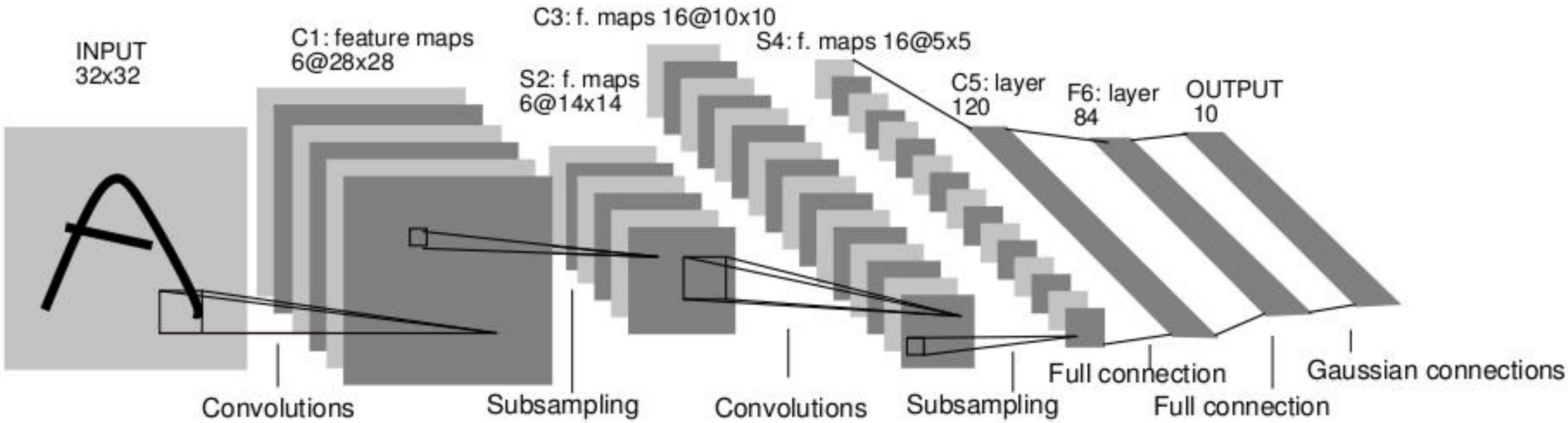
Reduce filter sizes (except possibly at the lowest layer), factorize filters aggressively

Use 1x1 convolutions to reduce and expand the number of feature maps judiciously

Use skip connections and/or create multiple paths through the network

Slide credit: Svetlana Lazebnik

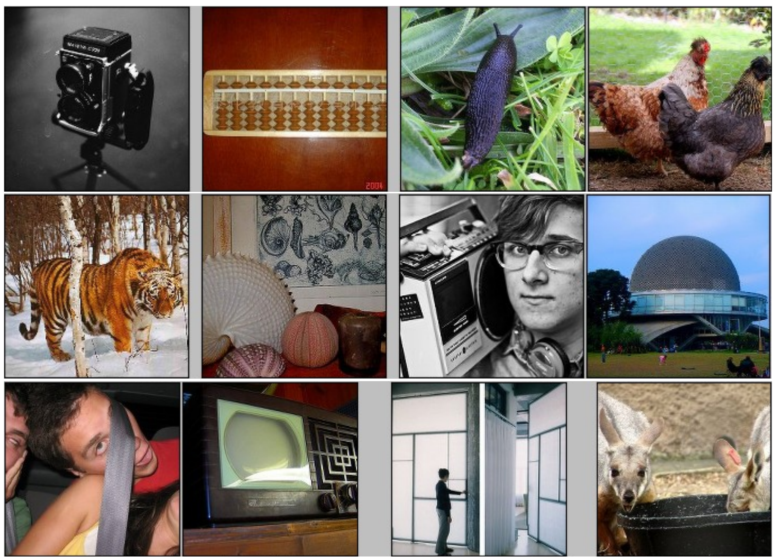
LeNet-5



Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, Gradient-based learning applied to document recognition, Proc. IEEE 86(11): 2278–2324, 1998.

Slide credit: Svetlana Lazebnik

ImageNet



~14 million labeled images, 20k classes

Images gathered from Internet

Human labels via Amazon MTurk

ImageNet Large-Scale Visual Recognition Challenge (ILSVRC): 1.2 million training images, 1000 classes

www.image-net.org/challenges/LSVRC/

Slide credit: Svetlana Lazebnik

Slide credit: Svetlana Lazebnik

http://www.inference.vc/deep-learning-is-easy/

Outline

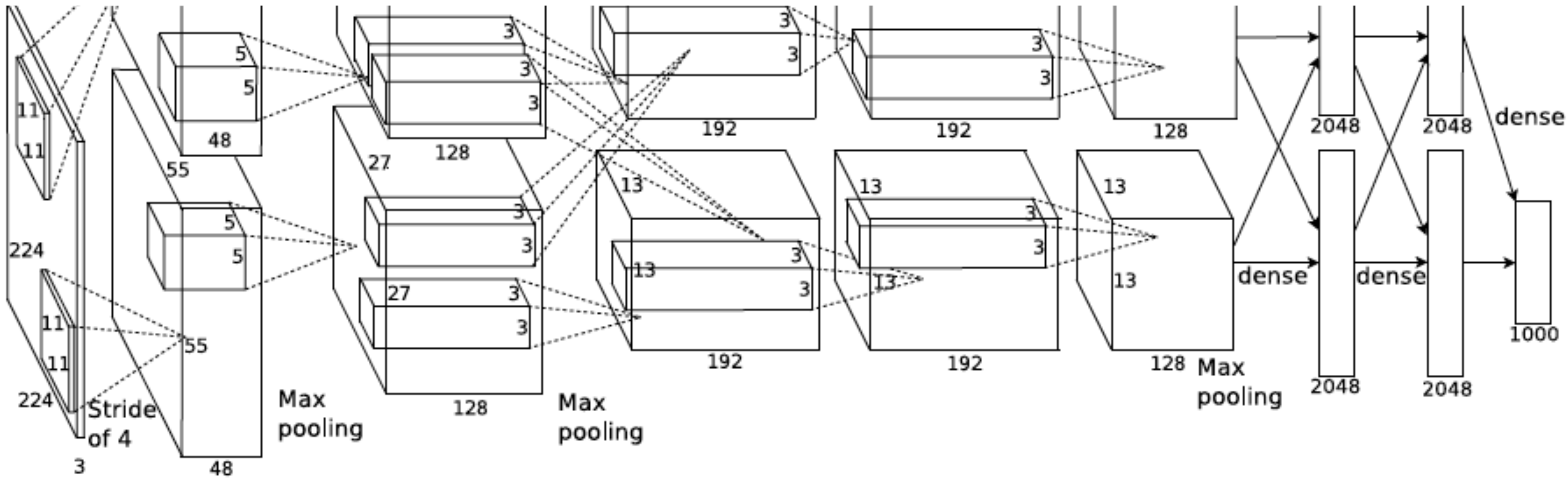
Convolutional Neural Networks

- 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
- Solving vanishing gradients problem

AlexNet: ILSVRC 2012 winner



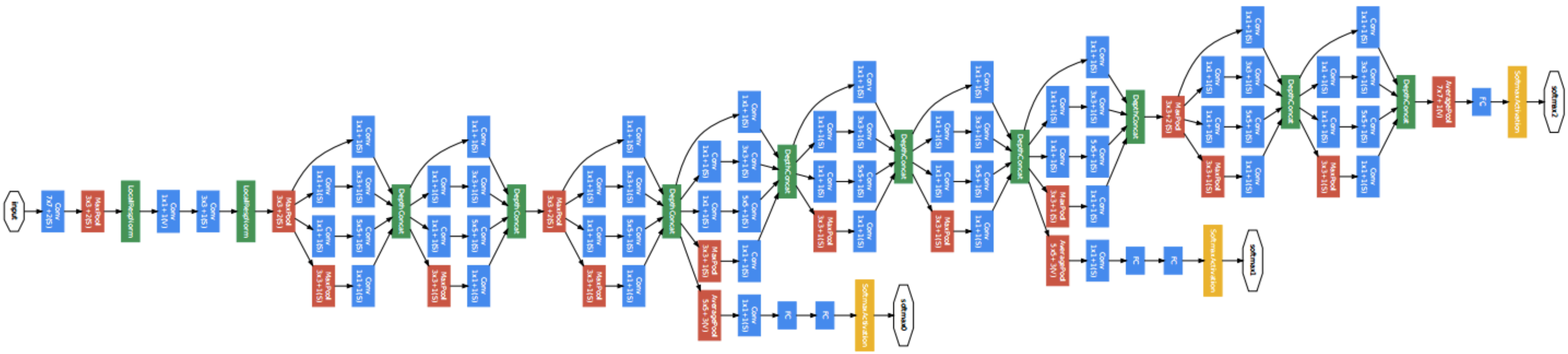
Similar framework to LeNet but:

- Max pooling, ReLU nonlinearity
- More data and bigger model (7 hidden layers, 650K units, 60M params)
- GPU implementation (50x speedup over CPU): Two GPUs for a week
- Dropout regularization

A. Krizhevsky, I. Sutskever, and G. Hinton, [ImageNet Classification with Deep Convolutional Neural Networks], NIPS 2012

Slide credit: Svetlana Lazebnik

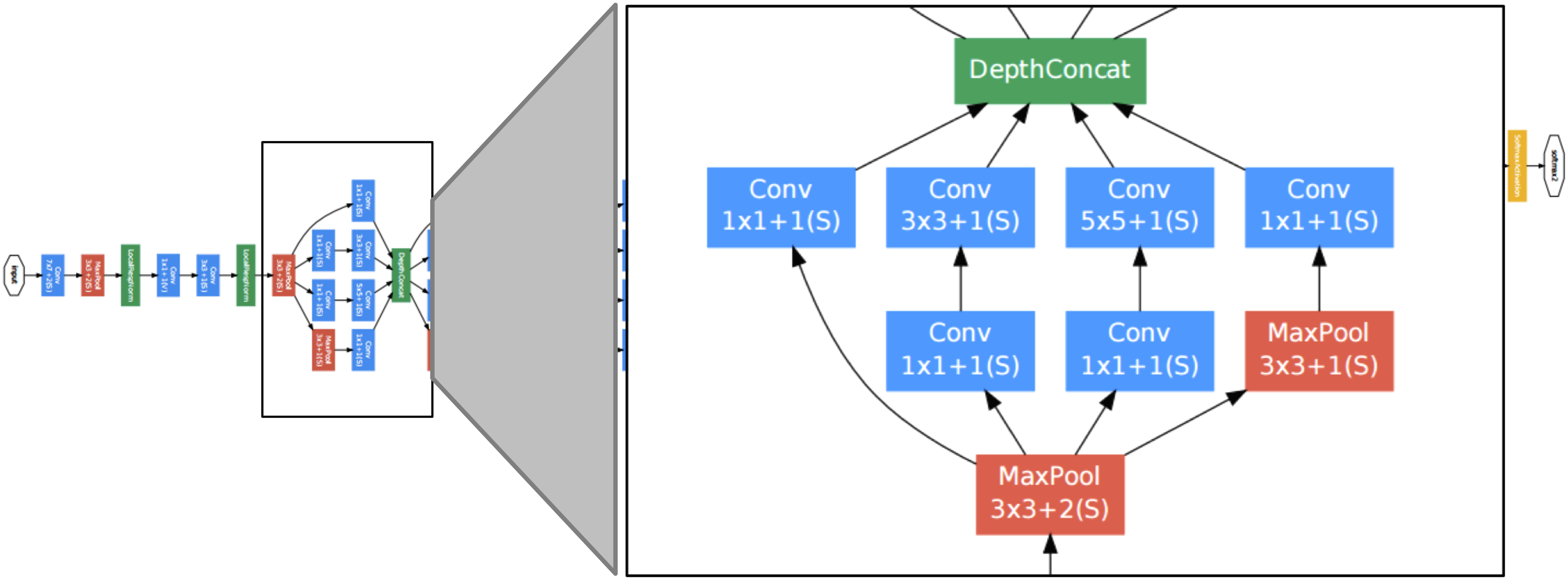
GoogLeNet



Szegedy et al., 2015

Slide credit: Svetlana Lazebnik

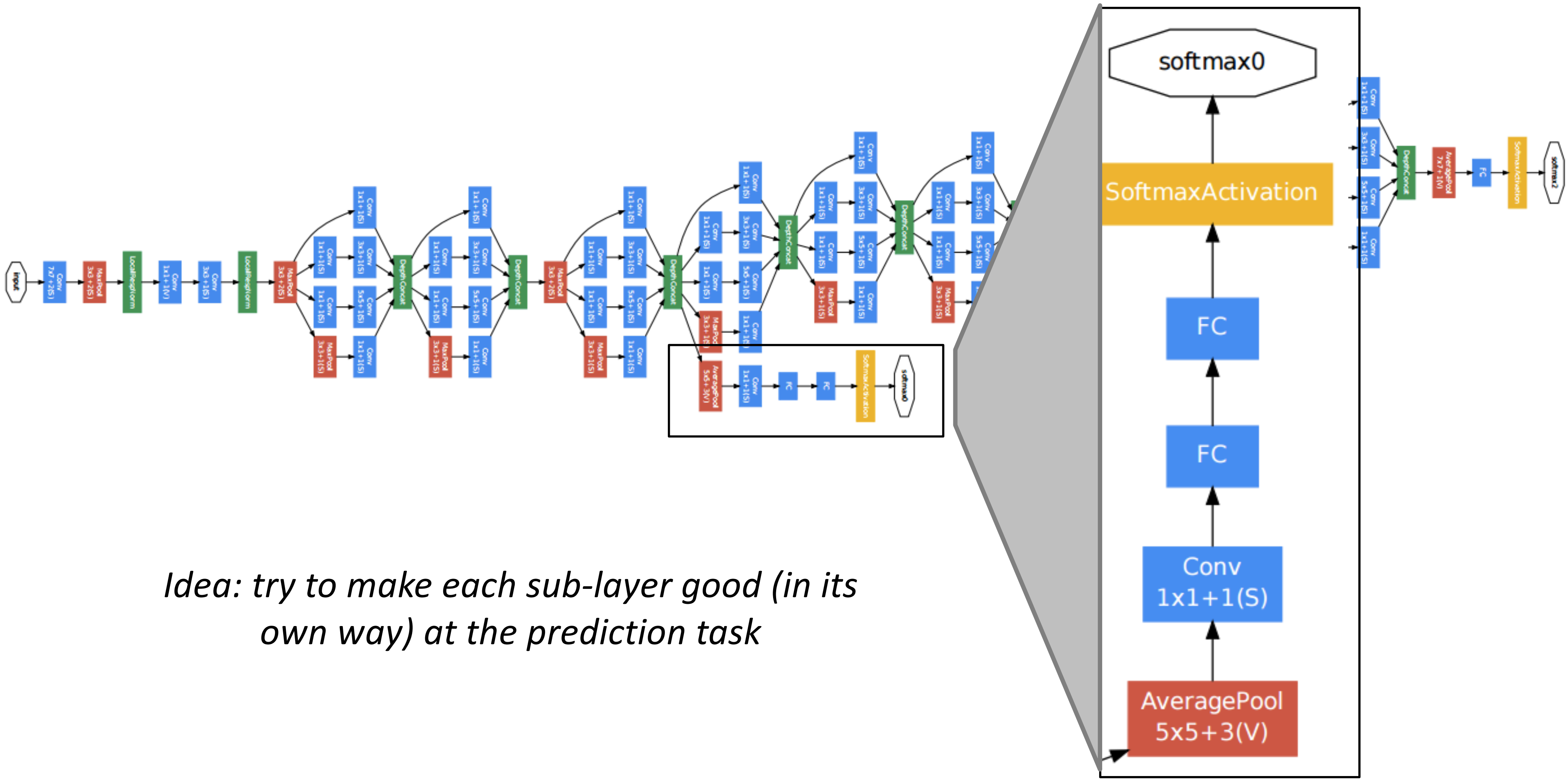
GoogLeNet



Slide credit: Svetlana Lazebnik

Szegedy et al., 2015

GoogLeNet: Auxiliary Classifier at Sub-levels



Idea: try to make each sub-layer good (in its own way) at the prediction task

Slide credit: Svetlana Lazebnik

Szegedy et al., 2015

GoogLeNet

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

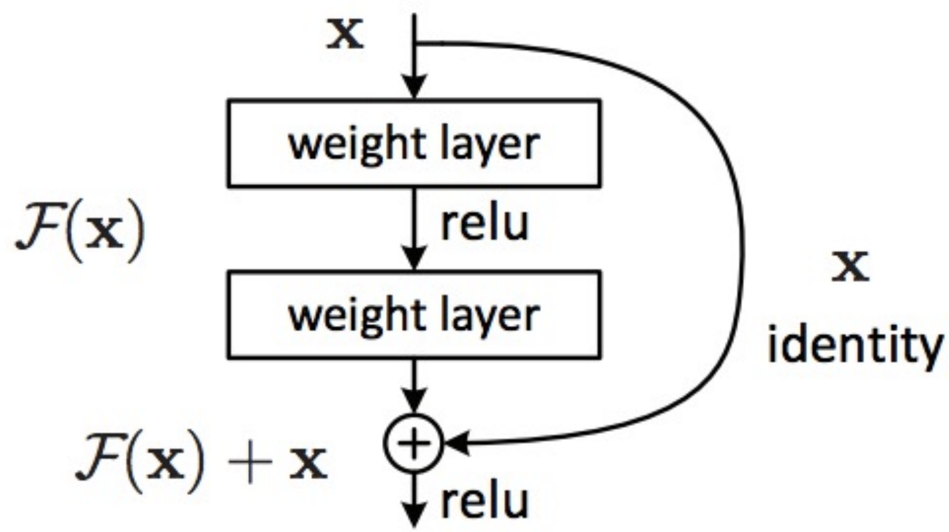
Slide credit: Svetlana Lazebnik

Szegedy et al., 2015

ResNet (Residual Network)

Make it easy for network layers to represent the identity mapping

Skipping 2+ layers is intentional & needed



He et al. “Deep Residual Learning for Image Recognition” (2016)

Slide credit: Svetlana Lazebnik

Summary: ILSVRC 2012-2015

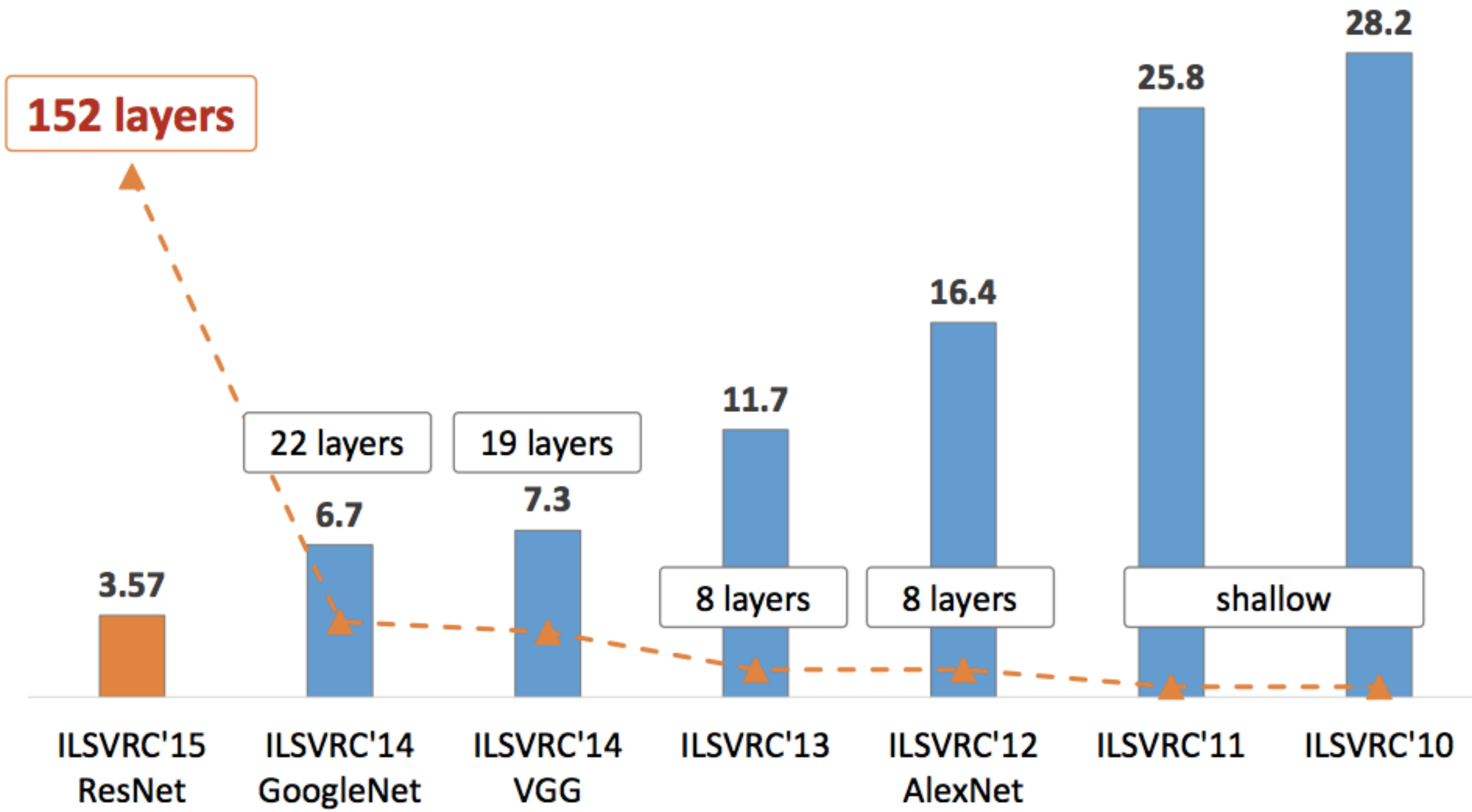
Team	Year	Place	Error (top-5)	External data
SuperVision	2012	-	16.4%	no
SuperVision	2012	1st	15.3%	ImageNet 22k
Clarifai (7 layers)	2013	-	11.7%	no
Clarifai	2013	1st	11.2%	ImageNet 22k
VGG (16 layers)	2014	2nd	7.32%	no
GoogLeNet (19 layers)	2014	1st	6.67%	no
ResNet (152 layers)	2015	1st	3.57%	
Human expert*			5.1%	

http://karpathy.github.io/2014/09/02/what-i-learned-from-competing-against-a-convnet-on-imagenet/

Slide credit: Svetlana Lazebnik

Rapid Progress due to CNNs

Classification: ImageNet Challenge top-5 error



Slide credit: Svetlana Lazebnik

Figure source: Kaiming He

Outline

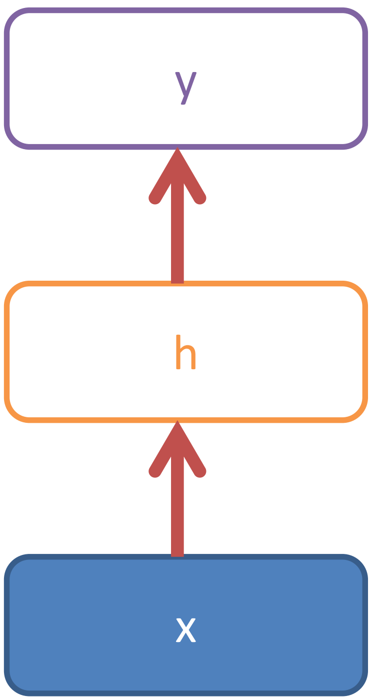
Convolutional Neural Networks

- 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

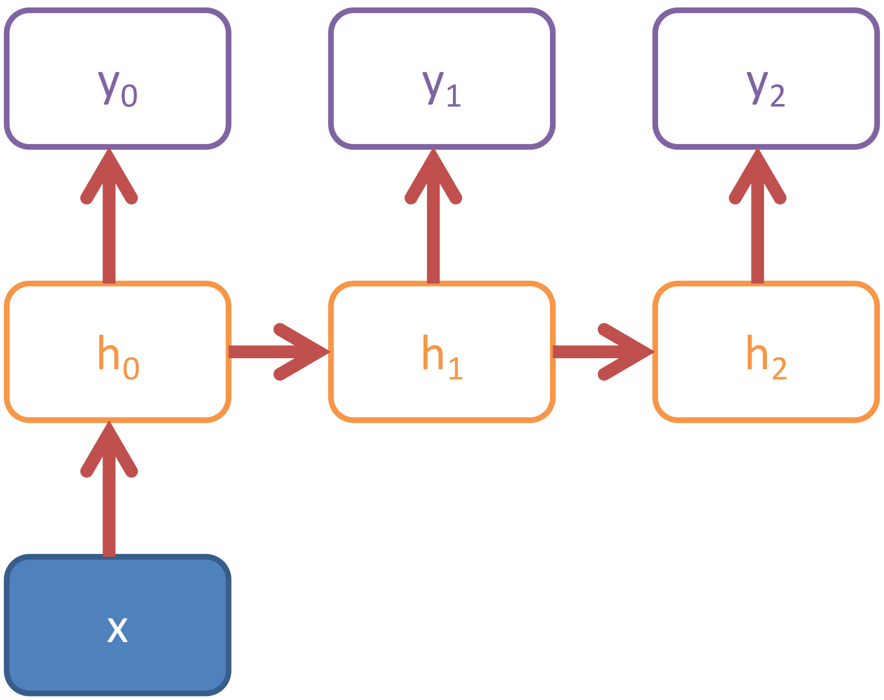
Network Types



Feed forward

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

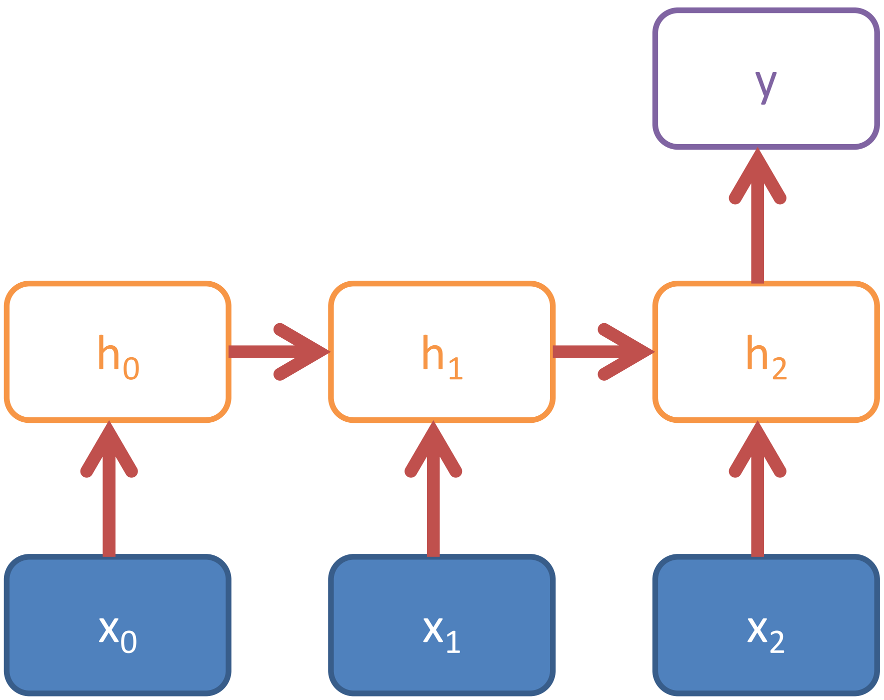
Network Types



Recursive: One input, Sequence output

Automated caption generation

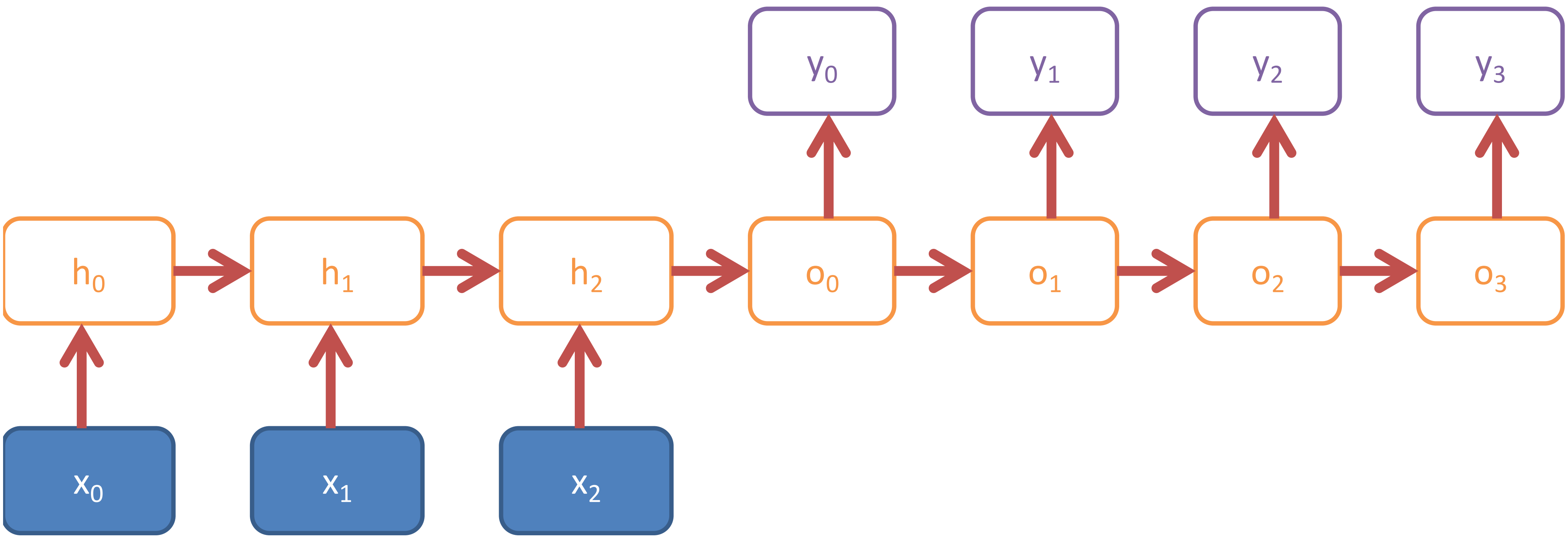
Network Types



Recursive: Sequence input, one output

Document classification
Action recognition in video (high-level)

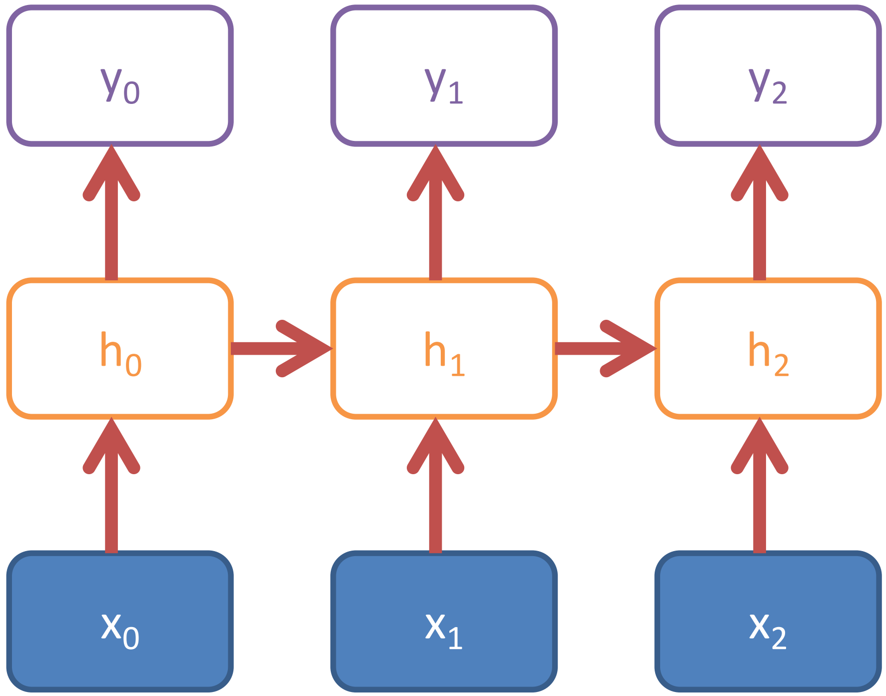
Network Types



Recursive: Sequence input, Sequence output (time delay)

Machine translation
Sequential description
Summarization

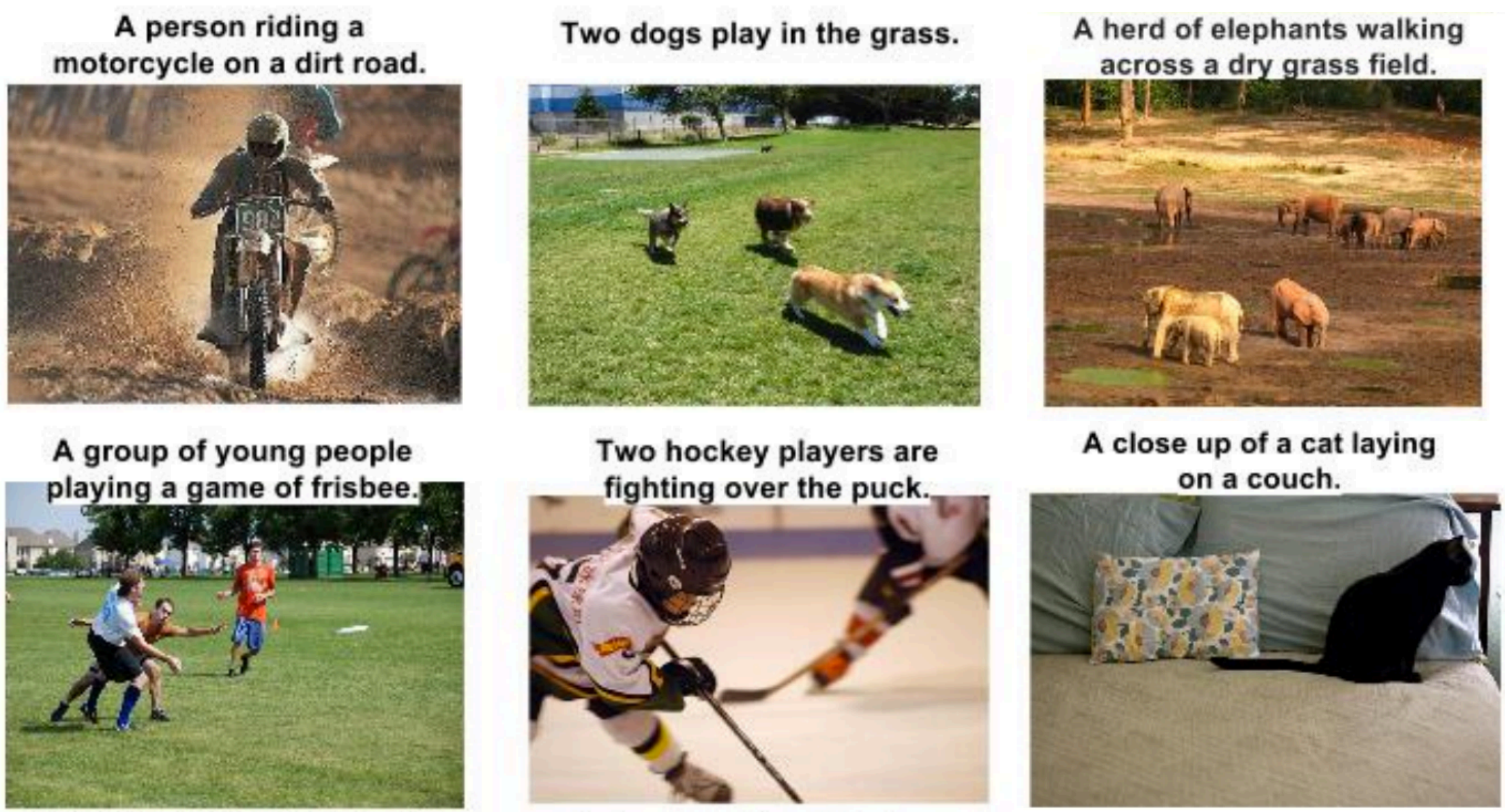
Network Types



Recursive: Sequence input, Sequence output

Part of speech tagging
Action recognition (fine grained)

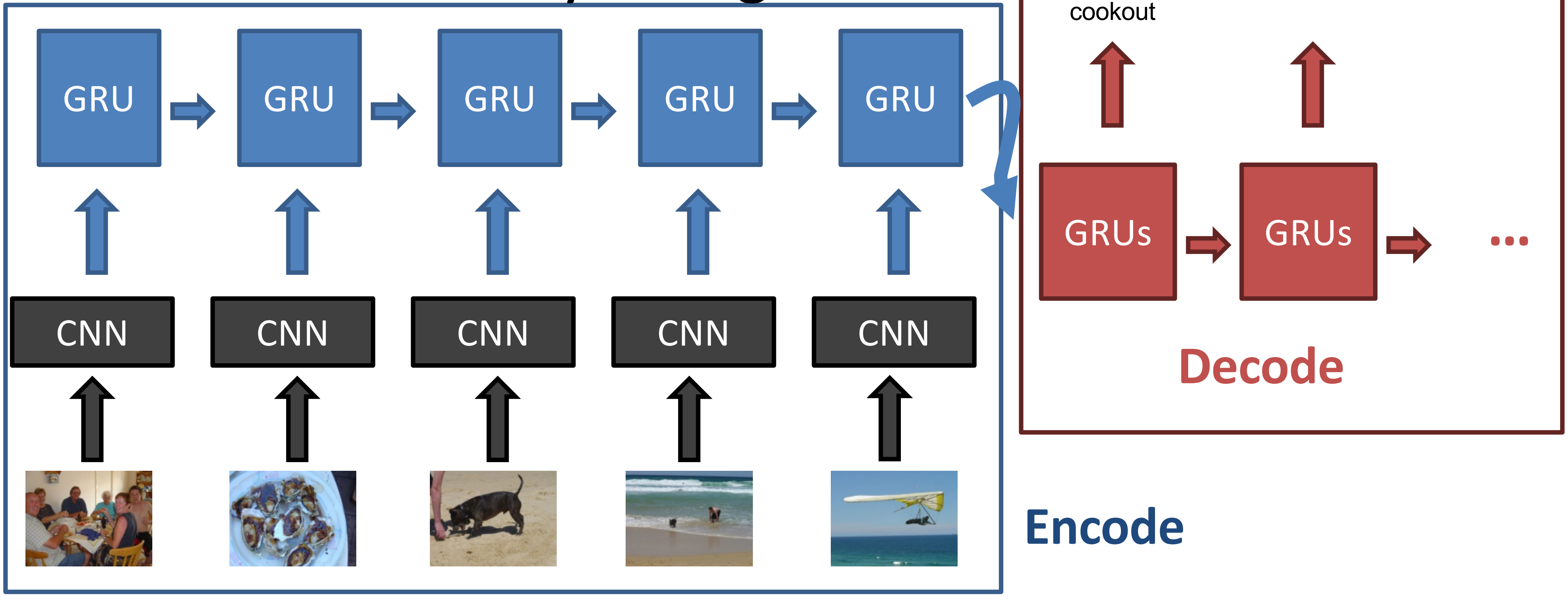
RNN Outputs: Image Captions



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

Slide credit: Arun Mallya

RNN Output: Visual Storytelling



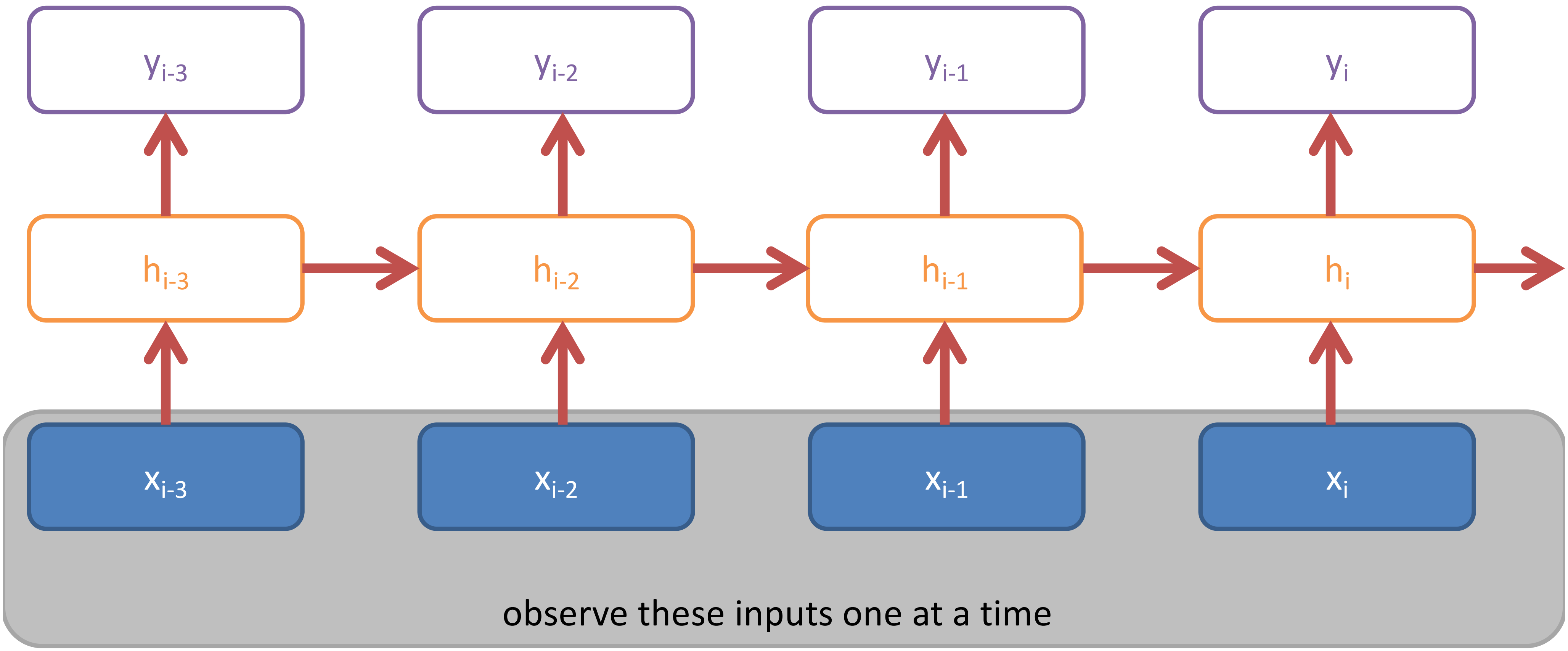
Human Reference

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!

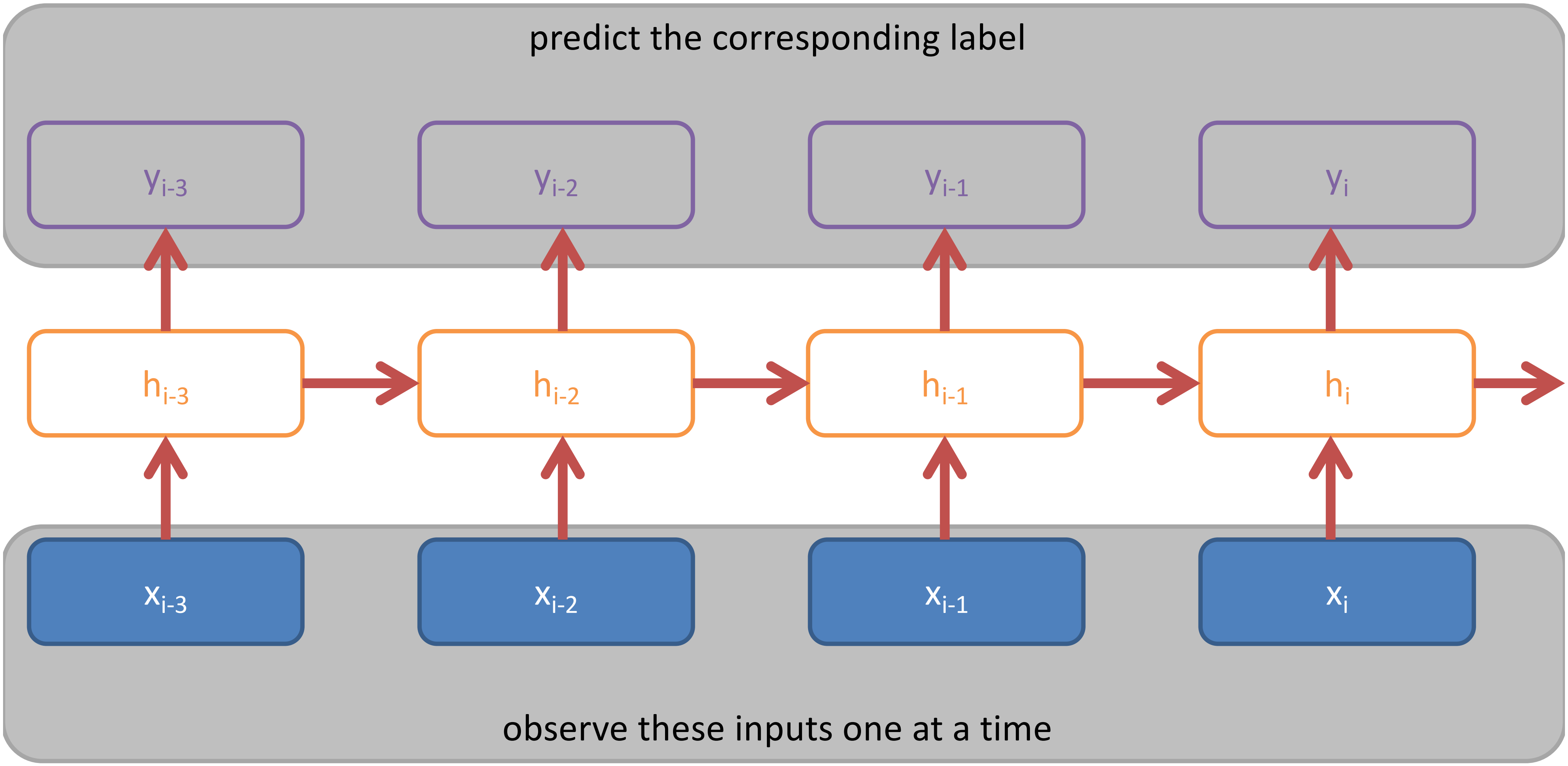
Huang et al. (2016)

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.

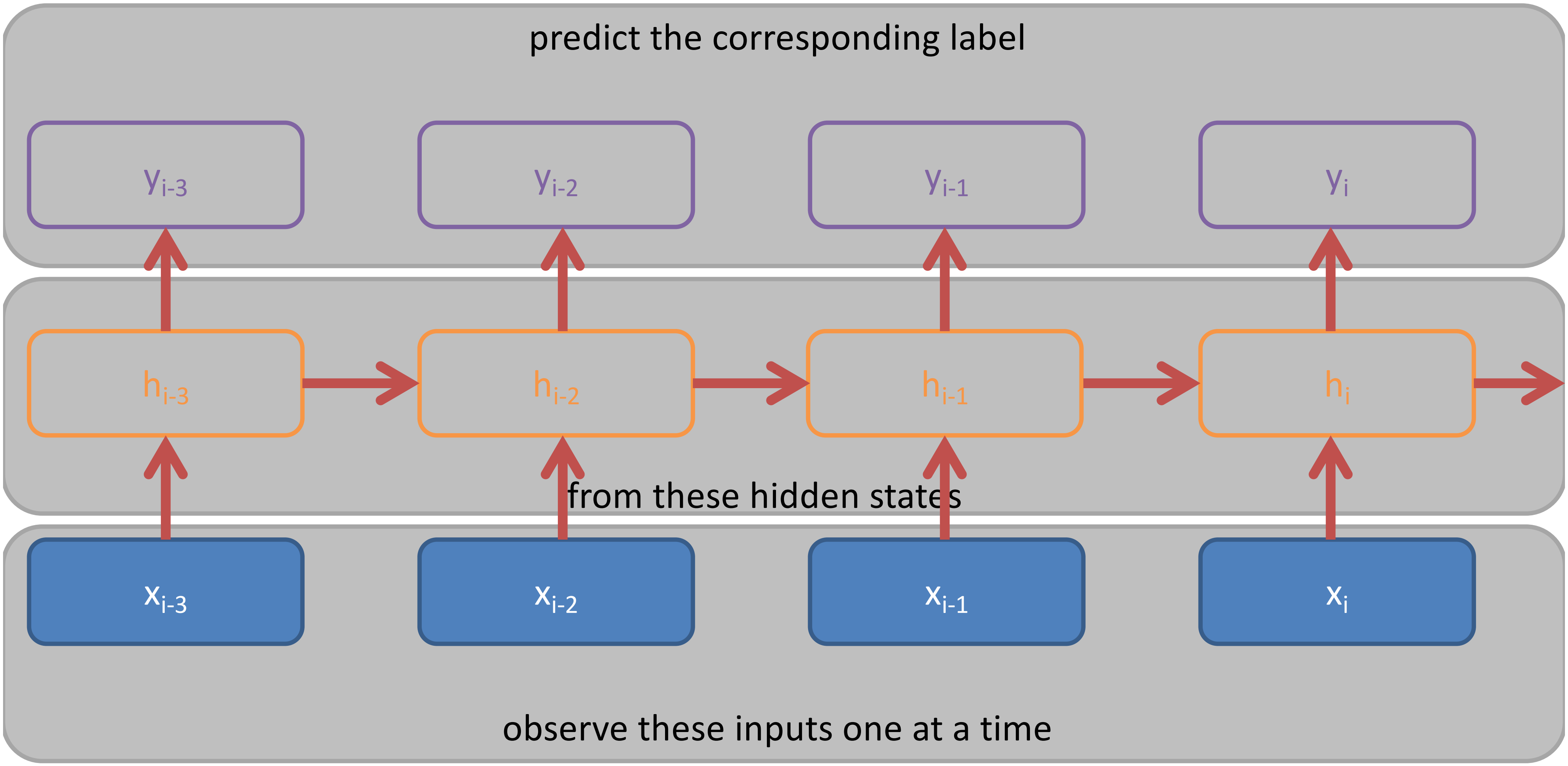
Recurrent Networks



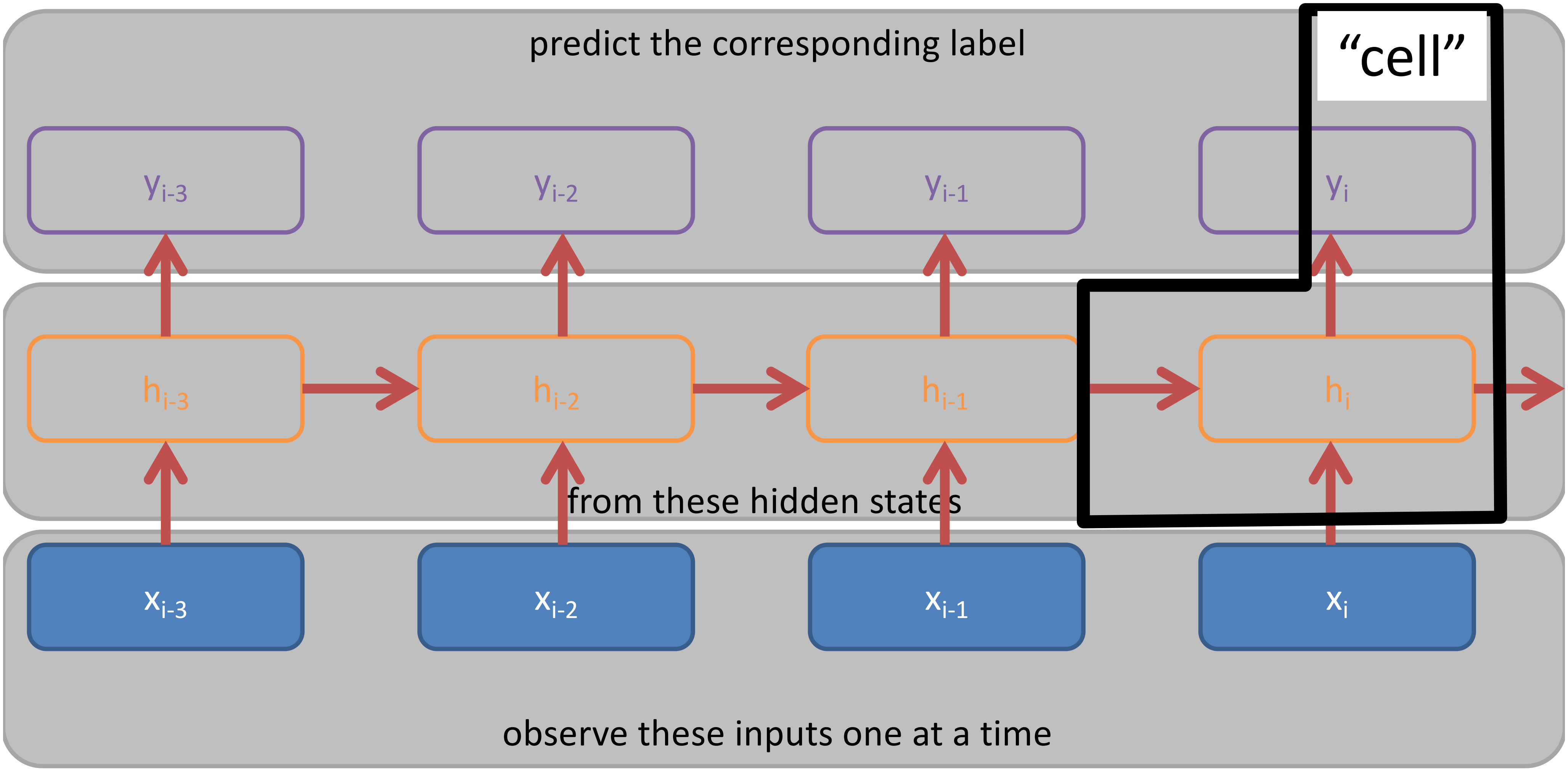
Recurrent Networks



Recurrent Networks



Recurrent Networks



Outline

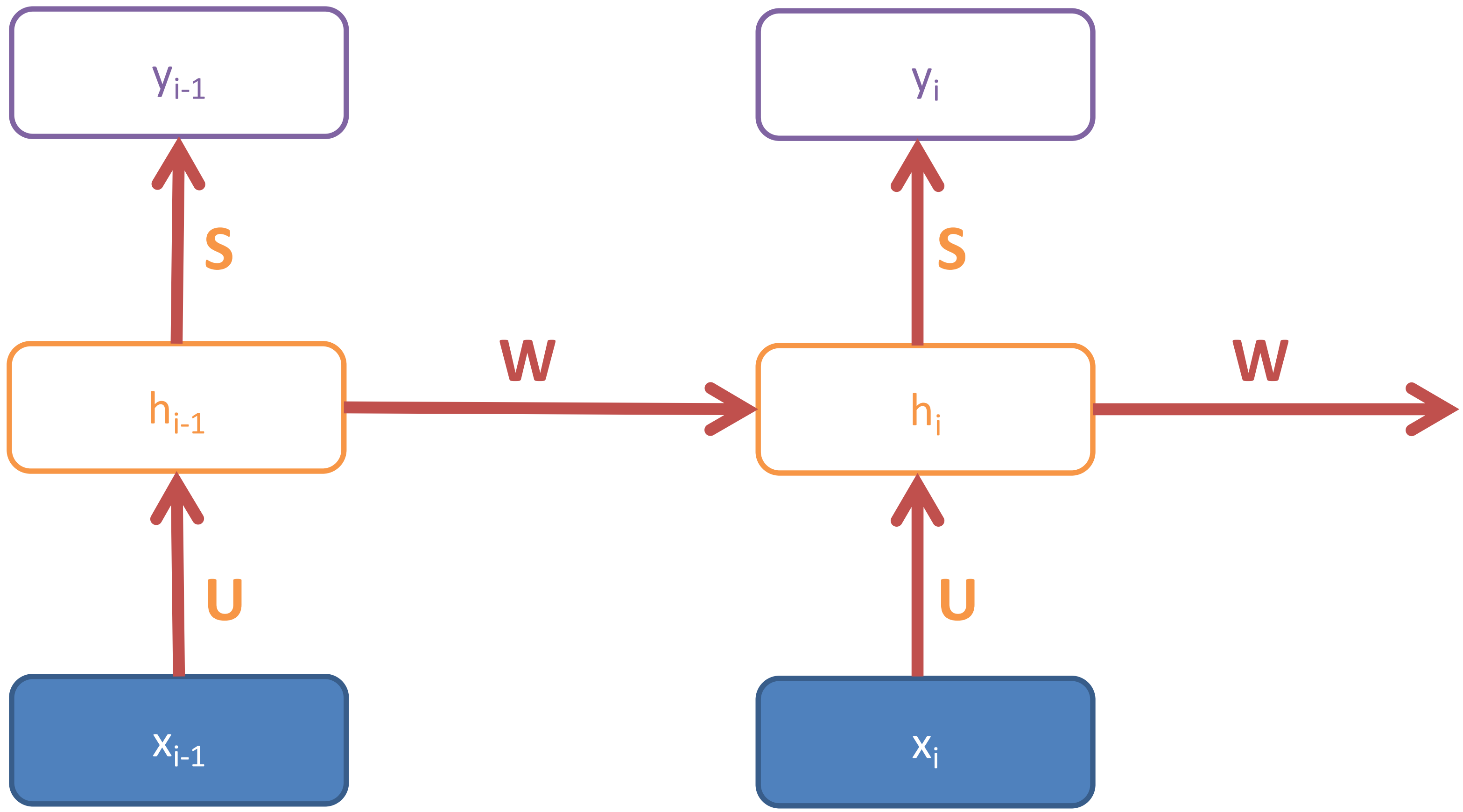
Convolutional Neural Networks

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

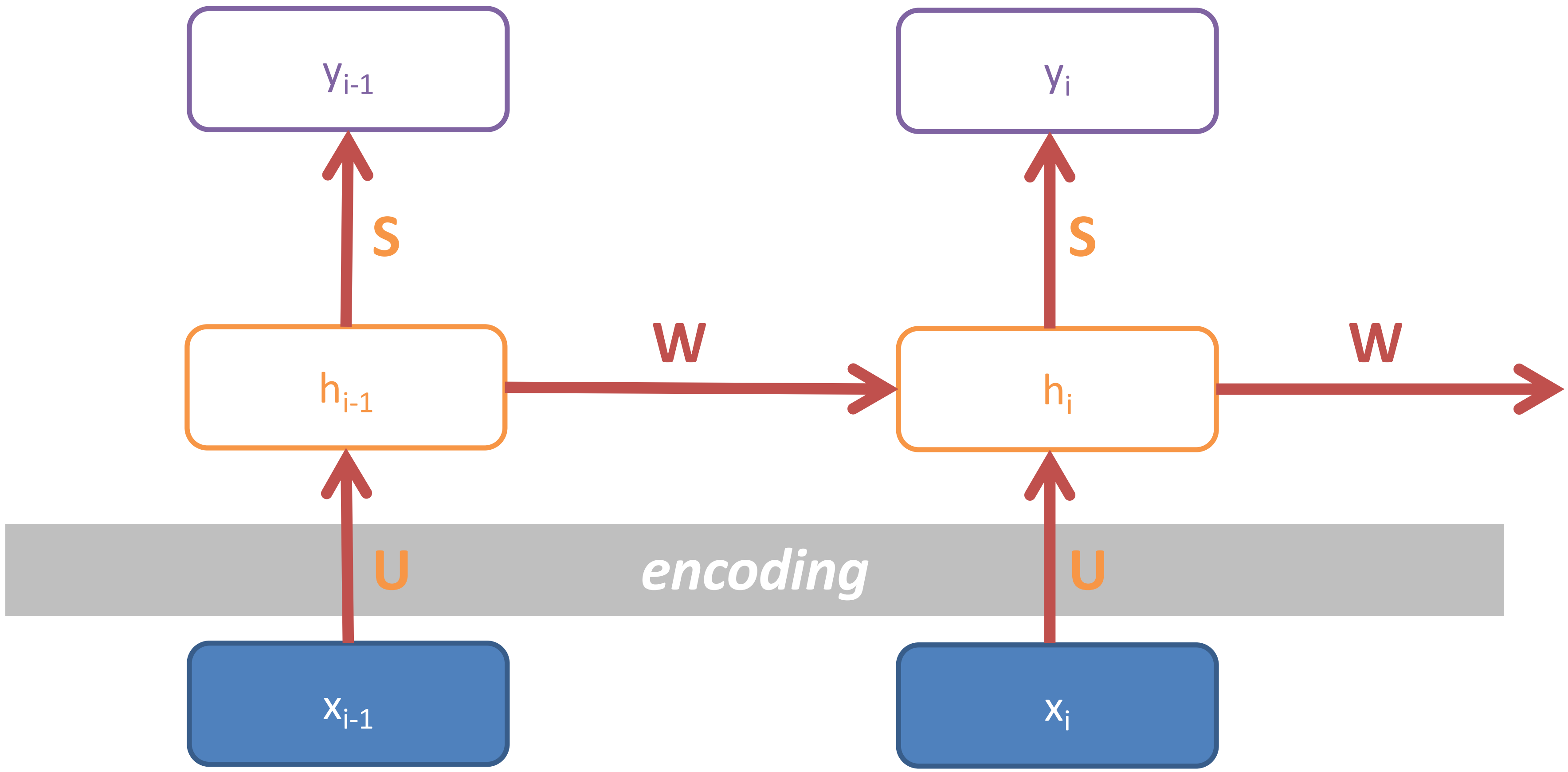
Recurrent Neural Networks

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

A *Simple* Recurrent Neural Network Cell

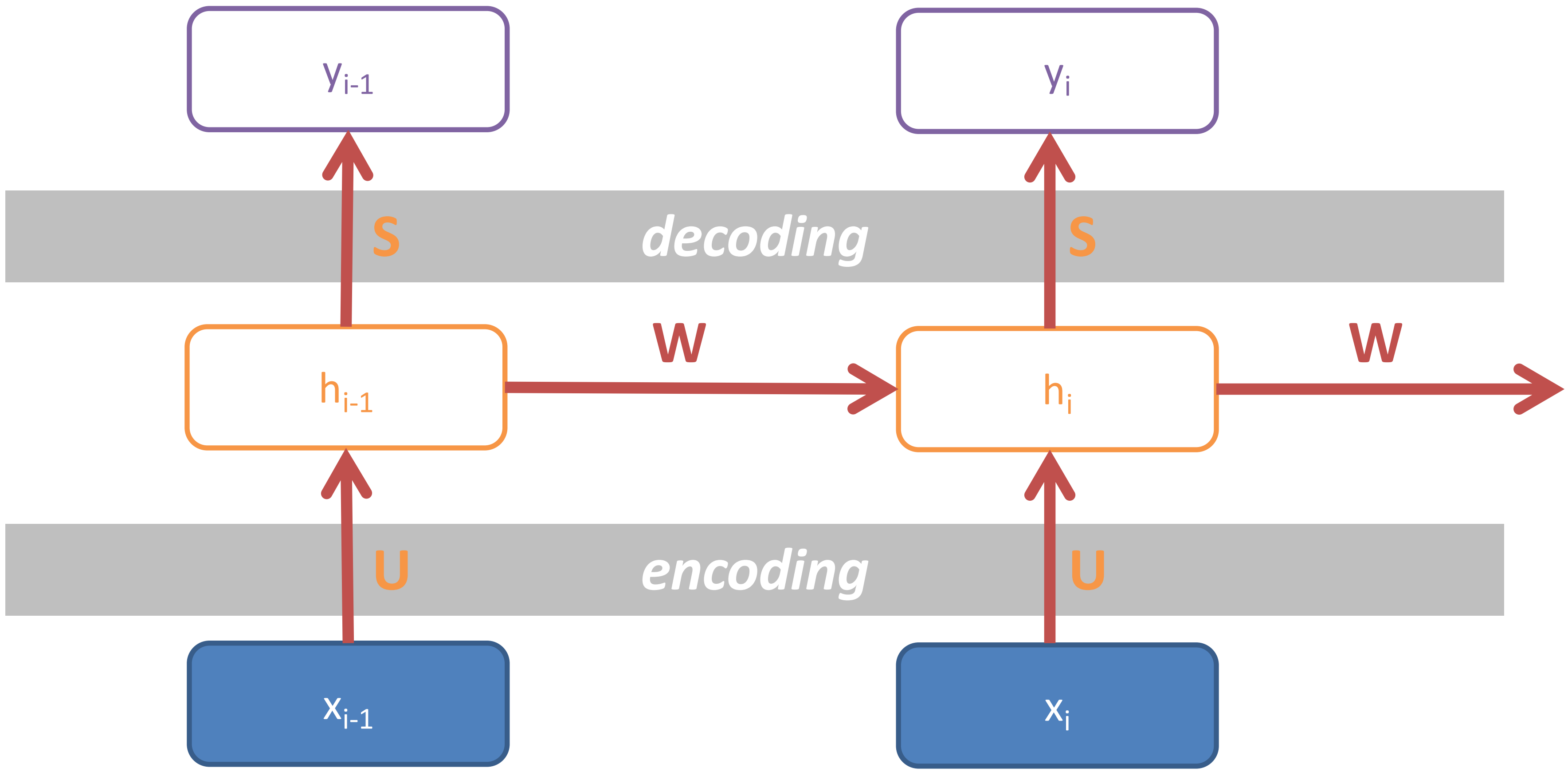


A *Simple* Recurrent Neural Network Cell



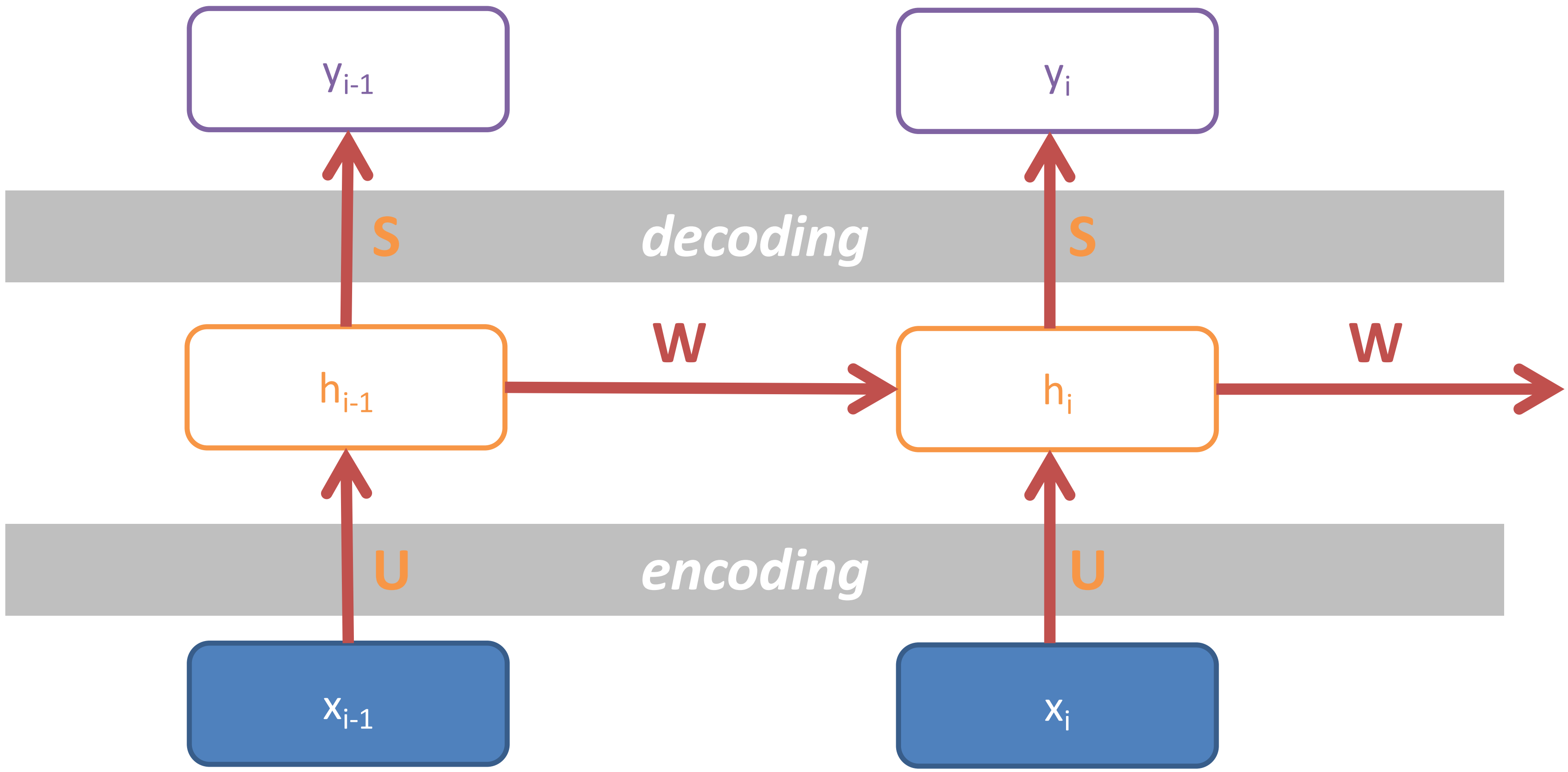
$$h_i = \tanh(W h_{i-1} + U x_i)$$

A *Simple* Recurrent Neural Network Cell



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

A *Simple* Recurrent Neural Network Cell



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

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

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

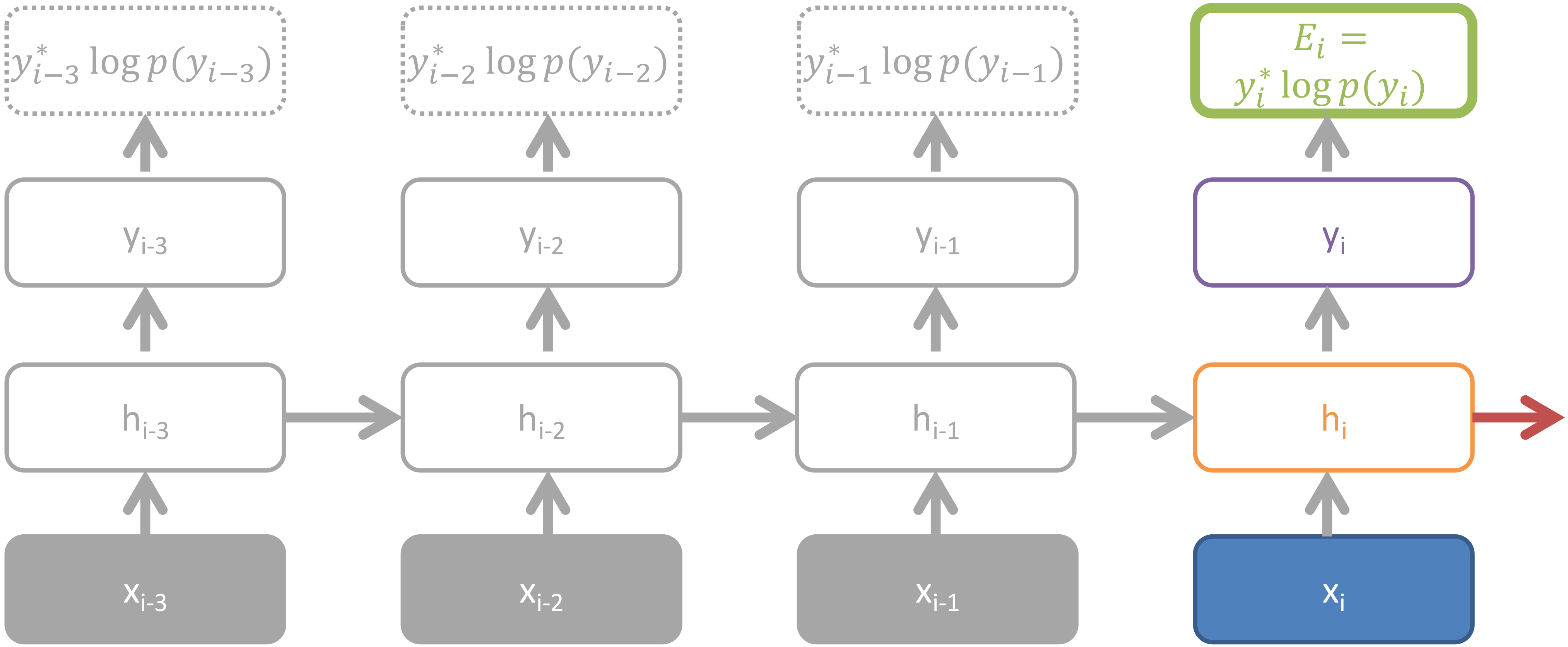
- Types of recurrence
- A basic recurrent cell
- BPTT: Backpropagation through time

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



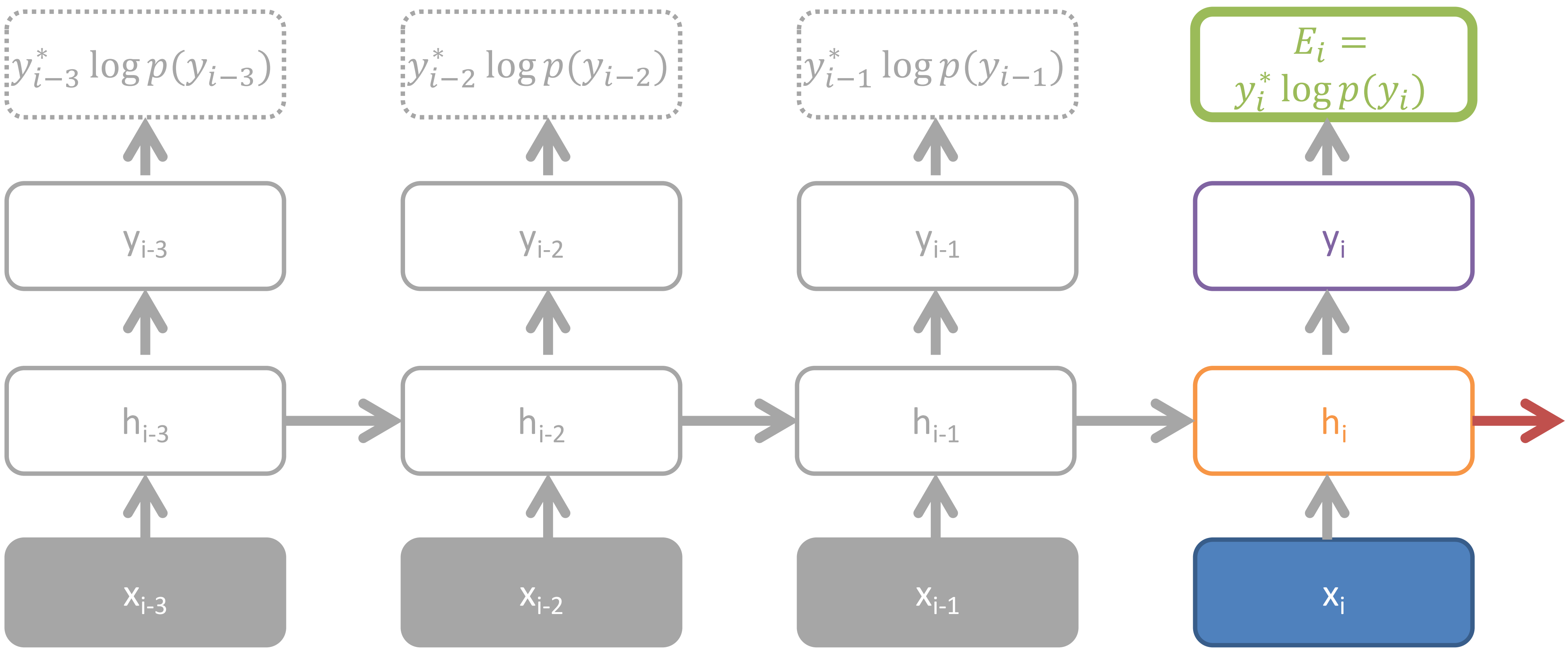
$$y_i = \text{softmax}(Sh_i)$$

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

per-step loss: cross entropy

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

BPTT



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

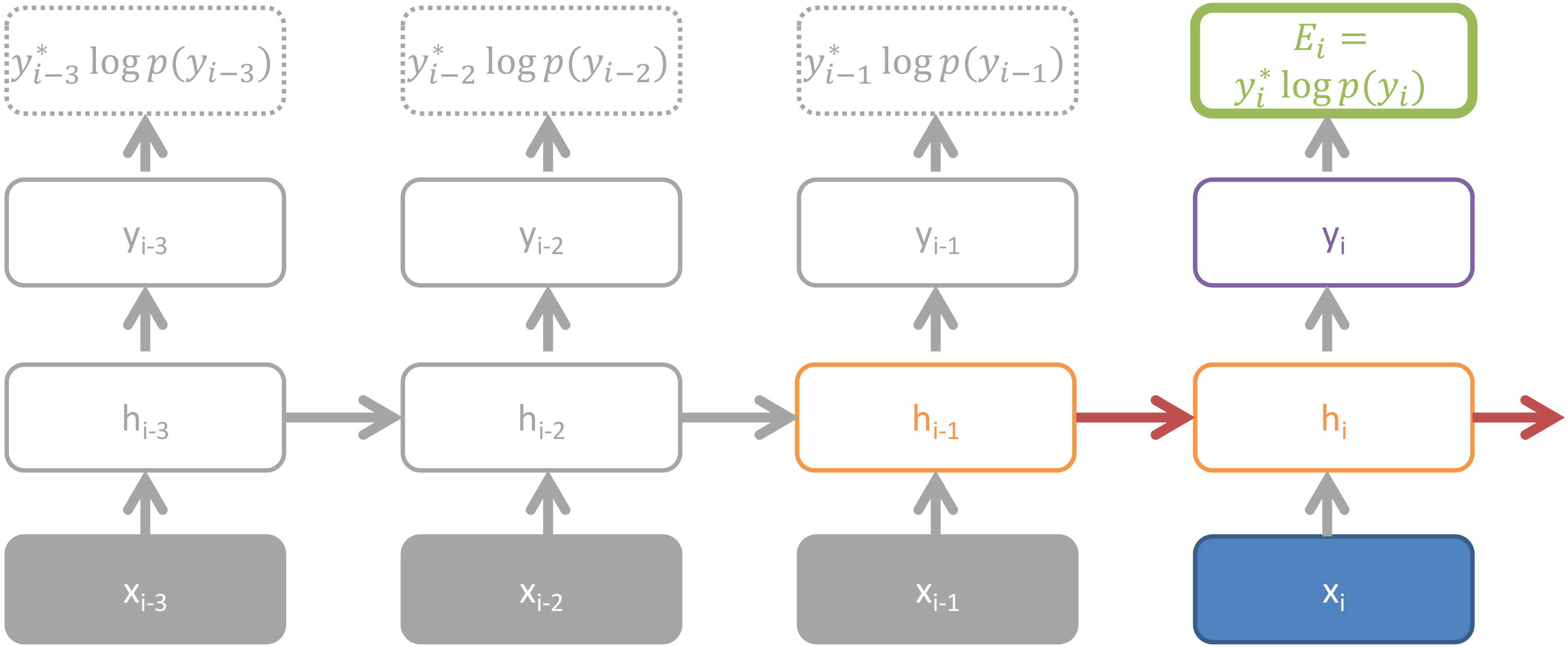
$$h_i = \tanh(Wh_{i-1} + Ux_i)$$

per-step loss: cross entropy

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

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

BPTT



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

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

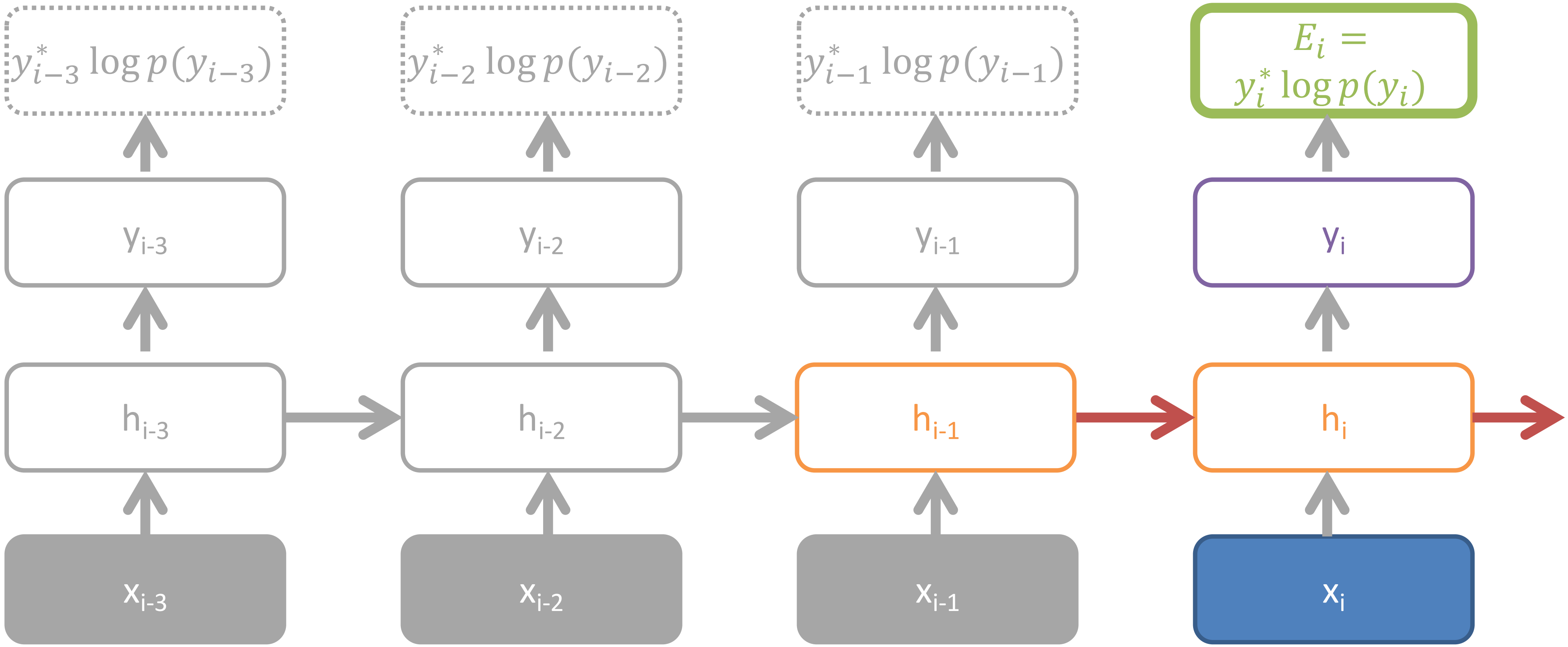
per-step loss: cross entropy

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

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

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

BPTT



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

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

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

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

per-step loss: cross entropy

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

BPTT

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

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

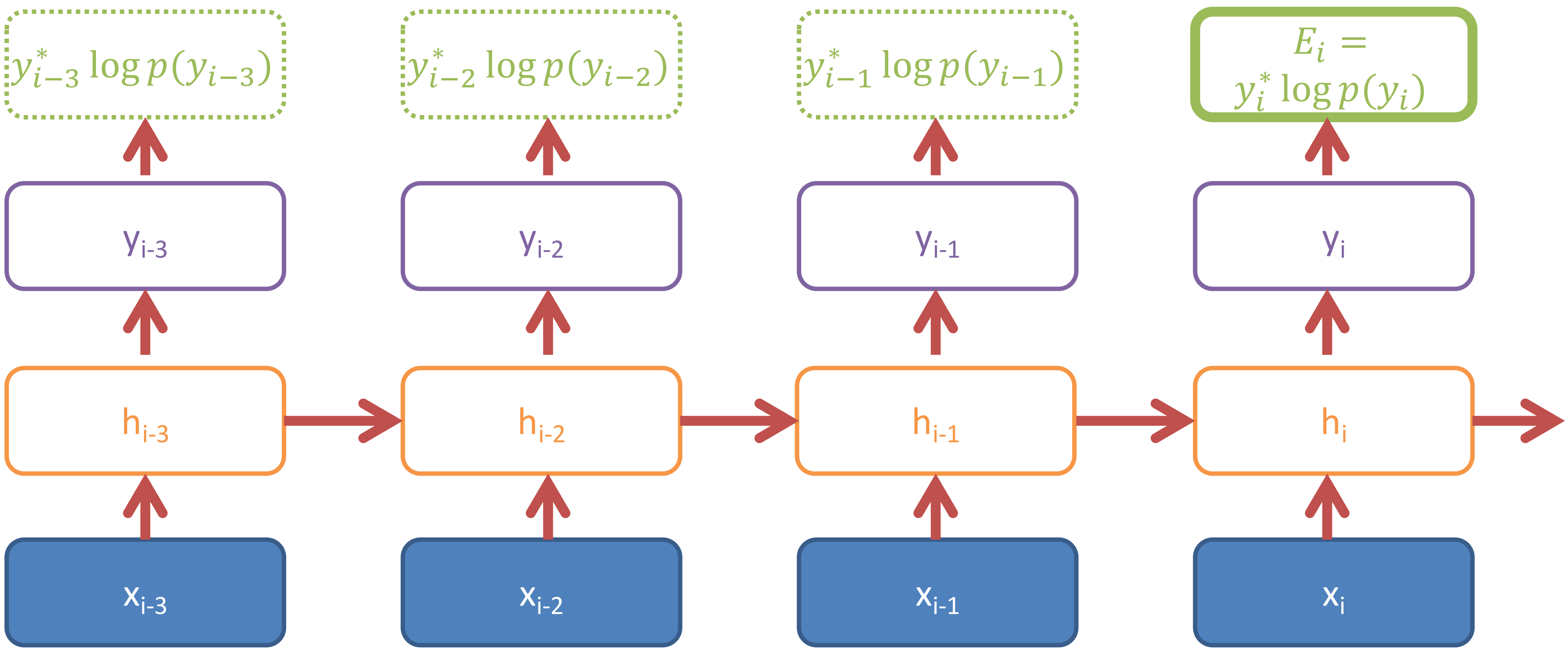
$$= \sum_j \frac{\partial E_i}{\partial y_i}\frac{\partial y_i}{\partial h_i}\frac{\partial h_i}{\partial h_l}\frac{\partial h_l}{\partial W^{(l)}}$$

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

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

per-loss, per-step backpropagation error

BPTT



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

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

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?

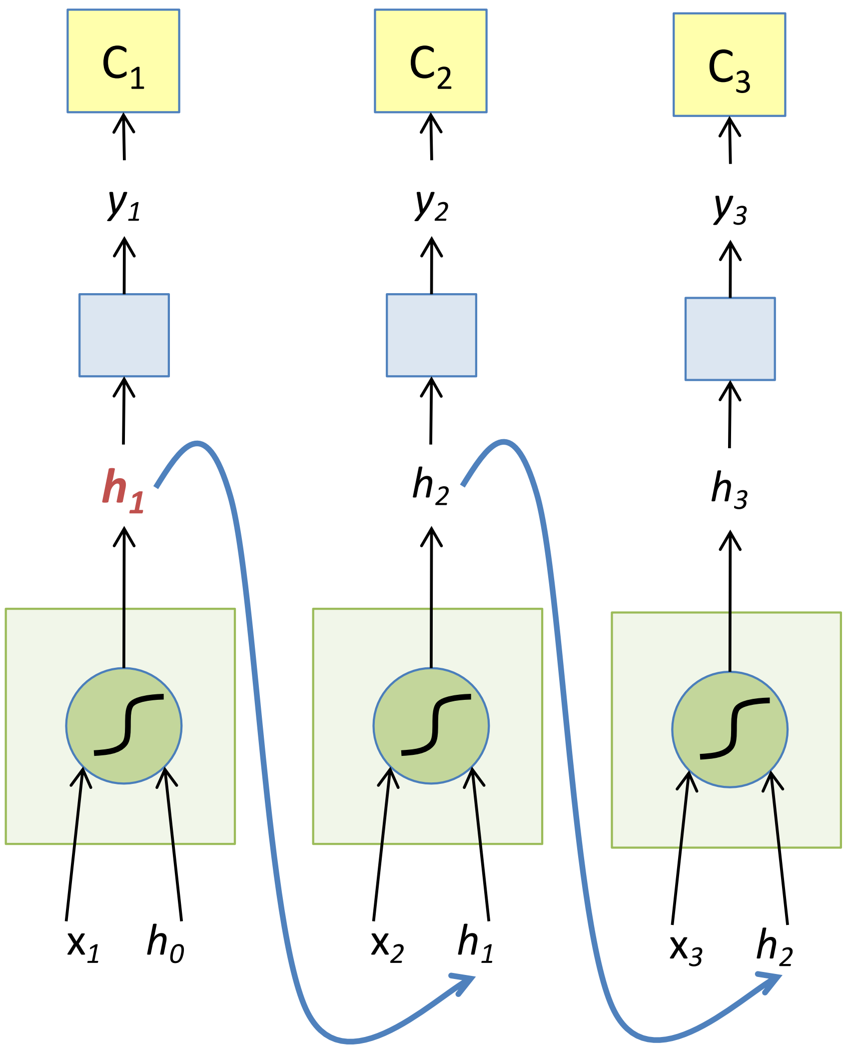
$$\frac{\partial C_t}{\partial h_1} = \left(\frac{\partial C_t}{\partial y_t}\right)\left(\frac{\partial y_t}{\partial h_1}\right)$$

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

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

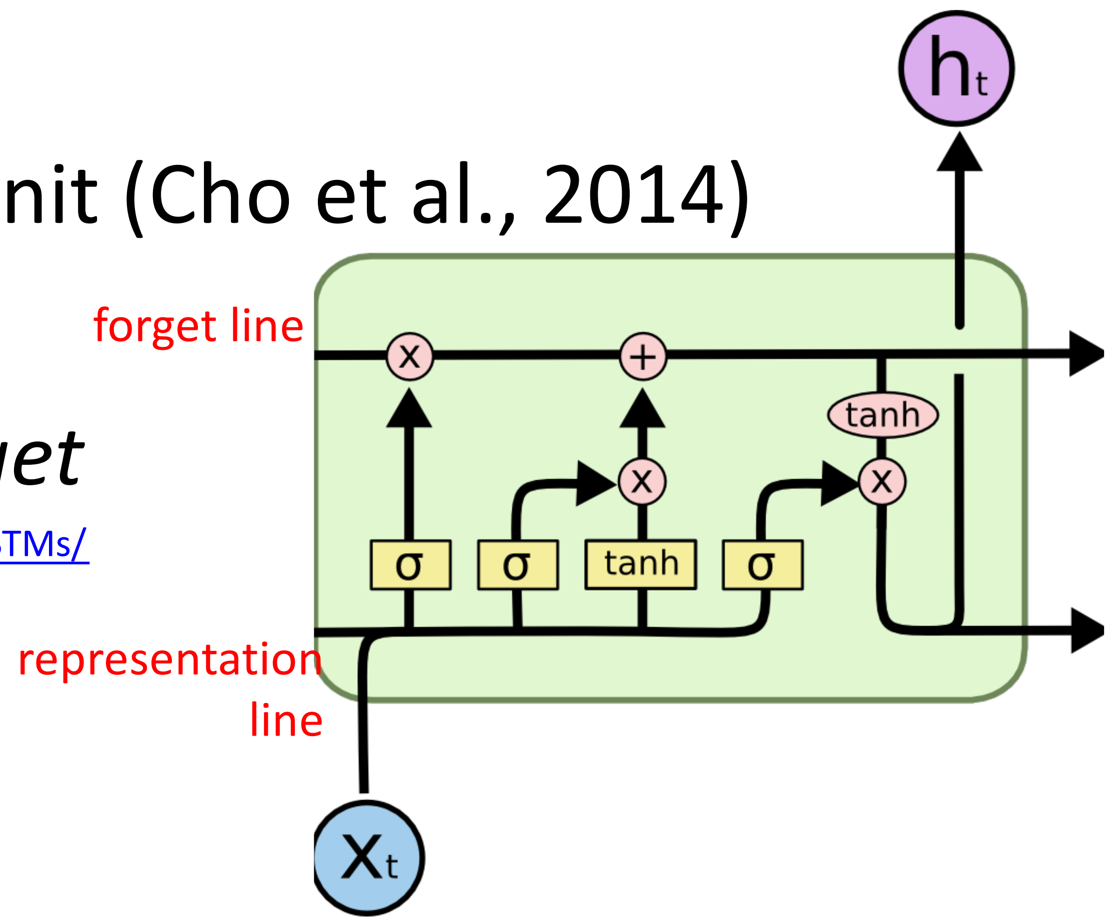
Vanishing Gradient Solution: Model Implementations

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

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

Basic Ideas: *learn to forget*

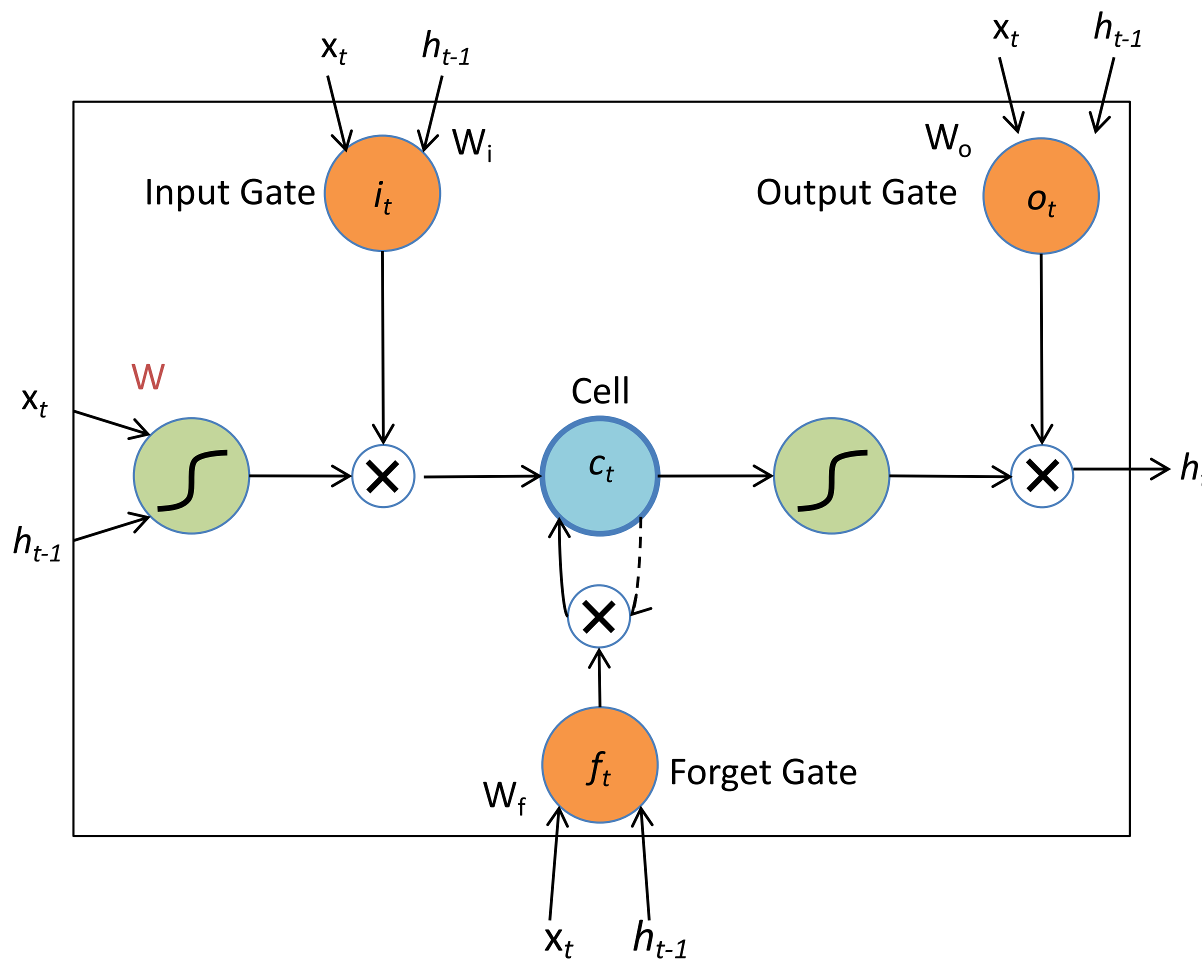
http://colah.github.io/posts/2015-08-Understanding-LSTMs/



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



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

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

```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_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