

Neural Networks

Machine Learning – CSE 446
Kevin Jamieson
University of Washington

May 29, 2019

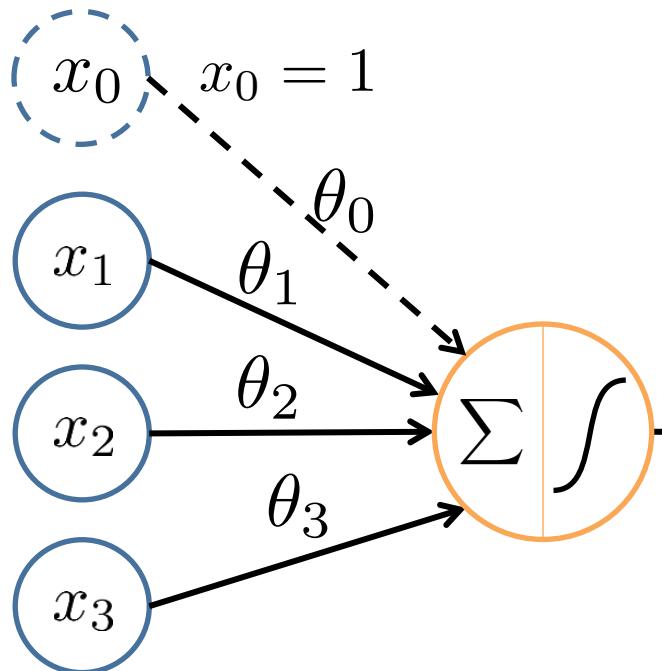
©Kevin Jamieson

Contains slides from...

- LeCun & Ranzato
- Russ Salakhutdinov
- Honglak Lee
- Andrew Ng
- Google images
- <https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/>
- <http://cs231n.github.io/convolutional-networks/>
-

Single Node

“bias unit”



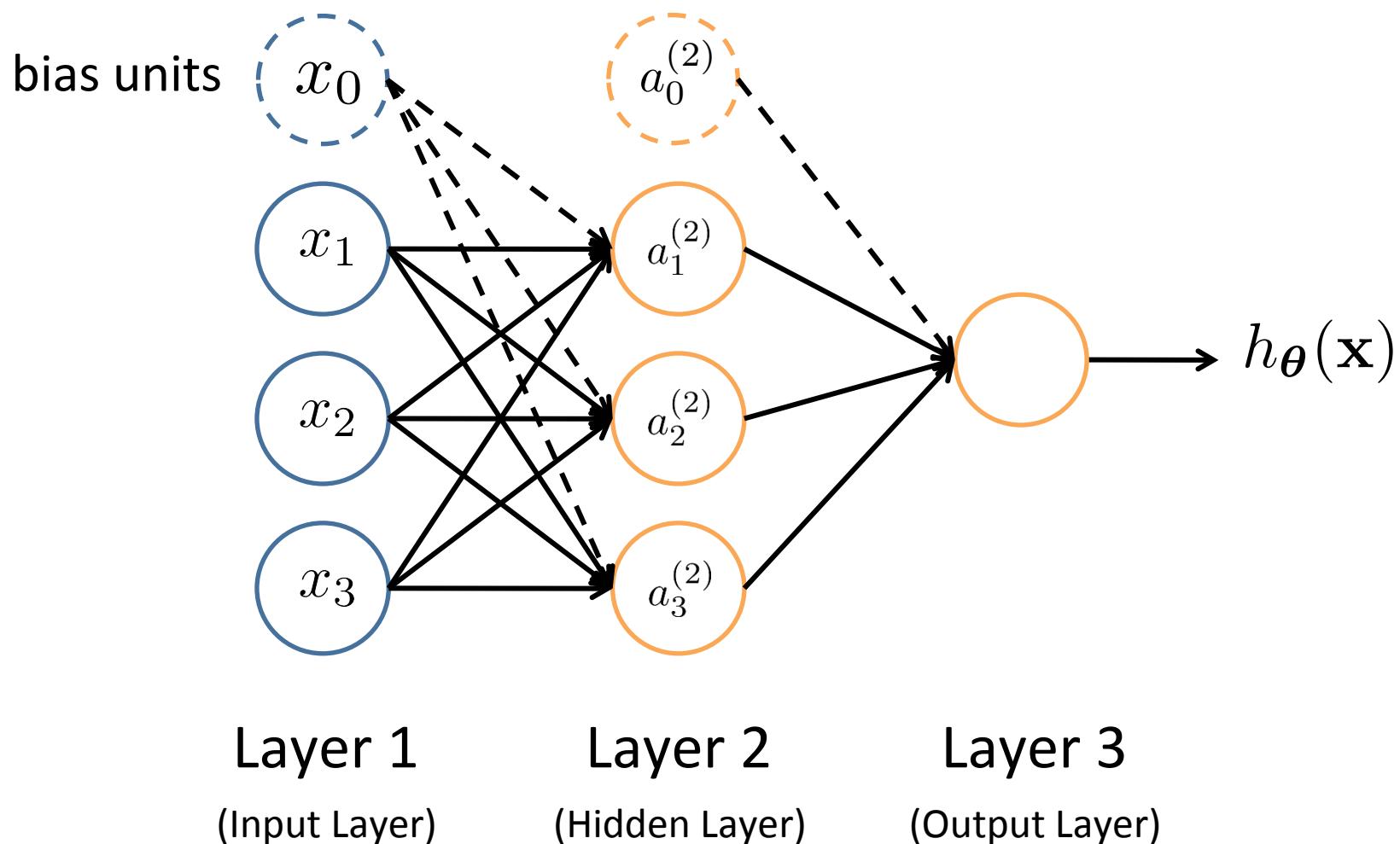
$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$

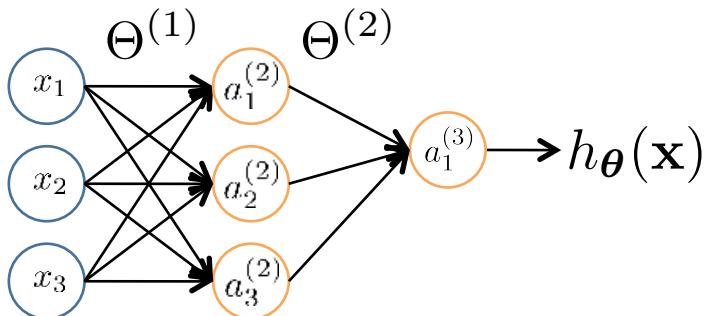
$$h_{\boldsymbol{\theta}}(\mathbf{x}) = g(\boldsymbol{\theta}^T \mathbf{x}) \\ = \frac{1}{1 + e^{-\boldsymbol{\theta}^T \mathbf{x}}}$$

Binary
Logistic
Regression

Sigmoid (logistic) activation function: $g(z) = \frac{1}{1 + e^{-z}}$

Neural Network





$a_i^{(j)}$ = “activation” of unit i in layer j
 $\Theta^{(j)}$ = weight matrix stores parameters from layer j to layer $j + 1$

Inner product of weights with activations in previous layer (or in the case of the first hidden layer)

$$a_1^{(2)} = g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3)$$

$$a_2^{(2)} = g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3)$$

$$a_3^{(2)} = g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3)$$

$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)})$$

If network has s_j units in layer j and s_{j+1} units in layer $j+1$,
then $\Theta^{(j)}$ has dimension $s_{j+1} \times (s_j + 1)$.

$$\Theta^{(1)} \in \mathbb{R}^{3 \times 4} \quad \Theta^{(2)} \in \mathbb{R}^{1 \times 4}$$

Multi-layer Neural Network

This final part is simply logistic regression (with this structure and using logistic loss function)

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g(z^{(2)})$$

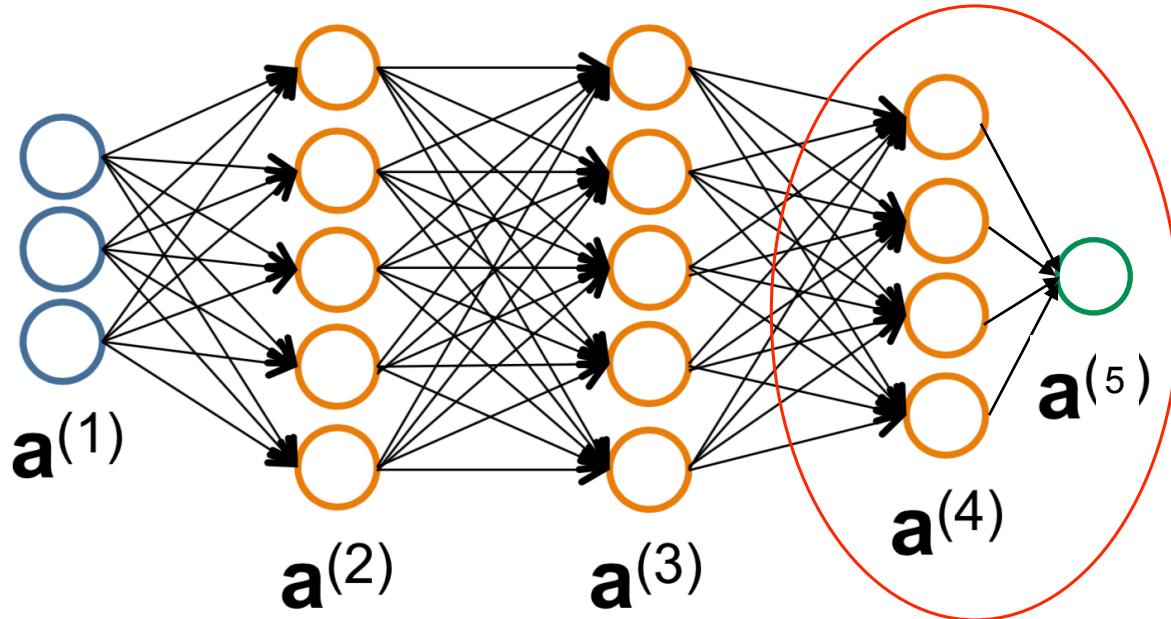
:

$$z^{(l+1)} = \Theta^{(l)} a^{(l)}$$

$$a^{(l+1)} = g(z^{(l+1)})$$

:

$$\hat{y} = a^{(L+1)}$$



$$L(y, \hat{y}) = y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})$$

$$g(z) = \frac{1}{1 + e^{-z}}$$

Binary
Logistic
Regression

Multiple Output Units: One-vs-Rest



Pedestrian



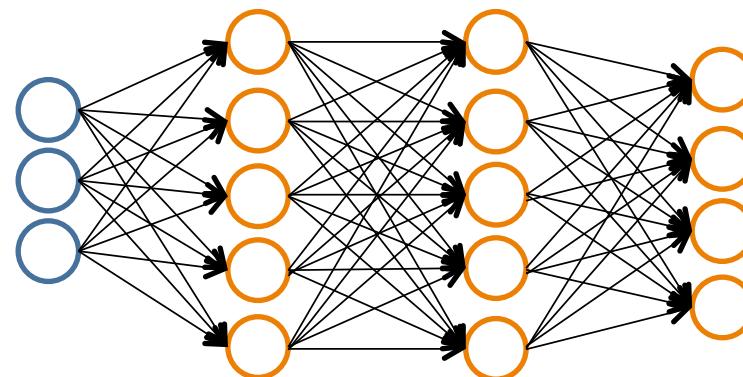
Car



Motorcycle



Truck



$$h_{\Theta}(\mathbf{x}) \in \mathbb{R}^K$$

Multi-class
Logistic
Regression

We want:

$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

when pedestrian

$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

when car

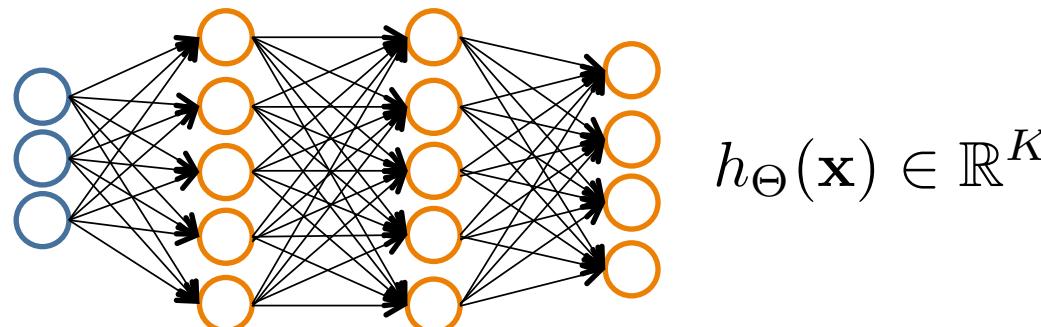
$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

when motorcycle

$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

when truck

Multiple Output Units: One-vs-Rest



We want:

$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

when pedestrian when car when motorcycle when truck

- Given $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_n, y_n)\}$
- Must convert labels to 1-of- K representation

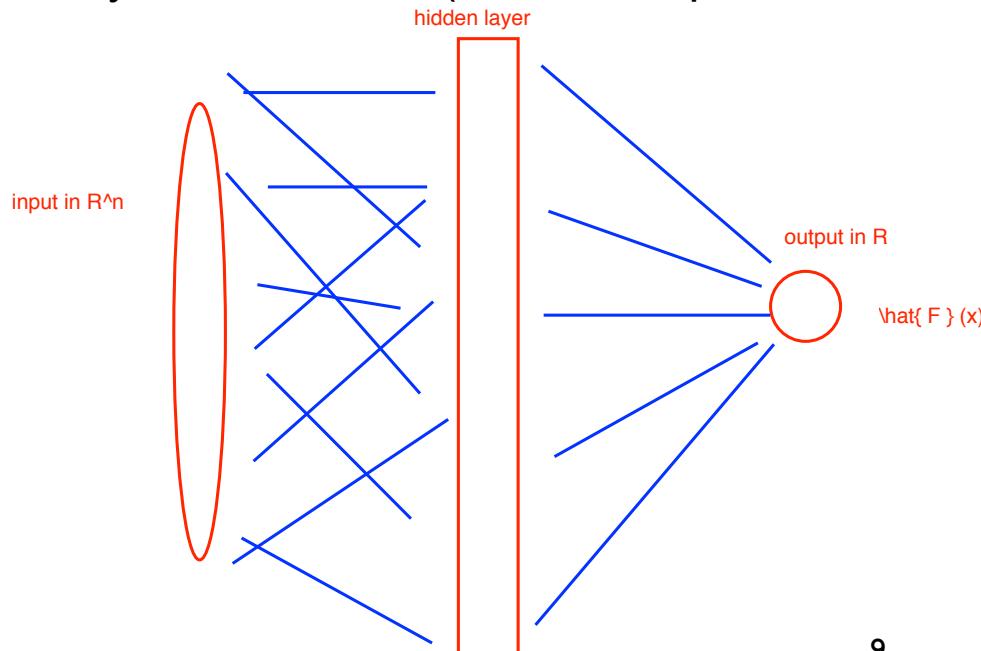
– e.g., $y_i = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$ when motorcycle, $y_i = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ when car, etc.

Neural Networks are arbitrary function approximators

one input layer, one hidden layer, one output layer

Theorem 10 (Two-Layer Networks are Universal Function Approximators). *Let F be a continuous function on a bounded subset of D -dimensional space. Then there exists a two-layer neural network \hat{F} with a finite number of hidden units that approximate F arbitrarily well. Namely, for all x in the domain of F , $|F(x) - \hat{F}(x)| < \epsilon$.*

Cybenko, Hornik (theorem reproduced from CML, Ch. 10)

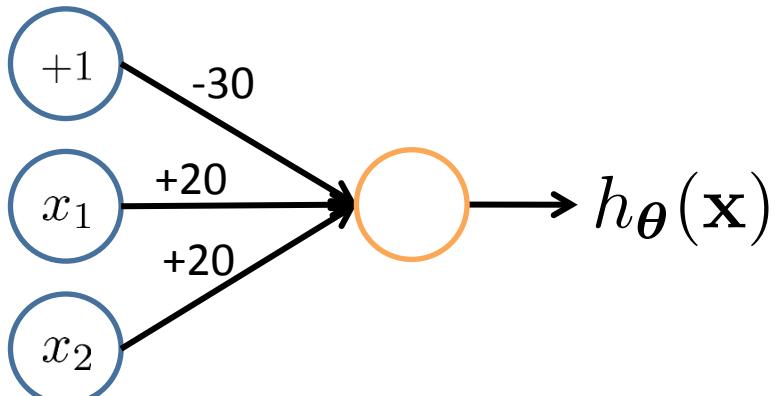


Representing Boolean Functions

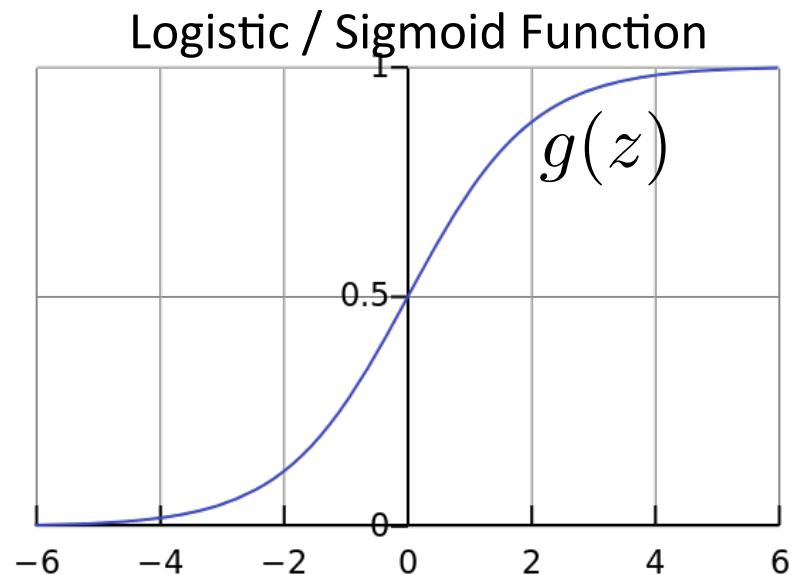
Simple example: AND

$$x_1, x_2 \in \{0, 1\}$$

$$y = x_1 \text{ AND } x_2$$



$$h_\theta(\mathbf{x}) = g(-30 + 20x_1 + 20x_2)$$

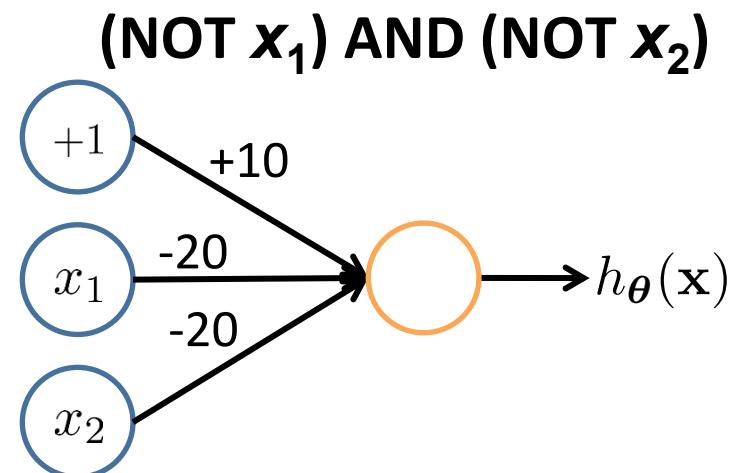
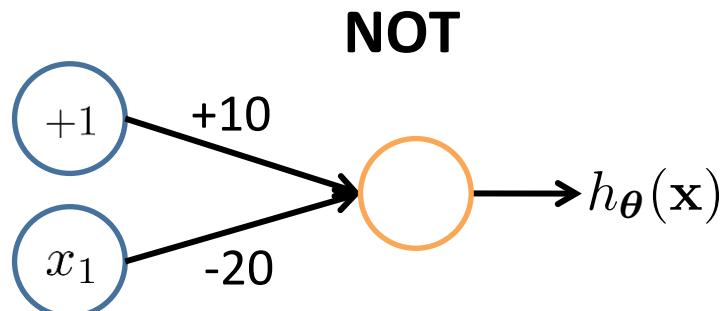
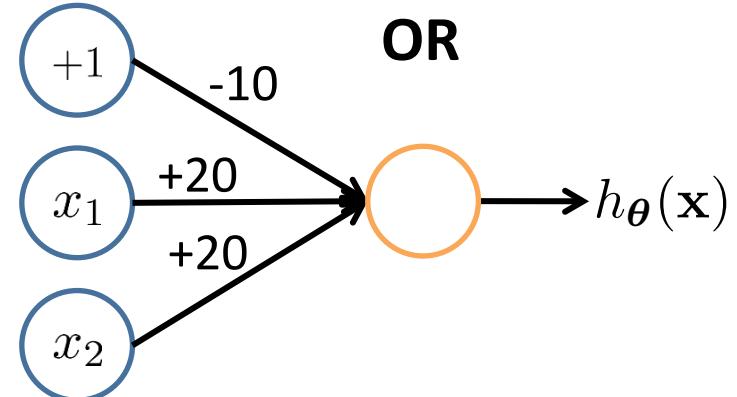
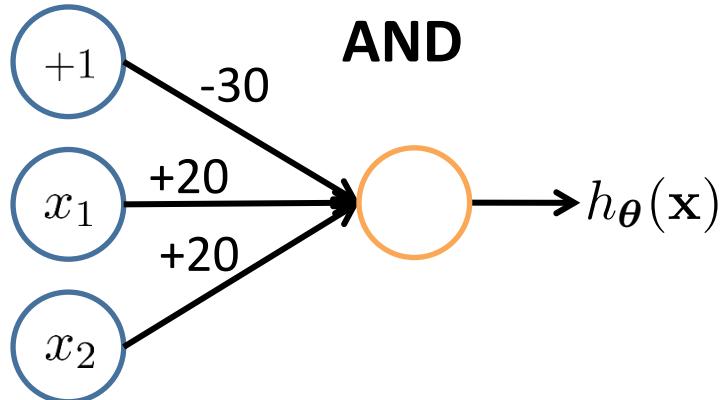


x_1	x_2	$h_\theta(\mathbf{x})$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

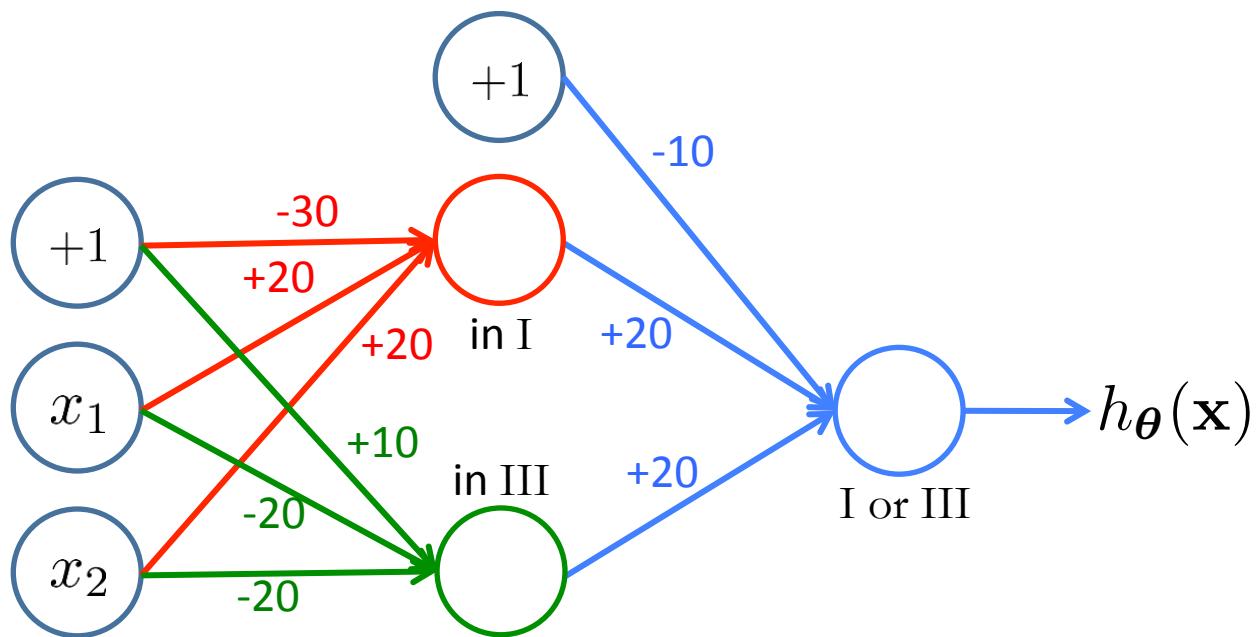
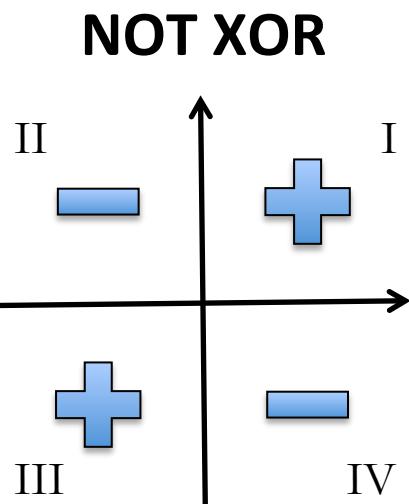
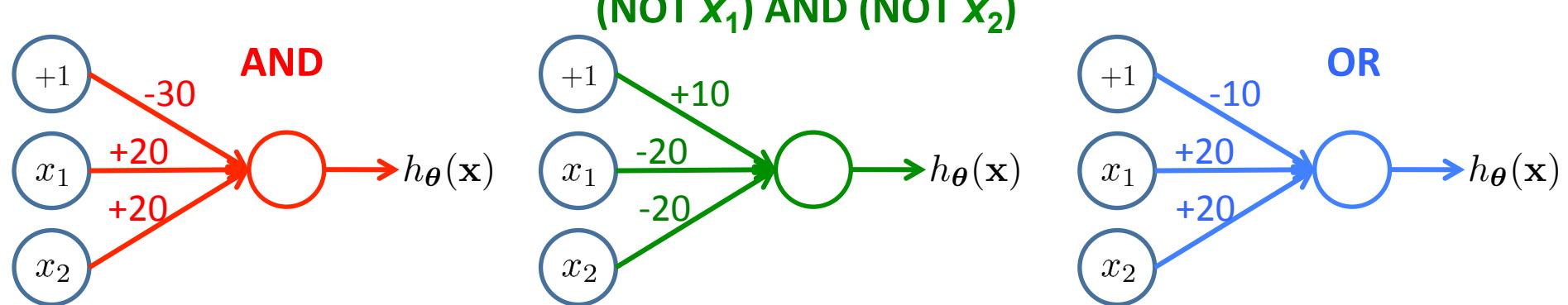
Representing Boolean Functions

With just one layer I can make this...

With two layers I can combine a series of AND / OR gates then I can make arbitrarily complex logical circuits given enough nodes in the hidden layer

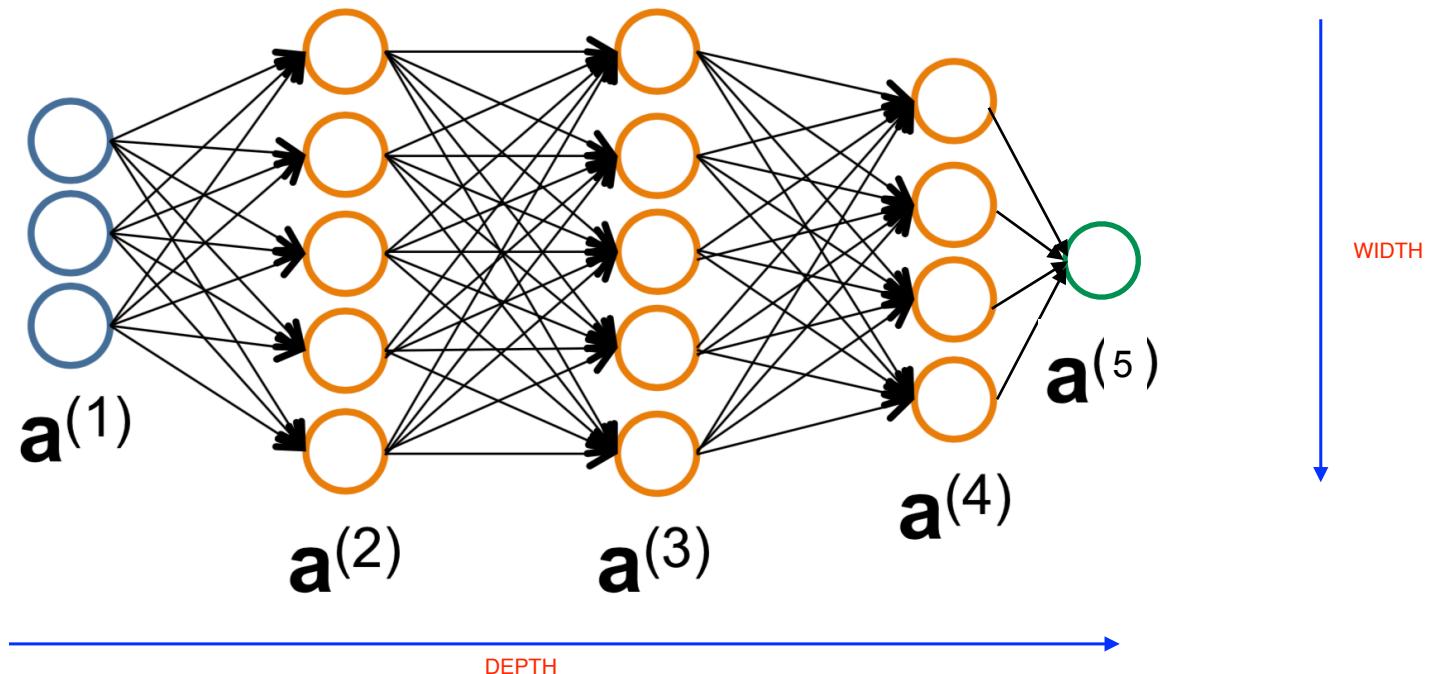


Combining Representations to Create Non-Linear Functions



Neural Network Architecture

The neural network architecture is defined by the number of layers, and the number of nodes in each layer, but also by **allowable edges**.

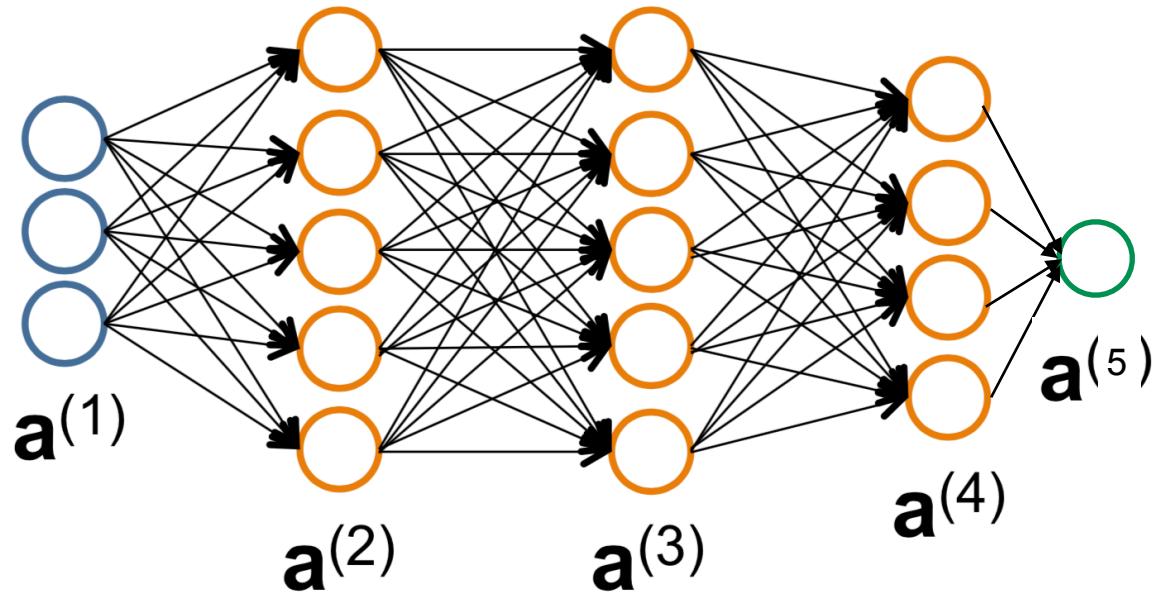


Why aren't all neural networks just 3 layers then? Well, the theorem says nothing about how large the hidden layer has to be... It could be HUGE. Plus, since the networks are fully connected you add more and more connections...

It might be that deeper networks can accomplish the same thing with less nodes!

Neural Network Architecture

The neural network architecture is defined by the number of layers, and the number of nodes in each layer, but also by **allowable edges**.



All nodes in each layer are connected to all nodes in previous layer

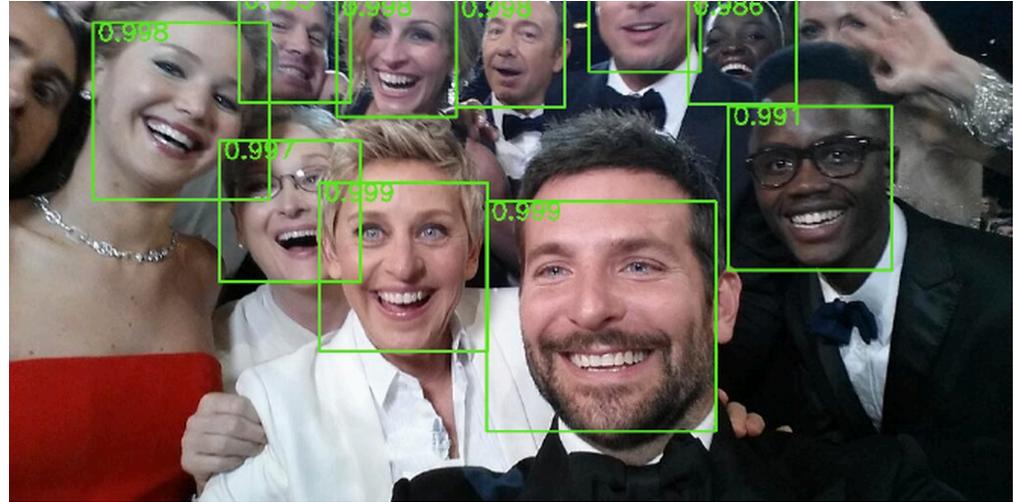
We say a layer is **Fully Connected (FC)** if all linear mappings from the current layer to the next layer are permissible.

$$a^{(k+1)} = g(\Theta a^{(k)}) \quad \text{for any } \Theta \in \mathbb{R}^{n_{k+1} \times n_k}$$

A lot of parameters!! $n_1 n_2 + n_2 n_3 + \cdots + n_L n_{L+1}$

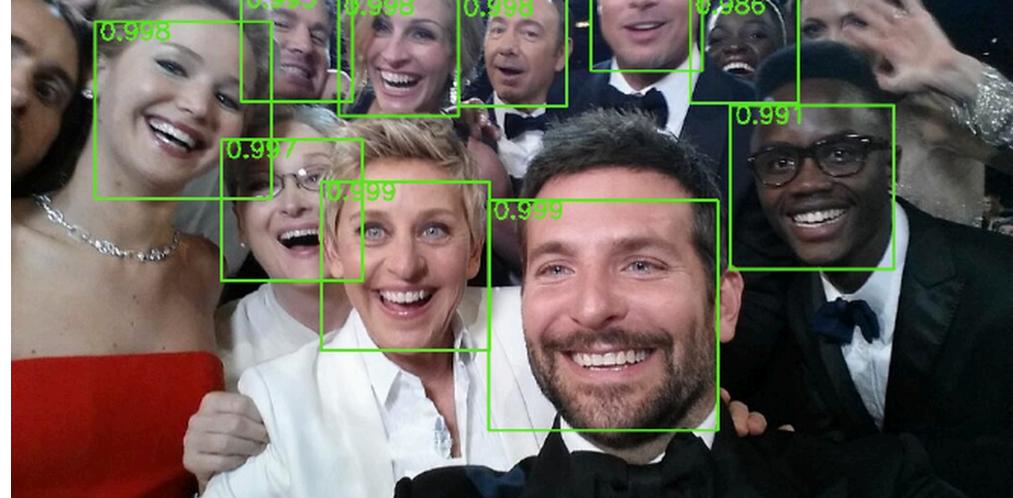
Neural Network Architecture

Objects are often
localized in space so to
find the faces in an
image, not every pixel is
important for classification
—makes sense to drag a
window across an image.

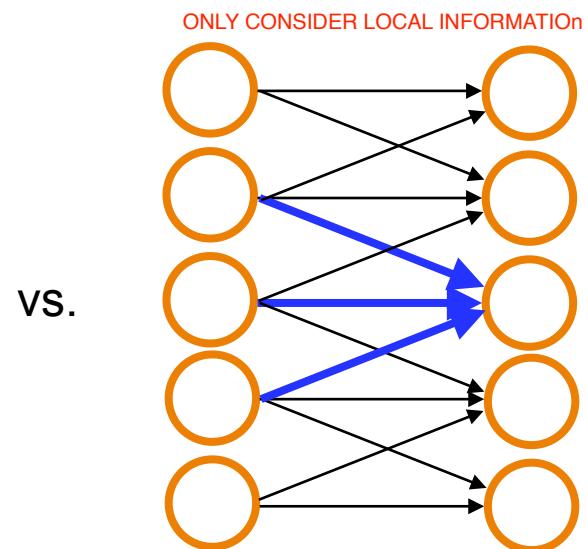
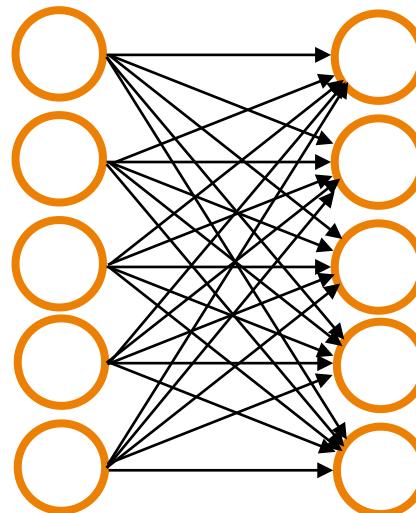


Neural Network Architecture

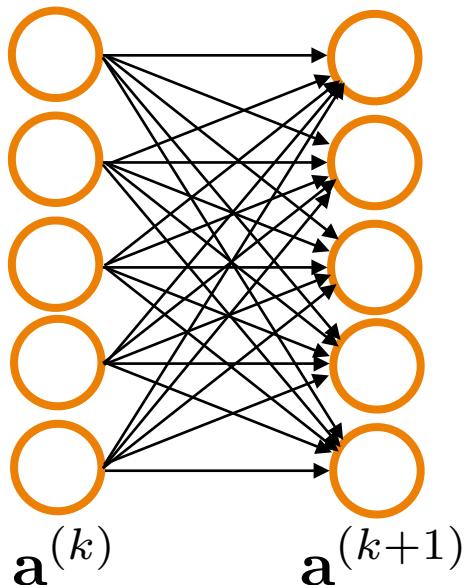
Objects are often **localized in space** so to find the faces in an image, not every pixel is important for classification —makes sense to drag a window across an image.



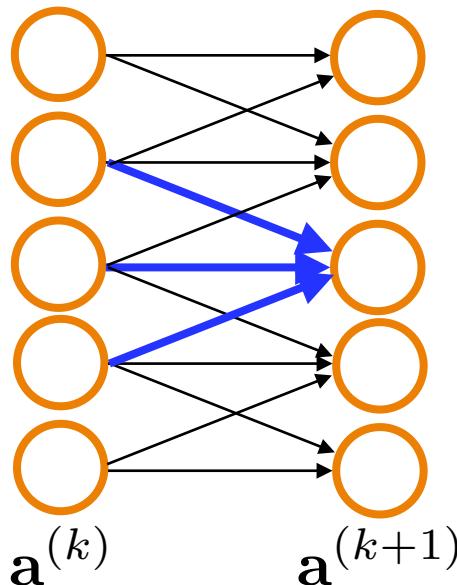
Similarly, to identify edges or other local structure, it makes sense to only look at **local information**



Neural Network Architecture



vs.



$$\begin{bmatrix} \Theta_{0,0} & \Theta_{0,1} & \Theta_{0,2} & \Theta_{0,3} & \Theta_{0,4} \\ \Theta_{1,0} & \Theta_{1,1} & \Theta_{1,2} & \Theta_{1,3} & \Theta_{1,4} \\ \Theta_{2,0} & \Theta_{2,1} & \Theta_{2,2} & \Theta_{2,3} & \Theta_{2,4} \\ \Theta_{3,0} & \Theta_{3,1} & \Theta_{3,2} & \Theta_{3,3} & \Theta_{3,4} \\ \Theta_{4,0} & \Theta_{4,1} & \Theta_{4,2} & \Theta_{4,3} & \Theta_{4,4} \end{bmatrix}$$

Parameters: n^2

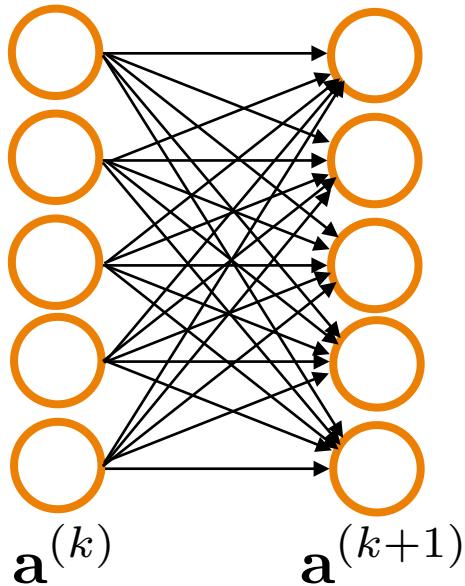
$$\begin{bmatrix} \Theta_{0,0} & \Theta_{0,1} & 0 & 0 & 0 \\ \Theta_{1,0} & \Theta_{1,1} & \Theta_{1,2} & 0 & 0 \\ 0 & \Theta_{2,1} & \Theta_{2,2} & \Theta_{2,3} & 0 \\ 0 & 0 & \Theta_{3,2} & \Theta_{3,3} & \Theta_{3,4} \\ 0 & 0 & 0 & \Theta_{4,3} & \Theta_{4,4} \end{bmatrix}$$

$3n - 2$

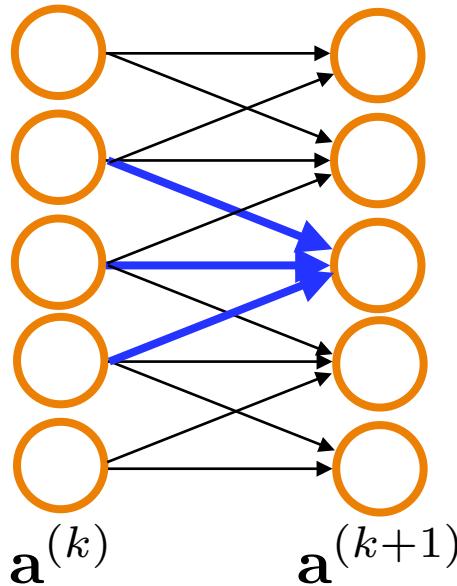
WAY less parameters...

$$\mathbf{a}_i^{(k+1)} = g \left(\sum_{j=0}^{n-1} \Theta_{i,j} \mathbf{a}_j^{(k)} \right)$$

Neural Network Architecture



vs.



Share weights in each receptive field in the image

Mirror/share local weights everywhere (e.g., structure equally likely to be anywhere in image)

$$\begin{bmatrix} \Theta_{0,0} & \Theta_{0,1} & \Theta_{0,2} & \Theta_{0,3} & \Theta_{0,4} \\ \Theta_{1,0} & \Theta_{1,1} & \Theta_{1,2} & \Theta_{1,3} & \Theta_{1,4} \\ \Theta_{2,0} & \Theta_{2,1} & \Theta_{2,2} & \Theta_{2,3} & \Theta_{2,4} \\ \Theta_{3,0} & \Theta_{3,1} & \Theta_{3,2} & \Theta_{3,3} & \Theta_{3,4} \\ \Theta_{4,0} & \Theta_{4,1} & \Theta_{4,2} & \Theta_{4,3} & \Theta_{4,4} \end{bmatrix}$$

Parameters: n^2

$$\begin{bmatrix} \Theta_{0,0} & \Theta_{0,1} & 0 & 0 & 0 \\ \Theta_{1,0} & \Theta_{1,1} & \Theta_{1,2} & 0 & 0 \\ 0 & \Theta_{2,1} & \Theta_{2,2} & \Theta_{2,3} & 0 \\ 0 & 0 & \Theta_{3,2} & \Theta_{3,3} & \Theta_{3,4} \\ 0 & 0 & 0 & \Theta_{4,3} & \Theta_{4,4} \end{bmatrix}$$

$3n - 2$

$$\begin{bmatrix} \theta_1 & \theta_2 & 0 & 0 & 0 \\ \theta_0 & \theta_1 & \theta_2 & 0 & 0 \\ 0 & \theta_0 & \theta_1 & \theta_2 & 0 \\ 0 & 0 & \theta_0 & \theta_1 & \theta_2 \\ 0 & 0 & 0 & \theta_0 & \theta_1 \end{bmatrix}$$

3

$$\mathbf{a}_i^{(k+1)} = g \left(\sum_{j=0}^{n-1} \Theta_{i,j} \mathbf{a}_j^{(k)} \right)$$

$$\mathbf{a}_i^{(k+1)} = g \left(\sum_{j=0}^{m-1} \theta_j \mathbf{a}_{i+j}^{(k)} \right)$$

Neural Network Architecture

Fully Connected (FC) Layer

$$\begin{bmatrix} \Theta_{0,0} & \Theta_{0,1} & \Theta_{0,2} & \Theta_{0,3} & \Theta_{0,4} \\ \Theta_{1,0} & \Theta_{1,1} & \Theta_{1,2} & \Theta_{1,3} & \Theta_{1,4} \\ \Theta_{2,0} & \Theta_{2,1} & \Theta_{2,2} & \Theta_{2,3} & \Theta_{2,4} \\ \Theta_{3,0} & \Theta_{3,1} & \Theta_{3,2} & \Theta_{3,3} & \Theta_{3,4} \\ \Theta_{4,0} & \Theta_{4,1} & \Theta_{4,2} & \Theta_{4,3} & \Theta_{4,4} \end{bmatrix}$$

Convolutional (CONV) Layer (1 filter)

$$\begin{bmatrix} \theta_1 & \theta_2 & 0 & 0 & 0 \\ \theta_0 & \theta_1 & \theta_2 & 0 & 0 \\ 0 & \theta_0 & \theta_1 & \theta_2 & 0 \\ 0 & 0 & \theta_0 & \theta_1 & \theta_2 \\ 0 & 0 & 0 & \theta_0 & \theta_1 \end{bmatrix}$$

m=3

cross correlation vs convolution?

$$\mathbf{a}_i^{(k+1)} = g \left(\sum_{j=0}^{n-1} \Theta_{i,j} \mathbf{a}_j^{(k)} \right)$$

$$\mathbf{a}_i^{(k+1)} = g \left(\sum_{j=0}^{m-1} \theta_j \mathbf{a}_{i+j}^{(k)} \right) = g([\theta * \mathbf{a}^{(k)}]_i)$$

Convolution*

$\theta = (\theta_0, \dots, \theta_{m-1}) \in \mathbb{R}^m$ is referred to as a “filter”

* Actually defined as the closely related quantity of “cross-correlation” but the deep learning literature just calls this “convolution”

Example (1d convolution)

$$(\theta * x)_i = \sum_{j=0}^{m-1} \theta_j x_{i+j}$$

i=0 to n-m

1	1	1	0	0
---	---	---	---	---

Input $x \in \mathbb{R}^n$

1	0	1
---	---	---

Filter $\theta \in \mathbb{R}^m$



Output $\theta * x$

Example (1d convolution)

$$(\theta * x)_i = \sum_{j=0}^{m-1} \theta_j x_{i+j}$$

1	1	1	0	0
---	---	---	---	---

Input $x \in \mathbb{R}^n$

1	0	1
---	---	---

Filter $\theta \in \mathbb{R}^m$

1	1	1	0	0
<small>$\times 1$</small>	<small>$\times 0$</small>	<small>$\times 1$</small>		



2		
---	--	--

Output $\theta * x$

Example (1d convolution)

$$(\theta * x)_i = \sum_{j=0}^{m-1} \theta_j x_{i+j}$$

1	1	1	0	0
---	---	---	---	---

Input $x \in \mathbb{R}^n$

1	0	1
---	---	---

Filter $\theta \in \mathbb{R}^m$

1	1	1	0	0
	<small>$\times 1$</small>	<small>$\times 0$</small>	<small>$\times 1$</small>	



2	1	
---	---	--

Output $\theta * x$

Example (1d convolution)

$$(\theta * x)_i = \sum_{j=0}^{m-1} \theta_j x_{i+j}$$

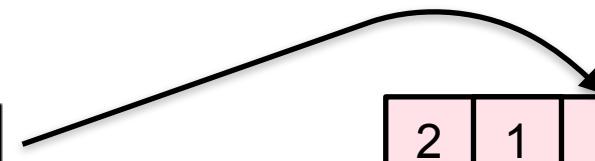
1	1	1	0	0
---	---	---	---	---

Input $x \in \mathbb{R}^n$

1	0	1
---	---	---

Filter $\theta \in \mathbb{R}^m$

1	1	1	0	0
		$\times 1$	$\times 0$	$\times 1$

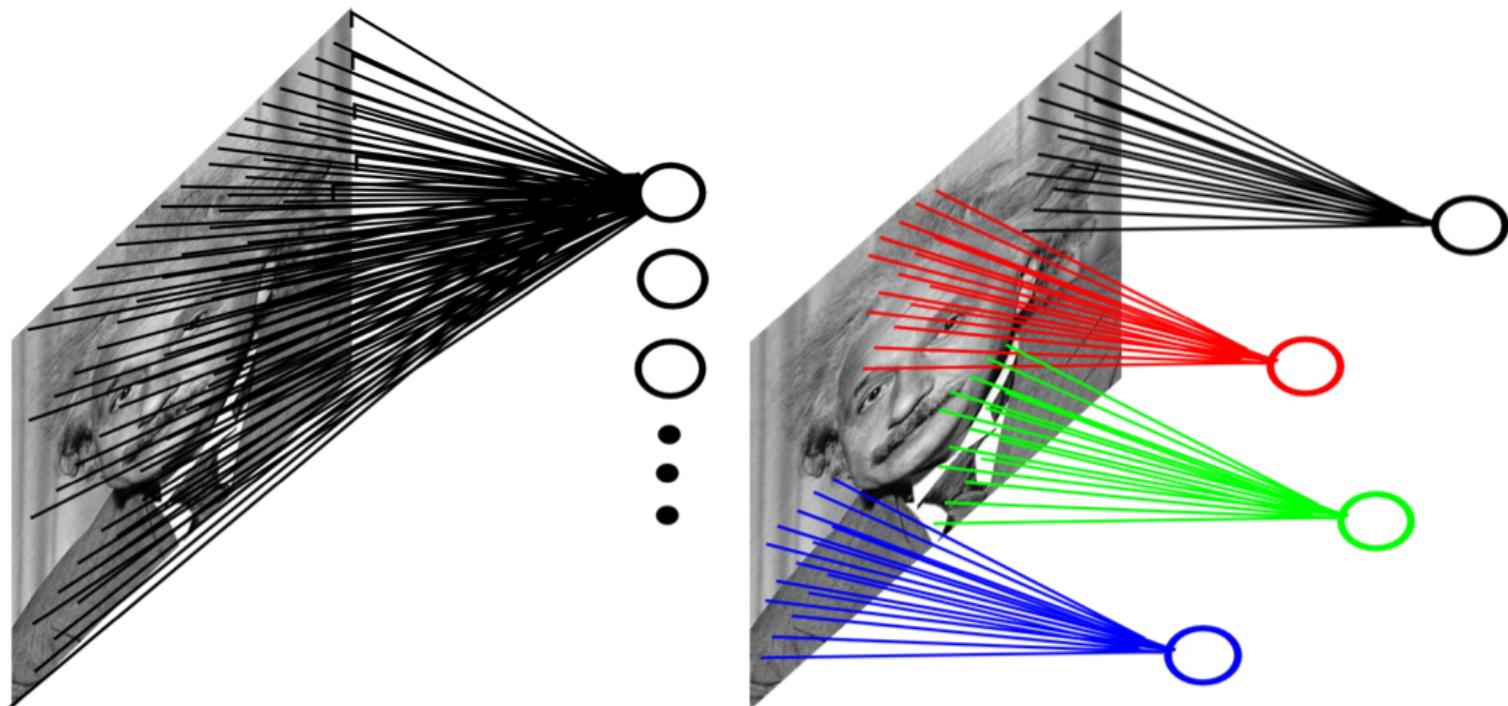


Output $\theta * x$

2d Convolution Layer

Example: 200x200 image

- ▶ Fully-connected, 400,000 hidden units = 16 billion parameters
- ▶ Locally-connected, 400,000 hidden units 10x10 fields = 40 million params
- ▶ Local connections capture local dependencies



Convolution of images (2d convolution)

$$(I * K)(i, j) = \sum_m \sum_n I(i + m, j + n)K(m, n)$$

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Image I

1	0	1
0	1	0
1	0	1

Filter K

1 x1	1 x0	1 x1	0	0
0 x0	1 x1	1 x0	1	0
0 x1	0 x0	1 x1	1	1
0	0	1	1	0
0	1	1	0	0

Image

4		

Convolved
Feature

$$I * K$$

Convolution of images

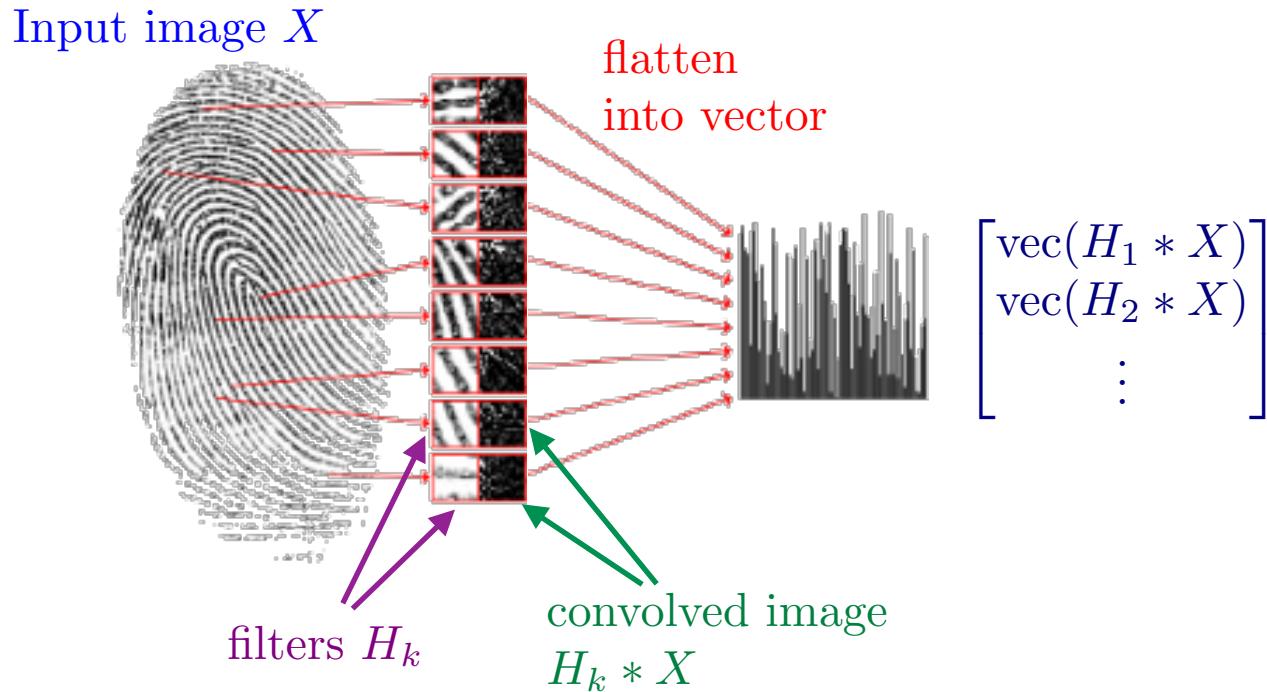
$$(I * K)(i, j) = \sum_m \sum_n I(i + m, j + n)K(m, n)$$

Image I

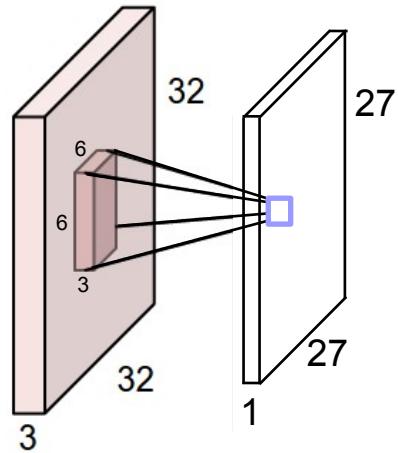


Operation	Filter K	Convolved Image $I * K$
	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Convolution of images



3d Convolution

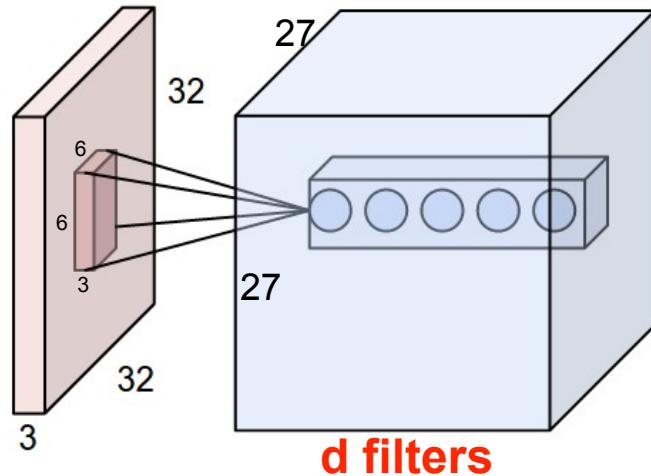


$$\Theta \in \mathbb{R}^{m \times m \times r}$$

$$x \in \mathbb{R}^{n \times n \times r}$$

$$(\Theta * x)_{s,t} = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} \sum_{k=0}^{r-1} \Theta_{i,j,k} x_{s+i, t+j}$$

Stacking convolved images



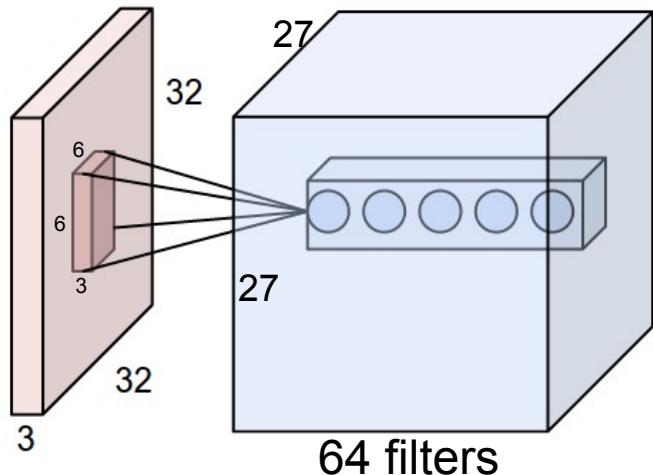
Repeat with d filters!

$$\Theta \in \mathbb{R}^{m \times m \times r}$$

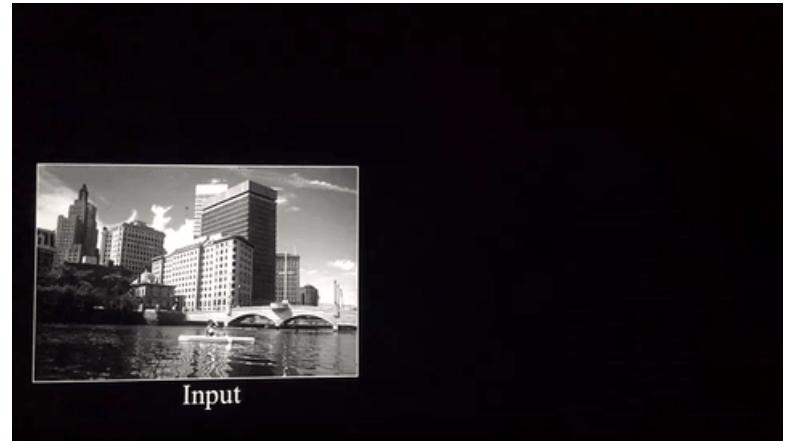
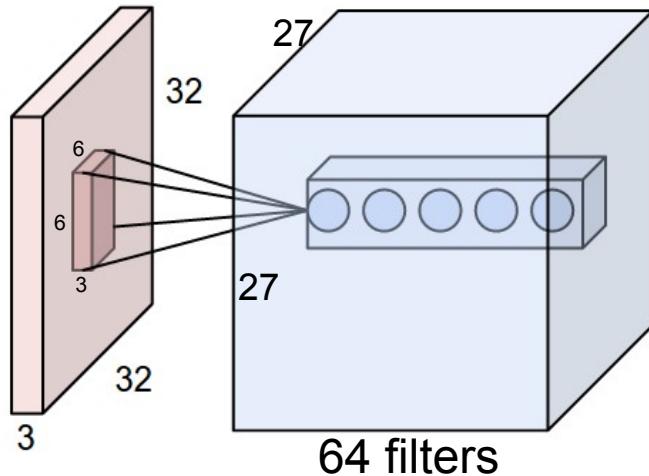
$$x \in \mathbb{R}^{n \times n \times r}$$

$$(\Theta * x)_{s,t} = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} \sum_{k=0}^{r-1} \Theta_{i,j,k} x_{s+i, t+j}$$

Stacking convolved images

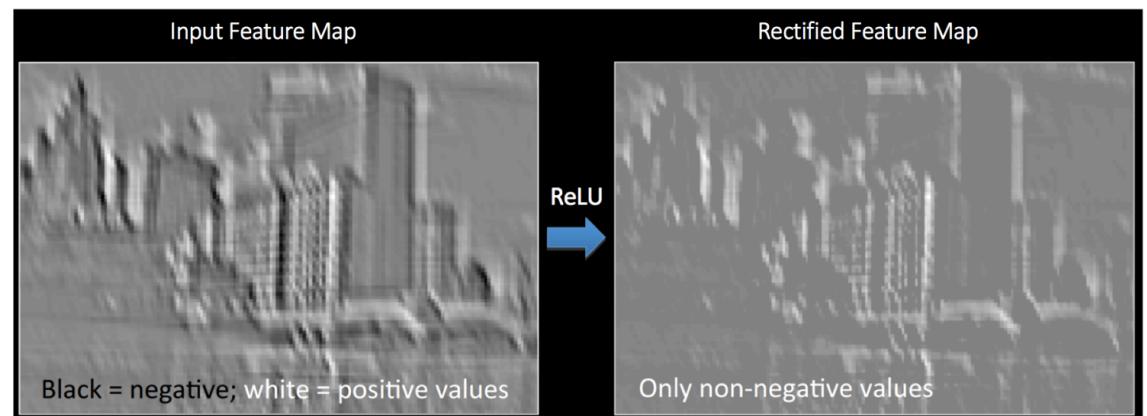


Stacking convolved images



Apply Non-linearity to
the output of each
layer, Here: ReLU
(rectified linear unit)

crucial for representing nonlinearities in response;;;
CANNOT achieve this success with simply linear transfer
functions...



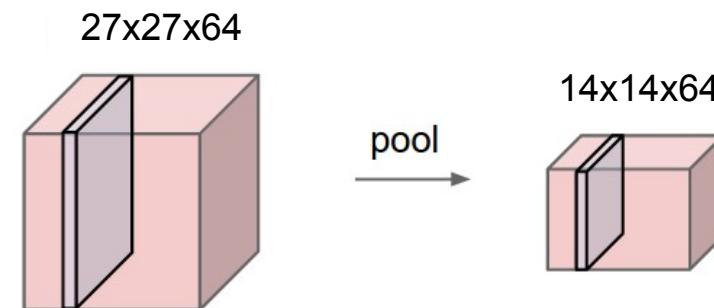
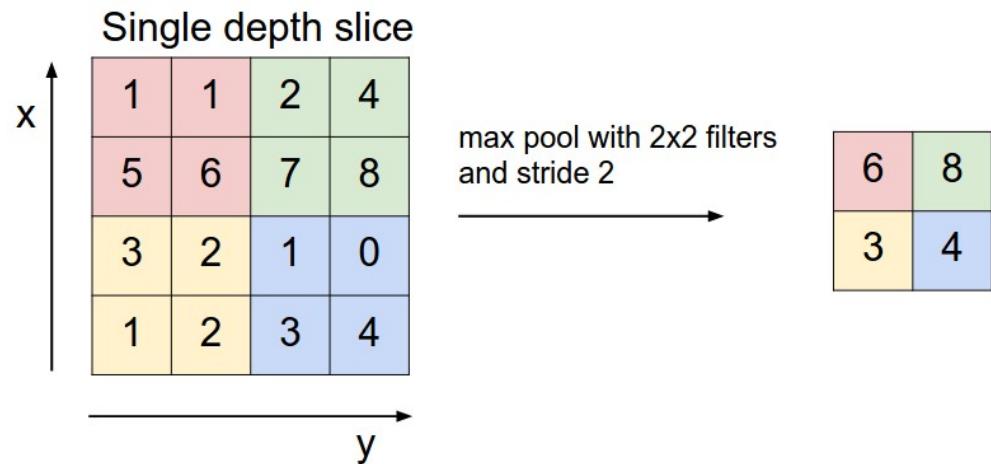
Other choices: sigmoid, arctan

Pooling

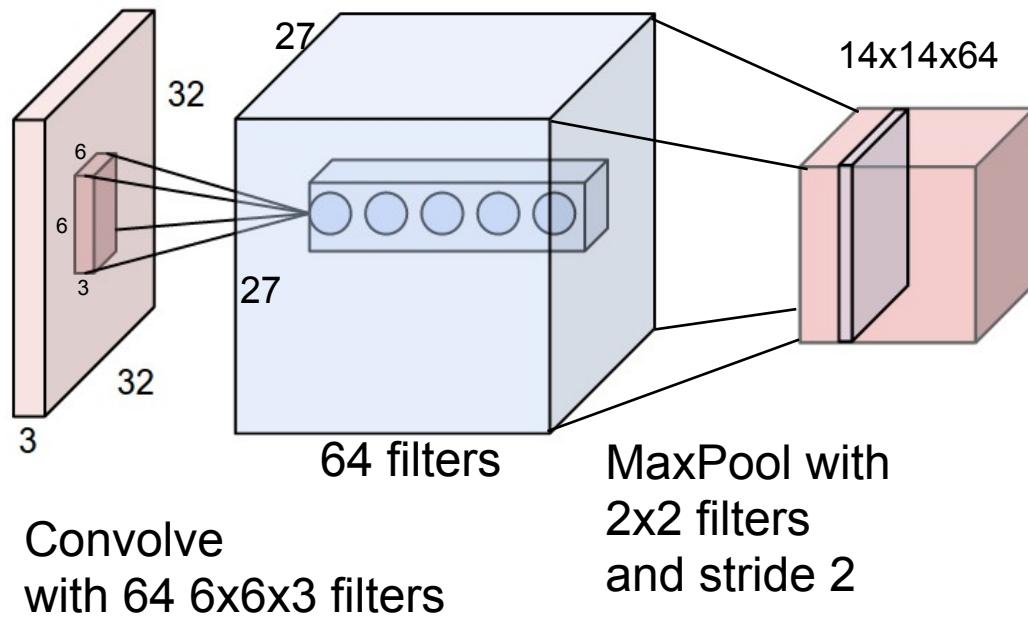
Reduce effect of translation on convolution.

This feature is somewhere around here in the image. Does not have to be at an exact (x,y) position, just in this general area.

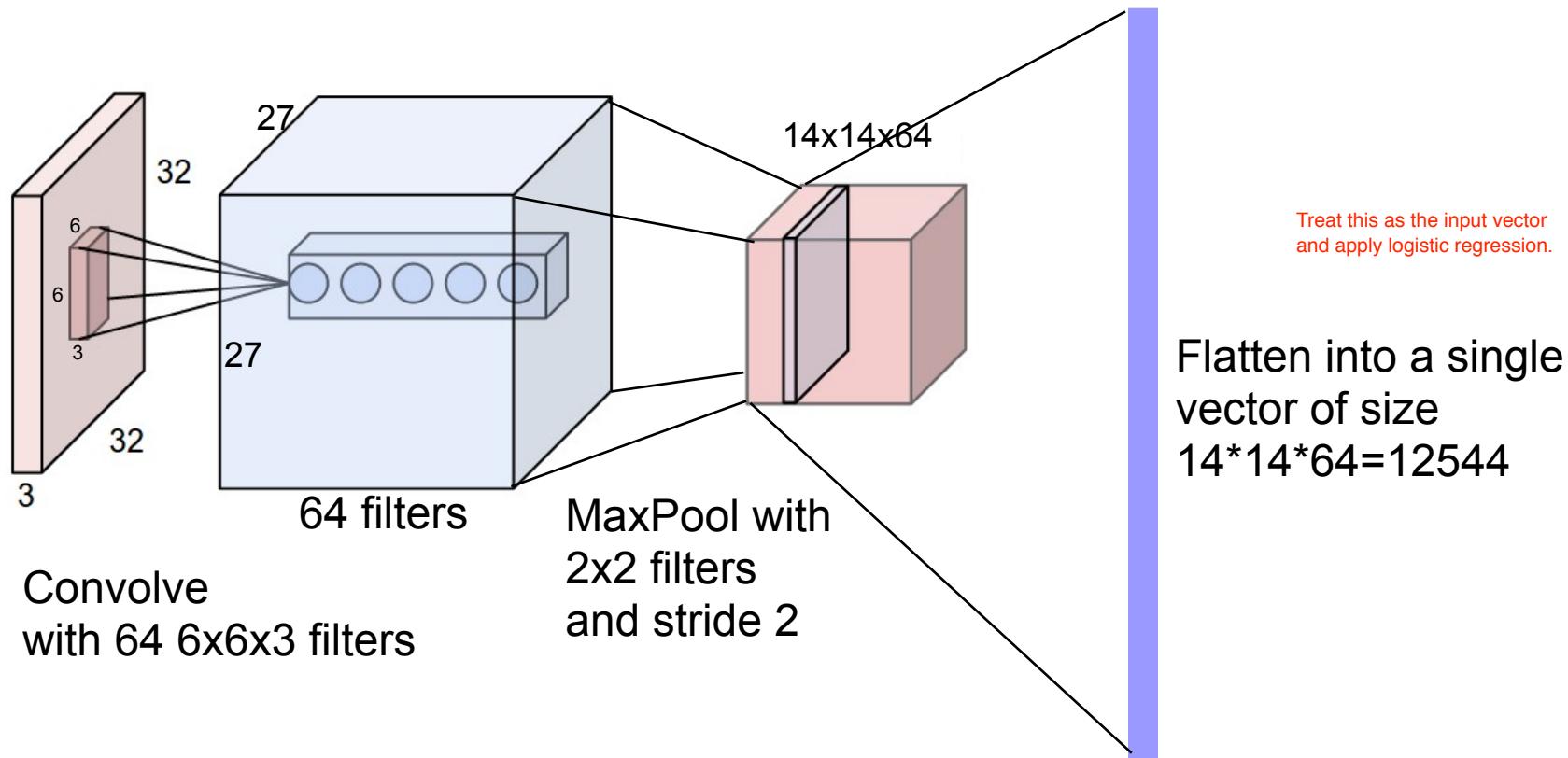
Pooling reduces the dimension and can be interpreted as “This filter had a high response in this general region”



Pooling Convolution layer



Simplest feature pipeline

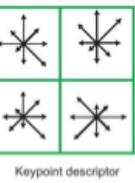
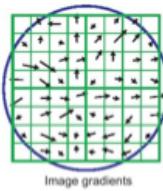


How do we choose all the hyperparameters?

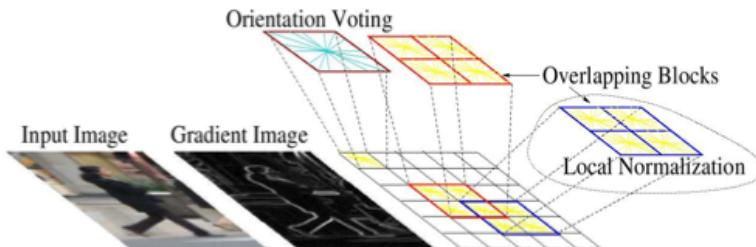
How do we choose the filters?

- Hand crafted (digital signal processing, c.f. wavelets)
- Learn them (deep learning)

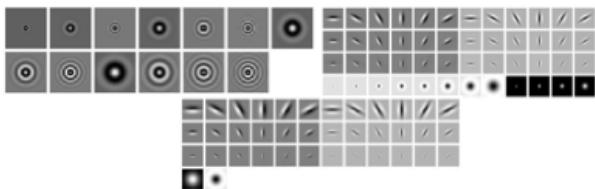
Some hand-created image features



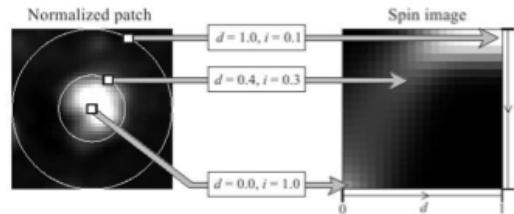
SIFT



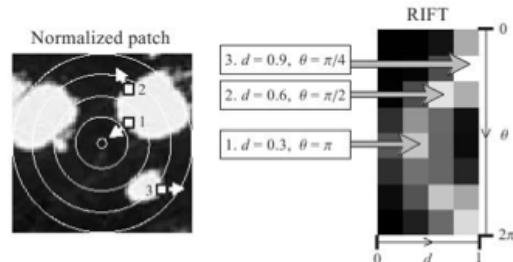
HoG



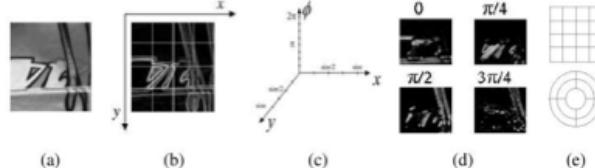
Texton



Spin Image



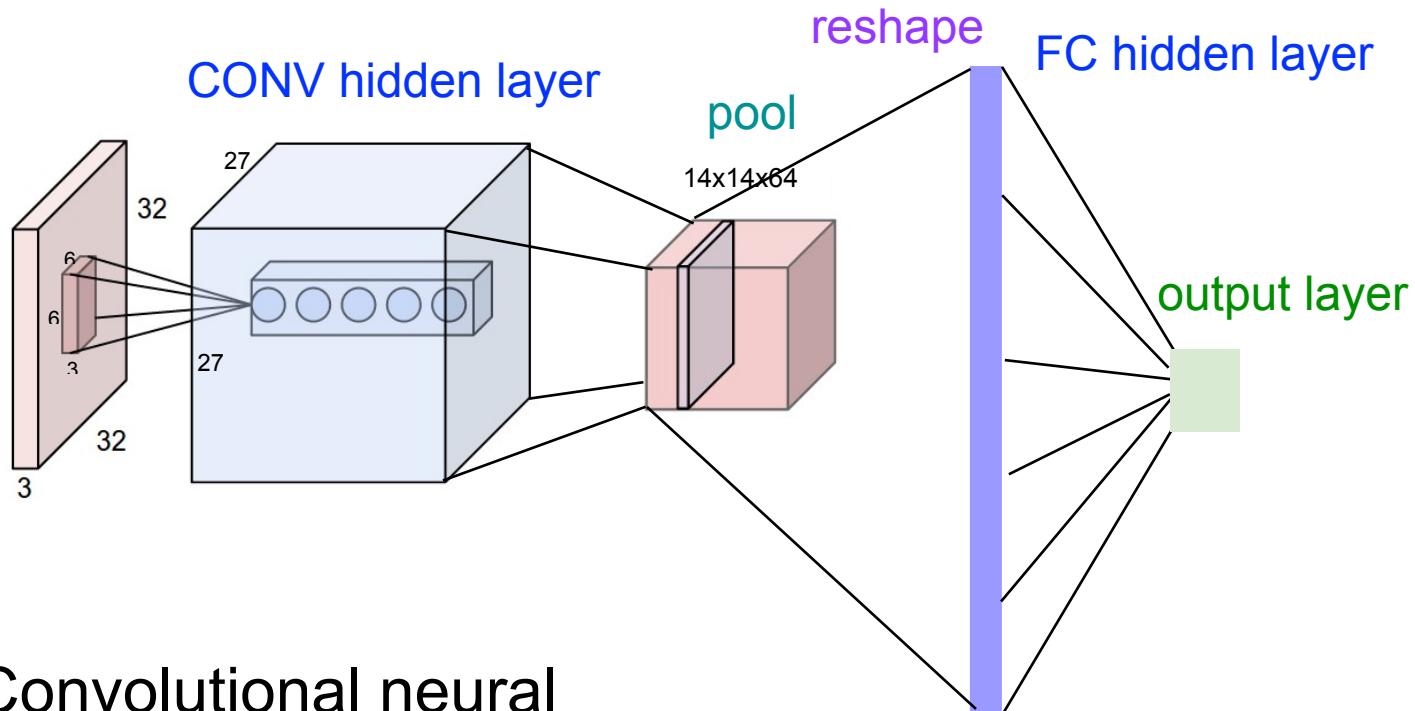
RIFT



GLOH

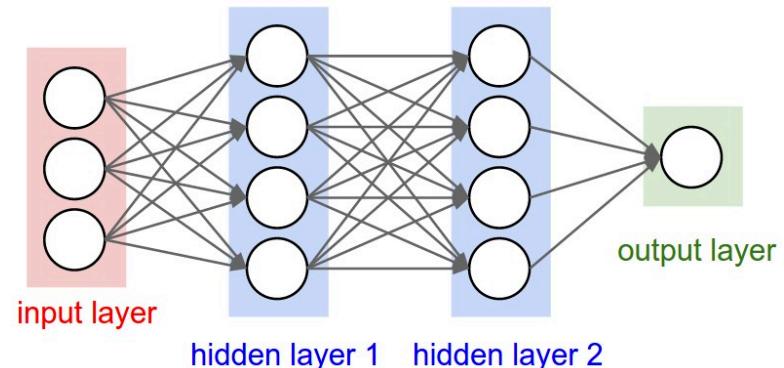
Slide from Honglak Lee

Learning Features with Convolutional Networks

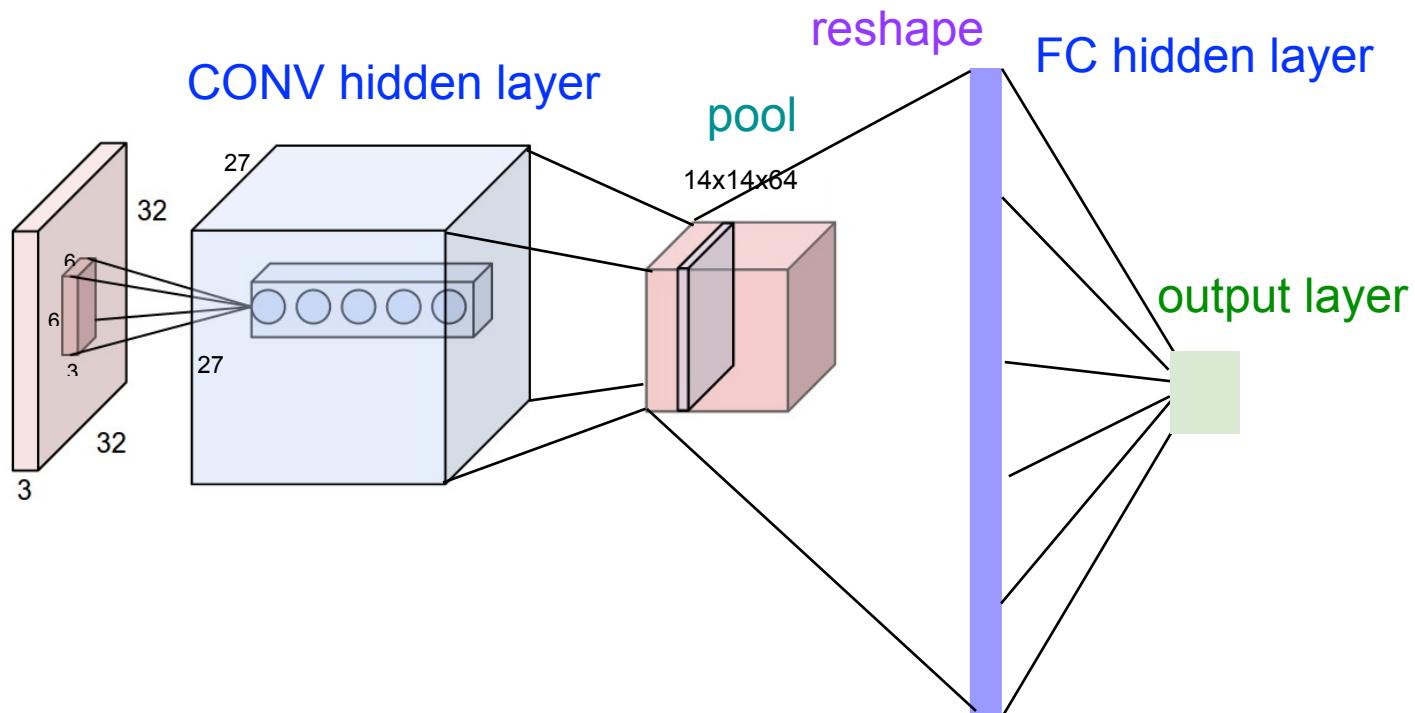


Recall: Convolutional neural networks (CNN) are just regular fully connected (FC) neural networks with some connections removed.

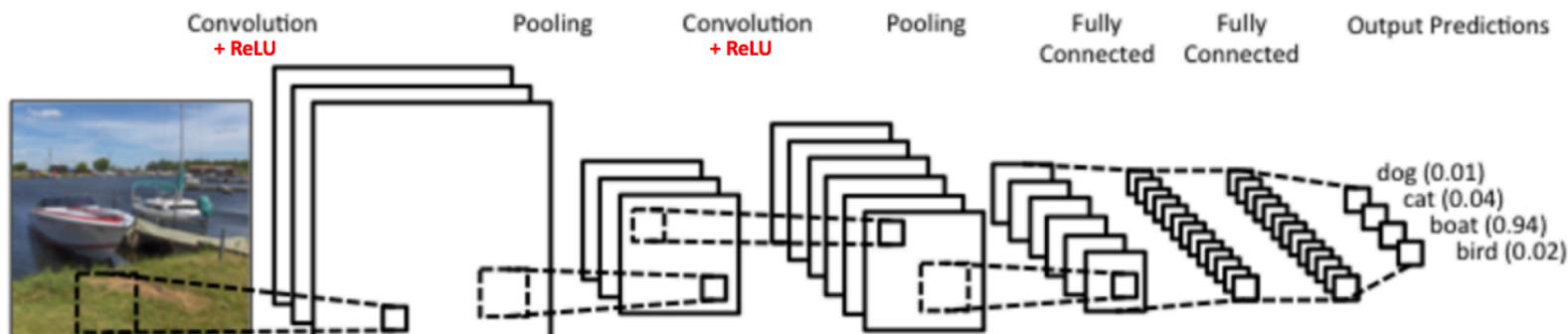
Train with SGD!

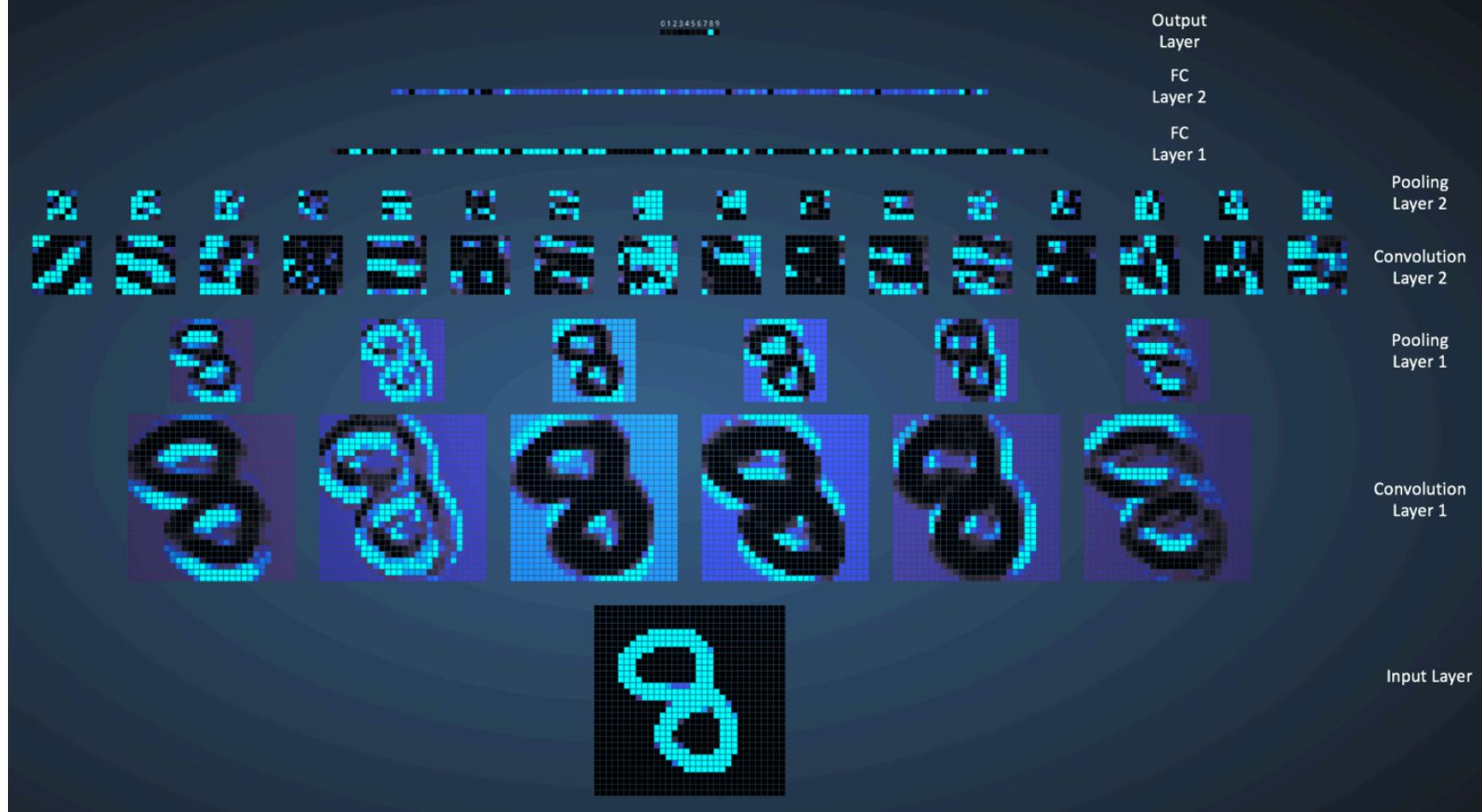


Training Convolutional Networks

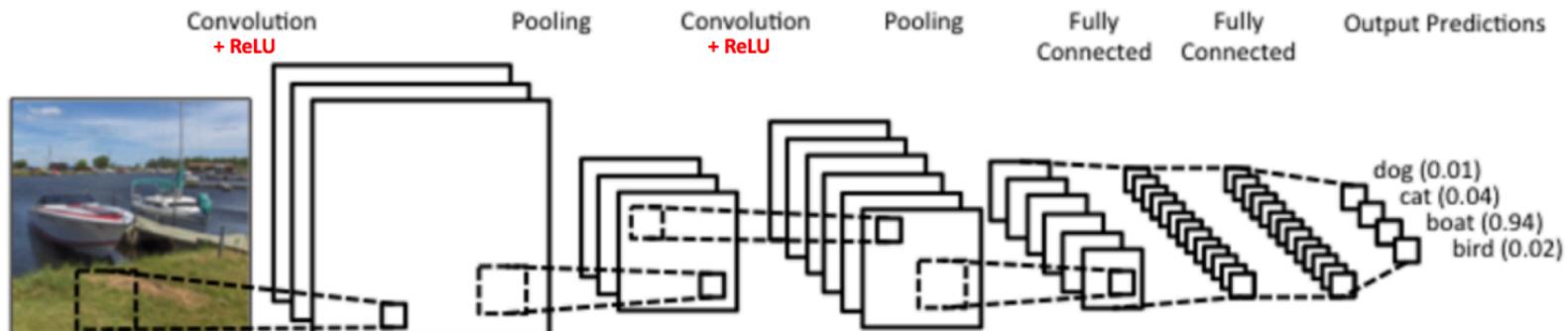


Real example network: LeNet





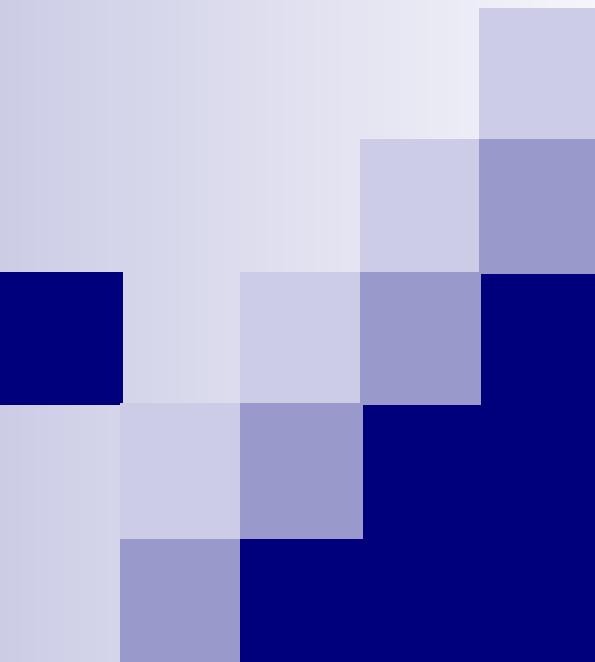
Real example network: LeNet



Remarks

Automatic way to generate features; Build feature space that is useful for classification.

- Convolution is a fundamental operation in signal processing. Instead of hand-engineering the filters (e.g., Fourier, Wavelets, etc.) **Deep Learning learns the filters and CONV layers with back-propagation**, replacing fully connected (FC) layers with convolutional (CONV) layers
- **Pooling** is a dimensionality reduction operation that summarizes the output of convolving the input with a filter
- Typically the last few layers are **Fully Connected (FC)**, with the interpretation that the CONV layers are feature extractors, preparing input for the final FC layers. Can replace last layers and retrain on different dataset+task.
- Just as hard to train as regular neural networks.
- More exotic network architectures for specific tasks



END LECTURE WED MAY 29

Training Neural Networks

Machine Learning – CSE446
Kevin Jamieson
University of Washington

May 29, 2019

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g(z^{(2)})$$

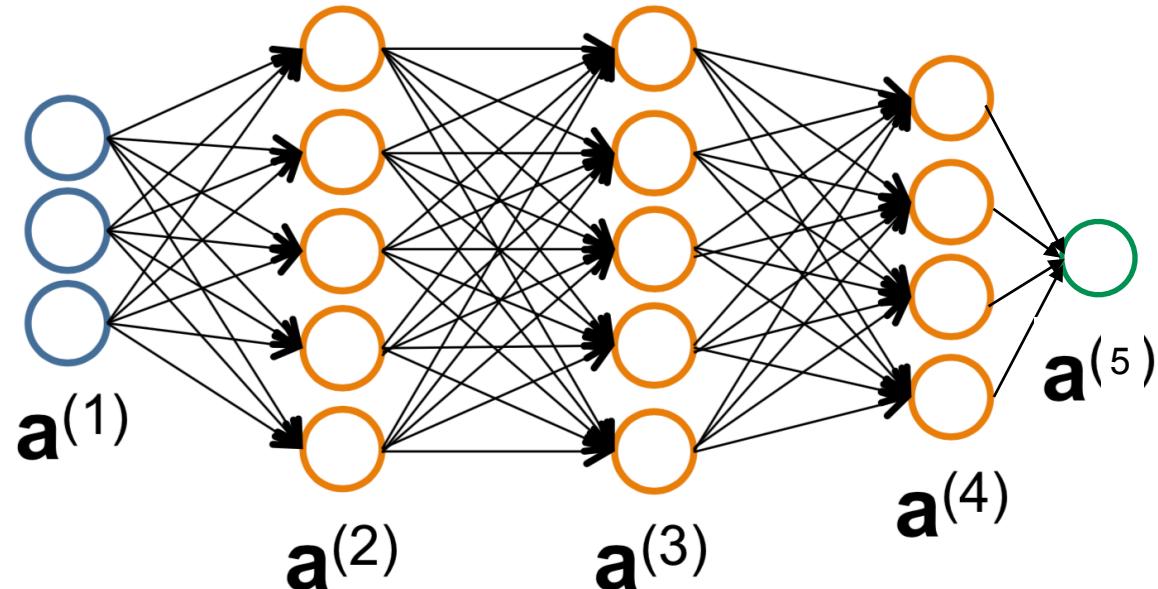
⋮

$$z^{(l+1)} = \Theta^{(l)} a^{(l)}$$

$$a^{(l+1)} = g(z^{(l+1)})$$

⋮

$$\hat{y} = a^{(L+1)}$$



$$L(y, \hat{y}) = y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})$$

$$g(z) = \frac{1}{1 + e^{-z}}$$

Gradient Descent: $\Theta^{(l)} \leftarrow \Theta^{(l)} - \eta \nabla_{\Theta^{(l)}} L(y, \hat{y}) \quad \forall l$

$$\text{Gradient Descent: } \Theta^{(l)} \leftarrow \Theta^{(l)} - \eta \nabla_{\Theta^{(l)}} L(y, \hat{y}) \quad \forall l$$

Seems simple enough, why are packages like PyTorch, Tensorflow, Theano, Cafe, MxNet synonymous with deep learning?

1. Automatic differentiation

$$\text{Gradient Descent: } \Theta^{(l)} \leftarrow \Theta^{(l)} - \eta \nabla_{\Theta^{(l)}} L(y, \hat{y}) \quad \forall l$$

Seems simple enough, why are packages like PyTorch, Tensorflow, Theano, Cafe, MxNet synonymous with deep learning?

1. Automatic differentiation

2. Convenient libraries

Gradient Descent:

Seems simple enough.
Tensorflow, Theano,
learning?

1. Automatic differentiation

2. Convenient library

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 3x3 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 3)
        self.conv2 = nn.Conv2d(6, 16, 3)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 6 * 6, 120)  # 6*6 from image dimension
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

```
# create your optimizer
optimizer = optim.SGD(net.parameters(), lr=0.01)

# in your training loop:
optimizer.zero_grad()    # zero the gradient buffers
output = net(input)
loss = criterion(output, target)
loss.backward()
optimizer.step()         # Does the update
```

$$\text{Gradient Descent: } \Theta^{(l)} \leftarrow \Theta^{(l)} - \eta \nabla_{\Theta^{(l)}} L(y, \hat{y}) \quad \forall l$$

Seems simple enough, why are packages like PyTorch, Tensorflow, Theano, Cafe, MxNet synonymous with deep learning?

1. Automatic differentiation

2. Convenient libraries

3. GPU support

Common training issues

Training is too slow:

- Use larger step sizes, develop step size reduction schedule
- Change batch size
- Use momentum and more exotic optimizers (e.g., Adam)
- Apply batch normalization
- Make network larger or smaller (# layers, # filters per layer, etc.)
- Use GPU

Test accuracy is bad

- Try modifying all of the above, plus changing other hyperparameters

Overfitting: it is not atypical to make your network large enough to achieve 100% training accuracy (which is a good stopping condition) even as test accuracy is 20%

Real networks

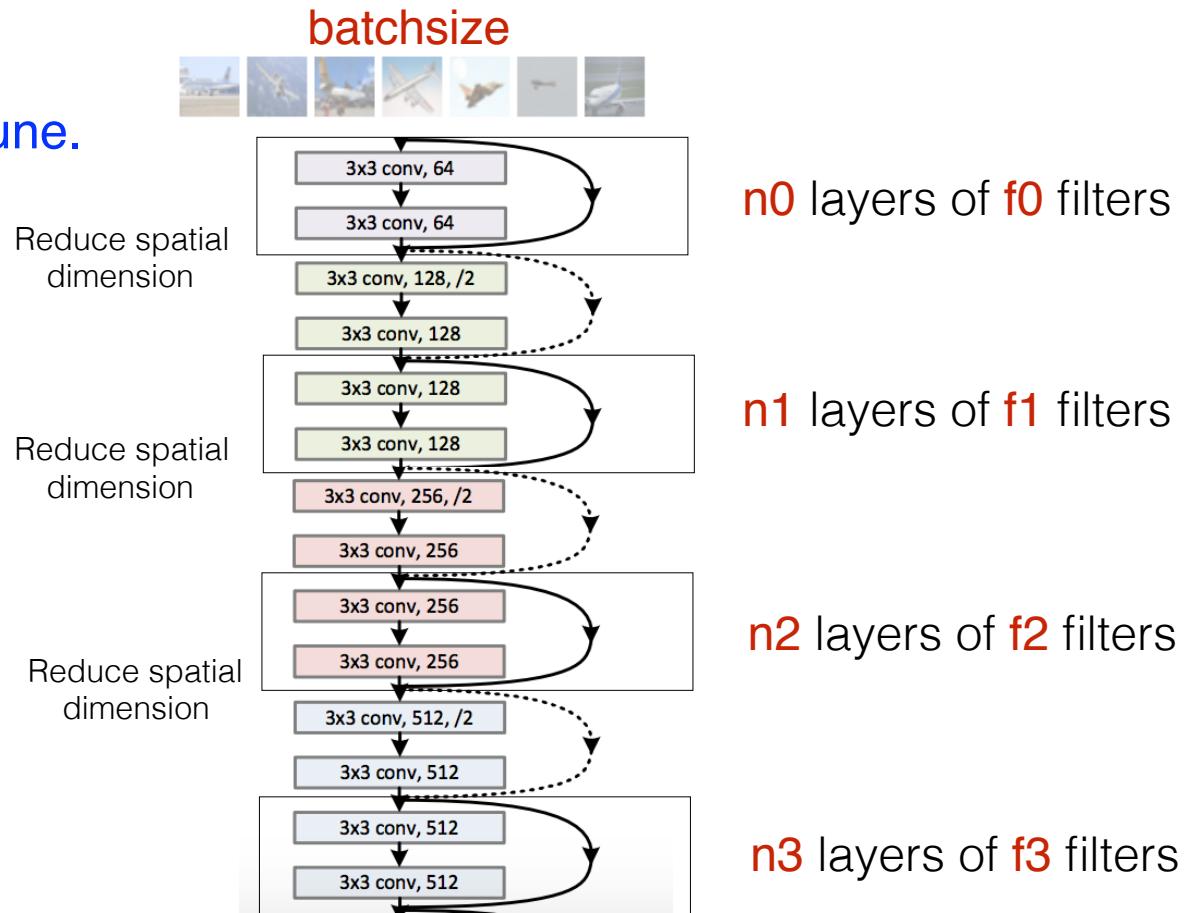
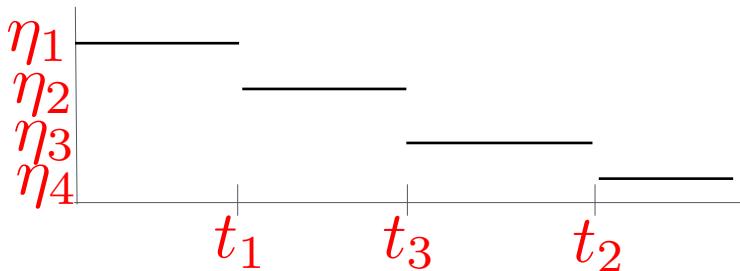
Residual Network of
[HeZhangRenSun'15]

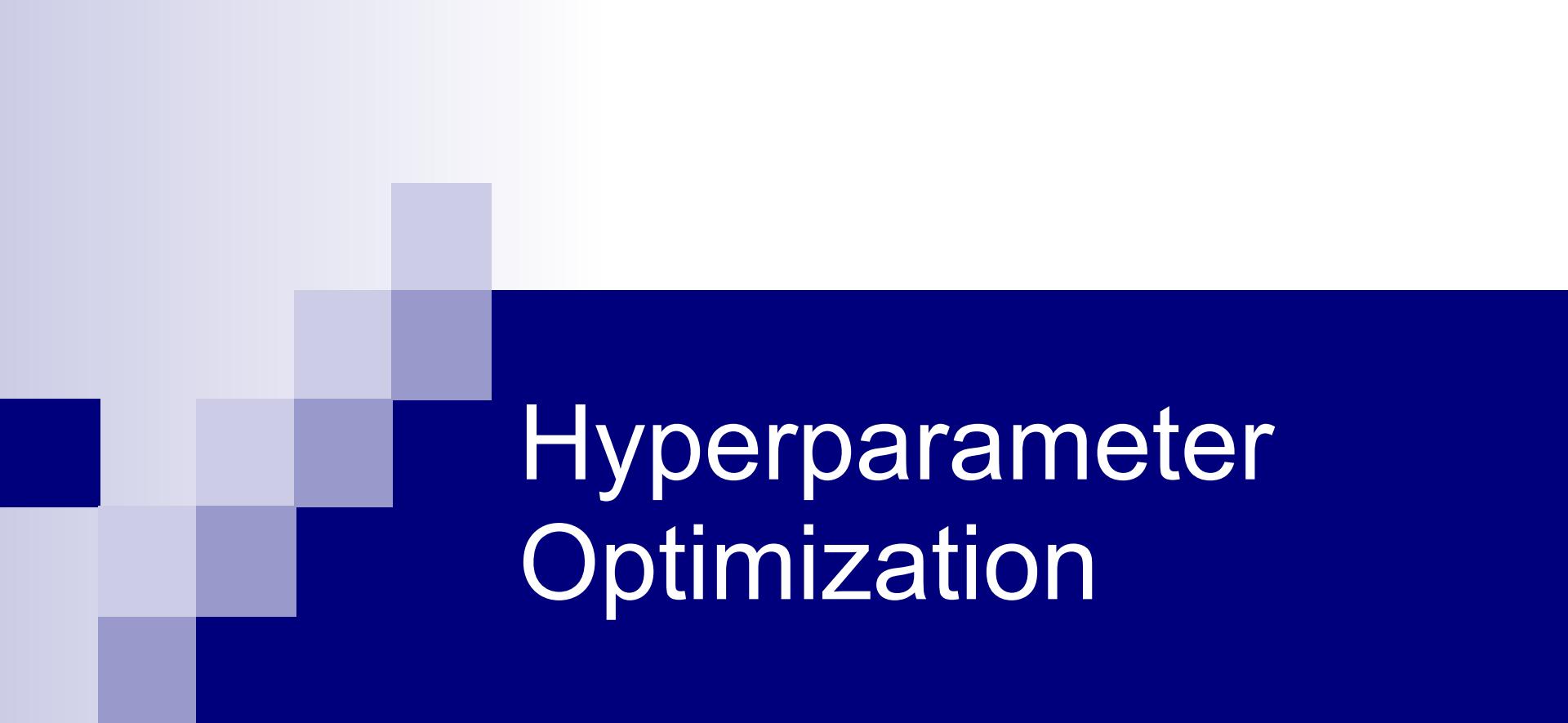
Modern networks have
dozens of parameters to tune.

Data augmentation?
Batch norm?

RELU leakiness
slope

Learning rate schedule





Hyperparameter Optimization

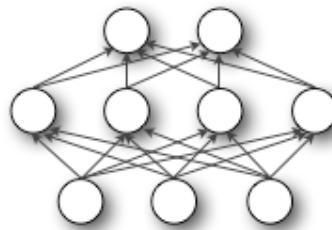
Machine Learning – CSE446
Kevin Jamieson
University of Washington

May 29, 2019

00000000000000000000
111111111111111111
222222222222222222
333333333333333333
444444444444444444
555555555555555555
666666666666666666
777777777777777777
888888888888888888
999999999999999999

0 0 0 0 0 0 0 0 0 0 0 0
 1 1 1 1 1 1 1 1 1 1 1 1
 2 2 2 2 2 2 2 2 2 2 2 2
 3 3 3 3 3 3 3 3 3 3 3 3
 4 4 4 4 4 4 4 4 4 4 4 4
 5 5 5 5 5 5 5 5 5 5 5 5
 6 6 6 6 6 6 6 6 6 6 6 6
 7 7 7 7 7 7 7 7 7 7 7 7
 8 8 8 8 8 8 8 8 8 8 8 8
 9 9 9 9 9 9 9 9 9 9 9 9

Training set



$$N_{out} = 10$$

$$N_{hid}$$

$$N_{in} = 784$$

0 0 0 0 0 0
 1 1 1 1 1 1
 2 2 2 2 2 2
 3 3 3 3 3 3
 6 6 6 6 6 6
 7 7 7 7 7 7
 8 8 8 8 8 8
 9 9 9 9 9 9

Eval set

hyperparameters

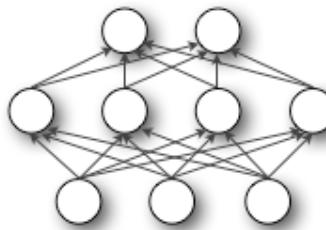
learning rate $\eta \in [10^{-3}, 10^{-1}]$

ℓ_2 -penalty $\lambda \in [10^{-6}, 10^{-1}]$

hidden nodes $N_{hid} \in [10^1, 10^3]$

0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3	3	3	3
4	4	4				4	4	4			
5	5	5				5	5	5			
6	6	6	6	6	6	6	6	6	6	6	6
7	7	7	7	7	7	7	7	7	7	7	7
8	8	8	8	8	8	8	8	8	8	8	8
9	9	9	9	9	9	9	9	9	9	9	9

Training set



$$N_{out} = 10$$

$$N_{hid}$$

$$N_{in} = 784$$

0	0	0	0	0	0
1	1	1	1	1	1
2	2	2	2	2	2
3	3	3	3	3	3
6	6	6	6	6	6
7	7	7	7	7	7
8	8	8	8	8	8
9	9	9	9	9	9

Eval set

Hyperparameters

$$(10^{-1.6}, 10^{-2.4}, 10^{1.7})$$

hyperparameters

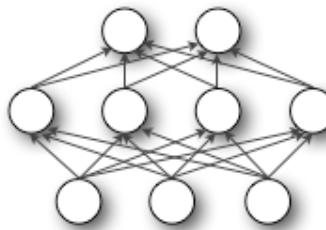
learning rate $\eta \in [10^{-3}, 10^{-1}]$

ℓ_2 -penalty $\lambda \in [10^{-6}, 10^{-1}]$

hidden nodes $N_{hid} \in [10^1, 10^3]$

0 0 0 0 0 0 0 0 0 0 0 0
 1 1 1 1 1 1 1 1 1 1 1 1
 2 2 2 2 2 2 2 2 2 2 2 2
 3 3 3 3 3 3 3 3 3 3 3 3
 4 4 4 4 4 4 4 4 4 4 4 4
 5 5 5 5 5 5 5 5 5 5 5 5
 6 6 6 6 6 6 6 6 6 6 6 6
 7 7 7 7 7 7 7 7 7 7 7 7
 8 8 8 8 8 8 8 8 8 8 8 8
 9 9 9 9 9 9 9 9 9 9 9 9

Training set



$$N_{out} = 10$$

$$N_{hid}$$

$$N_{in} = 784$$

0 0 0 0 0 0
 1 1 1 1 1 1
 2 2 2 2 2 2
 3 3 3 3 3 3
 6 6 6 6 6 6
 7 7 7 7 7 7
 8 8 8 8 8 8
 9 9 9 9 9 9

Eval set

Hyperparameters
 $(10^{-1.6}, 10^{-2.4}, 10^{1.7})$

Eval-loss
0.0577

$$\hat{f}$$

hyperparameters

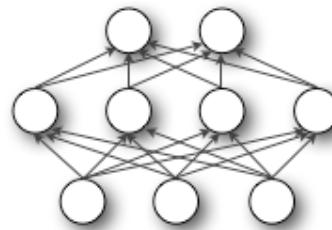
learning rate $\eta \in [10^{-3}, 10^{-1}]$

ℓ_2 -penalty $\lambda \in [10^{-6}, 10^{-1}]$

hidden nodes $N_{hid} \in [10^1, 10^3]$

0 0 0 0 0 0 0 0 0 0 0 0
 1 1 1 1 1 1 1 1 1 1 1 1
 2 2 2 2 2 2 2 2 2 2 2 2
 3 3 3 3 3 3 3 3 3 3 3 3
 4 4 4 4 4 4 4 4 4 4 4 4
 5 5 5 5 5 5 5 5 5 5 5 5
 6 6 6 6 6 6 6 6 6 6 6 6
 7 7 7 7 7 7 7 7 7 7 7 7
 8 8 8 8 8 8 8 8 8 8 8 8
 9 9 9 9 9 9 9 9 9 9 9 9

Training set



$$N_{out} = 10$$

$$N_{hid}$$

$$N_{in} = 784$$

0 0 0 0 0 0
 1 1 1 1 1 1
 2 2 2 2 2 2
 3 3 3 3 3 3
 6 6 6 6 6 6
 7 7 7 7 7 7
 8 8 8 8 8 8
 9 9 9 9 9 9

Eval set

Hyperparameters

$(10^{-1.6}, 10^{-2.4}, 10^{1.7})$	0.0577
$(10^{-1.0}, 10^{-1.2}, 10^{2.6})$	0.182
$(10^{-1.2}, 10^{-5.7}, 10^{1.4})$	0.0436
$(10^{-2.4}, 10^{-2.0}, 10^{2.9})$	0.0919
$(10^{-2.6}, 10^{-2.9}, 10^{1.9})$	0.0575
$(10^{-2.7}, 10^{-2.5}, 10^{2.4})$	0.0765
$(10^{-1.8}, 10^{-1.4}, 10^{2.6})$	0.1196
$(10^{-1.4}, 10^{-2.1}, 10^{1.5})$	0.0834
$(10^{-1.9}, 10^{-5.8}, 10^{2.1})$	0.0242
$(10^{-1.8}, 10^{-5.6}, 10^{1.7})$	0.029

Eval-loss

hyperparameters

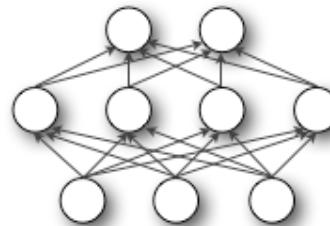
learning rate $\eta \in [10^{-3}, 10^{-1}]$

ℓ_2 -penalty $\lambda \in [10^{-6}, 10^{-1}]$

hidden nodes $N_{hid} \in [10^1, 10^3]$

0 0 0 0 0 0 0 0 0 0 0 0
 1 1 1 1 1 1 1 1 1 1 1 1
 2 2 2 2 2 2 2 2 2 2 2 2
 3 3 3 3 3 3 3 3 3 3 3 3
 4 4 4 4 4 4 4 4 4 4 4 4
 5 5 5 5 5 5 5 5 5 5 5 5
 6 6 6 6 6 6 6 6 6 6 6 6
 7 7 7 7 7 7 7 7 7 7 7 7
 8 8 8 8 8 8 8 8 8 8 8 8
 9 9 9 9 9 9 9 9 9 9 9 9

Training set



$$N_{out} = 10$$

$$N_{hid}$$

$$N_{in} = 784$$

0 0 0 0 0 0
 1 1 1 1 1 1
 2 2 2 2 2 2
 3 3 3 3 3 3
 6 6 6 6 6 6
 7 7 7 7 7 7
 8 8 8 8 8 8
 9 9 9 9 9 9

Eval set

Hyperparameters

- $(10^{-1.6}, 10^{-2.4}, 10^{1.7})$
- $(10^{-1.0}, 10^{-1.2}, 10^{2.6})$
- $(10^{-1.2}, 10^{-5.7}, 10^{1.4})$
- $(10^{-2.4}, 10^{-2.0}, 10^{2.9})$
- $(10^{-2.6}, 10^{-2.9}, 10^{1.9})$
- $(10^{-2.7}, 10^{-2.5}, 10^{2.4})$
- $(10^{-1.8}, 10^{-1.4}, 10^{2.6})$
- $(10^{-1.4}, 10^{-2.1}, 10^{1.5})$
- $(10^{-1.9}, 10^{-5.8}, 10^{2.1})$
- $(10^{-1.8}, 10^{-5.6}, 10^{1.7})$

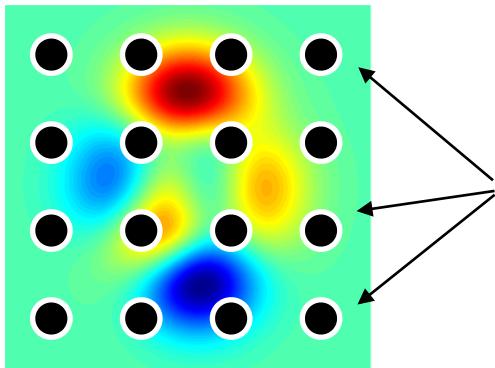
Eval-loss

- 0.0577**
- 0.182**
- 0.0436**
- 0.0919**
- 0.0575**
- 0.0765**
- 0.1196**
- 0.0834**
- 0.0242**
- 0.029**

How do we choose hyperparameters to train and evaluate?

How do we choose hyperparameters to train and evaluate?

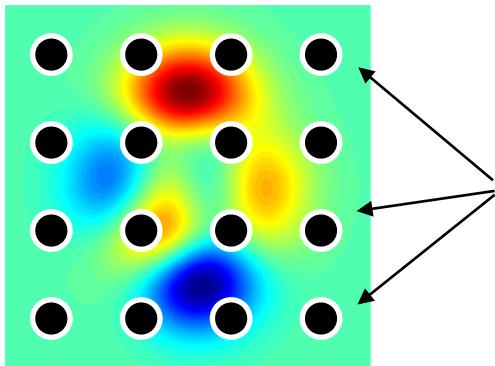
Grid search:



Hyperparameters
on 2d uniform grid

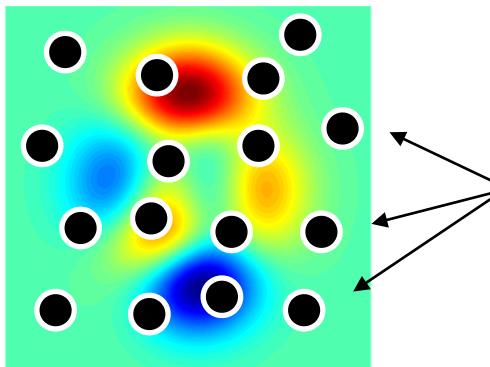
How do we choose hyperparameters to train and evaluate?

Grid search:



Hyperparameters
on 2d uniform grid

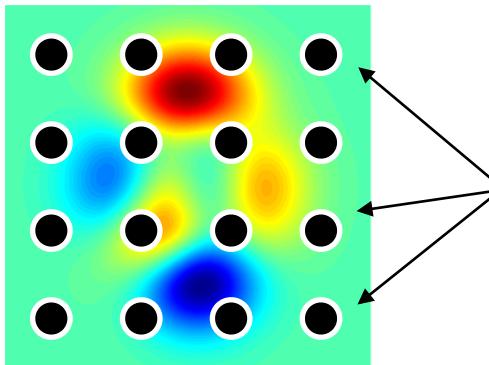
Random search:



Hyperparameters
randomly chosen

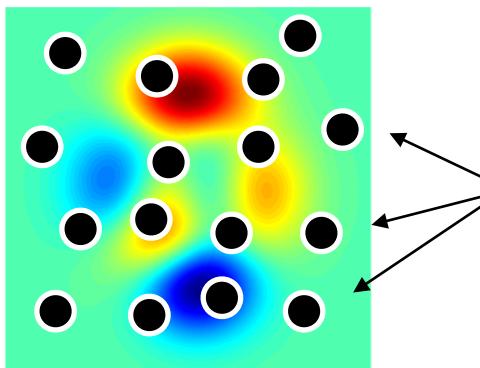
How do we choose hyperparameters to train and evaluate?

Grid search:



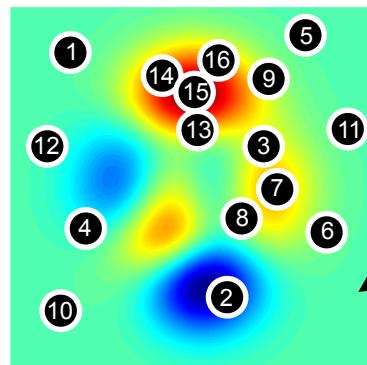
Hyperparameters
on 2d uniform grid

Random search:



Hyperparameters
randomly chosen

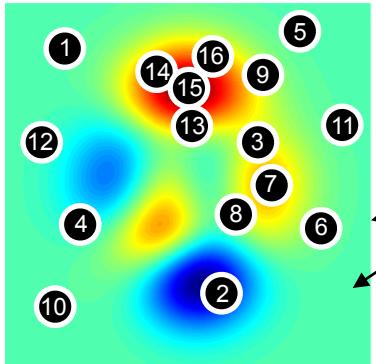
Bayesian Optimization:



Hyperparameters
adaptively chosen

Bayesian Optimization:

How does it work?



Hyperparameters
adaptively chosen

Recent work attempts to speed up hyperparameter evaluation by stopping poor performing settings before they are fully trained.

Kevin Swersky, Jasper Snoek, and Ryan Prescott Adams. Freeze-thaw bayesian optimization. arXiv:1406.3896, 2014.

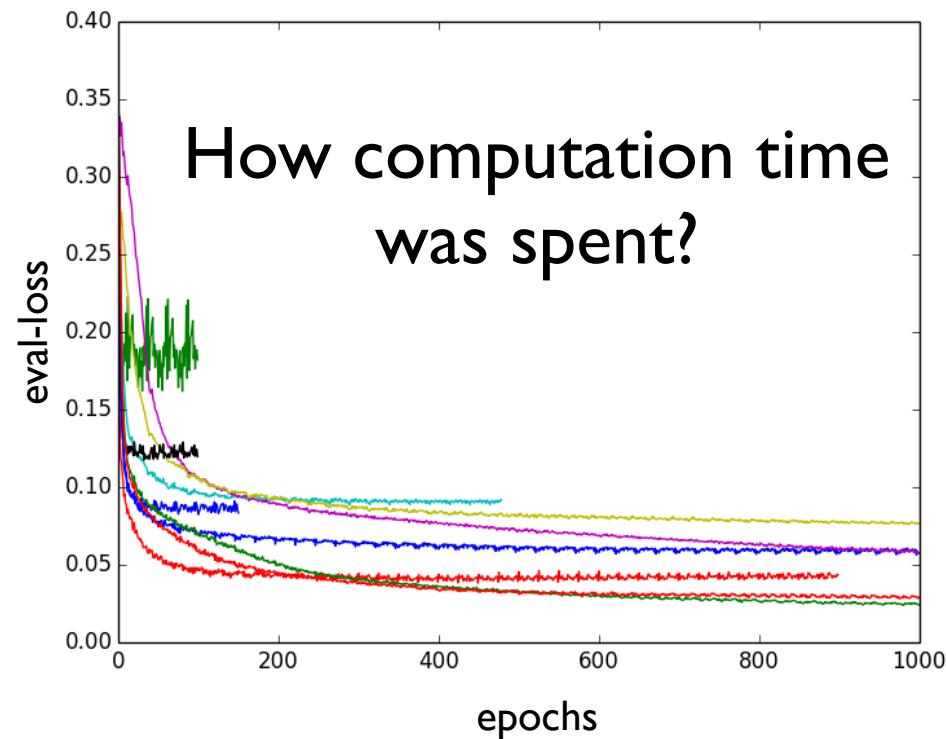
Alekh Agarwal, Peter Bartlett, and John Duchi. Oracle inequalities for computationally adaptive model selection. COLT, 2012.

Domhan, T., Springenberg, J. T., and Hutter, F. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *IJCAI*, 2015.

András György and Levente Kocsis. Efficient multi-start strategies for local search algorithms. JAIR, 41, 2011.

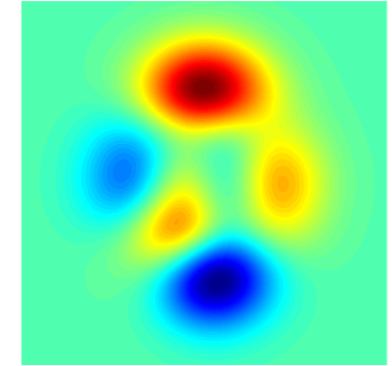
Li, Jamieson, DeSalvo, Rostamizadeh, Talwalkar. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. ICLR 2016.

Hyperparameters	Eval-loss
$(10^{-1.6}, 10^{-2.4}, 10^{1.7})$	0.0577
$(10^{-1.0}, 10^{-1.2}, 10^{2.6})$	0.182
$(10^{-1.2}, 10^{-5.7}, 10^{1.4})$	0.0436
$(10^{-2.4}, 10^{-2.0}, 10^{2.9})$	0.0919
$(10^{-2.6}, 10^{-2.9}, 10^{1.9})$	0.0575
$(10^{-2.7}, 10^{-2.5}, 10^{2.4})$	0.0765
$(10^{-1.8}, 10^{-1.4}, 10^{2.6})$	0.1196
$(10^{-1.4}, 10^{-2.1}, 10^{1.5})$	0.0834
$(10^{-1.9}, 10^{-5.8}, 10^{2.1})$	0.0242
$(10^{-1.8}, 10^{-5.6}, 10^{1.7})$	0.029



Hyperparameter Optimization

In general, hyperparameter optimization is non-convex optimization and little is known about the underlying function (only observe validation loss)



Your time is valuable, computers are cheap:

Do not employ “grad student descent” for hyper parameter search.
Write modular code that takes parameters as input and automate this embarrassingly parallel search.

Tools for different purposes:

- Very few evaluations: use random search (and pray) or be clever
- Few evaluations and long-running computations: see refs on last slide
- Moderate number of evaluations (but still $\exp(\# \text{params})$) and high accuracy needed: use Bayesian Optimization
- Many evaluations possible: use random search. Why overthink it?



Backprop

Machine Learning – CSE446
Kevin Jamieson
University of Washington

May 29, 2019

Backprop

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g(z^{(2)})$$

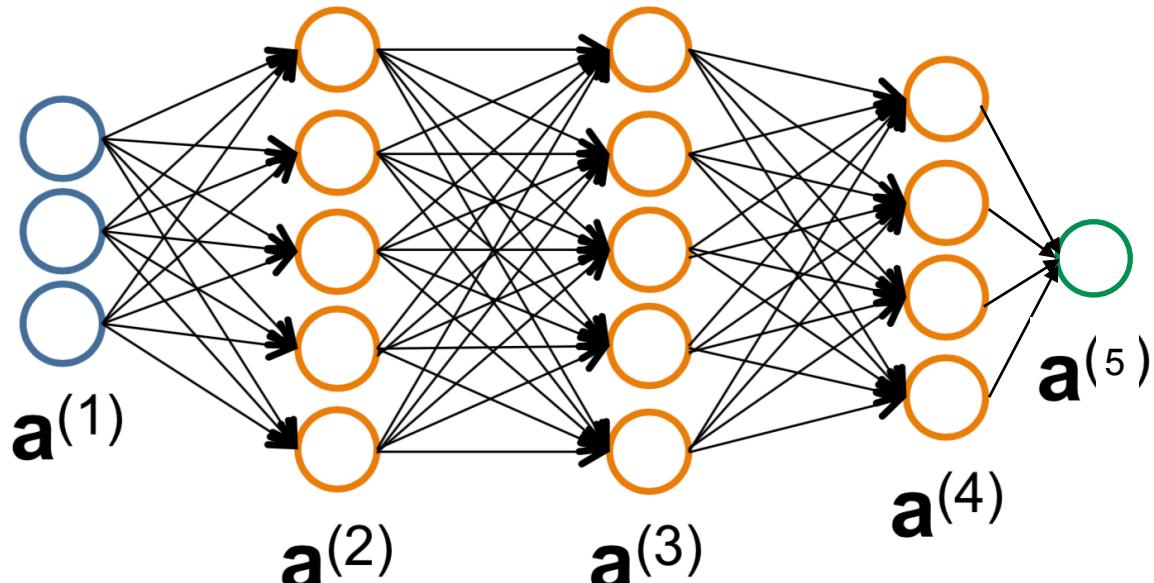
:

$$z^{(l+1)} = \Theta^{(l)} a^{(l)}$$

$$a^{(l+1)} = g(z^{(l+1)})$$

:

$$\hat{y} = a^{(L+1)}$$



$$L(y, \hat{y}) = y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})$$

$$g(z) = \frac{1}{1 + e^{-z}}$$

Backprop

$$\frac{\partial L(y, \hat{y})}{\partial \Theta_{i,j}^{(l)}} = \frac{\partial L(y, \hat{y})}{\partial z_i^{(l+1)}} \cdot \frac{\partial z_i^{(l+1)}}{\partial \Theta_{i,j}^{(l)}} =: \delta_i^{(l+1)} \cdot a_j^{(l)}$$

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g(z^{(2)})$$

⋮

$$z^{(l+1)} = \Theta^{(l)} a^{(l)}$$

$$a^{(l+1)} = g(z^{(l+1)})$$

⋮

$$\hat{y} = a^{(L+1)}$$

$$L(y, \hat{y}) = y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})$$

$$g(z) = \frac{1}{1 + e^{-z}} \quad \delta_i^{(l+1)} = \frac{\partial L(y, \hat{y})}{\partial z_i^{(l+1)}}$$

Backprop

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g(z^{(2)})$$

⋮

$$z^{(l+1)} = \Theta^{(l)} a^{(l)}$$

$$a^{(l+1)} = g(z^{(l+1)})$$

⋮

$$\hat{y} = a^{(L+1)}$$

$$\frac{\partial L(y, \hat{y})}{\partial \Theta_{i,j}^{(l)}} = \frac{\partial L(y, \hat{y})}{\partial z_i^{(l+1)}} \cdot \frac{\partial z_i^{(l+1)}}{\partial \Theta_{i,j}^{(l)}} =: \delta_i^{(l+1)} \cdot a_j^{(l)}$$

$$\begin{aligned} \delta_i^{(l)} &= \frac{\partial L(y, \hat{y})}{\partial z_i^{(l)}} = \sum_k \frac{\partial L(y, \hat{y})}{\partial z_k^{(l+1)}} \cdot \frac{\partial z_k^{(l+1)}}{\partial z_i^{(l)}} \\ &= \sum_k \delta_k^{(l+1)} \cdot \Theta_{k,i} \ g'(z_i^{(l)}) \\ &= a_i^{(l)}(1 - a_i^{(l)}) \sum_k \delta_k^{(l+1)} \cdot \Theta_{k,i} \end{aligned}$$

$$L(y, \hat{y}) = y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})$$

$$g(z) = \frac{1}{1 + e^{-z}} \quad \delta_i^{(l+1)} = \frac{\partial L(y, \hat{y})}{\partial z_i^{(l+1)}}$$

Backprop

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g(z^{(2)})$$

⋮

$$z^{(l+1)} = \Theta^{(l)} a^{(l)}$$

$$a^{(l+1)} = g(z^{(l+1)})$$

⋮

$$\hat{y} = a^{(L+1)}$$

$$\frac{\partial L(y, \hat{y})}{\partial \Theta_{i,j}^{(l)}} = \frac{\partial L(y, \hat{y})}{\partial z_i^{(l+1)}} \cdot \frac{\partial z_i^{(l+1)}}{\partial \Theta_{i,j}^{(l)}} =: \delta_i^{(l+1)} \cdot a_j^{(l)}$$

$$\delta_i^{(l)} = a_i^{(l)}(1 - a_i^{(l)}) \sum_k \delta_k^{(l+1)} \cdot \Theta_{k,i}$$

$$L(y, \hat{y}) = y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})$$

$$g(z) = \frac{1}{1 + e^{-z}} \quad \delta_i^{(l+1)} = \frac{\partial L(y, \hat{y})}{\partial z_i^{(l+1)}}$$

Backprop

$$\frac{\partial L(y, \hat{y})}{\partial \Theta_{i,j}^{(l)}} = \frac{\partial L(y, \hat{y})}{\partial z_i^{(l+1)}} \cdot \frac{\partial z_i^{(l+1)}}{\partial \Theta_{i,j}^{(l)}} =: \delta_i^{(l+1)} \cdot a_j^{(l)}$$

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g(z^{(2)})$$

⋮

$$z^{(l+1)} = \Theta^{(l)} a^{(l)}$$

$$a^{(l+1)} = g(z^{(l+1)})$$

⋮

$$\hat{y} = a^{(L+1)}$$

$$\delta_i^{(l)} = a_i^{(l)}(1 - a_i^{(l)}) \sum_k \delta_k^{(l+1)} \cdot \Theta_{k,i}$$

$$\begin{aligned} \delta_i^{(L+1)} &= \frac{\partial L(y, \hat{y})}{\partial z_i^{(L+1)}} = \frac{\partial}{\partial z_i^{(L+1)}} [y \log(g(z^{(L+1)})) + (1 - y) \log(1 - g(z^{(L+1)}))] \\ &= \frac{y}{g(z^{(L+1)})} g'(z^{(L+1)}) - \frac{1 - y}{1 - g(z^{(L+1)})} g'(z^{(L+1)}) \\ &= y - g(z^{(L+1)}) = y - a^{(L+1)} \end{aligned}$$

$$L(y, \hat{y}) = y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})$$

$$g(z) = \frac{1}{1 + e^{-z}} \quad \delta_i^{(l+1)} = \frac{\partial L(y, \hat{y})}{\partial z_i^{(l+1)}}$$

Backprop

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g(z^{(2)})$$

⋮

$$z^{(l+1)} = \Theta^{(l)} a^{(l)}$$

$$a^{(l+1)} = g(z^{(l+1)})$$

⋮

$$\hat{y} = a^{(L+1)}$$

$$\frac{\partial L(y, \hat{y})}{\partial \Theta_{i,j}^{(l)}} = \frac{\partial L(y, \hat{y})}{\partial z_i^{(l+1)}} \cdot \frac{\partial z_i^{(l+1)}}{\partial \Theta_{i,j}^{(l)}} =: \delta_i^{(l+1)} \cdot a_j^{(l)}$$

$$\delta_i^{(l)} = a_i^{(l)}(1 - a_i^{(l)}) \sum_k \delta_k^{(l+1)} \cdot \Theta_{k,i}$$

$$\delta^{(L+1)} = y - a^{(L+1)}$$

Recursive Algorithm!

$$L(y, \hat{y}) = y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})$$

$$g(z) = \frac{1}{1 + e^{-z}} \quad \delta_i^{(l+1)} = \frac{\partial L(y, \hat{y})}{\partial z_i^{(l+1)}}$$

Backpropagation

Set $\Delta_{ij}^{(l)} = 0 \quad \forall l, i, j$

(Used to accumulate gradient)

For each training instance (\mathbf{x}_i, y_i) :

Set $\mathbf{a}^{(1)} = \mathbf{x}_i$

Compute $\{\mathbf{a}^{(2)}, \dots, \mathbf{a}^{(L)}\}$ via forward propagation

Compute $\boldsymbol{\delta}^{(L)} = \mathbf{a}^{(L)} - y_i$

Compute errors $\{\boldsymbol{\delta}^{(L-1)}, \dots, \boldsymbol{\delta}^{(2)}\}$

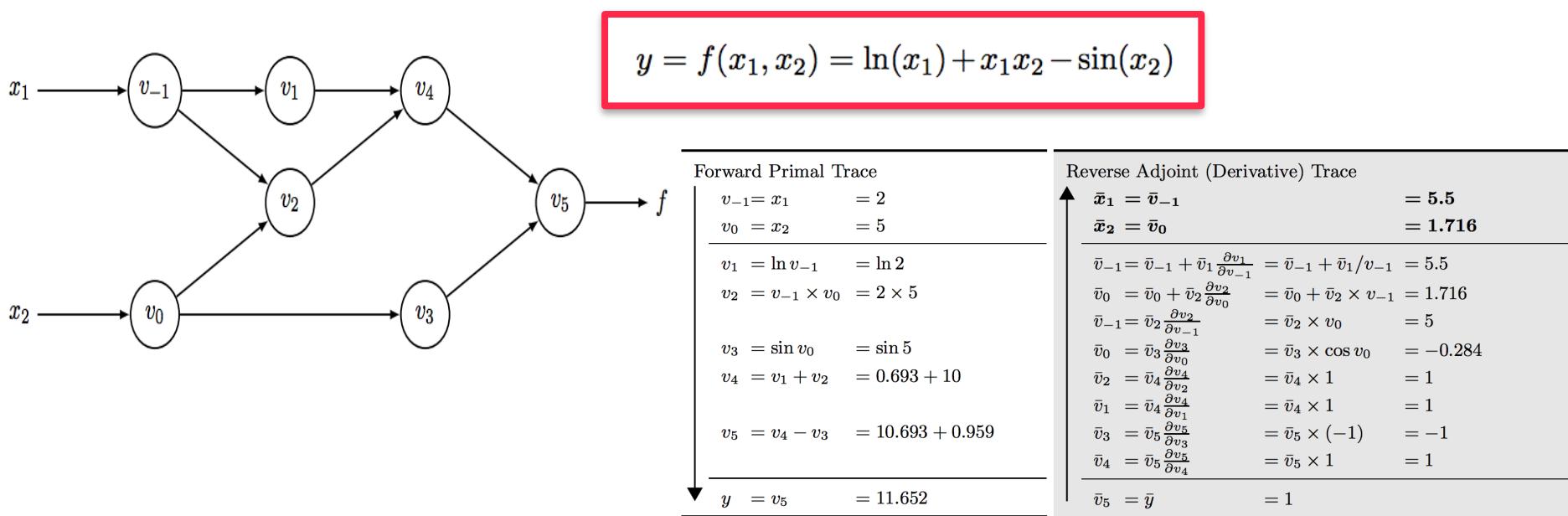
Compute gradients $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

Compute avg regularized gradient $D_{ij}^{(l)} = \begin{cases} \frac{1}{n} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} & \text{if } j \neq 0 \\ \frac{1}{n} \Delta_{ij}^{(l)} & \text{otherwise} \end{cases}$

$D^{(l)}$ is the matrix of partial derivatives of $J(\Theta)$

Autodiff

Backprop for this simple network architecture is a special case of ***reverse-mode auto-differentiation***:



This is the special sauce in Tensorflow, PyTorch, Theano, ...