

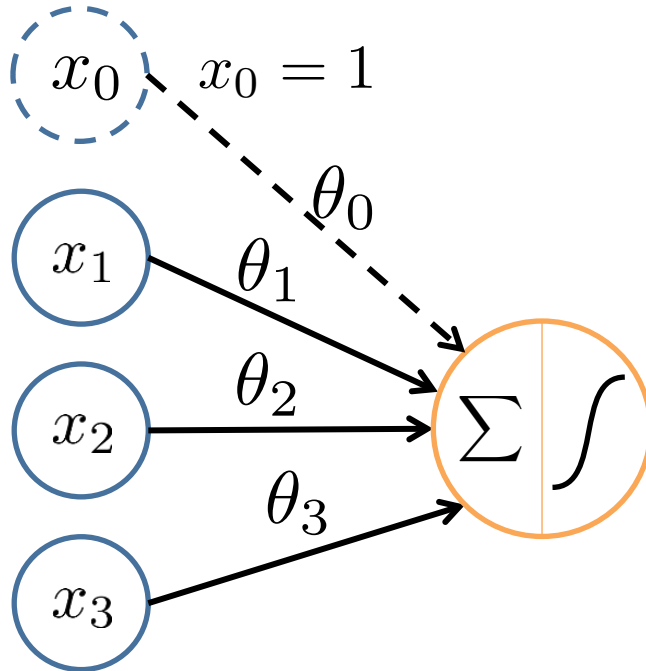
# Neural Networks

---



# Single Node

“bias unit”



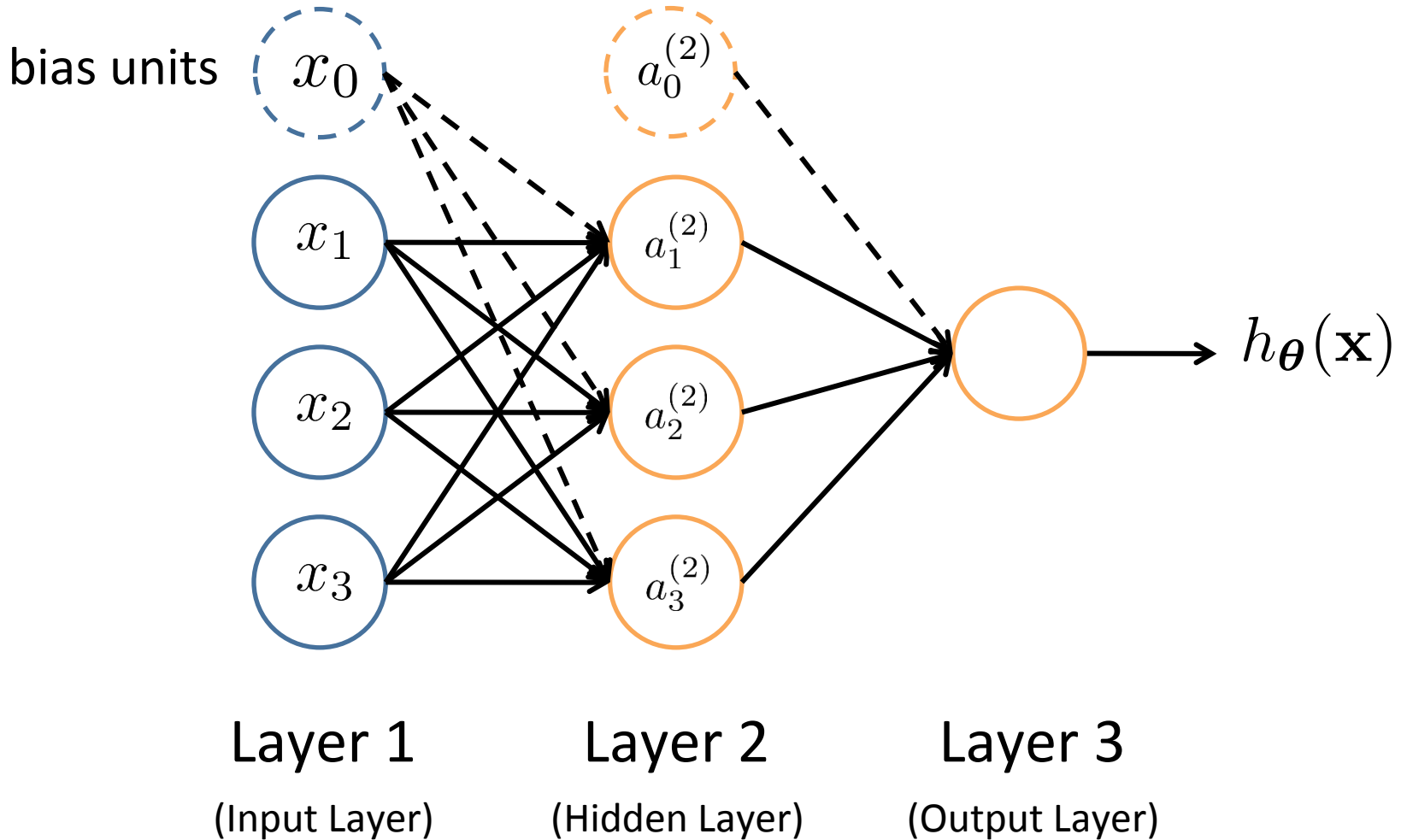
$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$

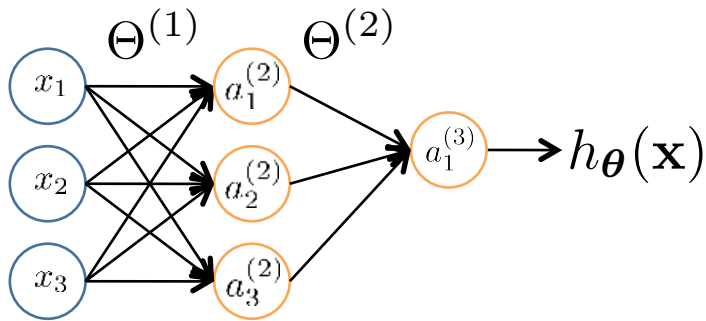
$$h_{\boldsymbol{\theta}}(\mathbf{x}) = g(\boldsymbol{\theta}^T \mathbf{x}) = \frac{1}{1 + e^{-\boldsymbol{\theta}^T \mathbf{x}}}$$

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

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

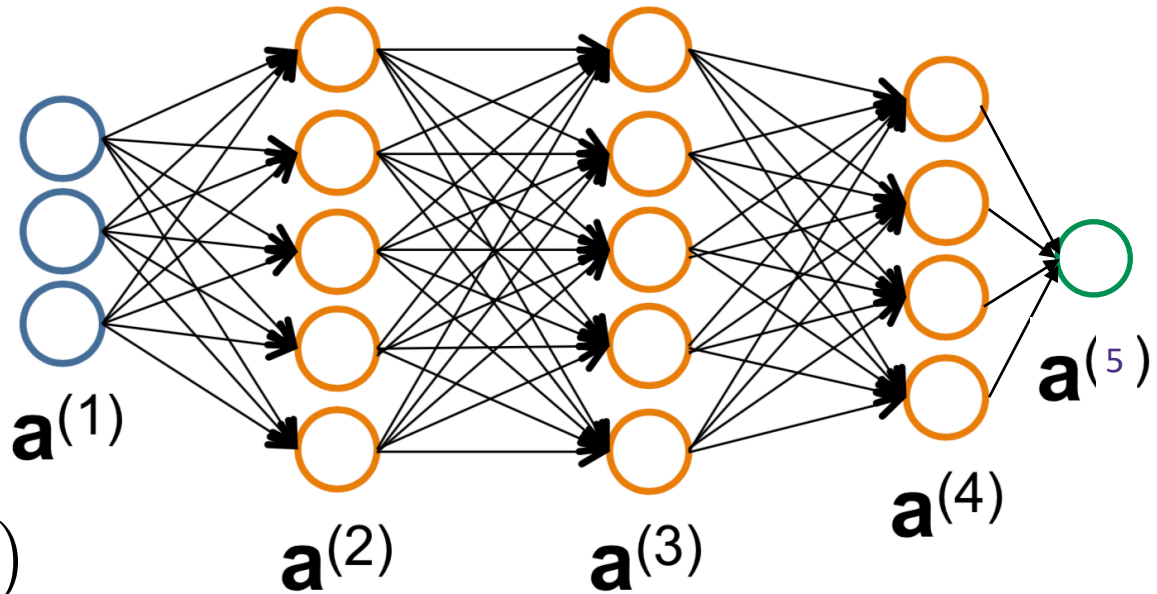
$$a^{(2)} = g(\Theta^{(1)} a^{(1)})$$

⋮

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

⋮

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



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

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

Binary  
Logistic  
Regression

# Multi-layer Neural Network - Binary Classification

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

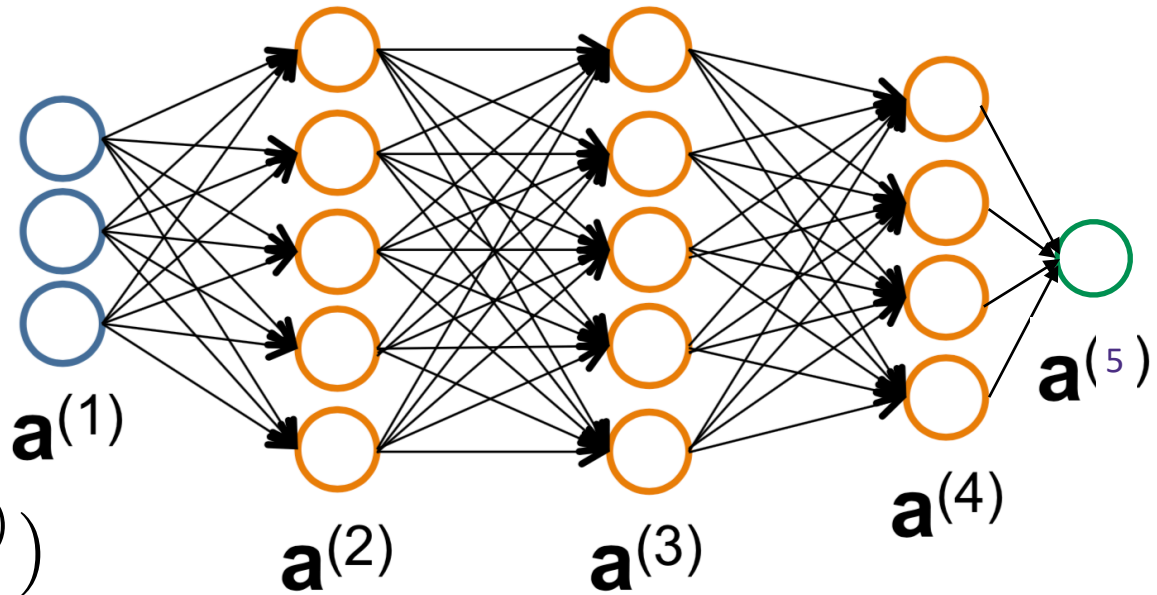
$$a^{(2)} = \sigma(\Theta^{(1)} a^{(1)})$$

⋮

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

⋮

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



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

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

# Multiple Output Units: One-vs-Rest



Pedestrian



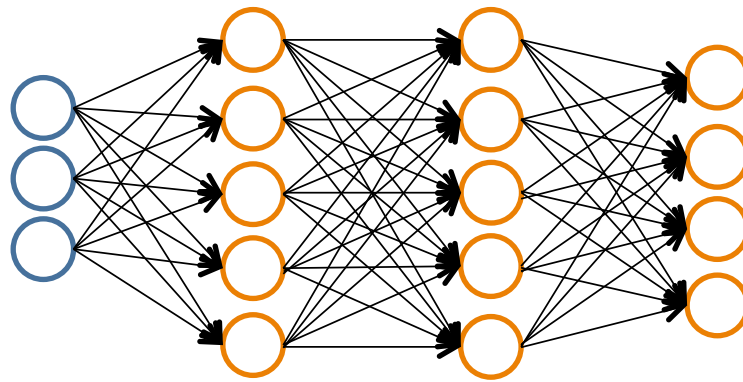
Car



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

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

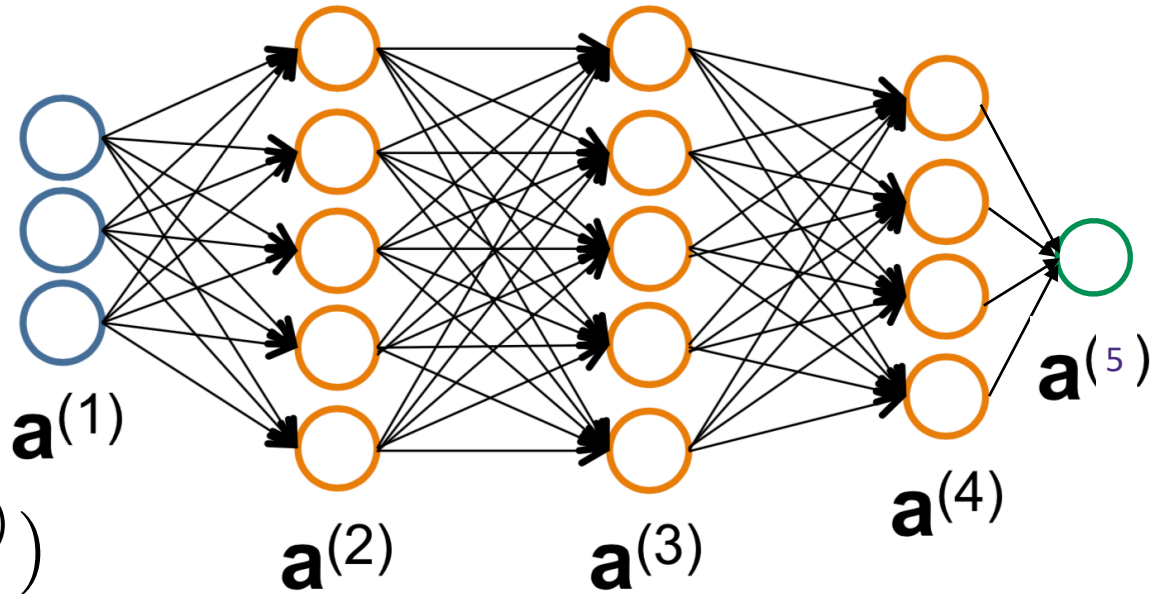
$$a^{(2)} = \sigma(\Theta^{(1)} a^{(1)})$$

⋮

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

⋮

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



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

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

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

---

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

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

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

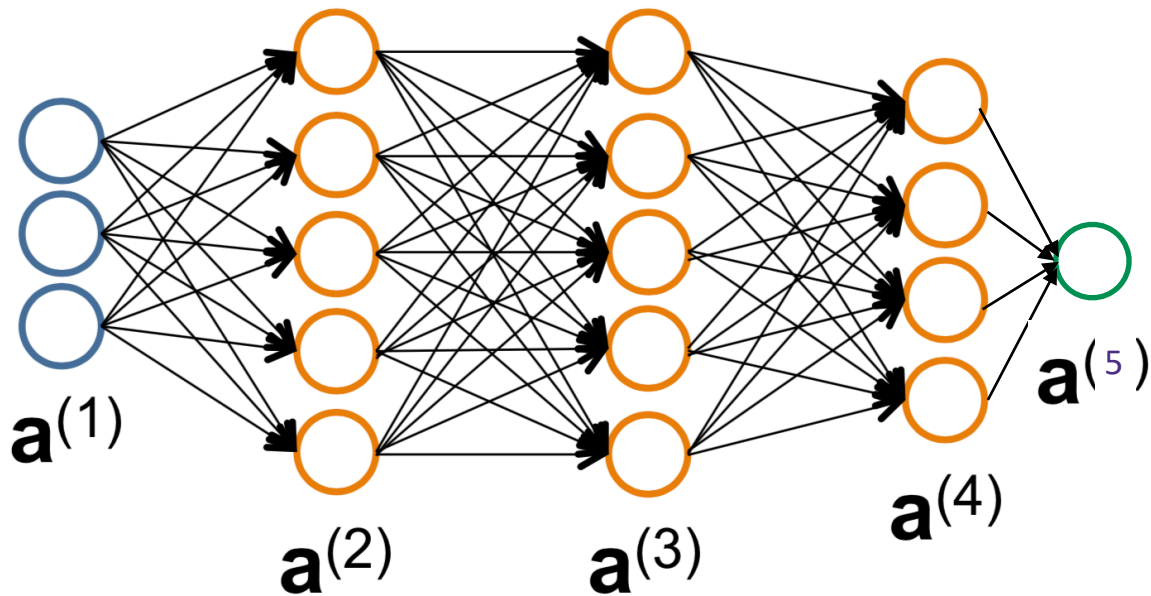
$$\vdots$$

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

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

$$\vdots$$

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



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

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

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

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 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

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

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

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

⋮

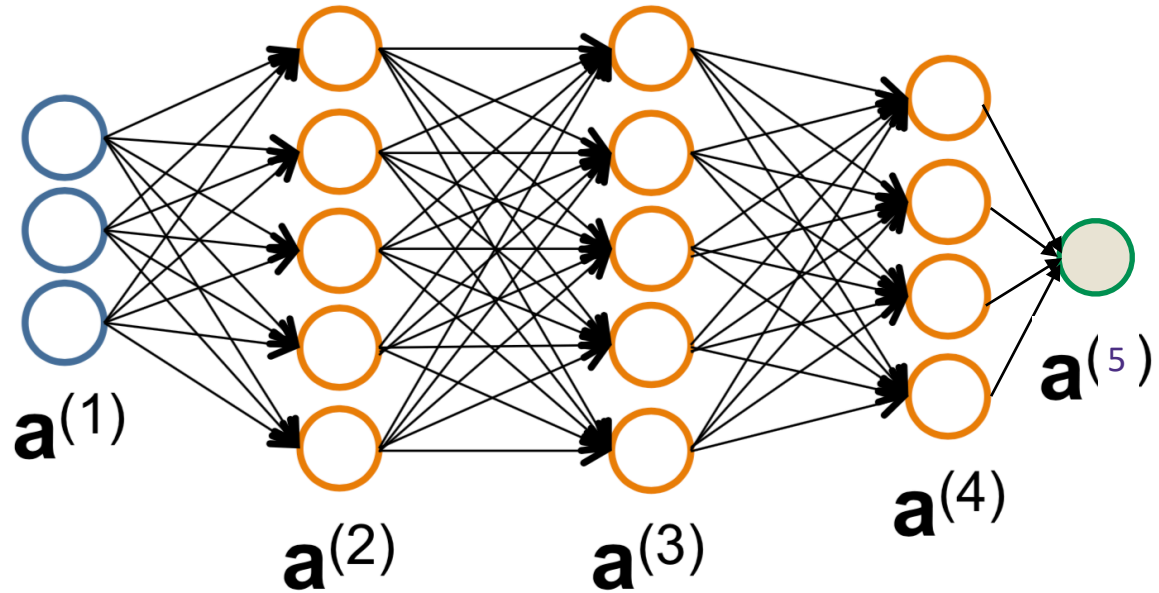
$$a^{(l)} = g(z^{(l)})$$

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

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

⋮

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



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

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

# Backprop

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

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

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

⋮

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

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

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

⋮

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

**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

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

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

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

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

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

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

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

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

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

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

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

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

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

⋮

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

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

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

⋮

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

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

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

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

# Backprop

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

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

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

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

⋮

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

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

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

⋮

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

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

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

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

# 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

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

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

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

$$\vdots$$

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

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

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

$$\vdots$$

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

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

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

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

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

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

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

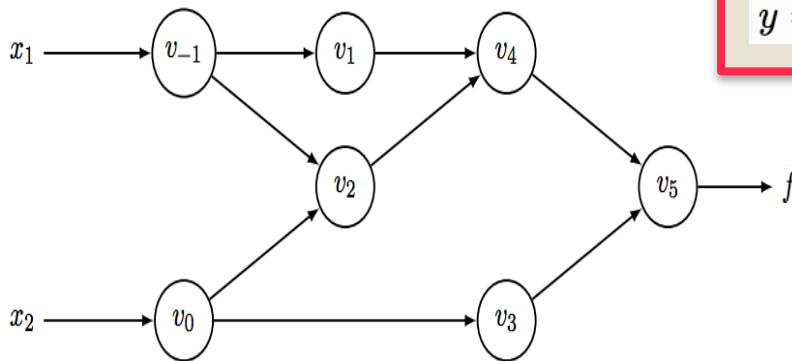
Compute gradients  $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

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/>		
$v_1 = \ln v_{-1}$	$=$	$\ln 2$
$v_2 = v_{-1} \times v_0$	$=$	$2 \times 5$
<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		
$\bar{x}_1 = \bar{v}_{-1}$	$=$	<b>5.5</b>
$\bar{x}_2 = \bar{v}_0$	$=$	<b>1.716</b>
<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

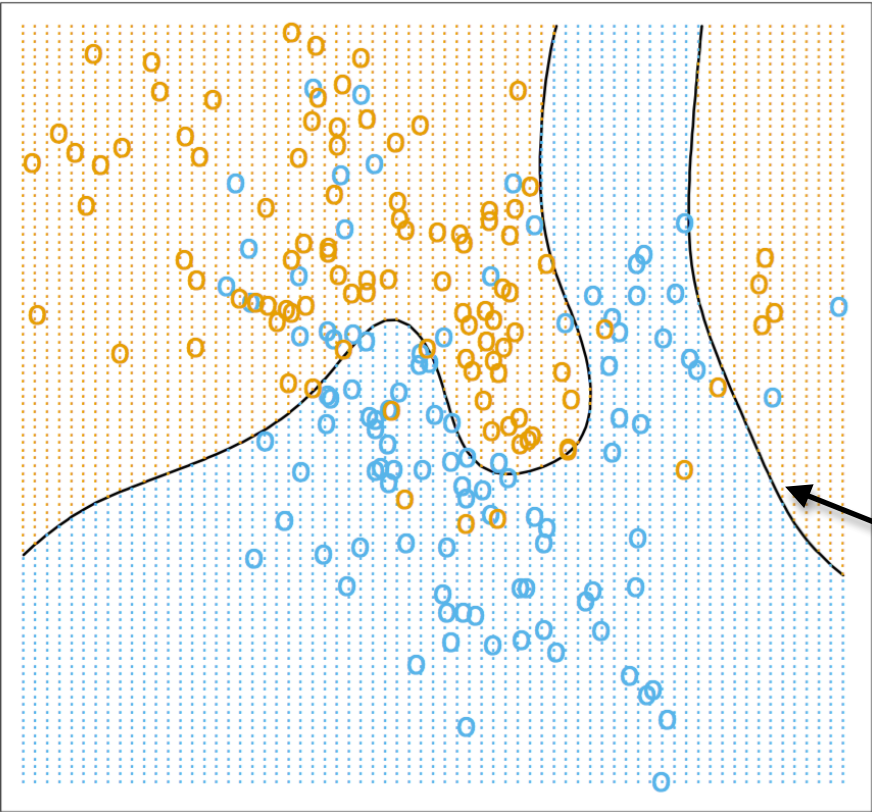
This is the special sauce in Tensorflow, PyTorch, Theano, ...

# Nearest Neighbor Methods

---

- A model is called “parametric” if the number of parameters do not depend on the number of samples
- A model is called “non-parametric” if the number of parameters increase with the number of samples (Does **not** mean absence of parameters!)

# Some data, Bayes Classifier



Training data:

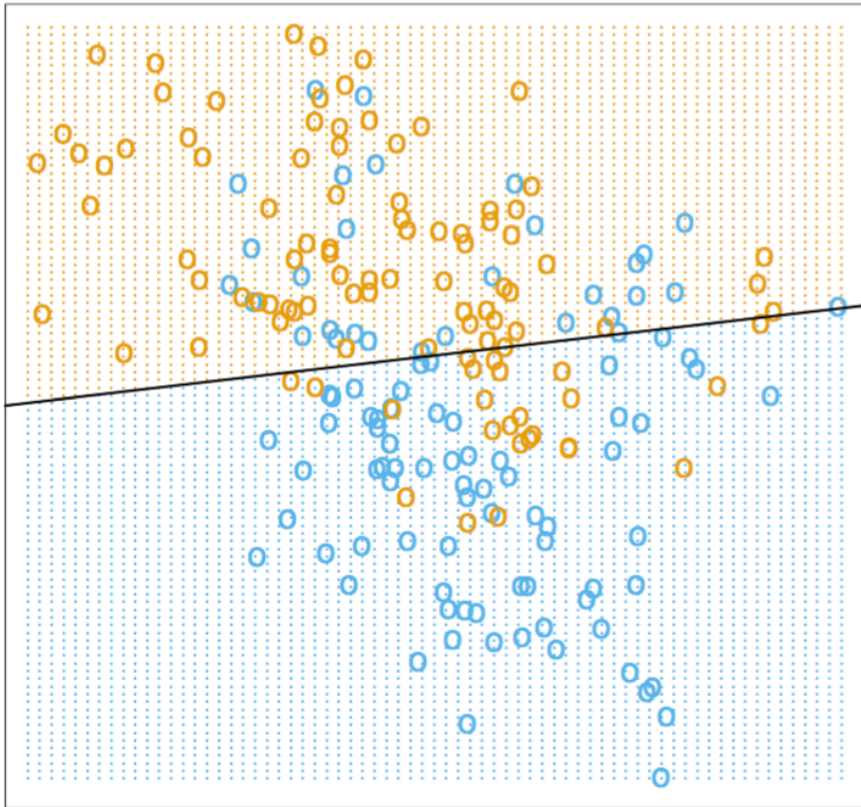
- True label: +1
- True label: -1

Optimal “Bayes” classifier:

$$\mathbb{P}(Y = 1|X = x) = \frac{1}{2}$$

- Predicted label: +1
- Predicted label: -1

# Linear Decision Boundary



Training data:

- True label: +1
- True label: -1

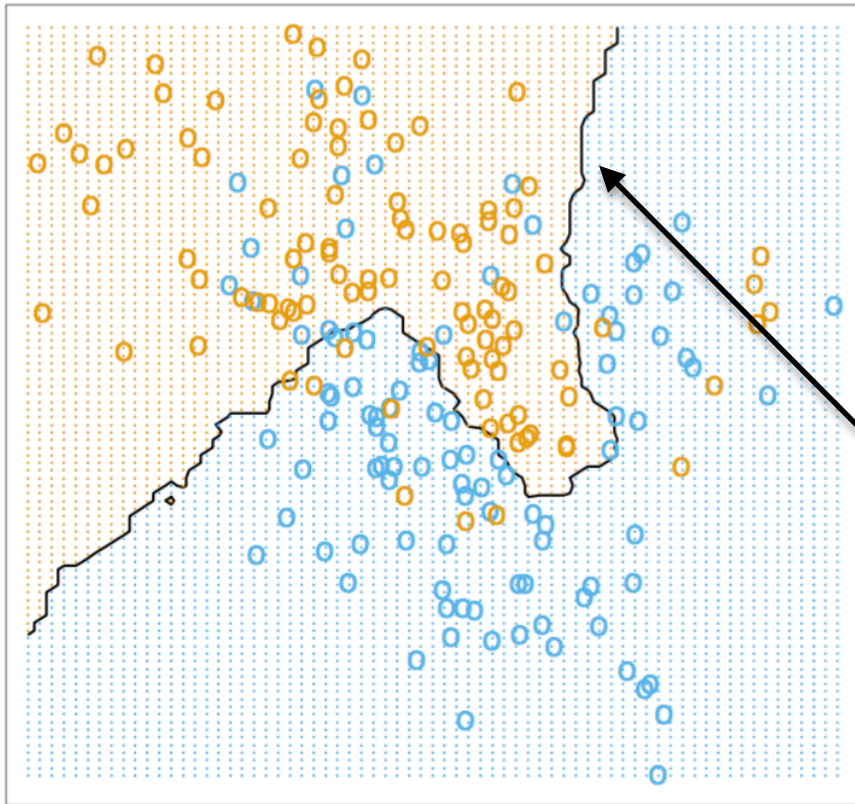
Learned:

Linear Decision boundary

$$x^T w + b = 0$$

- ▨ Predicted label: +1
- ▨ Predicted label: -1

# $k=15$ Nearest Neighbor Boundary



Training data:

○ True label: +1

○ True label: -1

Learned:

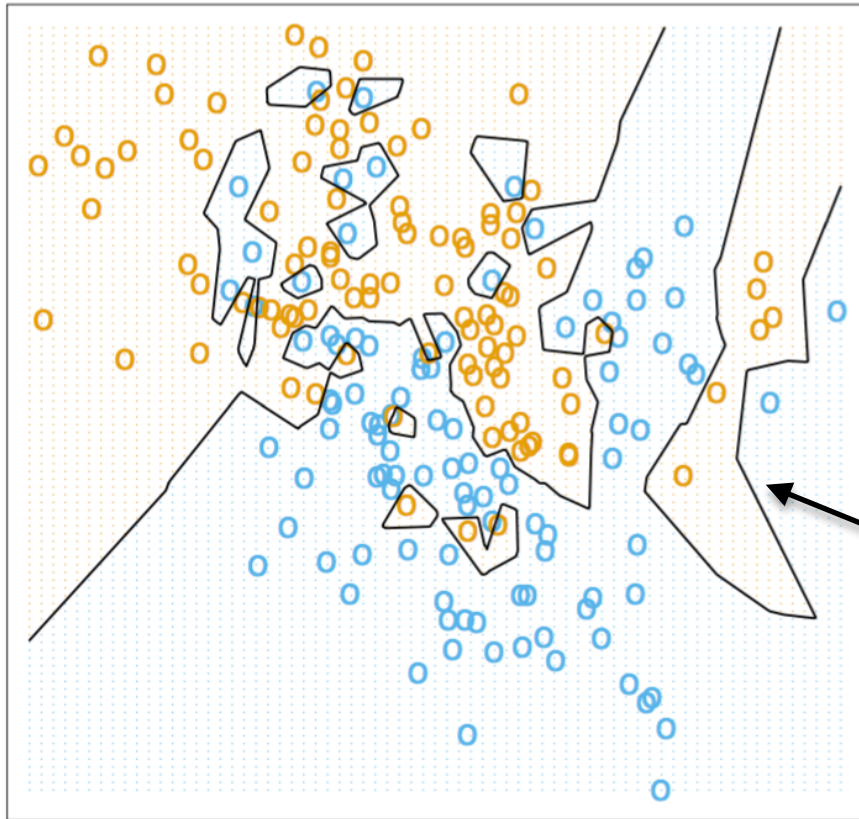
**15** nearest neighbor decision boundary (majority vote)

○ Predicted label: +1

○ Predicted label: -1

- Nearest neighbor gives non-linear decision boundaries
- What happens if we use a small  $k$  or a large  $k$ ?

# $k=1$ Nearest Neighbor Boundary



Training data:

○ True label: +1

○ True label: -1

Learned:

1 nearest neighbor decision boundary (majority vote)

■ Predicted label: +1

■ Predicted label: -1

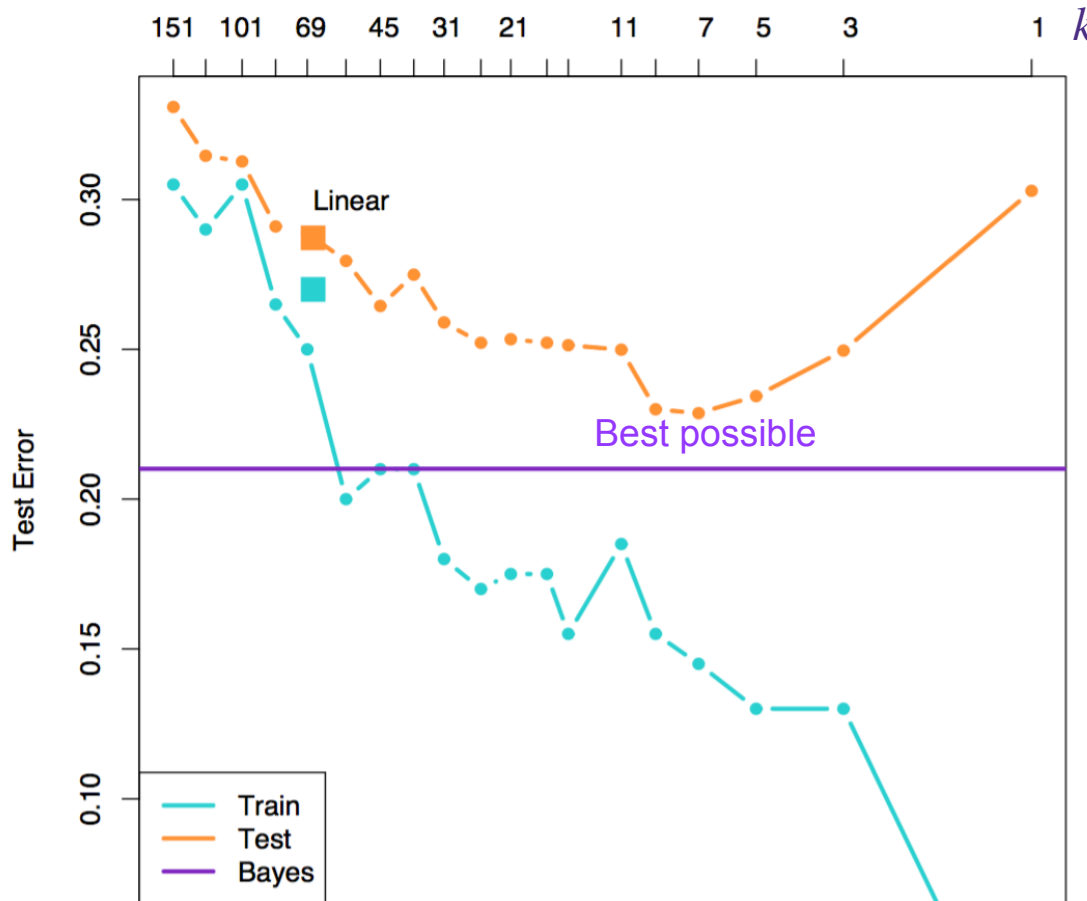
- With a small  $k$ , we tend to overfit.

# k-Nearest Neighbor Error

Model complexity low

Model complexity high

k – Number of Nearest Neighbors



Bias-Variance tradeoff

As  $k \rightarrow \infty$ ?

Bias:

Variance:

As  $k \rightarrow 1$ ?

Bias:

Variance:

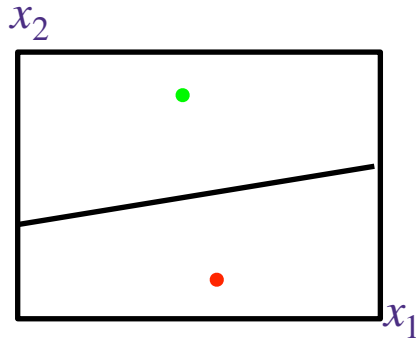
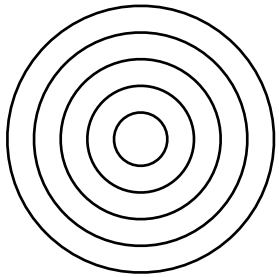
Figures from Hastie et al

- The error achieved by Bayes optimal classifier provides a lower bound on what any estimator can achieve

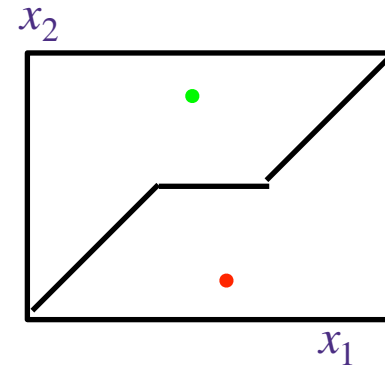
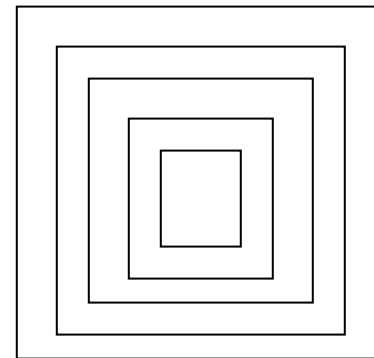
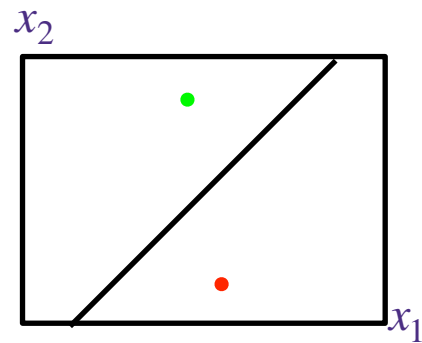
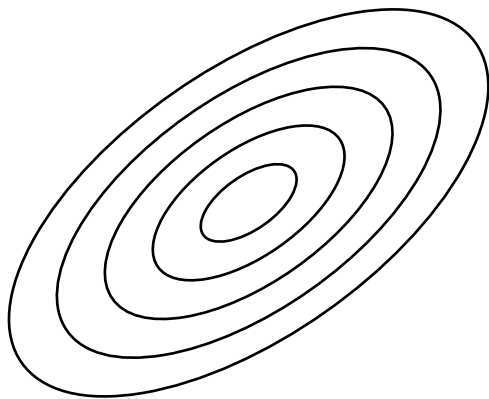
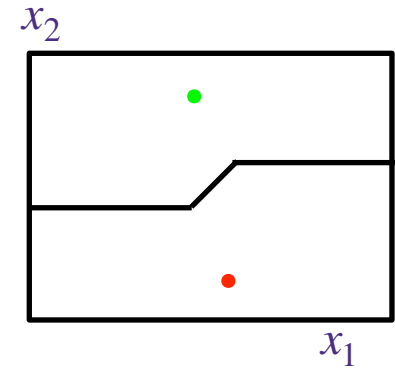
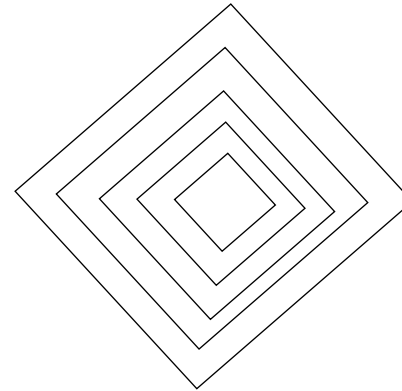
# Notable distance metrics (and their level sets)

Consider 2 dimensional example with 2 data points with labels green, red, and we show  $k = 1$  nearest neighbor decision boundaries for various choices of distances

**L<sub>2</sub> norm** :  $d(x, y) = \|x - y\|_2$



**L<sub>1</sub> norm (taxi-cab)**

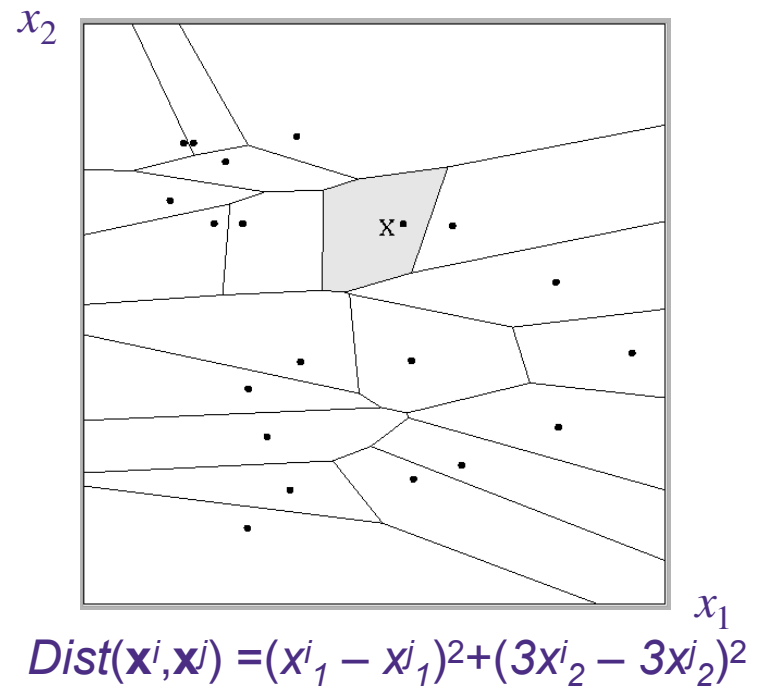
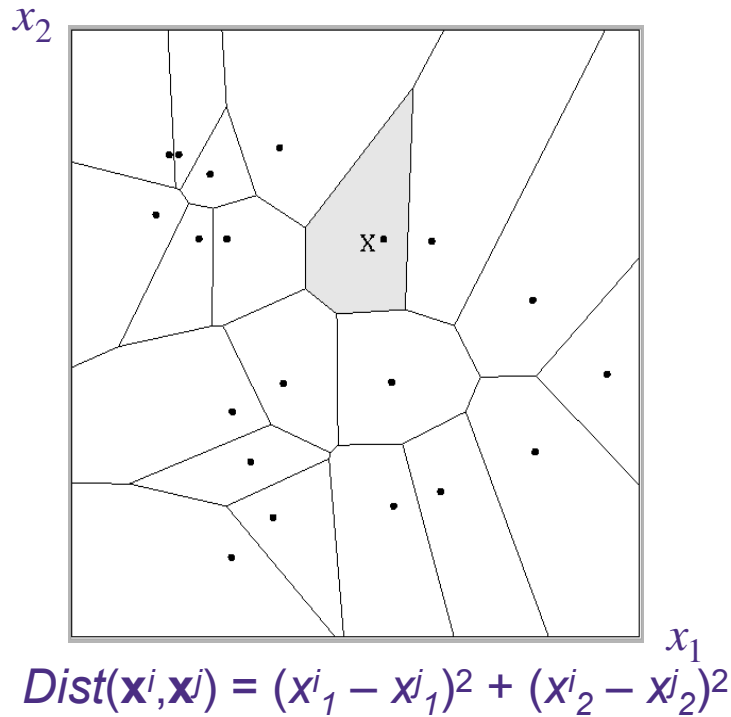


**Mahalanobis norm**:  $d(x, y) = (x - y)^T M (x - y)$

**L-infinity (max) norm**

# $k = 1$ nearest neighbor

One can draw the nearest-neighbor regions in input space.



The relative scalings in the distance metric affect region shapes

# 1 nearest neighbor guarantee - classification

---

$$\{(x_i, y_i)\}_{i=1}^n \quad x_i \in \mathbb{R}^d, \quad y_i \in \{0, 1\} \quad (x_i, y_i) \stackrel{iid}{\sim} P_{XY}$$

**Theorem**[Cover, Hart, 1967] If  $P_X$  is supported everywhere in  $\mathbb{R}^d$  and  $P(Y = 1|X = x)$  is smooth everywhere, then as  $n \rightarrow \infty$  the 1-NN classification rule has error at most twice the Bayes error rate.

# 1 nearest neighbor guarantee - classification

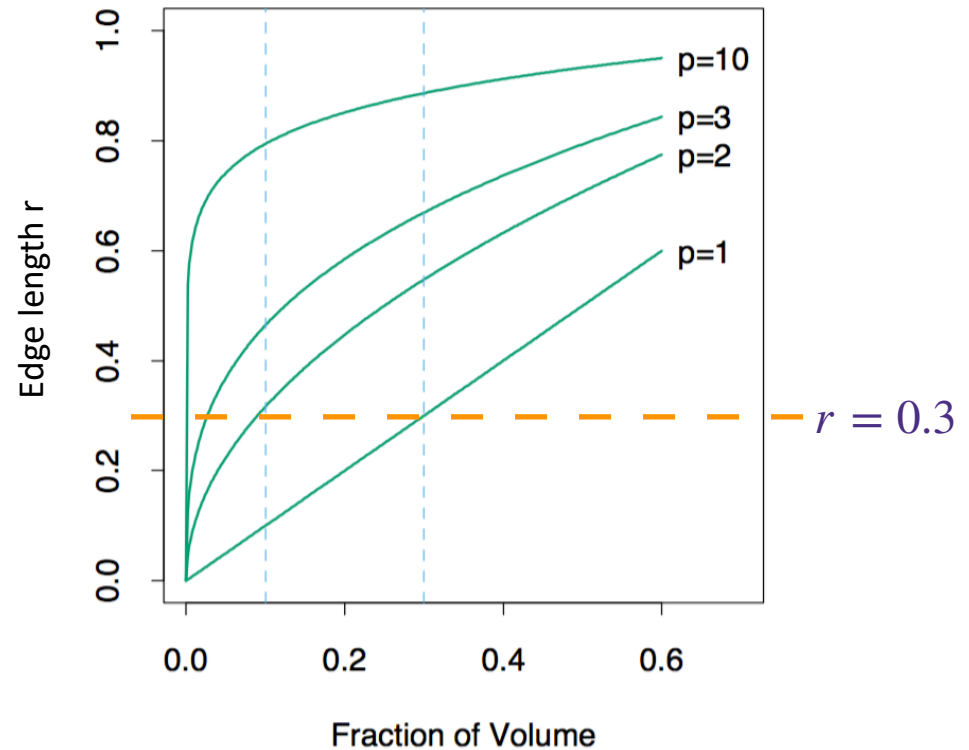
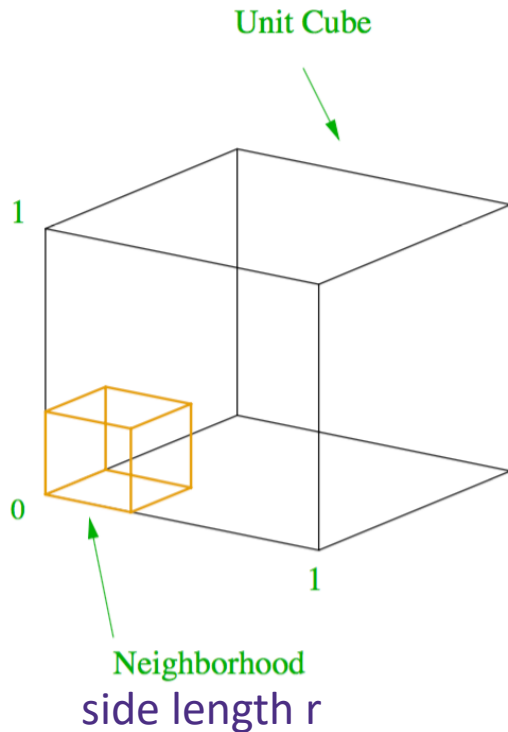
$$\{(x_i, y_i)\}_{i=1}^n \quad x_i \in \mathbb{R}^d, \quad y_i \in \{0, 1\} \quad (x_i, y_i) \stackrel{iid}{\sim} P_{XY}$$

**Theorem**[Cover, Hart, 1967] If  $P_X$  is supported everywhere in  $\mathbb{R}^d$  and  $P(Y = 1|X = x)$  is smooth everywhere, then as  $n \rightarrow \infty$  the 1-NN classification rule has error at most twice the Bayes error rate.

- Let  $x_{NN}$  denote the nearest neighbor at a point  $x$
- First note that as  $n \rightarrow \infty$ ,  $P(y = +1 | x_{NN}) \rightarrow P(y = +1 | x)$
- Let  $p^* = \min\{P(y = +1 | x), P(y = -1 | x)\}$  denote the Bayes error rate
- At a point  $x$ ,
  - Case 1: nearest neighbor is +1, which happens with  $P(y = +1 | x)$  and the error rate is  $P(y = -1 | x)$
  - Case 2: nearest neighbor is -1, which happens with  $P(y = -1 | x)$  and the error rate is  $P(y = +1 | x)$
- The average error of a 1-NN is

$$P(y = +1 | x) P(y = -1 | x) + P(y = -1 | x) P(y = +1 | x) = 2p^*(1 - p^*)$$

# Curse of dimensionality Ex. 1

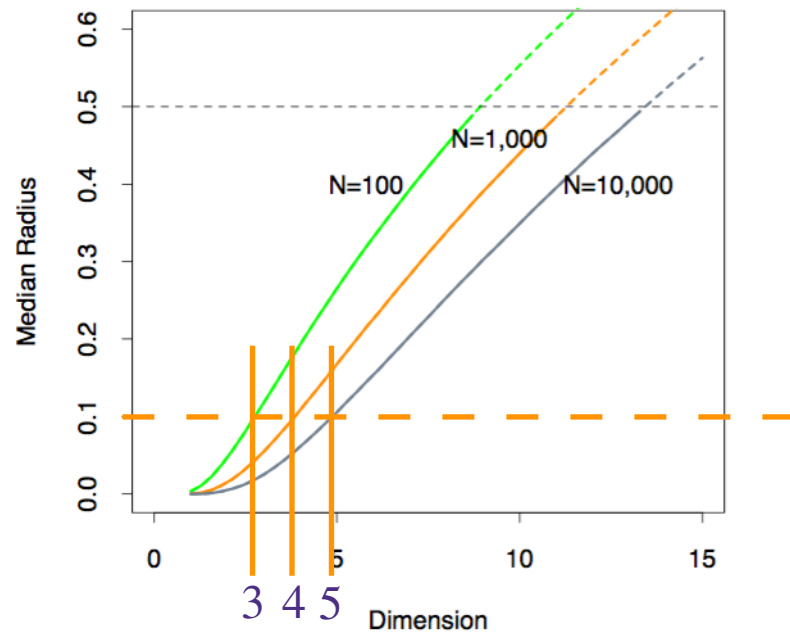
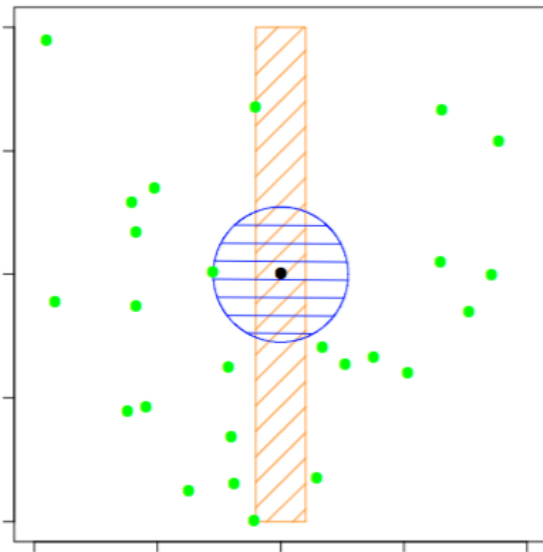


$X$  is uniformly distributed over  $[0, 1]^p$ . What is  $\mathbb{P}(X \in [0, r]^p)$ ?

How many samples do we need so that a nearest neighbor is within a cube of side length  $r$ ?

# Curse of dimensionality Ex. 2

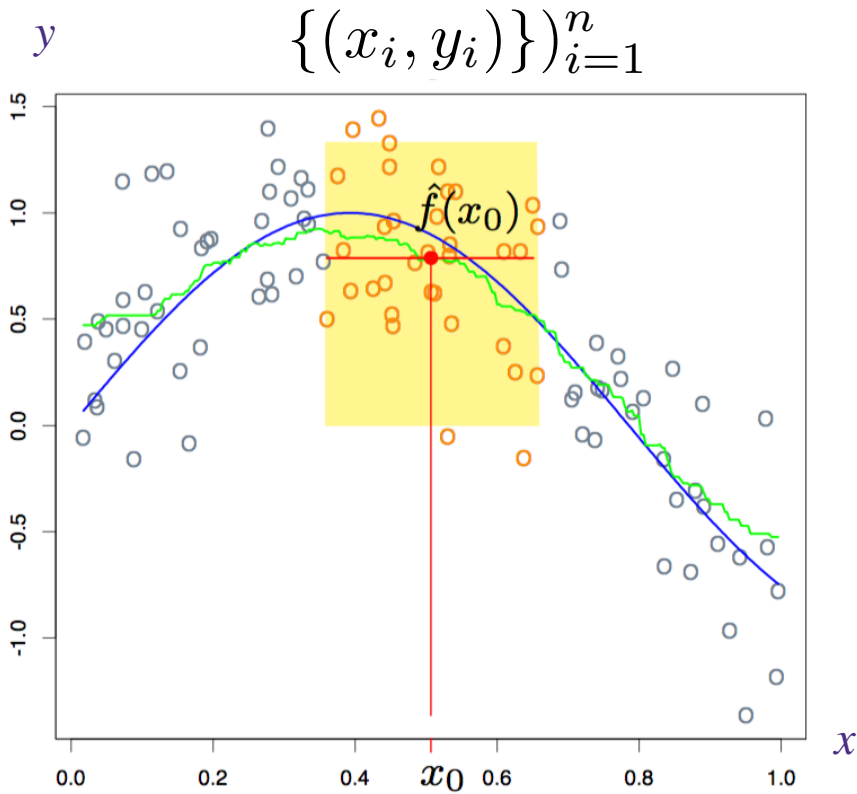
$\{X_i\}_{i=1}^n$  are uniformly distributed over  $[-.5, .5]^p$ .



What is the median distance from a point at origin to its 1NN?

How many samples do we need so that a median Euclidean distance is within  $r$ ?

# Nearest neighbor regression



- What is the optimal classifier that minimizes MSE  $\mathbb{E}[(\hat{y} - y)^2]$ ?

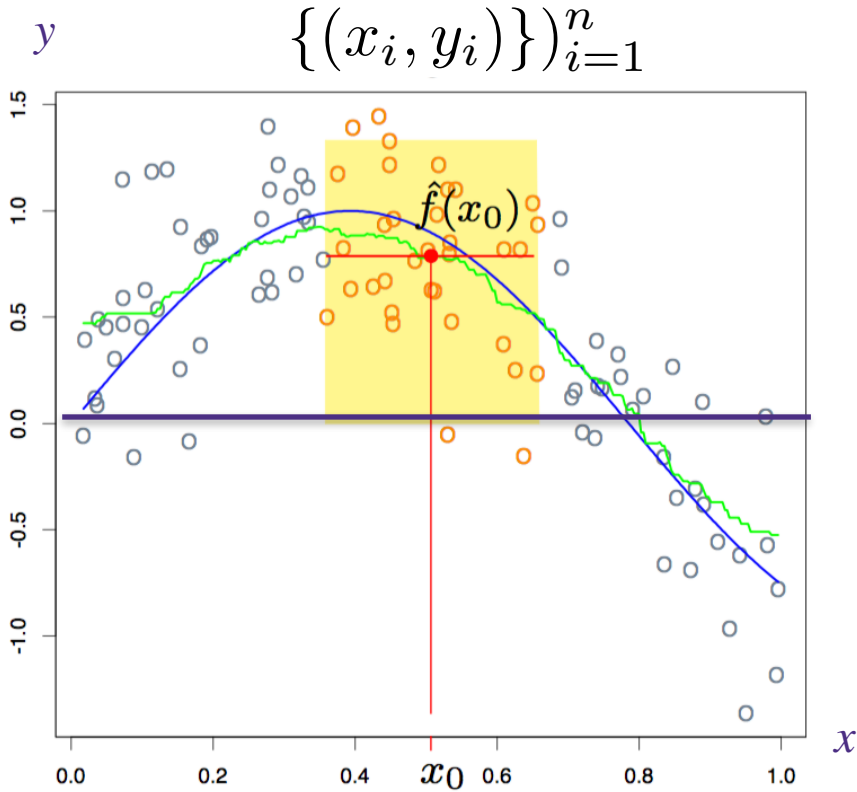
$$\hat{y} = \mathbb{E}[y | x]$$

- $k$ -nearest neighbor regressor is

$$\hat{f}(x) = \frac{1}{k} \sum_{j \in \text{nearest neighbor}} y_j$$

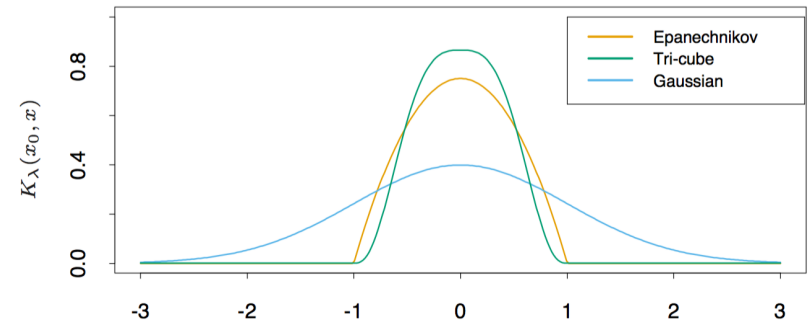
$$= \frac{\sum_{i=1}^n y_i \times \text{Ind}(x_i \text{ is a } k \text{ nearest neighbor})}{\sum_{i=1}^n \text{Ind}(x_i \text{ is a } k \text{ nearest neighbor})}$$

# Nearest neighbor regression



In nearest neighbor methods, the “weight” changes abruptly

smoothing:  $K(x, y)$

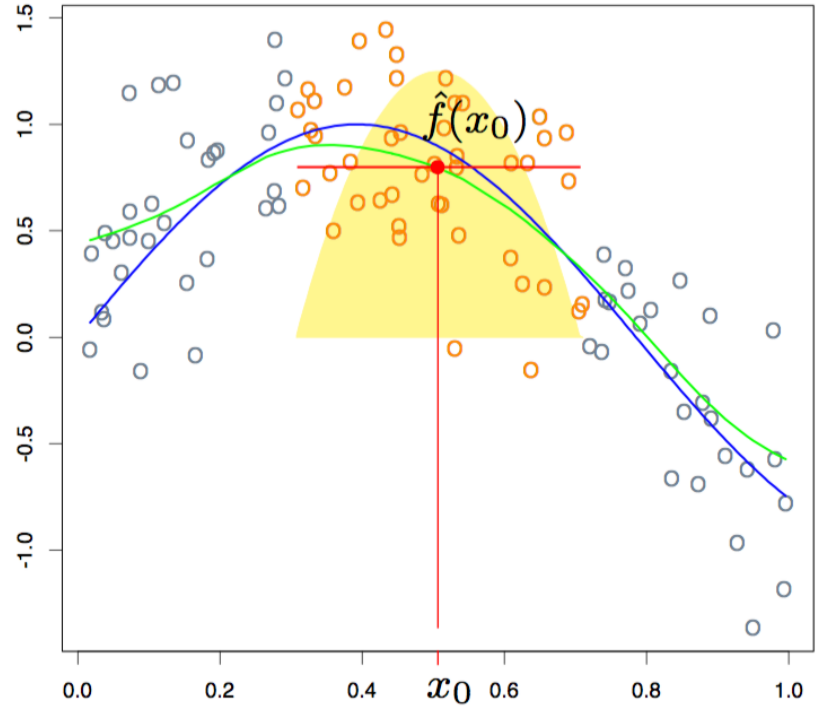
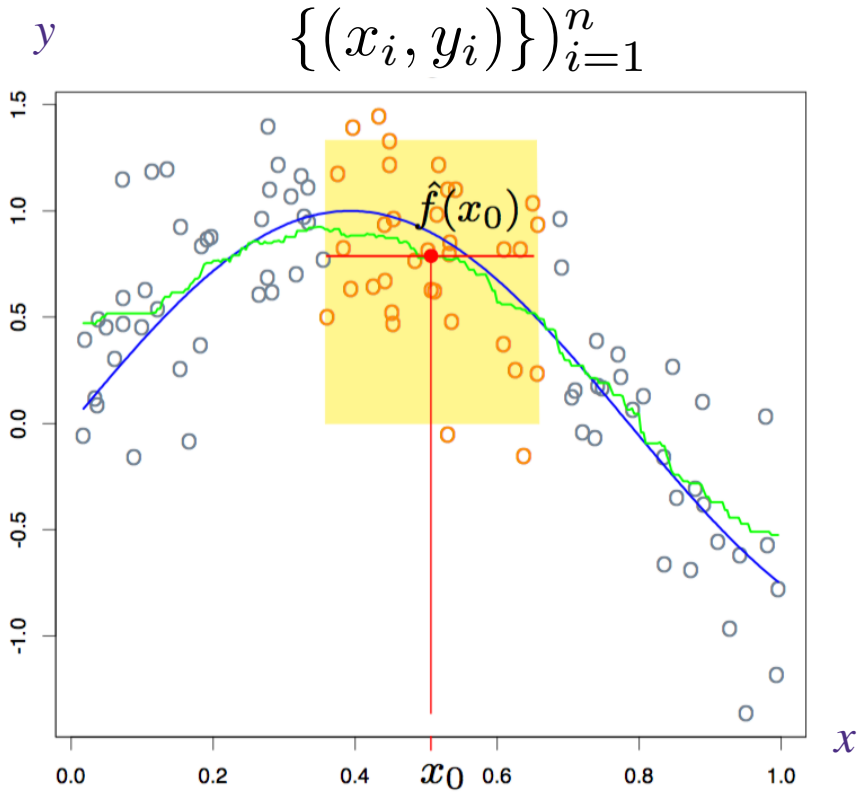


- $k$ -nearest neighbor regressor is

$$\hat{f}(x_0) = \frac{\sum_{i=1}^n y_i \times \text{Ind}(x_i \text{ is a } k \text{ nearest neighbor})}{\sum_{i=1}^n \text{Ind}(x_i \text{ is a } k \text{ nearest neighbor})}$$

$$\hat{f}(x_0) = \frac{\sum_{i=1}^n K(x_0, x_i) y_i}{\sum_{i=1}^n K(x_0, x_i)}$$

# Nearest neighbor regression

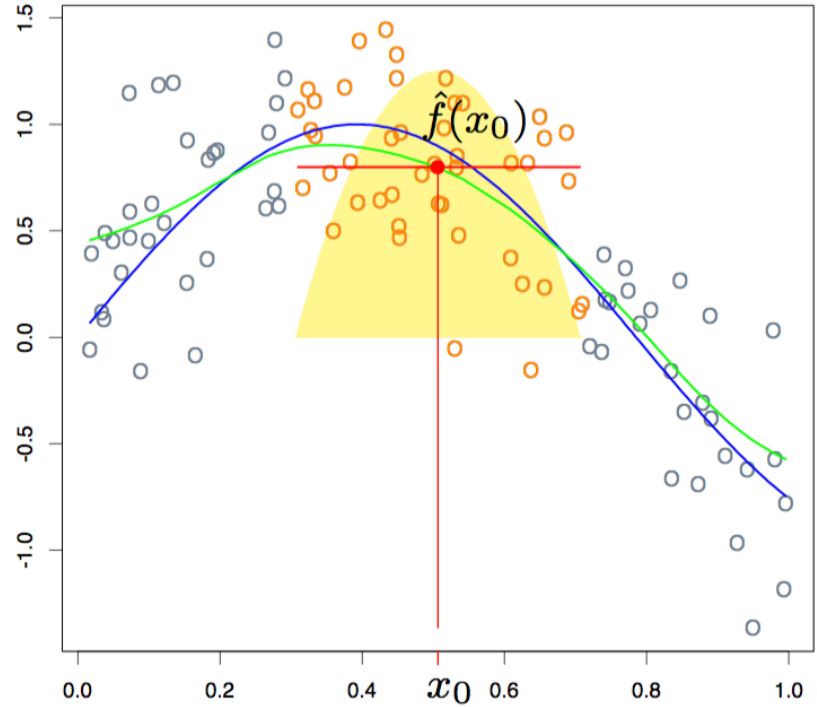
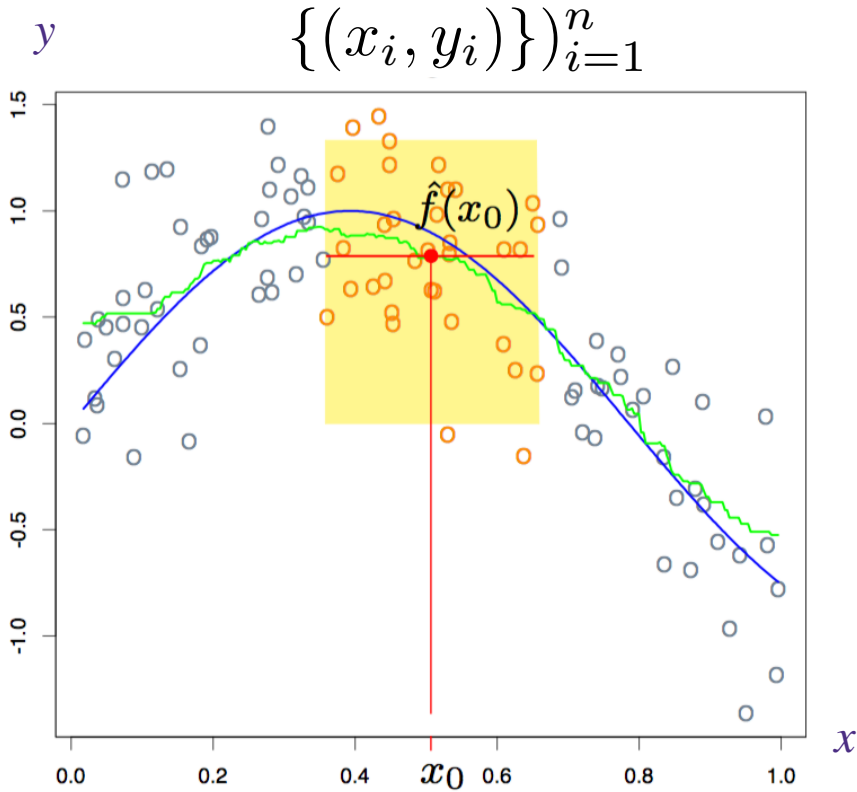


- $k$ -nearest neighbor regressor is

$$\hat{f}(x_0) = \frac{\sum_{i=1}^n y_i \times \text{Ind}(x_i \text{ is a } k \text{ nearest neighbor})}{\sum_{i=1}^n \text{Ind}(x_i \text{ is a } k \text{ nearest neighbor})}$$

$$\hat{f}(x_0) = \frac{\sum_{i=1}^n K(x_0, x_i) y_i}{\sum_{i=1}^n K(x_0, x_i)}$$

# Nearest neighbor regression



Why just average them?

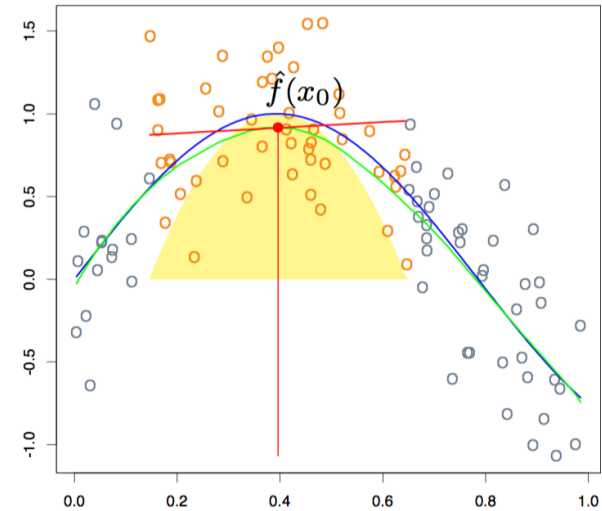
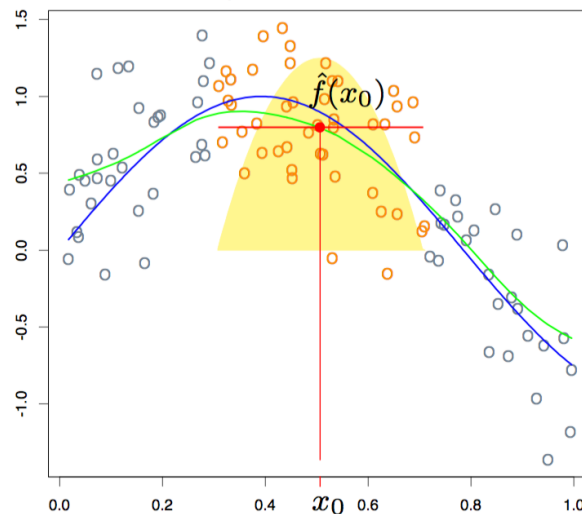
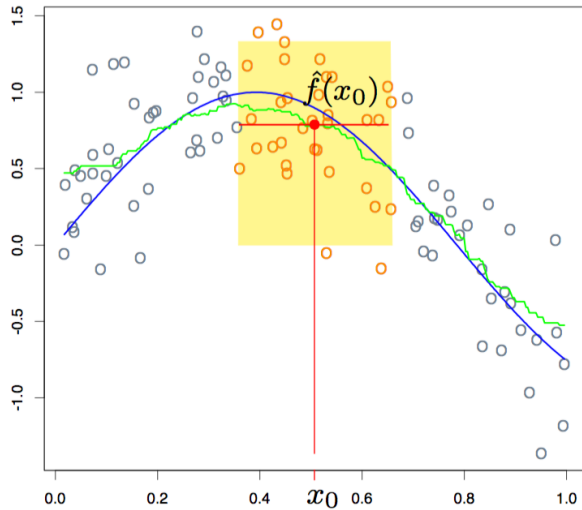
- $k$ -nearest neighbor regressor is

$$\hat{f}(x_0) = \frac{\sum_{i=1}^n y_i \times \text{Ind}(x_i \text{ is a } k \text{ nearest neighbor})}{\sum_{i=1}^n \text{Ind}(x_i \text{ is a } k \text{ nearest neighbor})}$$

$$\hat{f}(x_0) = \frac{\sum_{i=1}^n K(x_0, x_i) y_i}{\sum_{i=1}^n K(x_0, x_i)}$$

# Nearest neighbor regression

$$\{(x_i, y_i)\}_{i=1}^n$$



- $k$ -nearest neighbor regressor is

$$\hat{f}(x_0) = \frac{\sum_{i=1}^n y_i \times \text{Ind}(x_i \text{ is a } k \text{ nearest neighbor})}{\sum_{i=1}^n \text{Ind}(x_i \text{ is a } k \text{ nearest neighbor})}$$

$$\hat{f}(x_0) = \frac{\sum_{i=1}^n K(x_0, x_i) y_i}{\sum_{i=1}^n K(x_0, x_i)}$$

$$\hat{f}(x_0) = b(x_0) + w(x_0)^T x_0$$

$$w(x_0), b(x_0) = \arg \min_{w, b} \sum_{i=1}^n K(x_0, x_i) (y_i - (b + w^T x_i))^2$$

**Local Linear Regression**

# Nearest Neighbor Overview

---

- Very simple to explain and implement
- No training! But finding nearest neighbors in large dataset at test can be computationally demanding (KD-trees help)
- You can use other forms of distance (not just Euclidean)
- Smoothing and local linear regression can improve performance (at the cost of higher variance)
- With a lot of data, “local methods” have strong, simple theoretical guarantees.
- Without a lot of data, neighborhoods aren’t “local” and methods suffer (curse of dimensionality).

# Questions?

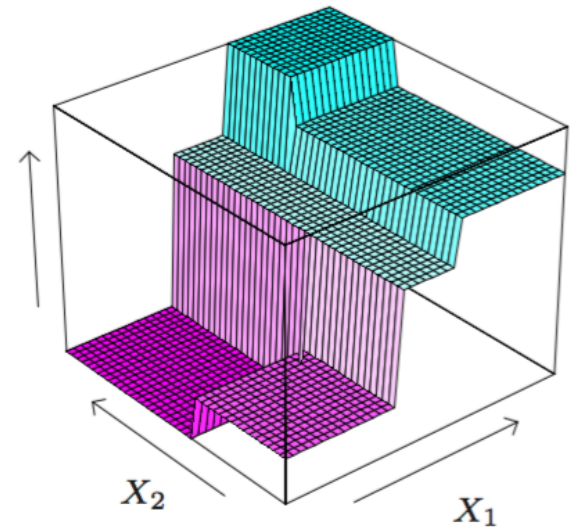
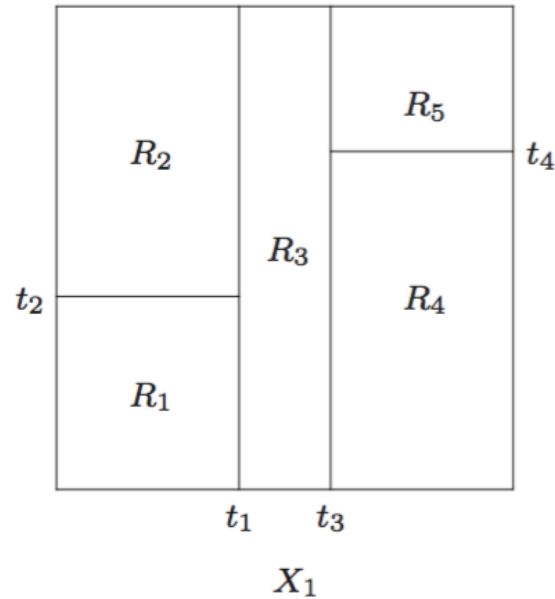
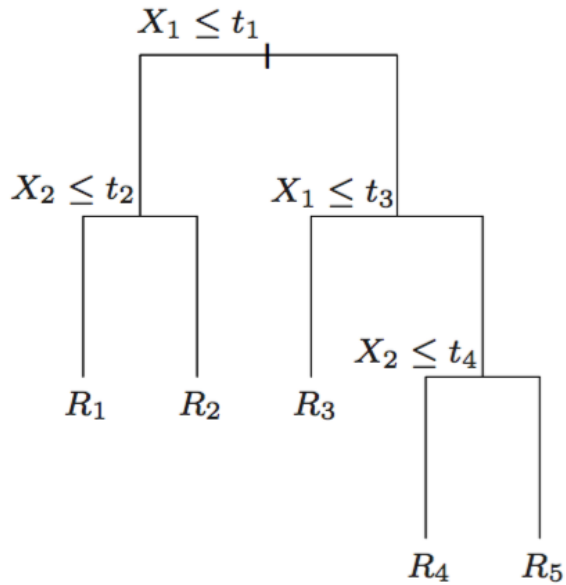
---

# Trees

---

# Trees

Example: binary tree with splits along axes



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

# Regression Trees

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

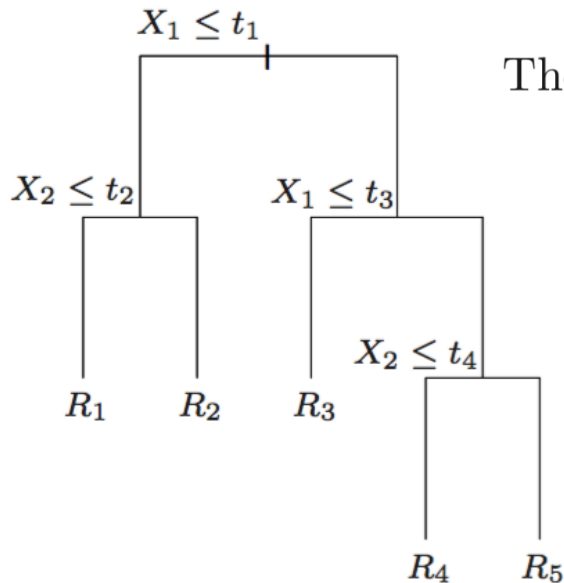
Binary tree with splits along axes.

How do you build the tree / find the splits?

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

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

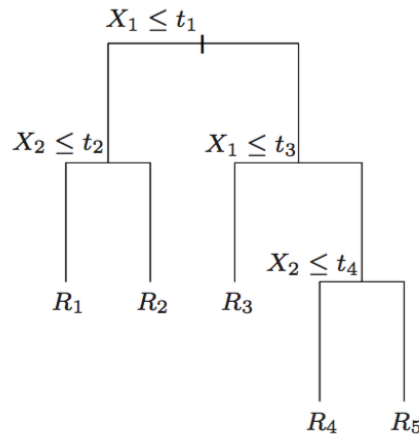
Then we seek the splitting variable  $j$  and split point  $s$  that solve



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

# Learning decision trees

- > Start from empty decision tree
- > Split on next best attribute (feature)
  - Use, for example, information gain to select attribute
  - Split on  $\arg \max_i IG(X_i) = \arg \max_i H(Y) - H(Y | X_i)$
- > Recurse
- > Prune



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



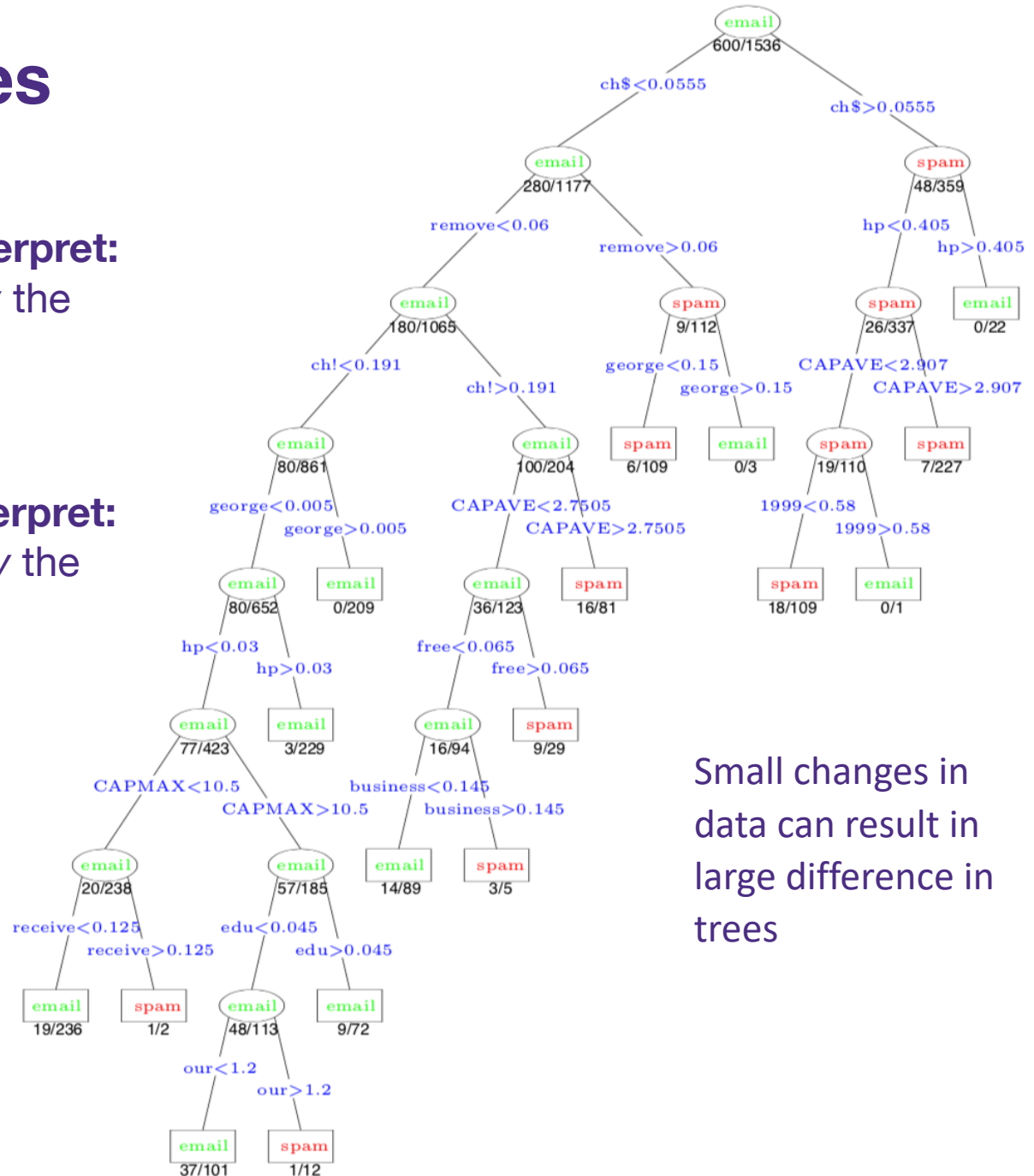
# Decision Trees

## Trees are easy to interpret:

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

## Trees are hard to interpret:

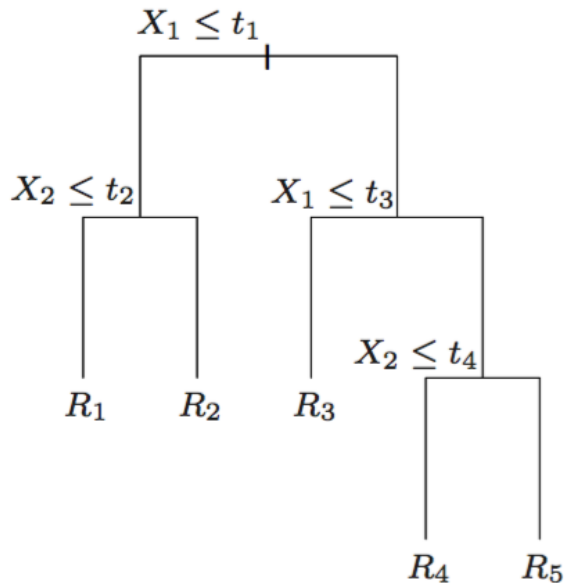
- Tough to explain *why* the classifier came to the conclusion it did



Small changes in data can result in large difference in trees

# Trees

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



- Trees
  - **have low bias, high variance**
  - deal with categorical variables well
  - intuitive, interpretable
  - good software exists
  - Some theoretical guarantees

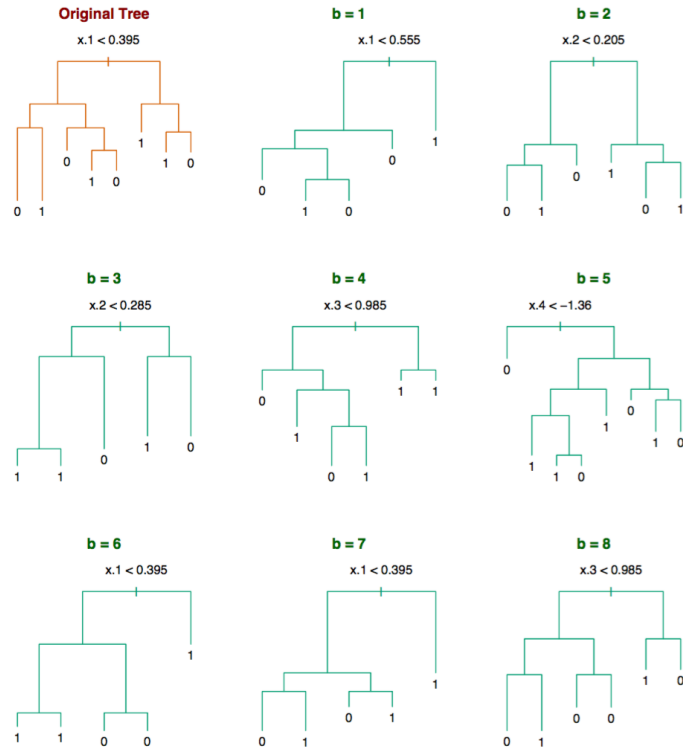
# Random Forests

---

# Random Forests

Tree methods have **low bias** but **high variance**.

One way to reduce variance is to construct a lot of “lightly correlated” trees and average them:



“Bagging:” Bootstrap aggregating

# Random Forests

---

**Algorithm 15.1** *Random Forest for Regression or Classification.*

---

1. For  $b = 1$  to  $B$ :
  - (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.
2. Output the ensemble of trees  $\{T_b\}_1^B$ .

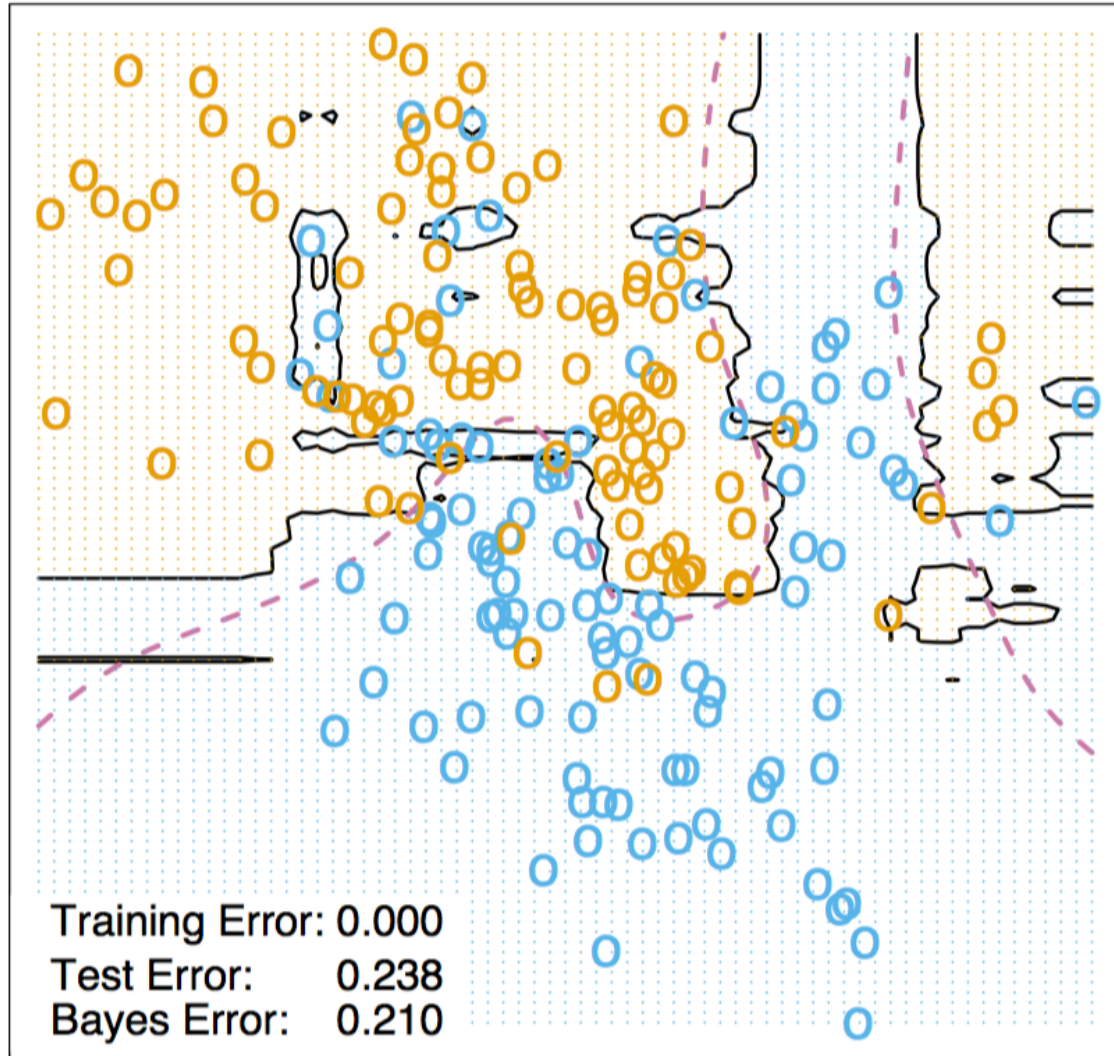
To make a prediction at a new point  $x$ :

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

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

# Random Forest - Decision Boundary Example

---



# Random Forest

---

Given random variables  $Y_1, Y_2, \dots, Y_B$  with  
 $\mathbb{E}[Y_i] = y$ ,  $\mathbb{E}[(Y_i - y)^2] = \sigma^2$ ,  $\mathbb{E}[(Y_i - y)(Y_j - y)] = \rho\sigma^2$

$\sigma^2$  Variance of individual predictor

Assume bias = 0

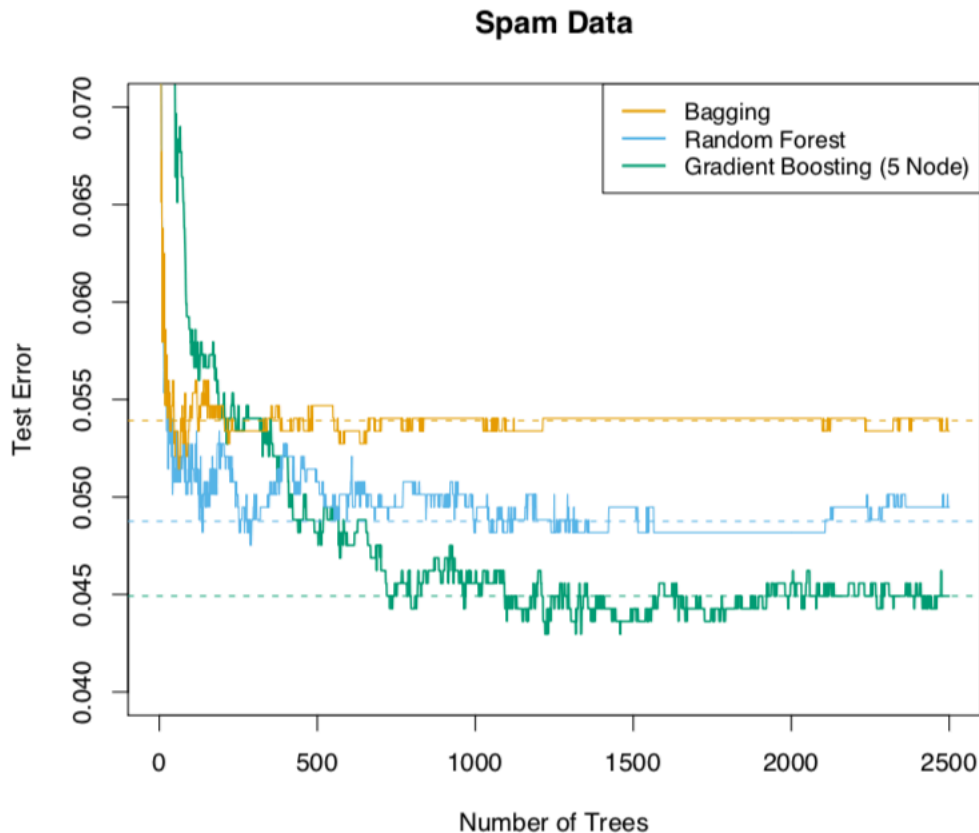
$\rho\sigma^2$  Correlation between predictors

The  $Y_i$ 's are identically distributed but **not** independent

$$\mathbb{E}\left[\left(\frac{1}{B} \sum_{i=1}^B Y_i - y\right)^2\right] =$$

# Random Forest

The power of weakly correlated predictors:



Bagging: Averaged trees trained on bootstrapped datasets that used **all  $d$  variables**

Random forest: Averaged trees trained on bootstrapped datasets that used  **$m < d$  random variables**

Gradient boosting: ignore for now

**Takeaway: reducing correlation improves performance!**

# Random Forests

---

- Random Forests
  - **have low bias, low variance**
  - deal with categorical variables well
  - not that intuitive or interpretable
  - Notion of confidence estimates
  - good software exists
  - Some theoretical guarantees
  - **works well with default hyperparameters**

# Boosting and Additive Models

---

# Boosting

---

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

## Weak learner definition (informal):

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

---

- **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

- **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  $\hat{w}_t$

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

is in general computationally hard

# Forward Stagewise Additive models

---

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

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

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

# Forward Stagewise Additive models

---

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

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  $\gamma$

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  $\gamma$       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!

# 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.
- Gradient boosting generalization with good software packages (e.g., *XGBoost*). Effective on Kaggle

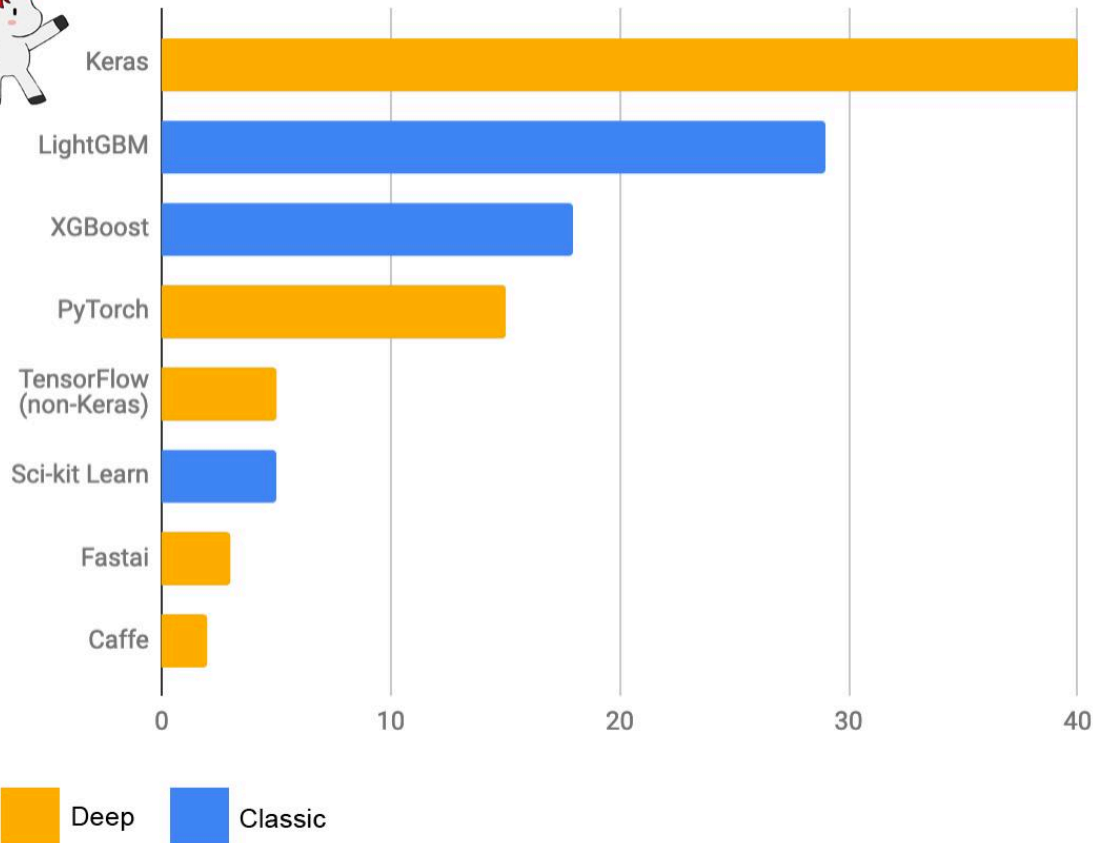
# Additive models



**François Chollet**  @fchollet · Apr 3, 2019

What machine learning tools do Kaggle champions use? We ran a survey among teams that ranked in the \*top 5\* of a competition since 2016.

**Primary ML software tool used by top-5 teams on Kaggle in each competition (n=120)**



# 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