

So far

Normally, I'd do the later part of this lecture first.

But I want to make sure you know about neural networks before I release the HW.

We've talked about parametric models (for regression and classification) and nonparametric models (kernels and k-NNs)

Now we're going to talk about another, more complex class of parametric models

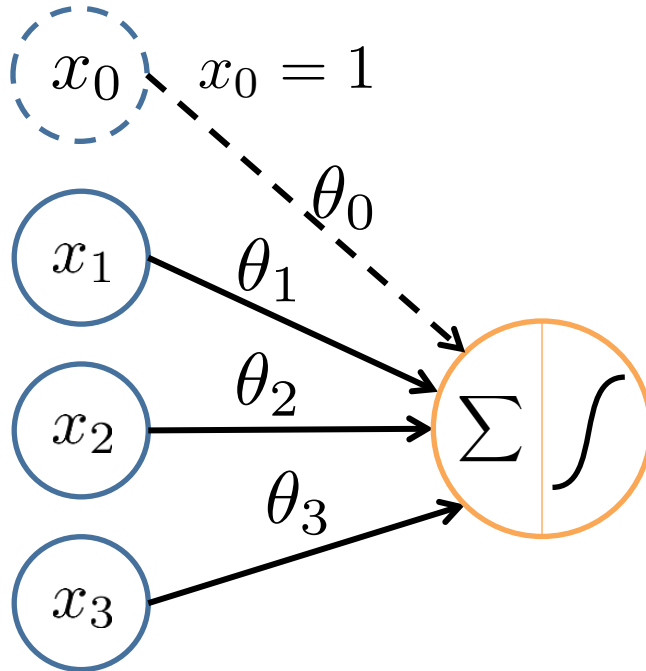
At the end, we'll revisit kernels/some other nonparametric model classes and methods.

Neural Networks



Single Node

“bias unit”



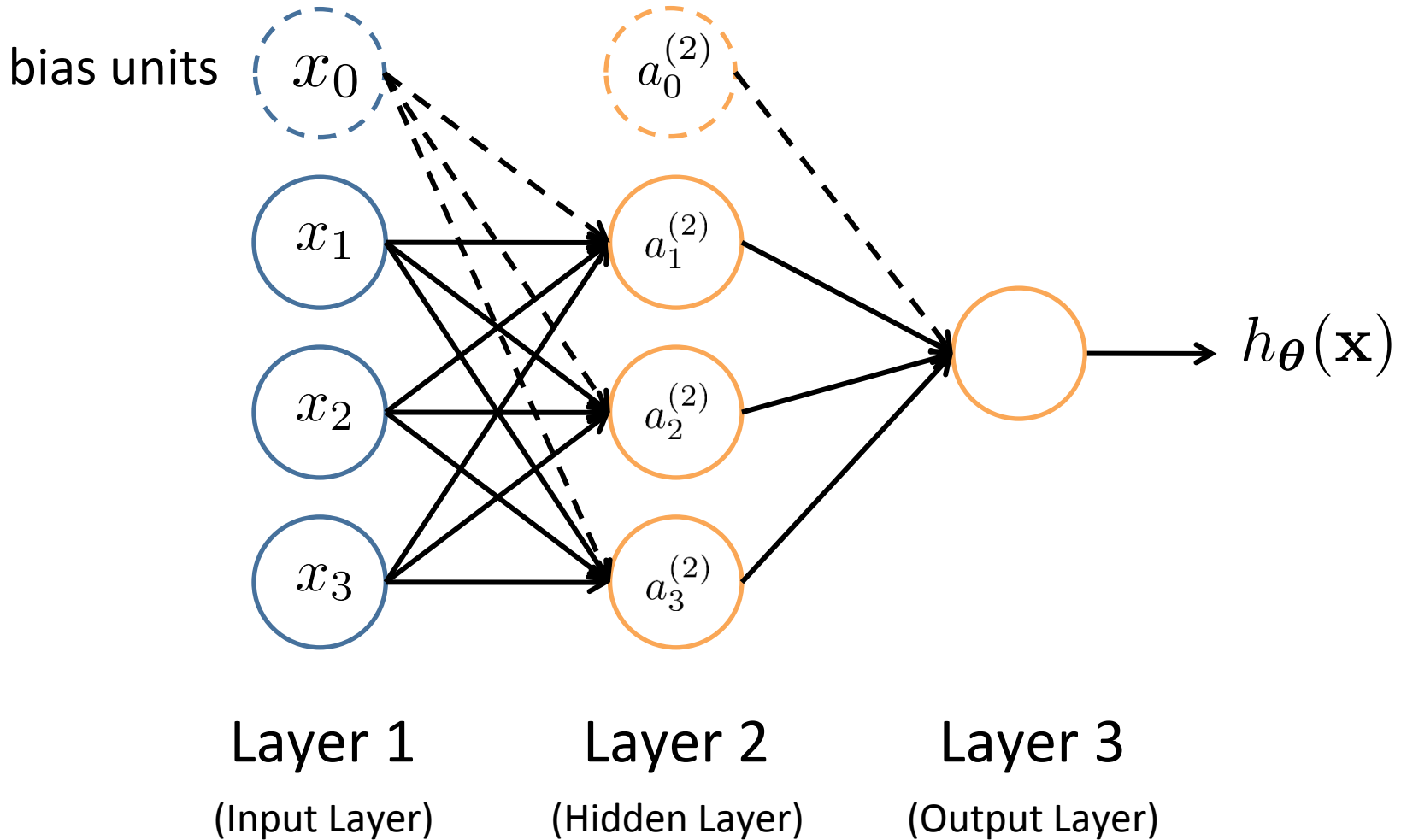
$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$

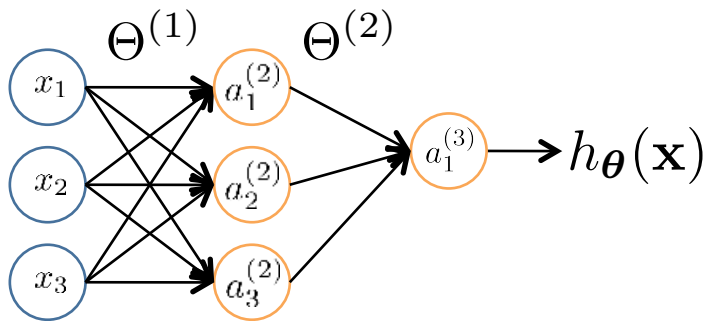
$$h_{\boldsymbol{\theta}}(\mathbf{x}) = g(\boldsymbol{\theta}^T \mathbf{x}) = \frac{1}{1 + e^{-\boldsymbol{\theta}^T \mathbf{x}}}$$

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 $j + 1$

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

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

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

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

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 - Binary Classification

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

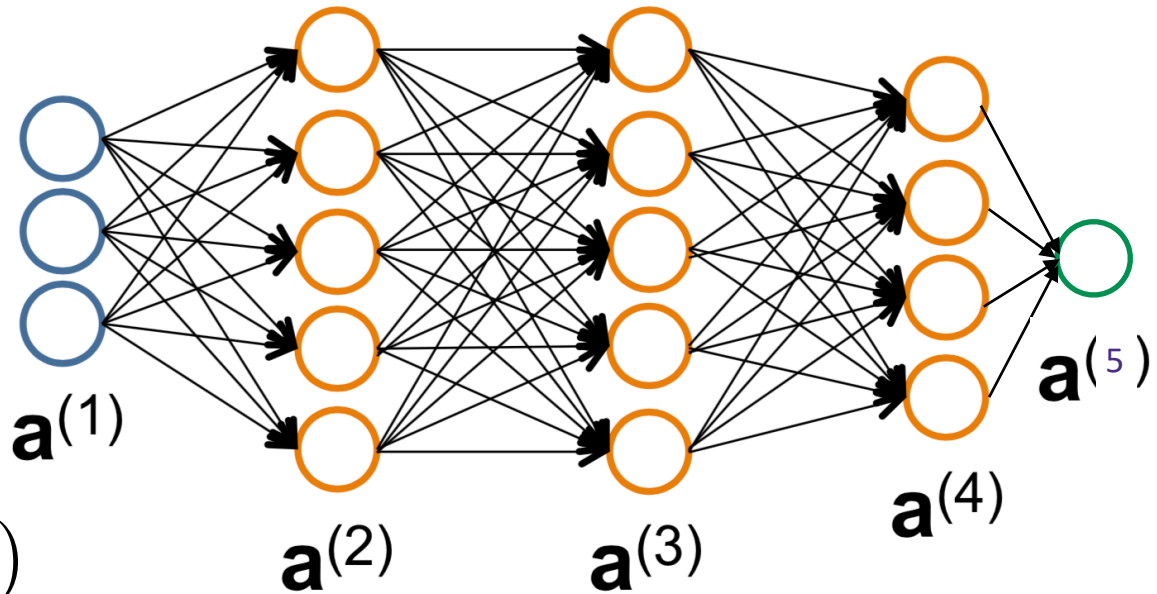
$$a^{(2)} = g(\Theta^{(1)} a^{(1)})$$

⋮

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

⋮

$$\hat{y} = g(\Theta^{(L)} a^{(L)})$$



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

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

Binary
Logistic
Regression

Multi-layer Neural Network - Binary Classification

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

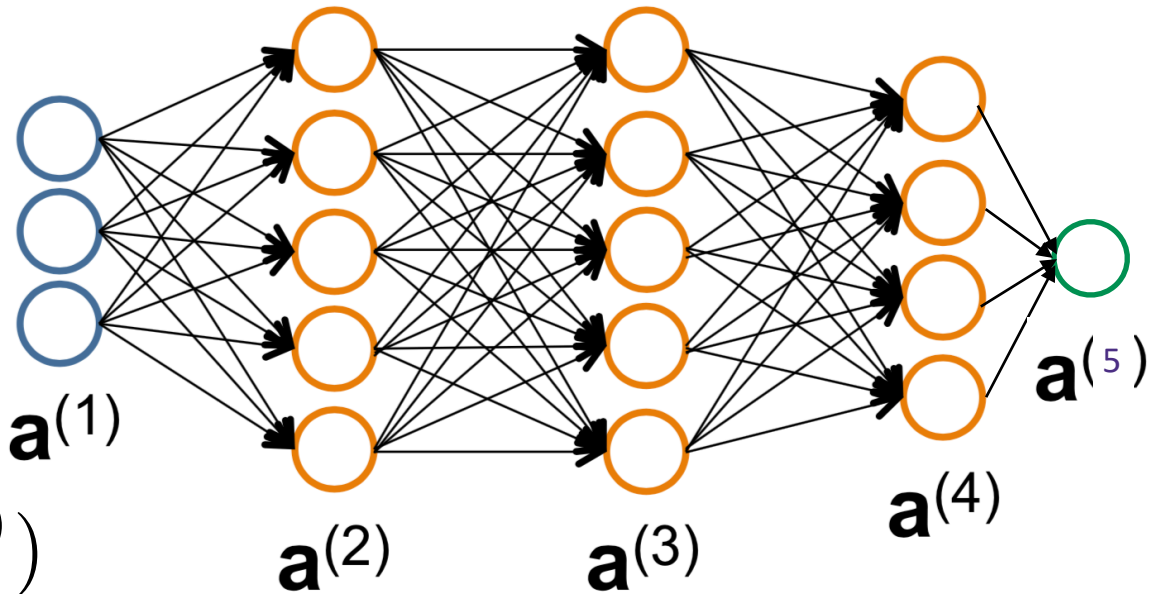
$$a^{(2)} = \sigma(\Theta^{(1)} a^{(1)})$$

⋮

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

⋮

$$\hat{y} = g(\Theta^{(L)} a^{(L)})$$



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

$$\sigma(z) = \max\{0, z\} \quad g(z) = \frac{1}{1 + e^{-z}} \quad \text{Binary Logistic Regression}$$

Multiple Output Units: One-vs-Rest



Pedestrian



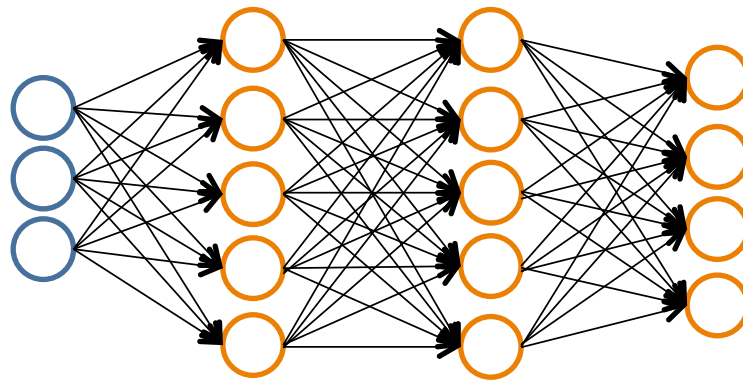
Car



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

Multi-layer Neural Network - Regression

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

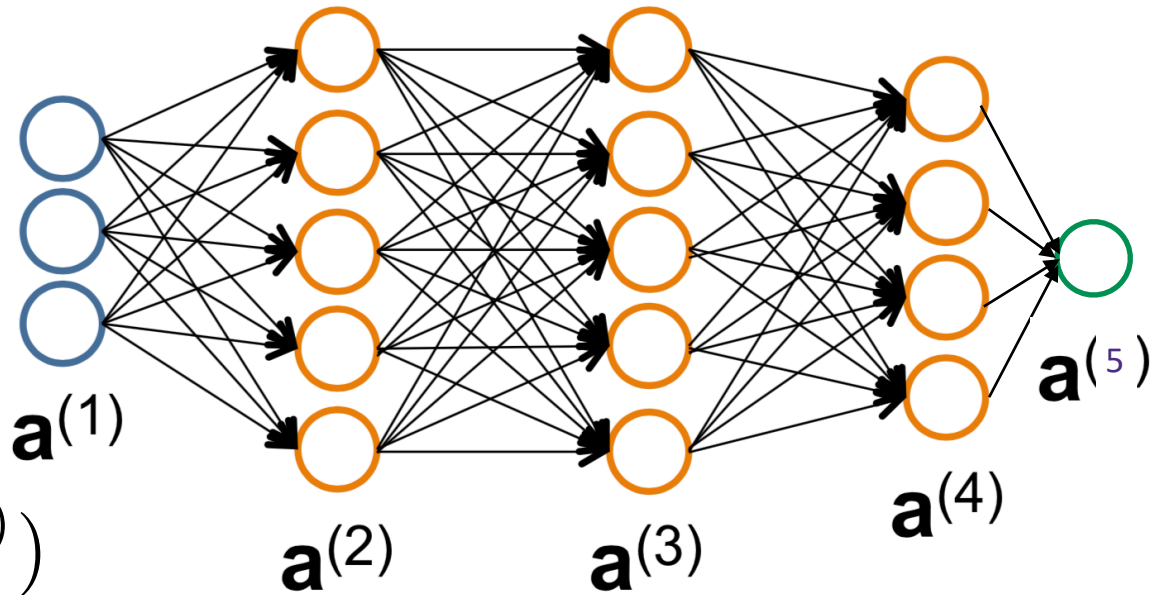
$$a^{(2)} = \sigma(\Theta^{(1)} a^{(1)})$$

⋮

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

⋮

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



$$L(y, \hat{y}) = (y - \hat{y})^2$$

$$\sigma(z) = \max\{0, z\}$$

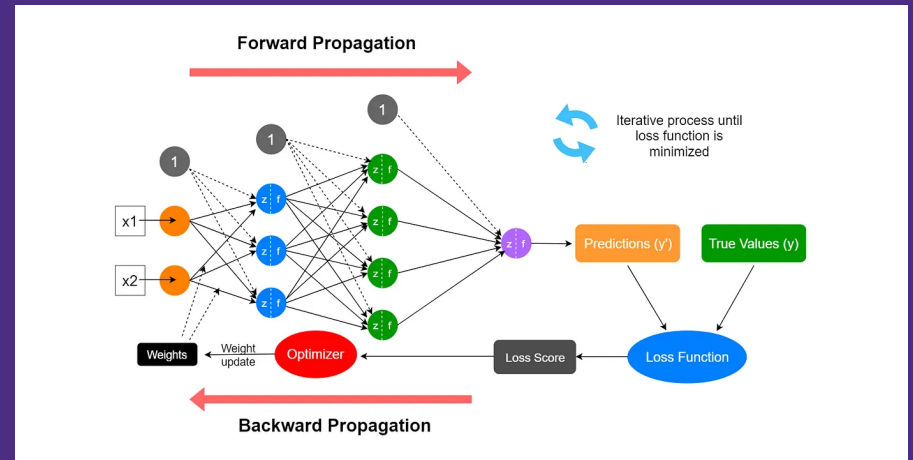
Regression

Neural Networks are arbitrary function approximators

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 \mathbf{x} in the domain of F , $|F(\mathbf{x}) - \hat{F}(\mathbf{x})| < \epsilon$.*

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

Training Neural Networks



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

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

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

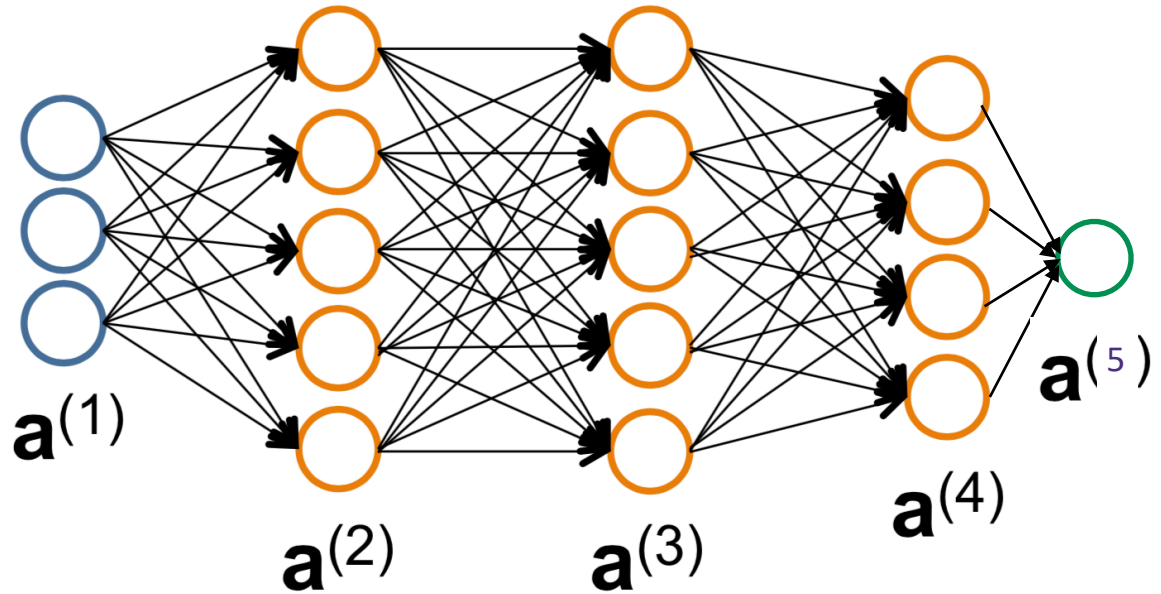
$$\vdots$$

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

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

$$\vdots$$

$$\hat{y} = g(\Theta^{(L)} a^{(L)})$$



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

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

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

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

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,
Theano, Cafe, MxNet s

1. Automatic differ

2. Convenient libra

```
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
```

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

Neural networks are **non-convex**

- For large networks, **gradients** can **blow up** or **go to zero**. This can be helped by **batchnorm** or ResNet architecture
- **Stepsize**, **batchsize**, **momentum** all have large impact on optimizing the training error *and* generalization performance
- Fancier alternatives to SGD (Adagrad, Adam, LAMB, etc.) can significantly improve training
- Overfitting is common and not always undesirable: typical to achieve 100% training accuracy even if test accuracy is just 80%
- Making the network *bigger* may make training *faster!*

Common training issues

Training is too slow:

- Use larger step sizes, develop step size reduction schedule
- Use GPU resources
- 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.)

Test accuracy is low

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

Common training issues

<https://playground.tensorflow.org/>

Backprop



Backprop

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

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

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

$$\vdots$$

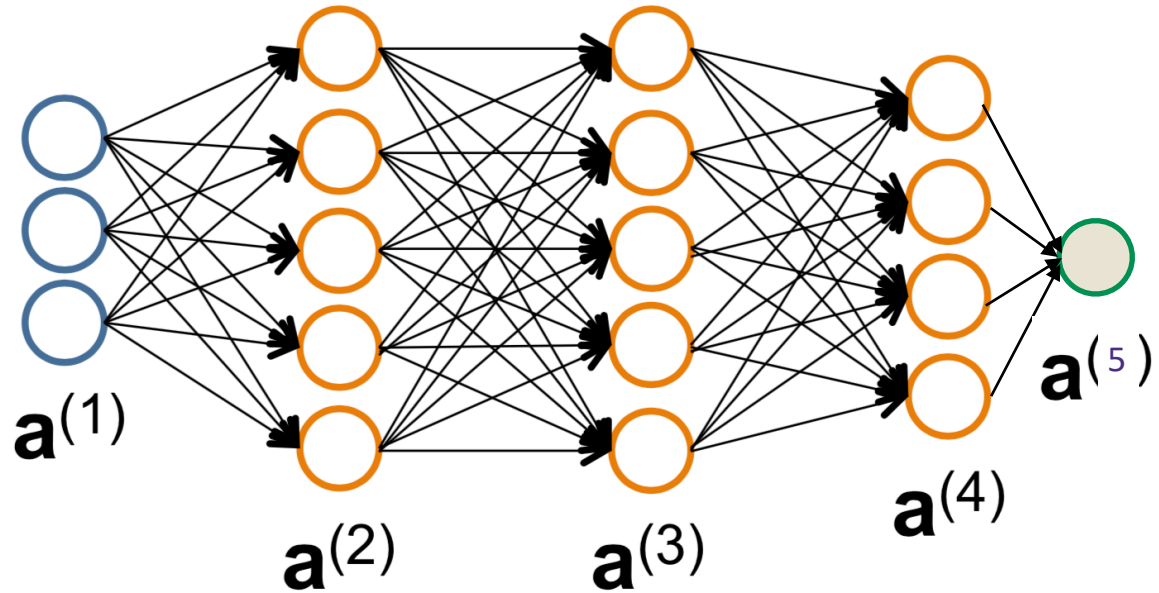
$$a^{(l)} = g(z^{(l)})$$

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

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

$$\vdots$$

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



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

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

Backprop

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

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

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

$$\vdots$$

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

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

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

$$\vdots$$

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

Train by Stochastic Gradient Descent:

$$\Theta_{i,j}^{(l)} \leftarrow \Theta_{i,j}^{(l)} - \eta \frac{\partial L(y, \hat{y})}{\partial \Theta_{i,j}^{(l)}}$$

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

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

Backprop

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

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

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

$$\vdots$$
$$a^{(l)} = g(z^{(l)})$$

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

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

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

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

Train by Stochastic Gradient Descent:

$$\Theta_{i,j}^{(l)} \leftarrow \Theta_{i,j}^{(l)} - \eta \frac{\partial L(y, \hat{y})}{\partial \Theta_{i,j}^{(l)}}$$

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

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

Backprop

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

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

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

⋮

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

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

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

⋮

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

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

$$\delta_i^{(l)} = \frac{\partial L(y, \hat{y})}{\partial z_i^{(l)}} = \sum_k \frac{\partial L(y, \hat{y})}{\partial z_k^{(l+1)}} \cdot \frac{\partial z_k^{(l+1)}}{\partial z_i^{(l)}}$$

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

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

Backprop

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

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

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

⋮

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

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

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

⋮

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

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

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

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

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

Backprop

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

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

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

$$\vdots$$
$$a^{(l)} = g(z^{(l)})$$

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

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

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

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

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

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

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

Backprop

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

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

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

⋮

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

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

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

⋮

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

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

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

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

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

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

Backprop

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

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

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

⋮

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

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

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

⋮

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

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

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

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

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

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

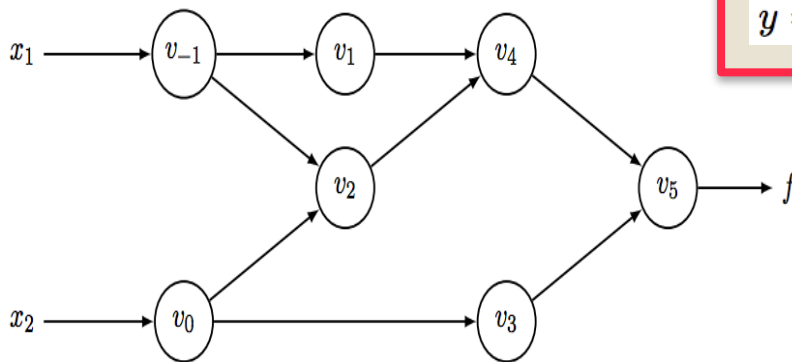
Compute gradients $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

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

$\mathbf{D}^{(l)}$ 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*:



$$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$$

Forward Primal Trace		
$v_{-1} = x_1$	$=$	2
$v_0 = x_2$	$=$	5
<hr/>		
$v_1 = \ln v_{-1}$	$=$	$\ln 2$
$v_2 = v_{-1} \times v_0$	$=$	2×5
<hr/>		
$v_3 = \sin v_0$	$=$	$\sin 5$
$v_4 = v_1 + v_2$	$=$	$0.693 + 10$
<hr/>		
$v_5 = v_4 - v_3$	$=$	$10.693 + 0.959$
<hr/>		
$y = v_5$	$=$	11.652

Reverse Adjoint (Derivative) Trace		
$\bar{x}_1 = \bar{v}_{-1}$	$=$	5.5
$\bar{x}_2 = \bar{v}_0$	$=$	1.716
<hr/>		
$\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}}$	$=$	$\bar{v}_{-1} + \bar{v}_1 / v_{-1} = 5.5$
$\bar{v}_0 = \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0}$	$=$	$\bar{v}_0 + \bar{v}_2 \times v_{-1} = 1.716$
$\bar{v}_{-1} = \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}}$	$=$	$\bar{v}_2 \times v_0 = 5$
$\bar{v}_0 = \bar{v}_3 \frac{\partial v_3}{\partial v_0}$	$=$	$\bar{v}_3 \times \cos v_0 = -0.284$
$\bar{v}_2 = \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	$=$	$\bar{v}_4 \times 1 = 1$
$\bar{v}_1 = \bar{v}_4 \frac{\partial v_4}{\partial v_1}$	$=$	$\bar{v}_4 \times 1 = 1$
$\bar{v}_3 = \bar{v}_5 \frac{\partial v_5}{\partial v_3}$	$=$	$\bar{v}_5 \times (-1) = -1$
$\bar{v}_4 = \bar{v}_5 \frac{\partial v_5}{\partial v_4}$	$=$	$\bar{v}_5 \times 1 = 1$
<hr/>		
$\bar{v}_5 = \bar{y}$	$=$	1

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

Convolutional Neural Network



Multi-layer Neural Network

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

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

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

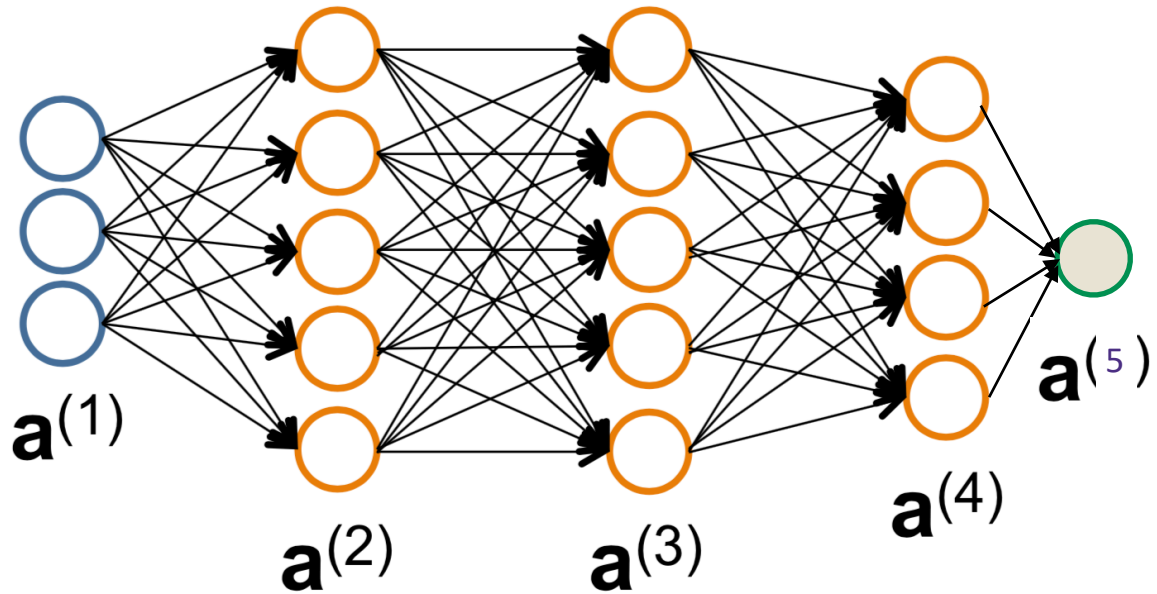
⋮

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

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

⋮

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



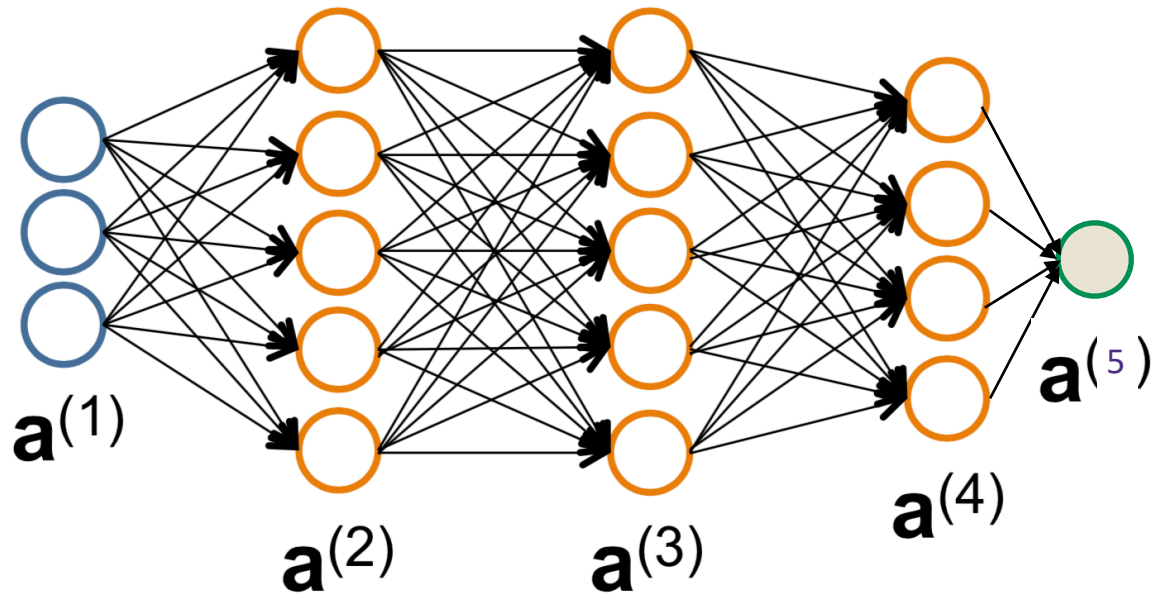
$$L(y, \hat{y}) = y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})$$

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

Binary
Logistic
Regression

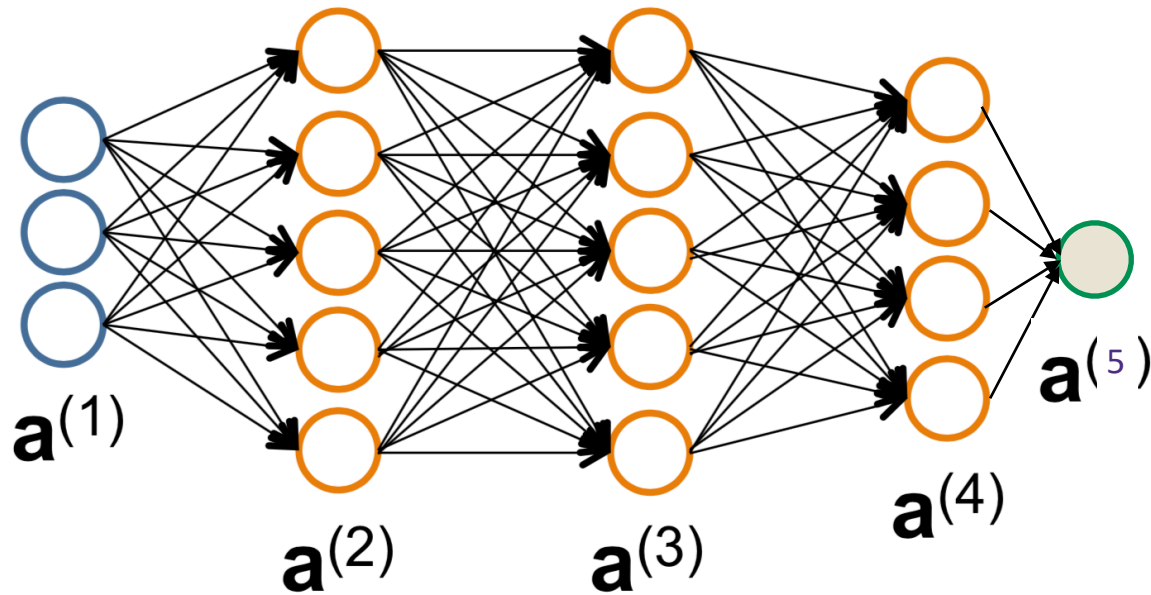
Neural Network Architecture

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



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**.



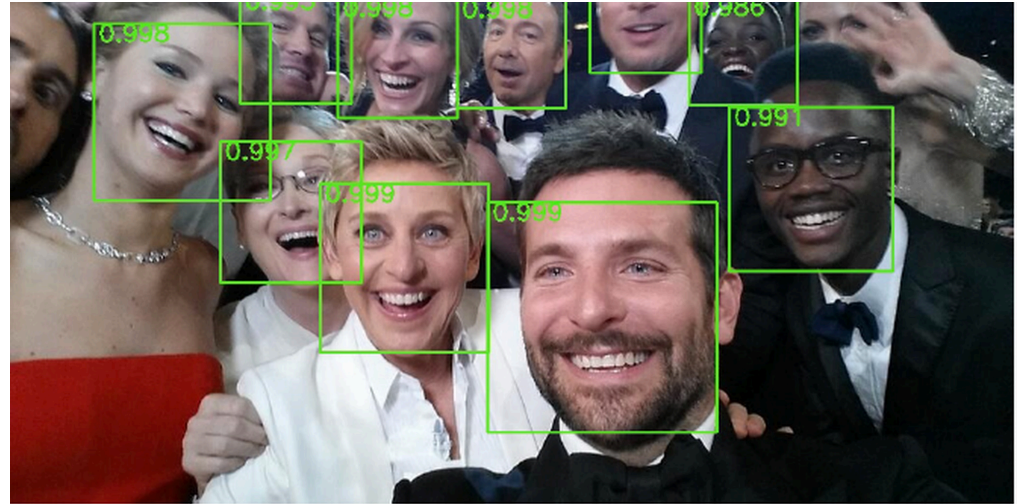
We say a layer is **Fully Connected (FC)** if all linear mappings from the current layer to the next layer are permissible.

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

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

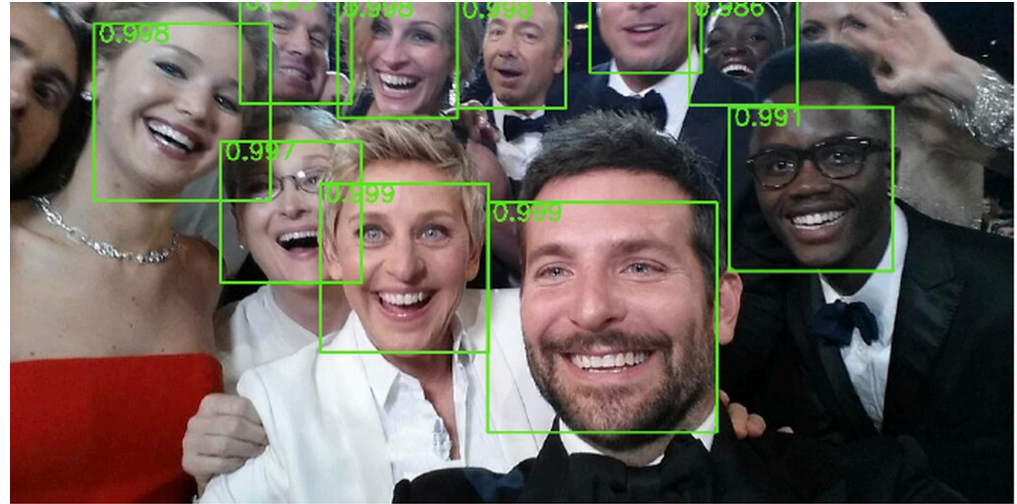
Neural Network Architecture

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.

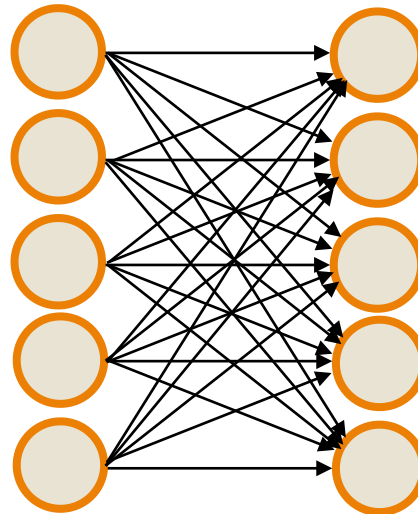


Neural Network Architecture

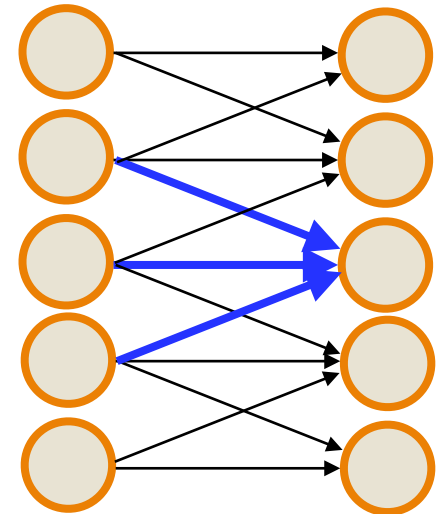
Objects are often **localized in space** so to find the faces in an image, not every pixel is important for classification—makes sense to drag a window across an image.



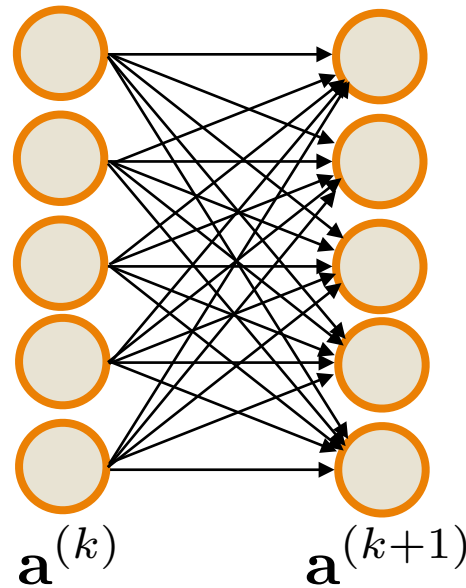
Similarly, to identify edges or other local structure, it makes sense to only look at **local information**



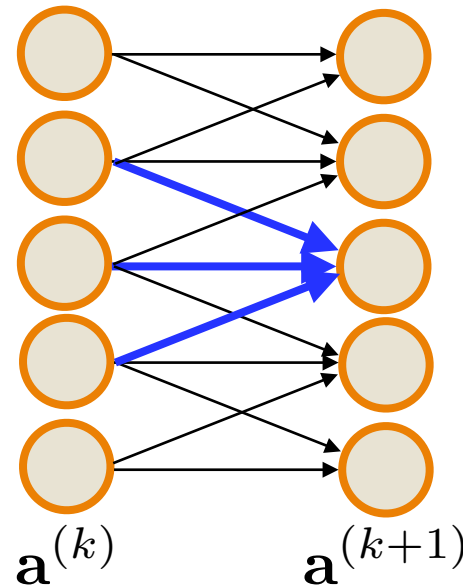
vs.



Neural Network Architecture



vs.



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

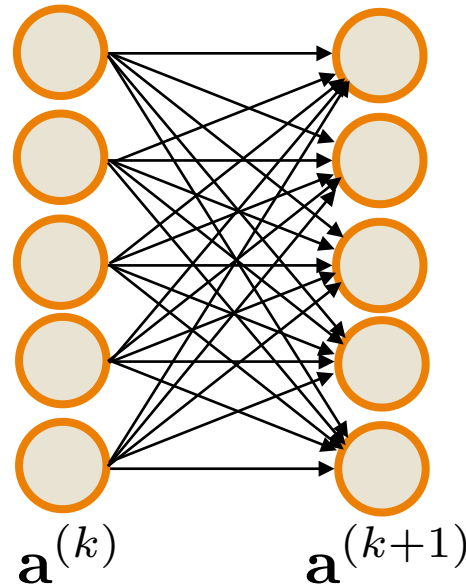
$$\begin{bmatrix} \Theta_{0,0} & \Theta_{0,1} & 0 & 0 & 0 \\ \Theta_{1,0} & \Theta_{1,1} & \Theta_{1,2} & 0 & 0 \\ 0 & \Theta_{2,1} & \Theta_{2,2} & \Theta_{2,3} & 0 \\ 0 & 0 & \Theta_{3,2} & \Theta_{3,3} & \Theta_{3,4} \\ 0 & 0 & 0 & \Theta_{4,3} & \Theta_{4,4} \end{bmatrix}$$

Parameters: n^2

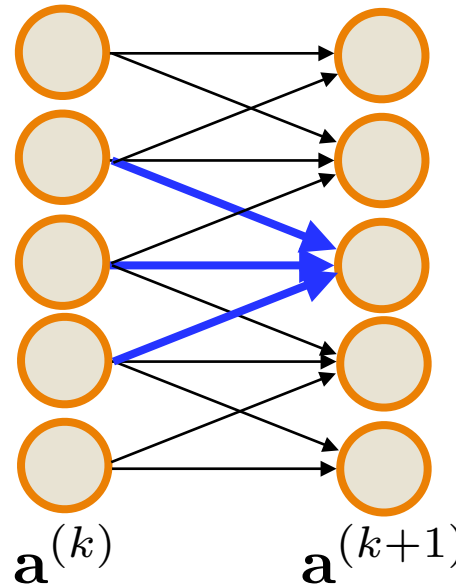
$3n - 2$

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

Neural Network Architecture



vs.



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)

$$\begin{bmatrix} \theta_1 & \theta_2 & 0 & 0 & 0 \\ \theta_0 & \theta_1 & \theta_2 & 0 & 0 \\ 0 & \theta_0 & \theta_1 & \theta_2 & 0 \\ 0 & 0 & \theta_0 & \theta_1 & \theta_2 \\ 0 & 0 & 0 & \theta_0 & \theta_1 \end{bmatrix} \quad m=3$$

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

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

Convolution*

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

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

--	--	--

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 _{x1}	1 _{x0}	1 _{x1}	0	0
-----------------	-----------------	-----------------	---	---

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 _{x1}	1 _{x0}	0 _{x1}	0
---	-----------------	-----------------	-----------------	---

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 _{x1}	0 _{x0}	0 _{x1}
---	---	-----------------	-----------------	-----------------

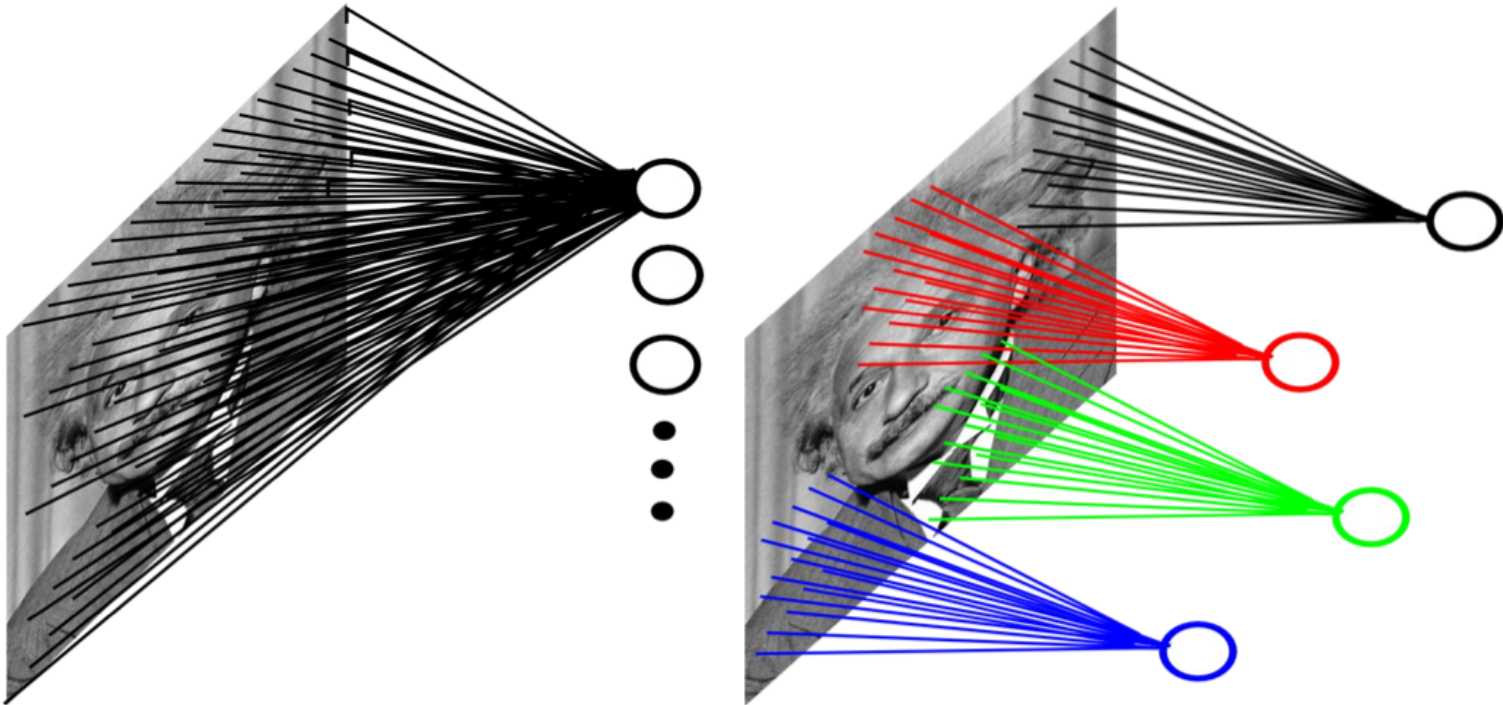
2	1	1
---	---	---

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

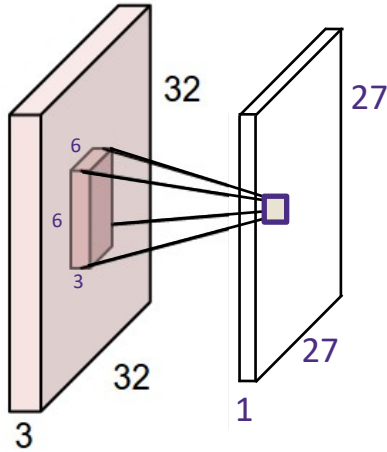
$$(I * K)(i, j) = \sum_m \sum_n I(i + m, j + n)K(m, n)$$

Image I



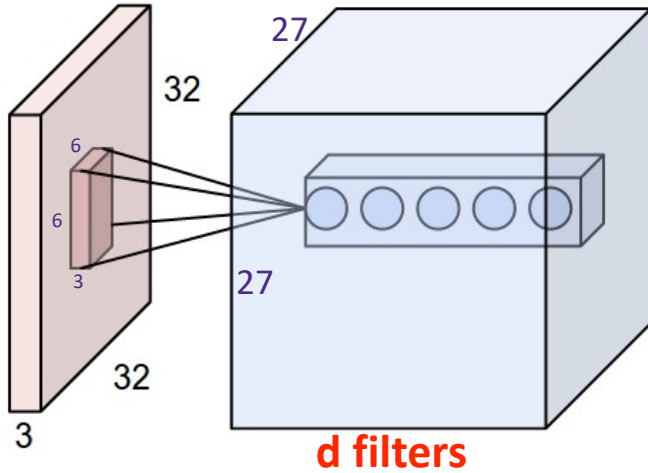
Operation	Filter K	Convolved Image $I * K$
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix}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

Stacking convolved images



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

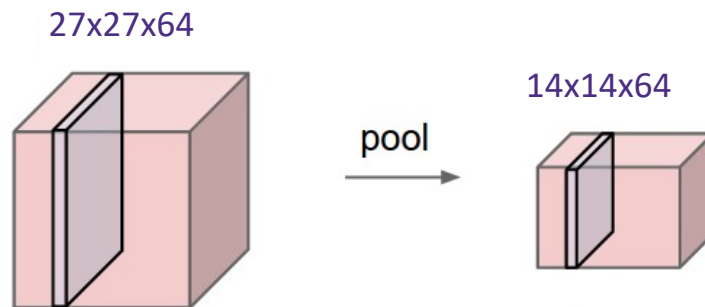
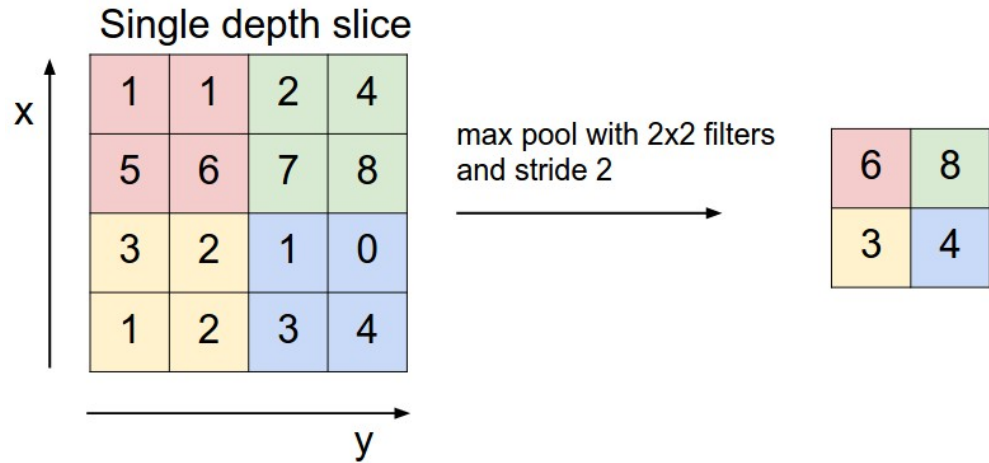
Stacking convolved images



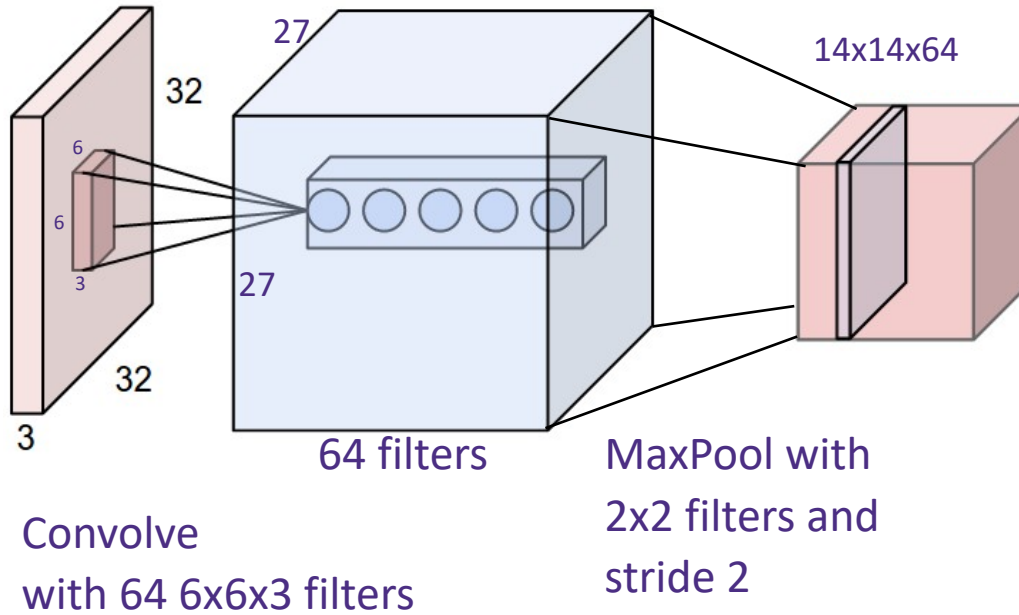
Repeat with d filters!

Pooling

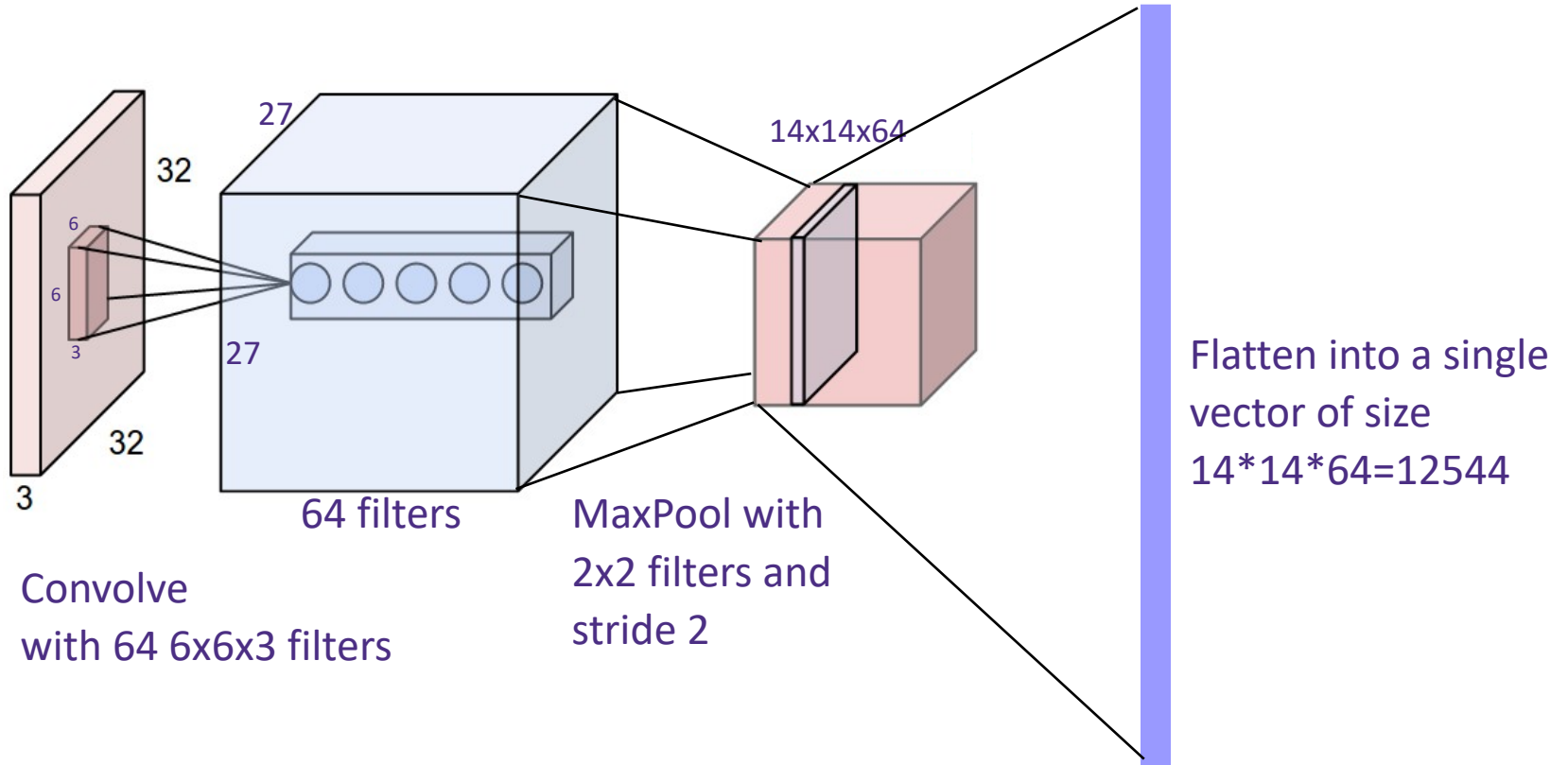
Pooling reduces the dimension and can be interpreted as “This filter had a high response in this general region”



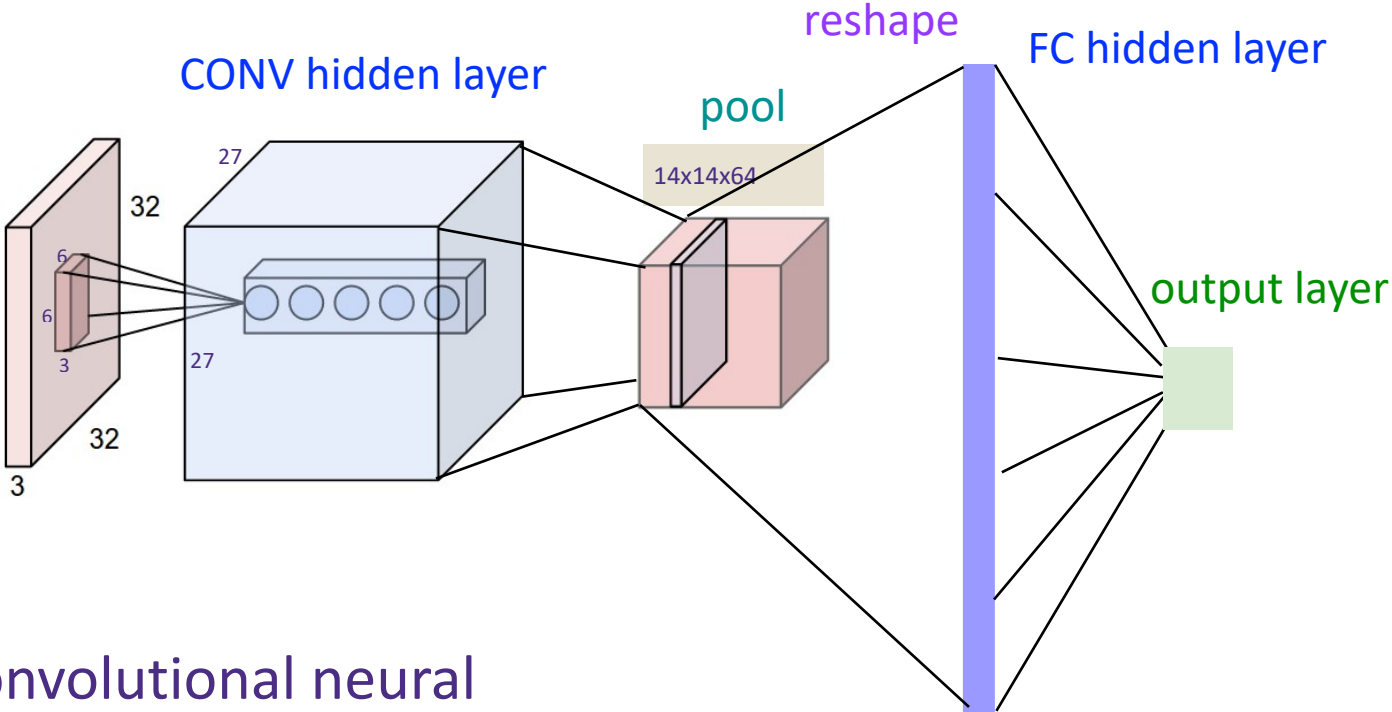
Pooling Convolution layer



Flattening

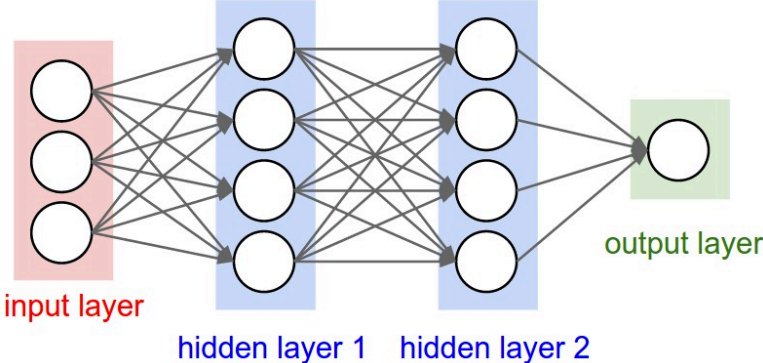


Training Convolutional Networks

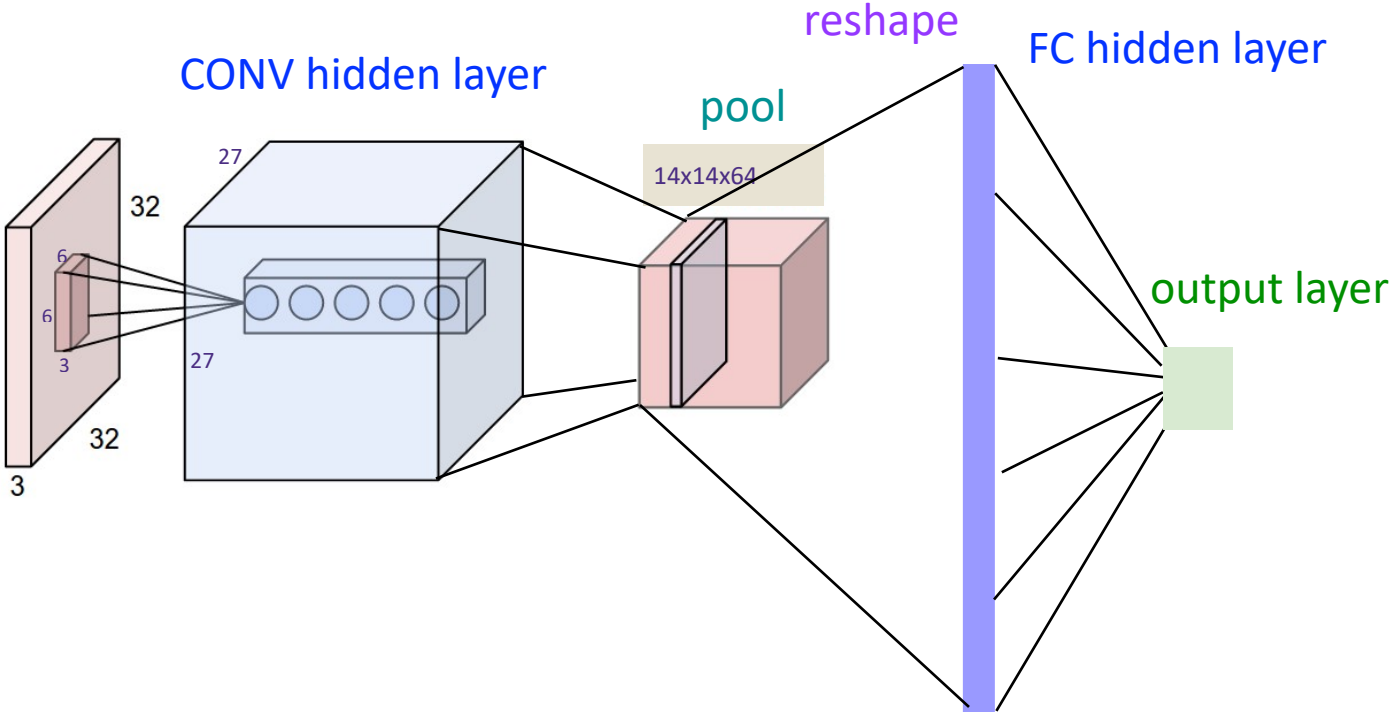


Recall: Convolutional neural networks (CNN) are just regular fully connected (FC) neural networks with some connections removed.

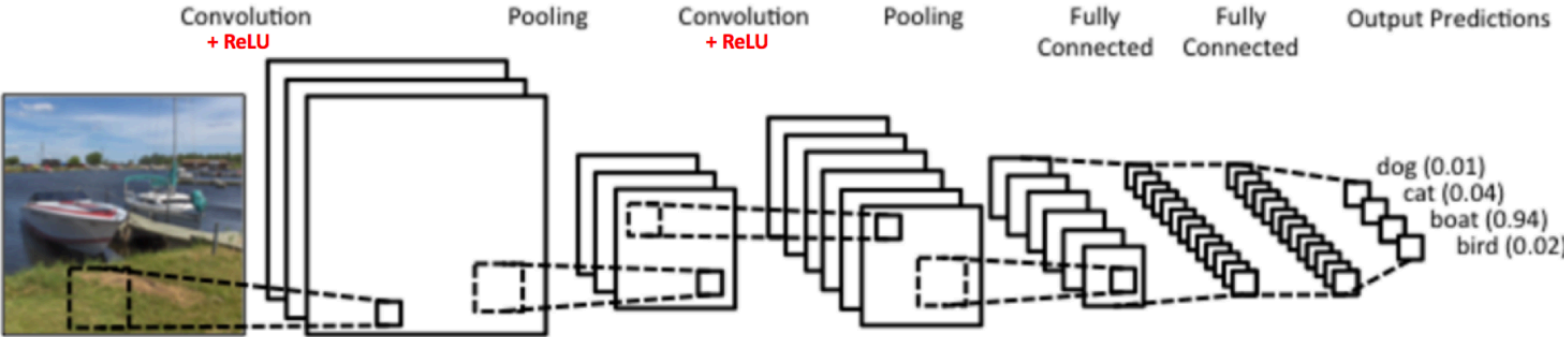
Train with SGD!

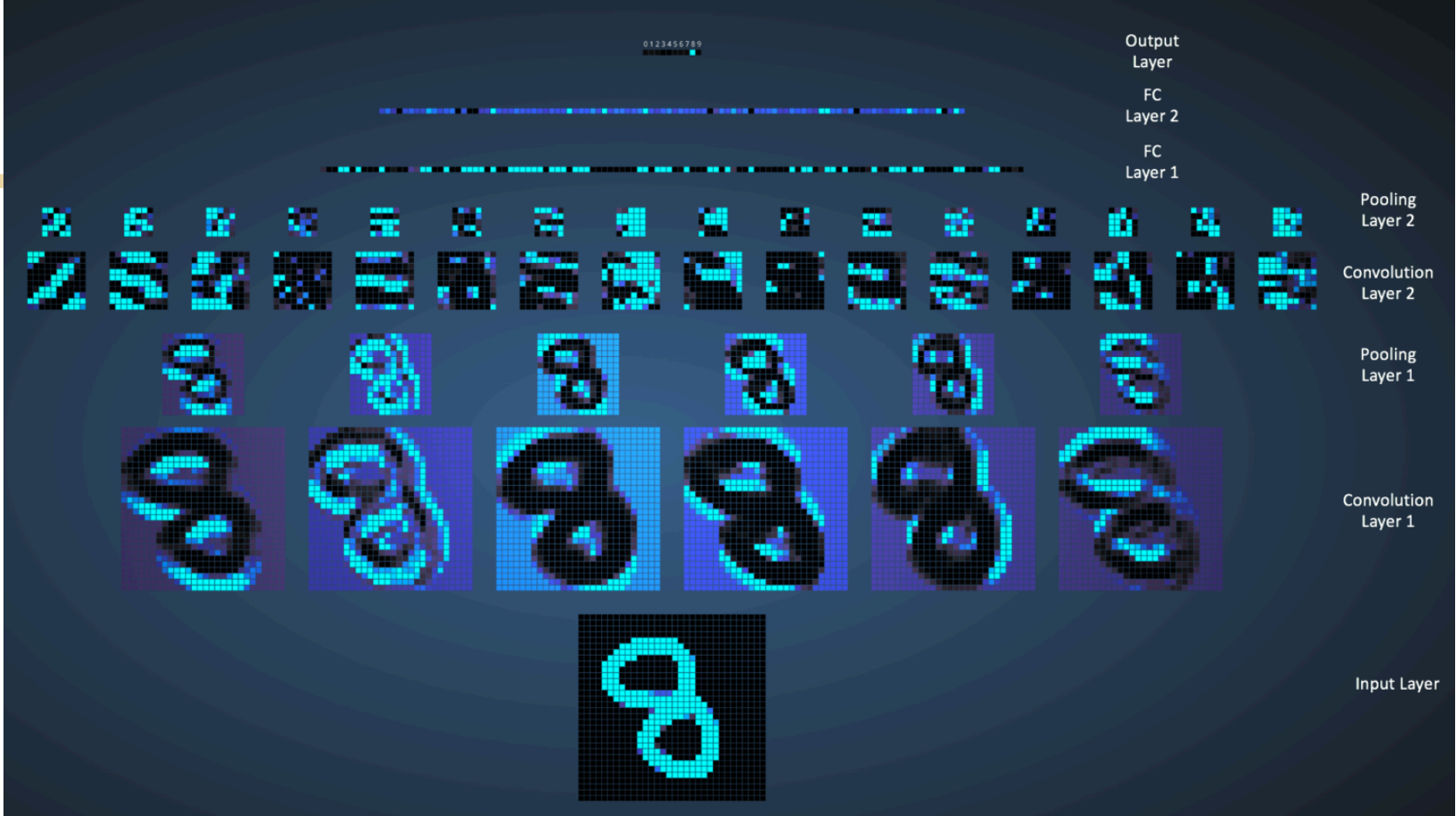


Training Convolutional Networks

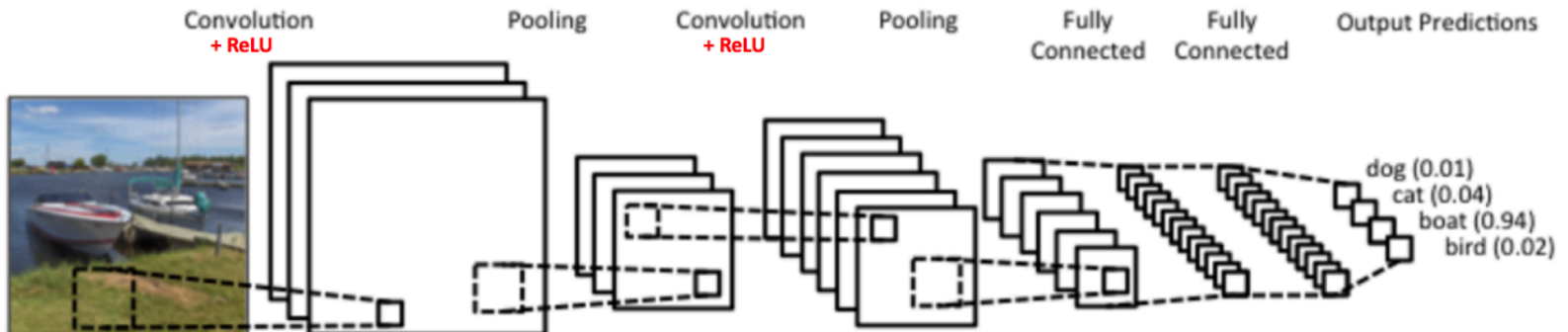


Real example network: LeNet





Real example network: LeNet



Remarks

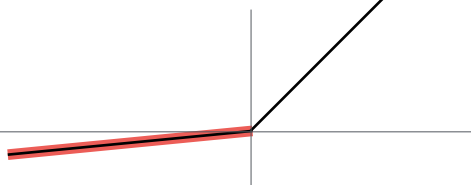
- Convolution is a fundamental operation in signal processing. Instead of hand-engineering the filters (e.g., Fourier, Wavelets, etc.) **Deep Learning *learns* the filters and CONV layers with back-propagation**, replacing fully connected (FC) layers with convolutional (CONV) layers
- **Pooling** is 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

Real networks

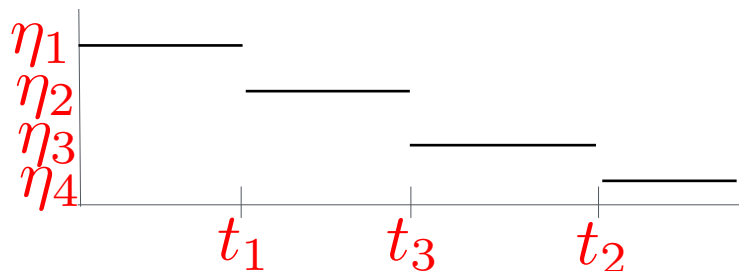
Modern networks have dozens of parameters to tune.

Data augmentation?
Batch norm?

RELU leakiness
slope



Learning rate schedule



What is a kernel?

A function $K : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}$ is a kernel for a map $\phi : \mathbb{R}^d \rightarrow \mathbb{R}^p$ if

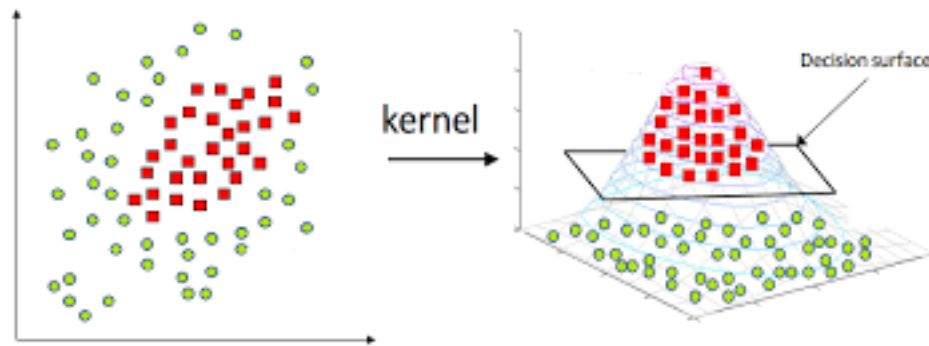
$$K(x, x') = \phi(x) \cdot \phi(x')$$

For all x, x' .

One can think of this as a similarity measurement.

What is the “goal” of using a kernel?

- Allows us to use simple (linear) methods while representing our data in more complicated ways
 - E.g. polynomial expansions of features
- Often, computationally cheaper than explicitly representing the data in those spaces



How do we use a kernel for a given method?

Method-specific

- Prove your model (nonkernelized) can be written as a linear combination of training examples

$$w = \sum_i \alpha_i x_i$$

- Show training algorithms can be written only touching data in terms of dot products

Decision rule is easy: why?

Substitute $K(x_i, x_j)$ for $(x_i^T x_j)$ everywhere

Linear → Kernelized Least Squares

	Linear LS	Kernelized
Solution	$\hat{w} = (X^T X)^{-1} X^T y$	$\hat{w} = \phi^T(X) \cdot \alpha$
Inference	$\hat{y} = \hat{w}x$	$\hat{y} = \phi(x') \cdot \phi^T(X) \cdot \alpha$ $= \sum_i \alpha_i K(x_i, x')$

- **Prove**

$$w = \sum_i \alpha_i x_i$$

- **Show training algorithms can be written only touching data in terms of dot products**
- **Substitute $K(x_i, x_j)$ for $(x_i^T x_j)$ everywhere**

Linear → Kernelized, Ridge Regression

	Ridge Regression	Kernelized
Solution	$\hat{w} = (X^T X + \lambda I)^{-1} X^T y$	$\hat{w} = \phi^T(X) \cdot \alpha$
Inference	$\hat{y} = \hat{w}x$	$\hat{y} = \phi(x') \cdot \phi^T(X) \cdot \alpha$ $= \sum_i \alpha_i K(x_i, x')$

- **Prove**

$$w = \sum_i \alpha_i x_i$$

- **Show training algorithms can be written only touching data in terms of dot products**
- **Substitute $K(x_i, x_j)$ for $(x_i^T x_j)$ everywhere**

Linear → Kernelized SVMs

	Linear LS	Kernelized
Solution	Solution to optimization	$\hat{w} = \phi^T(X) \cdot \alpha$
Inference	$\hat{y} = \text{sign}(\hat{w}x)$	$\hat{y} = \text{sign}(\phi(x') \cdot \phi^T(X))$ $= \text{sign}(\sum_i \alpha_i K(x_i, x'))$

- **Prove**

$$w = \sum_i \alpha_i x_i$$

- **Show training algorithms can be written only touching data in terms of dot products**
- **Substitute $K(x_i, x_j)$ for $(x_i^T x_j)$ everywhere**

Kernel methods

$$f(x) = \sum_{i=1}^n \alpha_i K(x^{(i)}, x)$$

- Can be seen as a soft, learned version of “nearest” neighbors
- $K(x^{(i)}, x) = \phi(x^{(i)})^\top \phi(x)$ defines “similarity” between $x^{(i)}$ and x
- How many parameters?

Trees



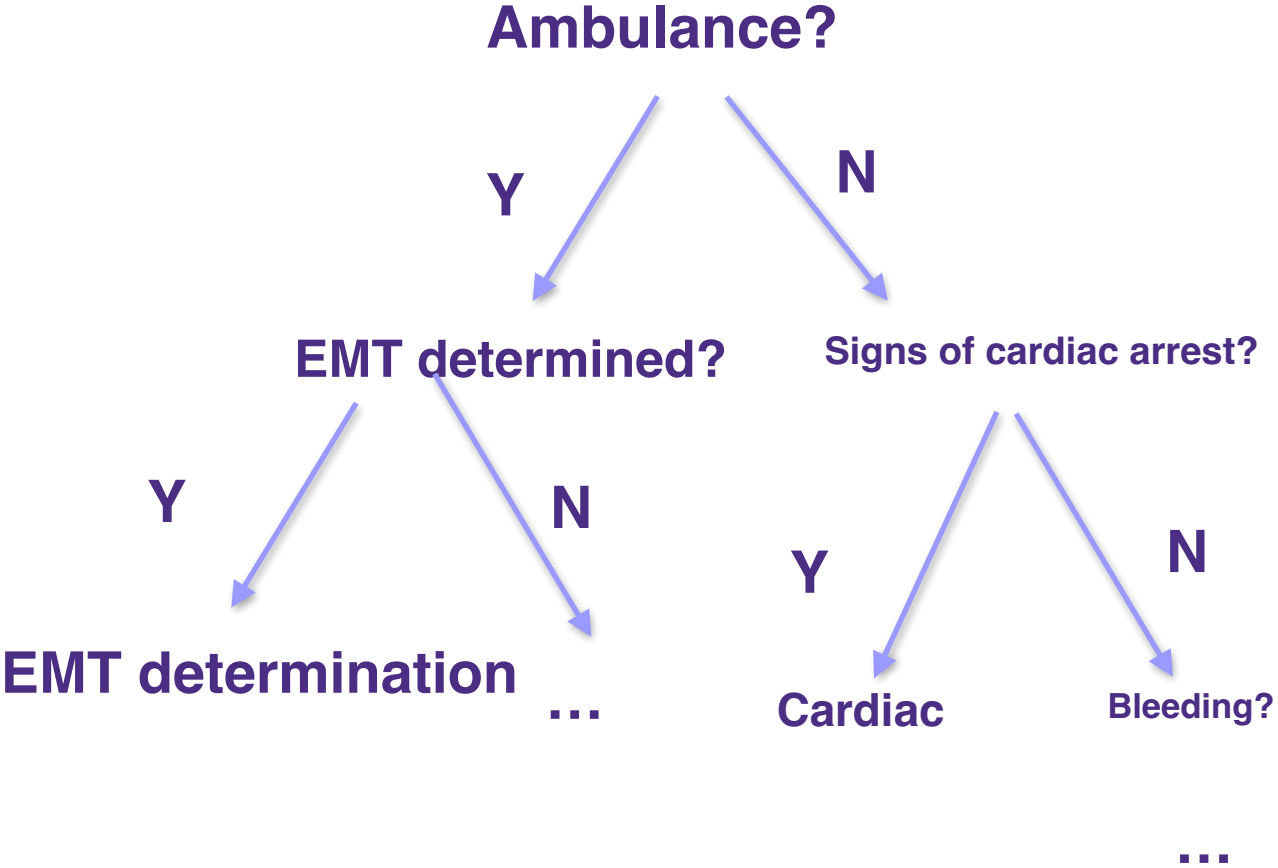
UNIVERSITY *of* WASHINGTON



Decision trees

Want to make a prediction (usually classification)

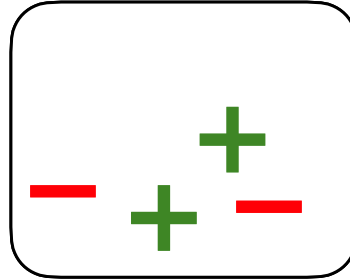
What is the cause of admission to the hospital?



Decision trees: How to build them

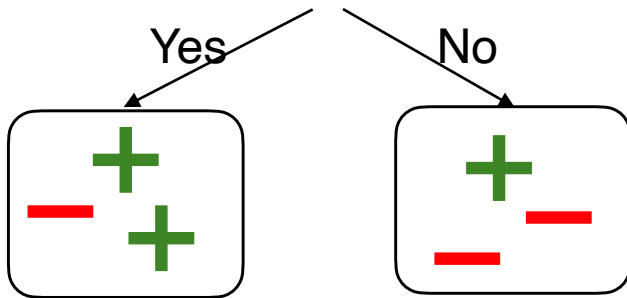
Want to make a prediction (usually classification)

Want to classify spam from not spam using text-based features. What feature gives me the best split?



Full dataset: 50% spam, 50% not
Entropy: 1.0

Split on feature: contains "Limited offer"



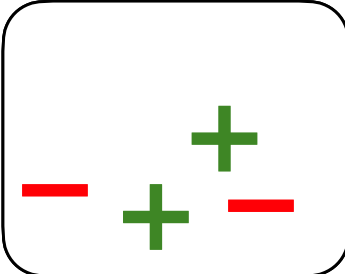
66% spam, 33% not
Entropy: 0.918

33% spam, 66% not
Entropy: 0.918

Decision trees: How to build them

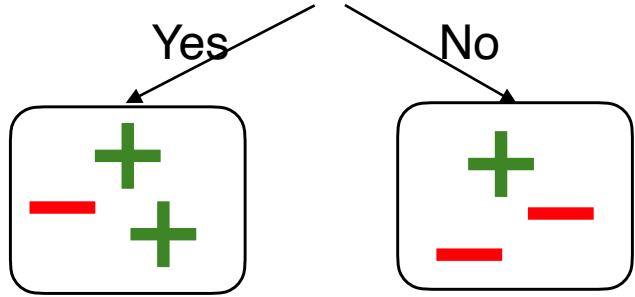
Want to make a prediction (usually classification)

Want to classify spam from not spam using text-based features. What feature gives me the best split?



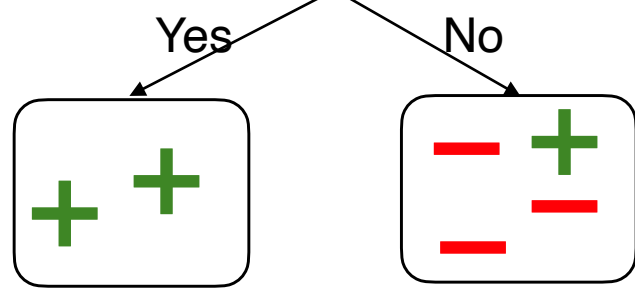
Full dataset: 50% spam, 50% not
Entropy: 1.0

Split on feature: contains "Limited offer"



66% spam, 33% not 33% spam, 66% not
Entropy: 0.918 Entropy: 0.918

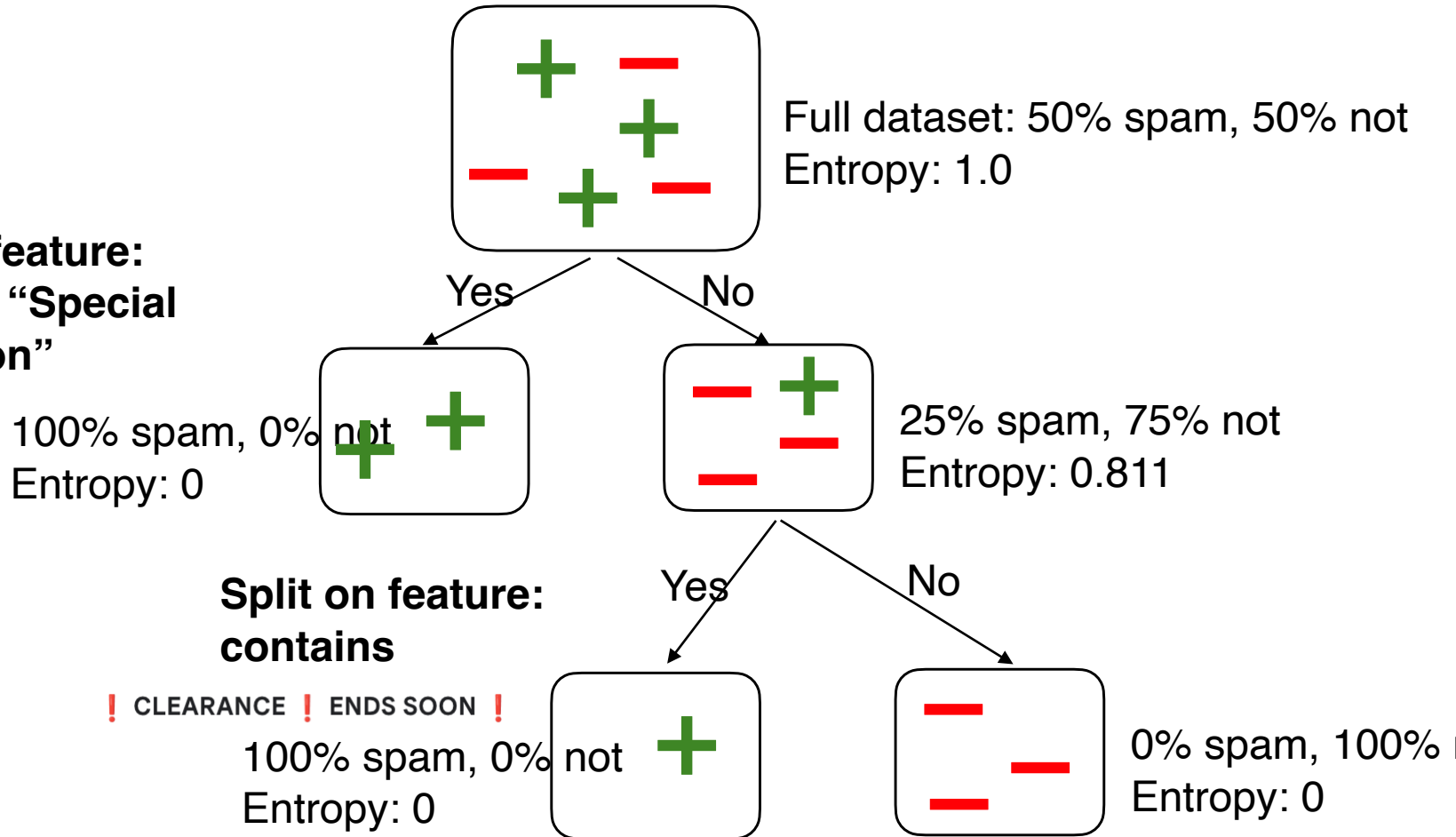
Split on feature: contains "Special P



100% spam, 0% not 25% spam, 75% not
Entropy: 0 Entropy: 0.811

Decision trees: How to build them

**Split on feature:
contains “Special
Promotion”**



Generic Tree-Building algorithm

1. Start from empty decision tree
2. Recursively, for each node:
 - Iterate through all features and compute how good it'd be to split on each feature
 - Split on the “best” feature
3. Prune

Design choices:

- Termination condition (max depth, entropy, train/val error)
- Tree complexity
- Splitting criterion
- Pruning

Generic Tree-Building algorithm

1. Start from empty decision tree
2. Recursively, for each node:
 - Iterate through all features and compute how good it'd be to split on each feature
 - Split on the “best” feature
3. Prune

Design choices:

- Termination condition (max depth, entropy, train/val error)
- Tree complexity
- Splitting criterion
- Pruning

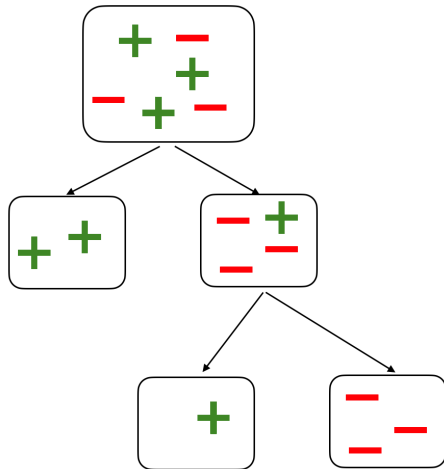
Generic Tree-Building algorithm

1. Start from empty decision tree
2. Recursively, for each node:
 - Iterate through all features and compute how good it'd be to split on each feature
 - Split on the “best” feature
3. Prune

Design choices:

- Termination condition (max depth, entropy, train/val error)
- Tree complexity
- Splitting criterion
- Pruning

Generic Tree-Building algorithm



Overall goal: choose features which can best sort positive and negative examples. For all the samples in a region R_m , predict the majority class c_m

Loss func? 0/1 error / binary classification error

$$f(x) = \sum_{m=1}^M c_m 1\{x \in R_m\}$$

split on $x_j \leq s, \quad x_j > s$

Same

Same for illustrative purposes. In reality you can also have categorical features. E.g. “Islet Antibody Positive Test”, or “Blood Type”

Define “impurity” measure I (measures how well positive class examples are sorted from negative)
Find split that minimizes impurity

$$R_1(j, s) = \{X|X_j \leq s\} \quad \text{and} \quad R_2(j, s) = \{X|X_j > s\}.$$

$$f(x) = \sum_{m=1}^M c_m I(x \in R_m).$$

Then we seek the splitting variable j and split point s that solve

$$\min_{j, s} \left[\min_{c_1} \sum_{x_i \in R_1(j, s)} (y_i - c_1)^2 + \min_{c_2} \sum_{x_i \in R_2(j, s)} (y_i - c_2)^2 \right].$$

$$\hat{c}_m = \text{ave}(y_i | x_i \in R_m).$$

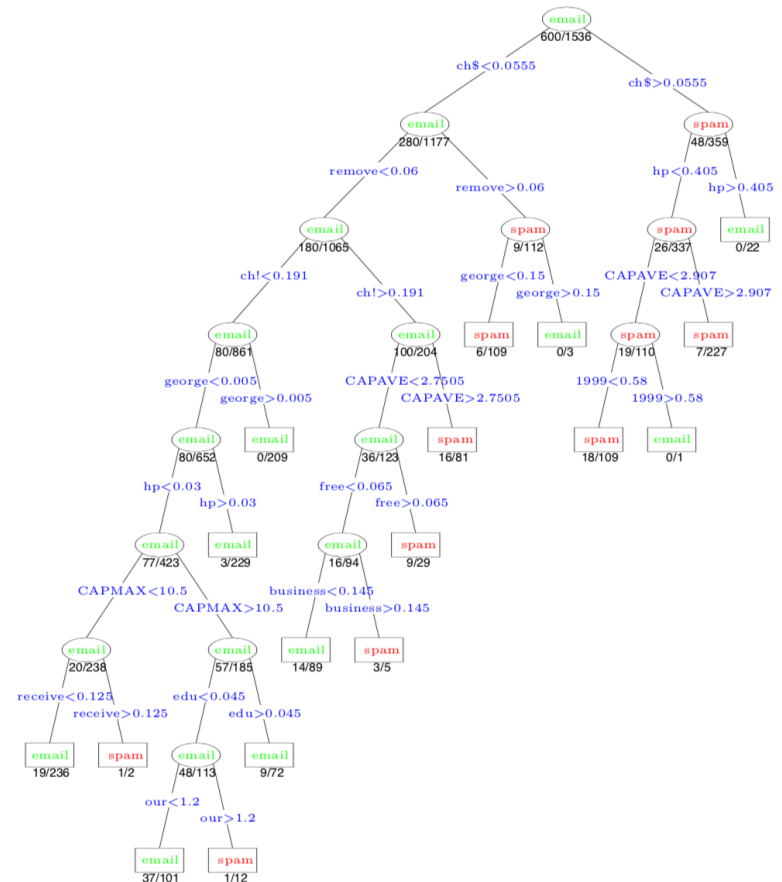
Trees: Interpretable (?)

Trees are “easy” to interpret:

- You can explain how the classifier came to the conclusion it did

But they can be complex

- Small changes in data can result in large difference in trees



Summary so far

- Trees have **low** bias, **high** variance
- Deal with categorical variables well # Where don't they work?
- Intuitive, “interpretable” Many related continuous features (pixels)
- Good software exists
- Some theoretical guarantees

Why low bias?

If you allow enough depth / splits, you can fit anything

How to regularize?

We will say more, but you can limit the number⁷⁴ of nodes, the depth, or the requirements on entropy for leaves

Ensembling methods

(Some slides from Natasha Jaques)



So far

1. Talked regression (linear, mostly)
2. Classification
 1. Logistic Regression
 2. SVMs
 3. Kernelized versions
 4. k-NN
 5. Trees

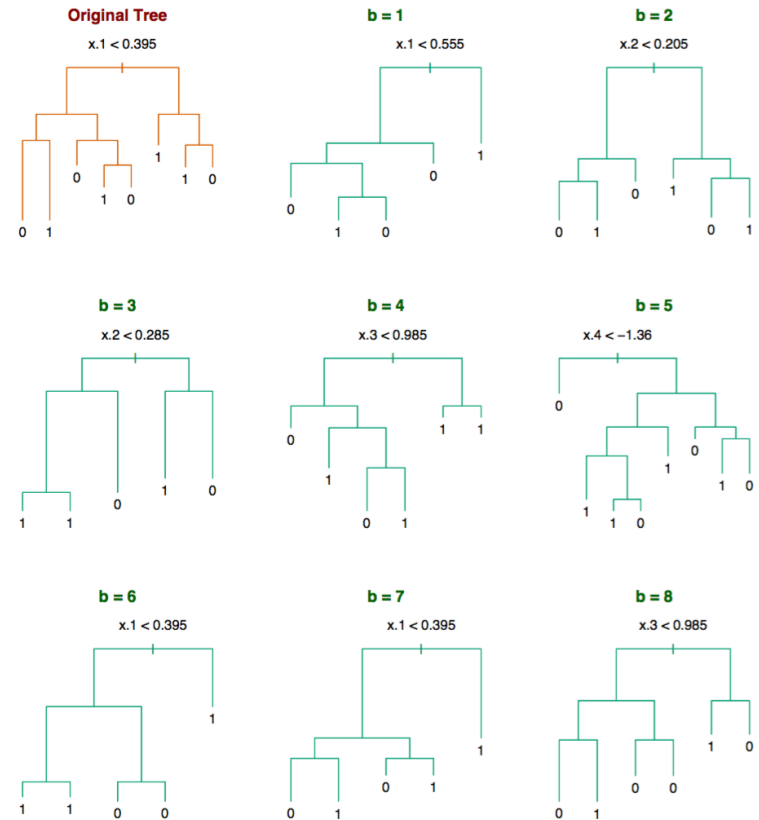
Most of these can be used for regression too

... about to go to more complex classes of models. Is doing so really necessary?

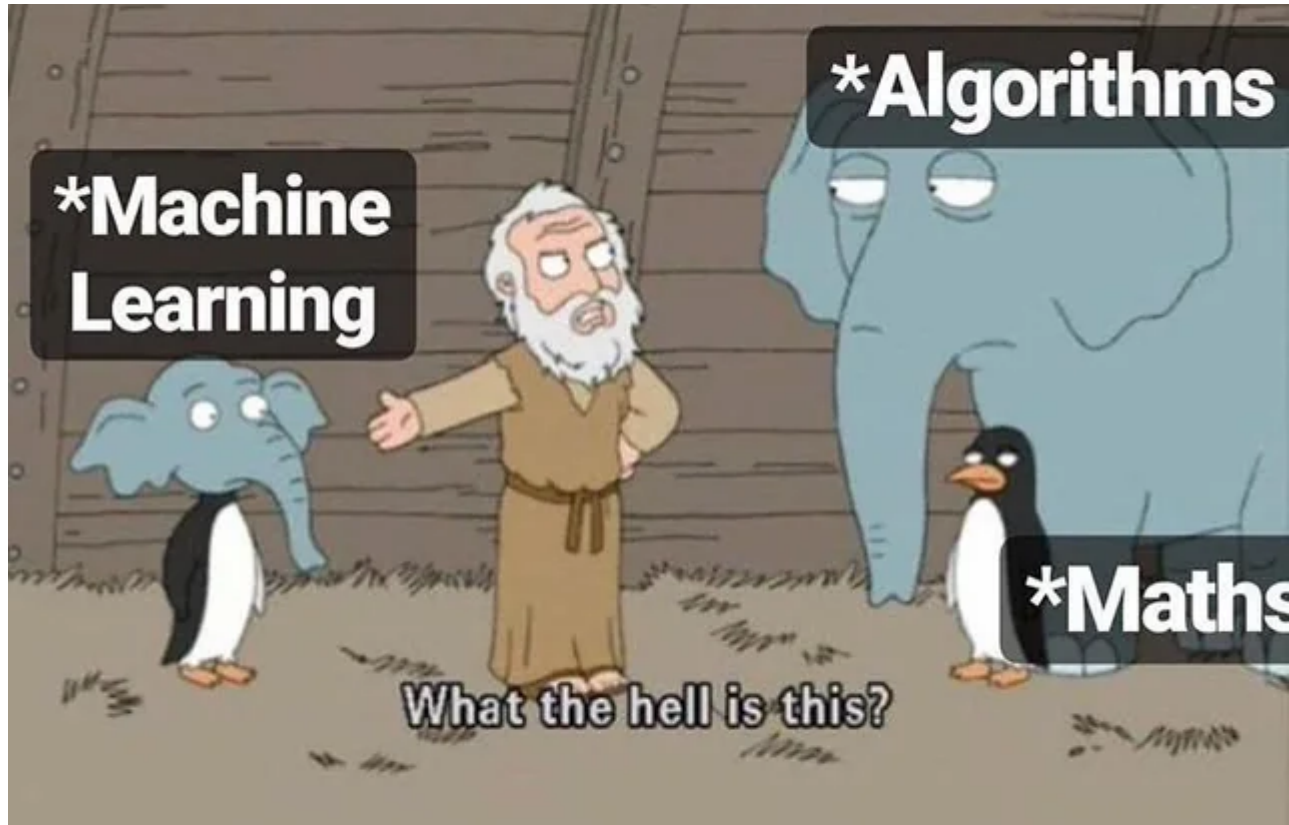
- If we have sufficiently complex kernels, or 1-NN, or high depth trees, we can get very low bias, high variance...
- What about combining weak learners into better learners?

Random Forests, aka “bagging” of trees

- Forest = many trees
- Tree methods have low bias but high variance
- We can reduce variance by constructing many “lightly correlated” trees and averaging them
- Bagging: Bootstrap aggregating
Each tree is trained on new bootstrap sample of data



Random Forests...



Random Forest learning algorithm

Algorithm 15.1 *Random Forest for Regression or Classification.*

1. For $b = 1$ to B :

with replacement

- (a) Draw a bootstrap sample \mathbf{Z}^* of size N from the training data.
- (b) Grow a random-forest tree T_b to the bootstrapped data, by recursively repeating the following steps for each terminal node of the tree, until the minimum node size n_{min} is reached.
 - i. Select m variables at random from the p variables.
 - ii. Pick the best variable/split-point among the m .
 - iii. Split the node into two daughter nodes.

b = train tree

2. Output the ensemble of trees $\{T_b\}_1^B$.

but you're using a random subset of the features

To make a prediction at a new point x :

Regression: $\hat{f}_{\text{rf}}^B(x) = \frac{1}{B} \sum_{b=1}^B T_b(x)$. **# Average**

Classification: Let $\hat{C}_b(x)$ be the class prediction of the b th random-forest tree. Then $\hat{C}_{\text{rf}}^B(x) = \text{majority vote } \{\hat{C}_b(x)\}_1^B$.

Other ensembling methods? Boosting!

Boosting!!! A personal favorite.

Take advantage of the fact that you can learn what parts of the space you are bad at, and do better there.

- 1988 Kearns and Valiant: “Can **weak learners** be combined to create a **strong learner**?”

An algorithm \mathcal{A} is a *weak learner* for a hypothesis class \mathcal{H} that maps \mathcal{X} to $\{-1, 1\}$ if for all input distributions over \mathcal{X} and $h \in \mathcal{H}$, we have that \mathcal{A} correctly classifies h with error at most $1/2 - \gamma$

1990 Robert Schapire: “Yup!”

1995 Schapire and Freund: “Practical for 0/1 loss” AdaBoost

2001 Friedman: “Practical for arbitrary losses”

2014 Tianqi Chen: “Scale it up!” XGBoost

Additive models

- Consider the first algorithm we used to get good classification for MNIST. Given: $\{(x_i, y_i)\}_{i=1}^n$ $x_i \in \mathbb{R}^d, y_i \in \{-1, 1\}$
- Generate **random** functions: $\phi_t : \mathbb{R}^d \rightarrow \mathbb{R} \quad t = 1, \dots, p$
- Learn some weights: $\hat{w} = \arg \min_w \sum_{i=1}^n \text{Loss} \left(y_i, \sum_{t=1}^p w_t \phi_t(x_i) \right)$
- Classify new data: $f(x) = \text{sign} \left(\sum_{t=1}^p \hat{w}_t \phi_t(x) \right)$

Additive models

- Consider the first algorithm we used to get good classification for MNIST. Given: $\{(x_i, y_i)\}_{i=1}^n$ $x_i \in \mathbb{R}^d, y_i \in \{-1, 1\}$
- Generate **random** functions: $\phi_t : \mathbb{R}^d \rightarrow \mathbb{R}$ $t = 1, \dots, p$
- Learn some weights: $\hat{w} = \arg \min_w \sum_{i=1}^n \text{Loss} \left(y_i, \sum_{t=1}^p w_t \phi_t(x_i) \right)$
- Classify new data: $f(x) = \text{sign} \left(\sum_{t=1}^p \hat{w}_t \phi_t(x) \right)$

An interpretation:

Each $\phi_t(x)$ is a classification rule that we are assigning some weight \hat{w}_t

Additive models

- Consider the first algorithm we used to get good classification for MNIST. Given: $\{(x_i, y_i)\}_{i=1}^n$ $x_i \in \mathbb{R}^d, y_i \in \{-1, 1\}$
- Generate **random** functions: $\phi_t : \mathbb{R}^d \rightarrow \mathbb{R}$ $t = 1, \dots, p$
- Learn some weights: $\hat{w} = \arg \min_w \sum_{i=1}^n \text{Loss} \left(y_i, \sum_{t=1}^p w_t \phi_t(x_i) \right)$
- Classify new data: $f(x) = \text{sign} \left(\sum_{t=1}^p \hat{w}_t \phi_t(x) \right)$

An interpretation:

Each $\phi_t(x)$ is a classification rule that we are assigning some weight \hat{w}_t

$$\hat{w}, \hat{\phi}_1, \dots, \hat{\phi}_t = \arg \min_{w, \phi_1, \dots, \phi_p} \sum_{i=1}^n \text{Loss} \left(y_i, \sum_{t=1}^p w_t \phi_t(x_i) \right)$$

is in general computationally hard

Forward Stagewise Additive models

$b(x, \gamma)$ is a function with parameters γ

Examples: $b(x, \gamma) = \frac{1}{1 + e^{-\gamma^T x}}$

Algorithm 10.2 *Forward Stagewise Additive Modeling.*

1. Initialize $f_0(x) = 0$.
2. For $m = 1$ to M :
 - (a) Compute

$$(\beta_m, \gamma_m) = \arg \min_{\beta, \gamma} \sum_{i=1}^N L(y_i, f_{m-1}(x_i) + \beta b(x_i; \gamma)).$$

- (b) Set $f_m(x) = f_{m-1}(x) + \beta_m b(x; \gamma_m)$.
-

Idea: greedily add one function at a time

Forward Stagewise Additive models

$b(x, \gamma)$ is a function with parameters γ

Examples: $b(x, \gamma) = \frac{1}{1 + e^{-\gamma^T x}}$

Algorithm 10.2 *Forward Stagewise Additive Modeling.*

1. Initialize $f_0(x) = 0$.
2. For $m = 1$ to M :
 - (a) Compute

$$(\beta_m, \gamma_m) = \arg \min_{\beta, \gamma} \sum_{i=1}^N L(y_i, f_{m-1}(x_i) + \beta b(x_i; \gamma)).$$

- (b) Set $f_m(x) = f_{m-1}(x) + \beta_m b(x; \gamma_m)$.
-

Idea: greedily add one function at a time

AdaBoost: $b(x, \gamma)$: classifiers to $\{-1, 1\}$

$$L(y, f(x)) = \exp(-yf(x))$$

Forward Stagewise Additive models

$b(x, \gamma)$ is a function with parameters γ

Examples: $b(x, \gamma) = \frac{1}{1 + e^{-\gamma^T x}}$

Algorithm 10.2 *Forward Stagewise Additive Modeling.*

1. Initialize $f_0(x) = 0$.
2. For $m = 1$ to M :
 - (a) Compute

$$(\beta_m, \gamma_m) = \arg \min_{\beta, \gamma} \sum_{i=1}^N L(y_i, f_{m-1}(x_i) + \beta b(x_i; \gamma)).$$

- (b) Set $f_m(x) = f_{m-1}(x) + \beta_m b(x; \gamma_m)$.
-

Idea: greedily add one function at a time

Boosted Regression Trees:

$$L(y, f(x)) = (y - f(x))^2$$

$b(x, \gamma)$: regression trees

Forward Stagewise Additive models

$b(x, \gamma)$ is a function with parameters γ

Examples: $b(x, \gamma) = \frac{1}{1 + e^{-\gamma^T x}}$

$$b(x, \gamma) = \gamma_1 \mathbf{1}\{x_3 \leq \gamma_2\}$$

Algorithm 10.2 *Forward Stagewise Additive Modeling.*

1. Initialize $f_0(x) = 0$.
2. For $m = 1$ to M :
 - (a) Compute

$$(\beta_m, \gamma_m) = \arg \min_{\beta, \gamma} \sum_{i=1}^N L(y_i, f_{m-1}(x_i) + \beta b(x_i; \gamma)).$$

- (b) Set $f_m(x) = f_{m-1}(x) + \beta_m b(x; \gamma_m)$.
-

Idea: greedily add one function at a time

Boosted Regression Trees:

$$L(y, f(x)) = (y - f(x))^2$$

$$\begin{aligned} L(y_i, f_{m-1}(x_i) + \beta b(x_i; \gamma)) &= (y_i - f_{m-1}(x_i) - \beta b(x_i; \gamma))^2 \\ &= (r_{im} - \beta b(x_i; \gamma))^2, \quad r_{im} = y_i - f_{m-1}(x_i) \end{aligned}$$

Efficient: No harder than learning regression trees!

Forward Stagewise Additive models

$b(x, \gamma)$ is a function with parameters γ

Examples: $b(x, \gamma) = \frac{1}{1 + e^{-\gamma^T x}}$

$$b(x, \gamma) = \gamma_1 \mathbf{1}\{x_3 \leq \gamma_2\}$$

Algorithm 10.2 *Forward Stagewise Additive Modeling.*

1. Initialize $f_0(x) = 0$.
2. For $m = 1$ to M :
 - (a) Compute

$$(\beta_m, \gamma_m) = \arg \min_{\beta, \gamma} \sum_{i=1}^N L(y_i, f_{m-1}(x_i) + \beta b(x_i; \gamma)).$$

- (b) Set $f_m(x) = f_{m-1}(x) + \beta_m b(x; \gamma_m)$.
-

Idea: greedily add one function at a time

Boosted Logistic Trees: $L(y, f(x)) = y \log(f(x)) + (1 - y) \log(1 - f(x))$

$b(x, \gamma)$: regression trees

Computationally hard to update

Gradient Boosting

Least squares, exponential loss easy. But what about cross entropy? Huber?

Algorithm 10.3 *Gradient Tree Boosting Algorithm.*

1. Initialize $f_0(x) = \arg \min_{\gamma} \sum_{i=1}^N L(y_i, \gamma)$.

2. For $m = 1$ to M :

(a) For $i = 1, 2, \dots, N$ compute

$$r_{im} = - \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} \right]_{f=f_{m-1}} .$$

(b) Fit a regression tree to the targets r_{im} giving terminal regions R_{jm} , $j = 1, 2, \dots, J_m$.

(c) For $j = 1, 2, \dots, J_m$ compute

$$\gamma_{jm} = \arg \min_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma) .$$

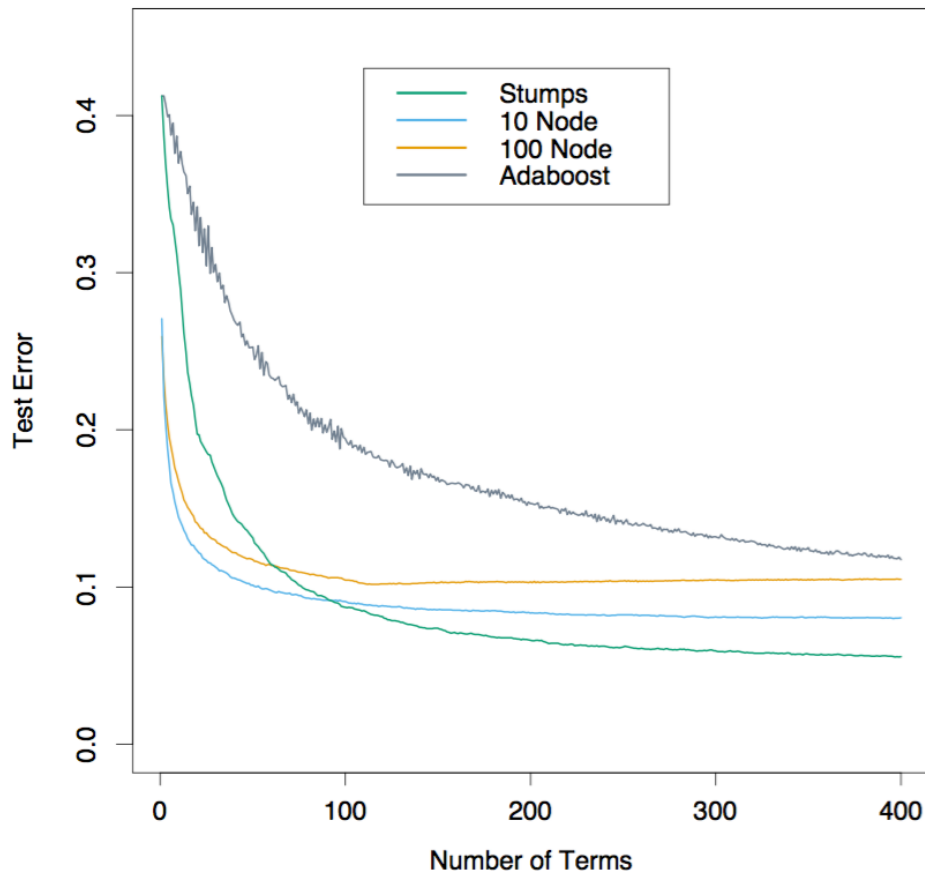
(d) Update $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$.

3. Output $\hat{f}(x) = f_M(x)$.

LS fit regression tree to n-dimensional gradient, take a step in that direction

Gradient Boosting

Least squares, 0/1 loss easy. But what about cross entropy? Huber?



AdaBoost uses 0/1 loss,
all other trees are minimizing
binomial deviance

Additive models



- Boosting is popular at parties: Invented by theorists, heavily adopted by practitioners.

Additive models



- Boosting is popular at parties: Invented by theorists, heavily adopted by practitioners.
- Computationally efficient with “weak” learners. But can also use trees! Boosting can scale.
- Kind of like sparsity?

Additive models

- Boosting is popular at parties: Invented by theorists, heavily adopted by practitioners.
- Computationally efficient with “weak” learners. But can also use trees! Boosting can scale.
- Kind of like sparsity?
- Gradient boosting generalization with good software packages (e.g., *XGBoost*). Effective on Kaggle
- Robust to overfitting and can be dealt with with “shrinkage” and “sampling”

Bagging versus Boosting

- Bagging *averages* many **low-bias, lightly dependent** classifiers to reduce the variance
- Boosting *learns* linear combination of **high-bias, highly dependent** classifiers to reduce error
- Empirically, boosting appears to outperform bagging