

Examples of popular Kernels

- **Polynomials of degree exactly k**

$$K(x, x') = (x^T x')^k$$

- **Polynomials of degree up to k**

$$K(x, x') = (1 + x^T x')^k$$

- **Gaussian (squared exponential) kernel**
(a.k.a RBF kernel for Radial Basis Function)

$$K(x, x') = \exp\left(-\frac{\|x - x'\|_2^2}{2\sigma^2}\right)$$

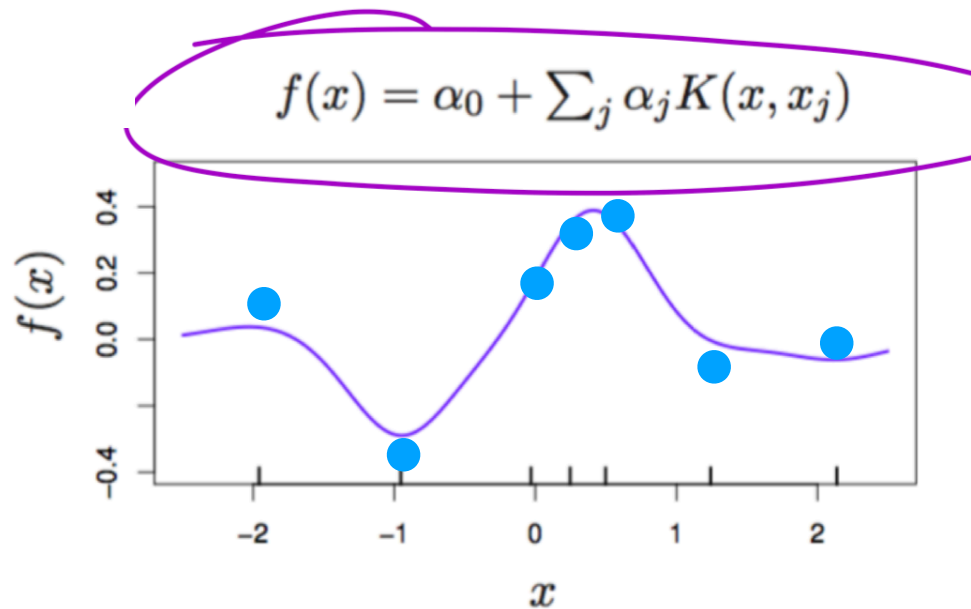
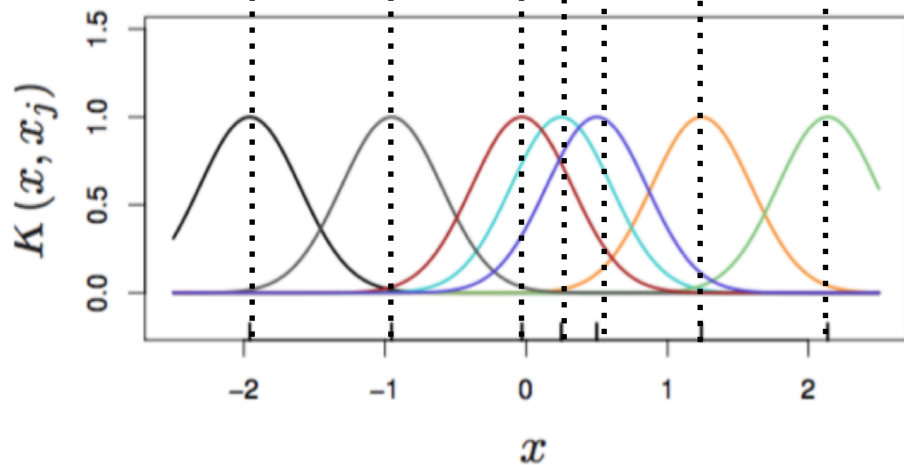
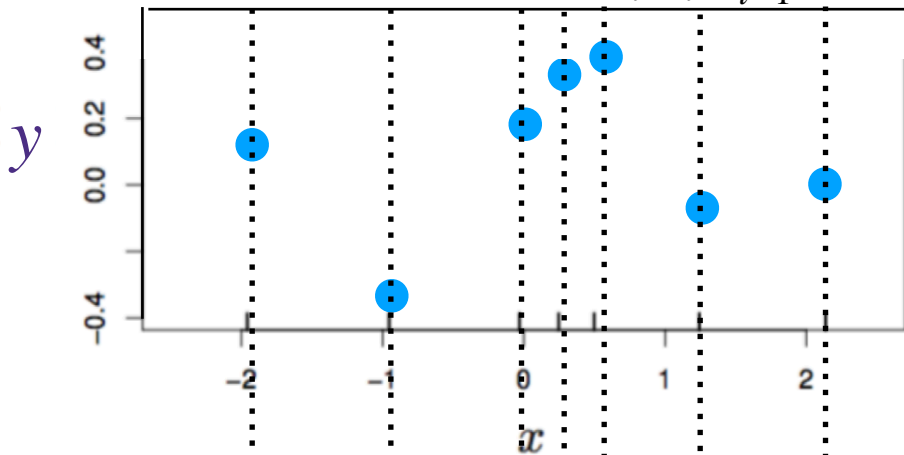
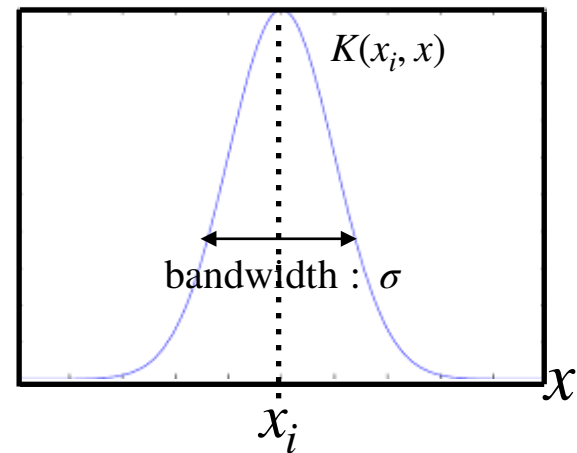
- $\phi(x) = \exp^{-x^2/2\sigma^2} \left[1, \frac{1}{\sqrt{1!\sigma^2}}x, \frac{1}{\sqrt{2!\sigma^4}}x^2, \frac{1}{\sqrt{3!\sigma^6}}x^3, \dots\right]^T$

- **Sigmoid**

$$K(x, x') = \tanh(\gamma x^T x' + r)$$

RBF kernel $k(x_i, x) = \exp\left\{-\frac{\|x_i - x\|_2^2}{2\sigma^2}\right\}$

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



- predictor $f(x) = \sum_{i=1}^n \alpha_i K(x_i, x)$ is taking weighted sum of n kernel functions centered at each sample points

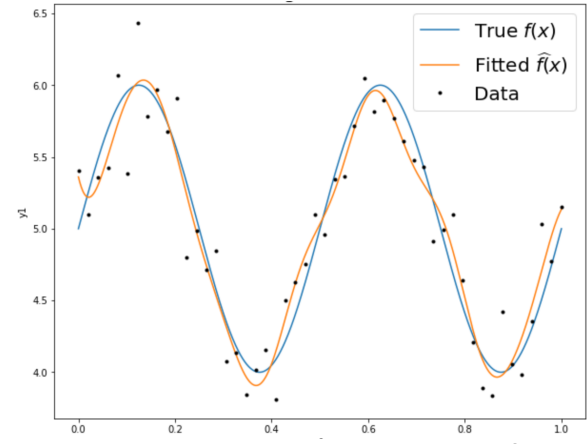
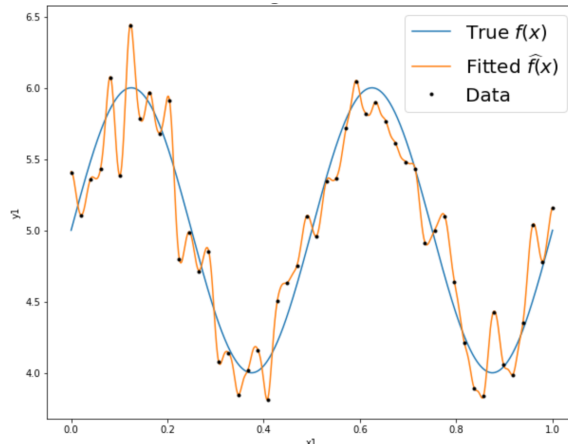
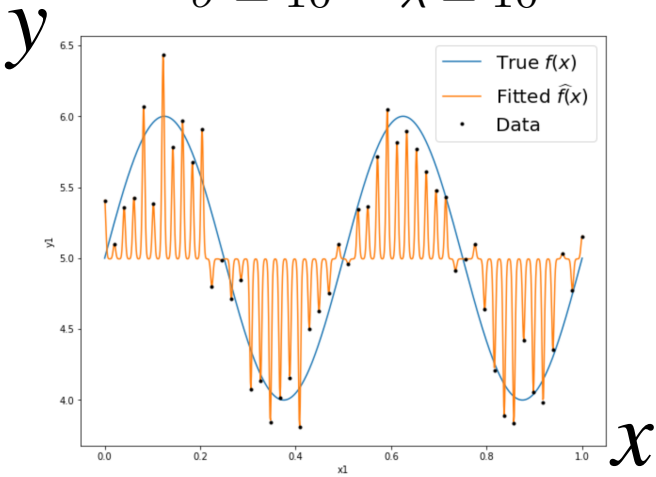
RBF kernel $k(x_i, x) = \exp\left\{-\frac{\|x_i - x\|_2^2}{2\sigma^2}\right\}$

- $\mathcal{L}(\alpha) = \|\mathbf{P}\alpha - \mathbf{y}\|_2^2 + \lambda\alpha^T\mathbf{P}\alpha$
- The bandwidth σ^2 of the kernel regularizes the predictor, and the regularization coefficient λ also regularizes the predictor

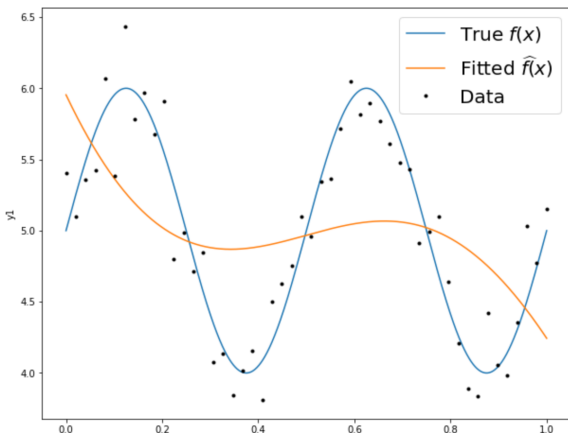
$\sigma = 10^{-3} \quad \lambda = 10^{-4}$

$\sigma = 10^{-2} \quad \lambda = 10^{-4}$

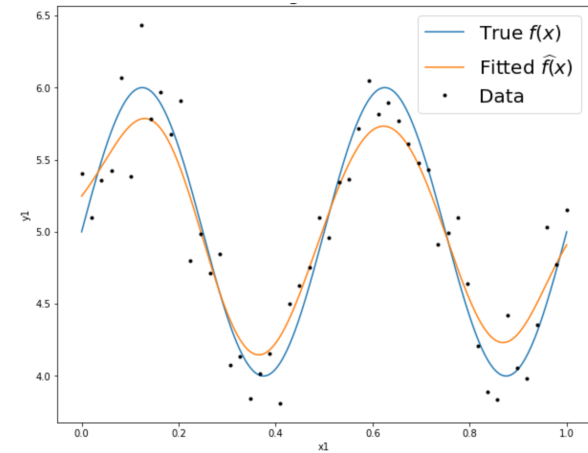
$\sigma = 10^{-1} \quad \lambda = 10^{-4}$



$\sigma = 10^{-0} \quad \lambda = 10^{-4}$



$\sigma = 10^{-1} \quad \lambda = 10^{-0}$



$$\hat{f}(x) = \sum_{i=1}^n \hat{\alpha}_i K(x_i, x)$$

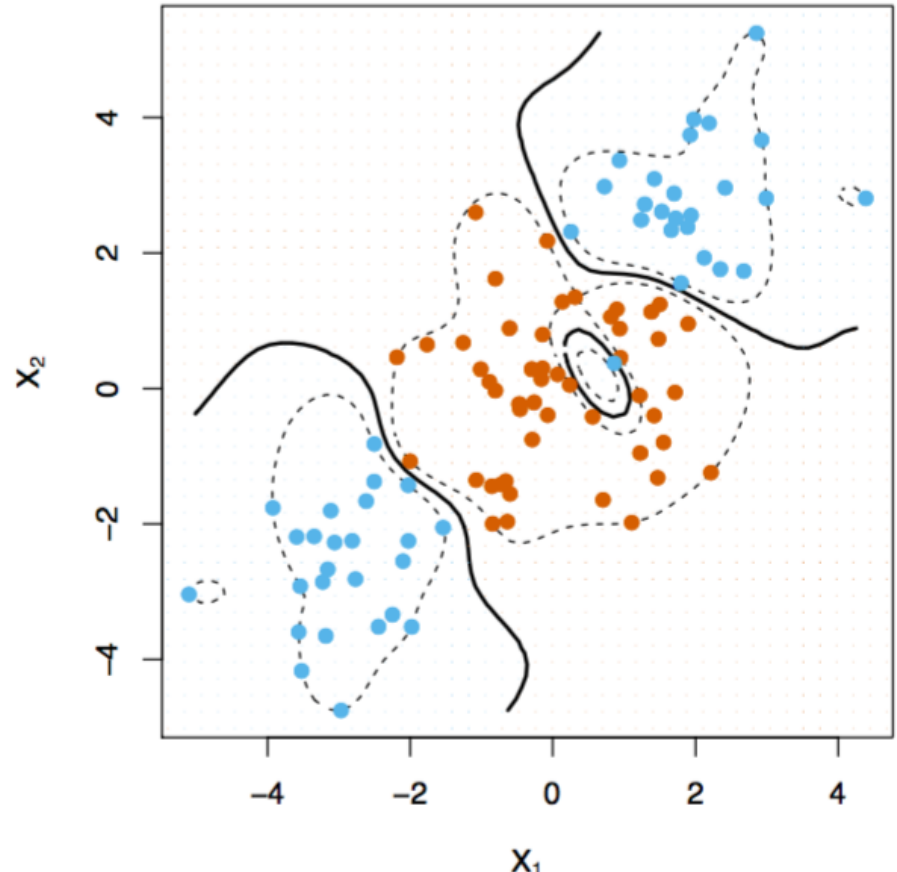
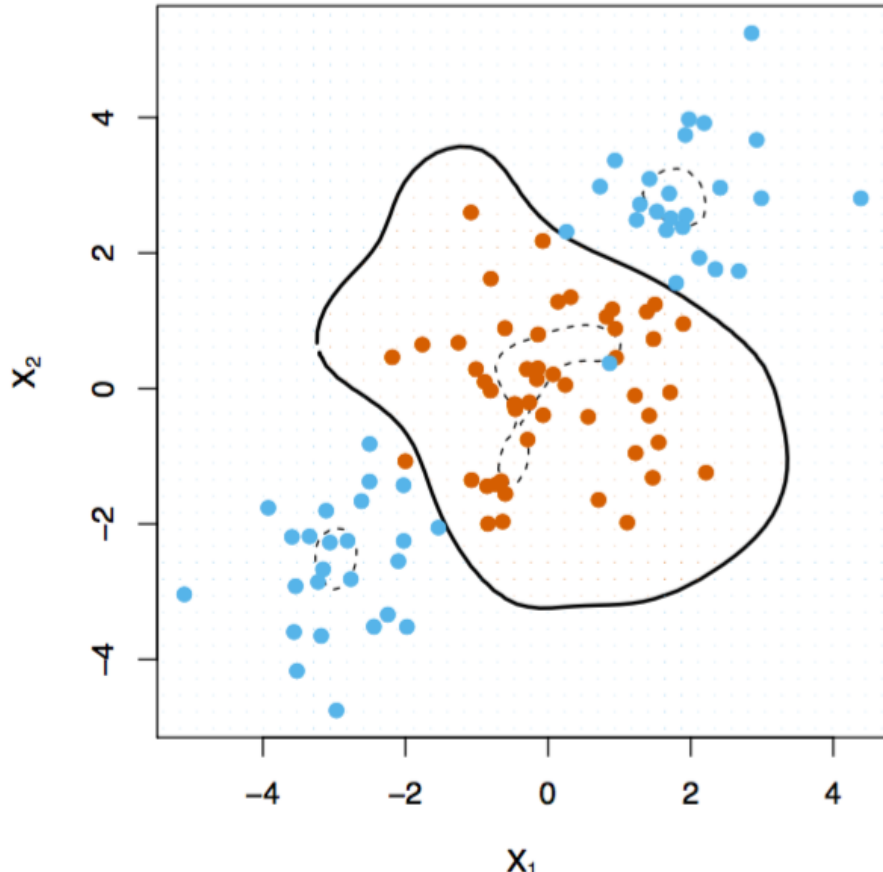
RBF kernel for SVMs

$$\hat{w} = \arg \min_{w,b} \frac{1}{n} \sum_{i=1}^n \max\{0, 1 - y_i(b + w^T x_i)\} + \lambda \|w\|_2^2$$

$$\hat{\alpha}, \hat{b} = \arg \min_{\alpha \in \mathbb{R}^n, b} \frac{1}{n} \sum_{i=1}^n \max\{0, 1 - y_i(b + \sum_{j=1}^n \alpha_j K(x_j, x_i))\} + \lambda \sum_{i=1, j=1}^n \alpha_i \alpha_j K(x_i, x_j)$$

Bandwidth σ is large enough

Bandwidth σ is small

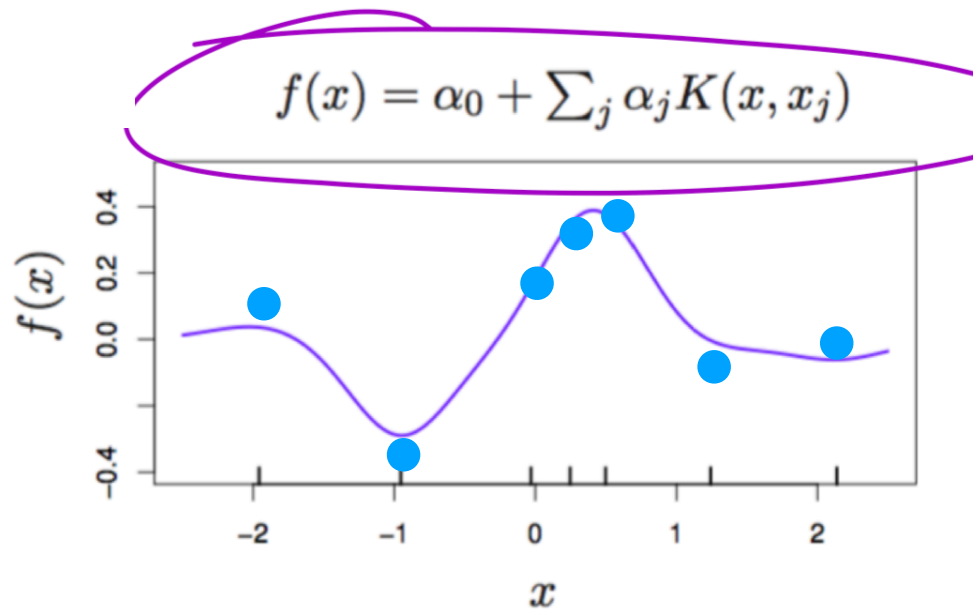
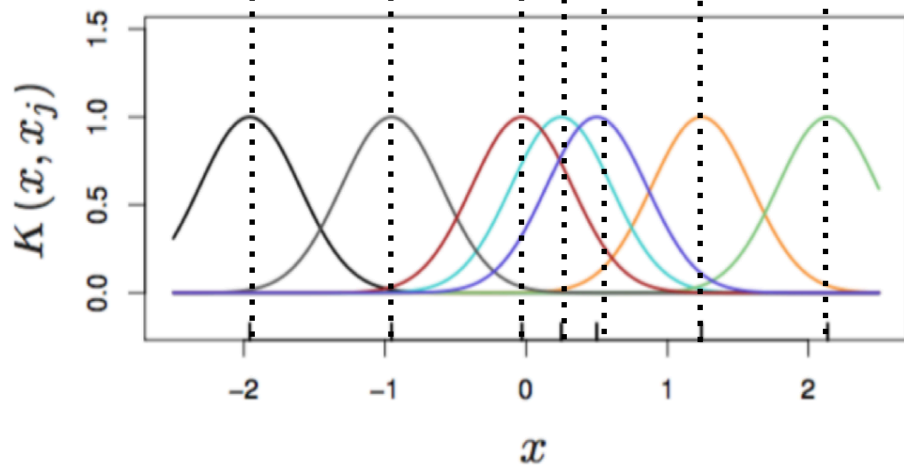
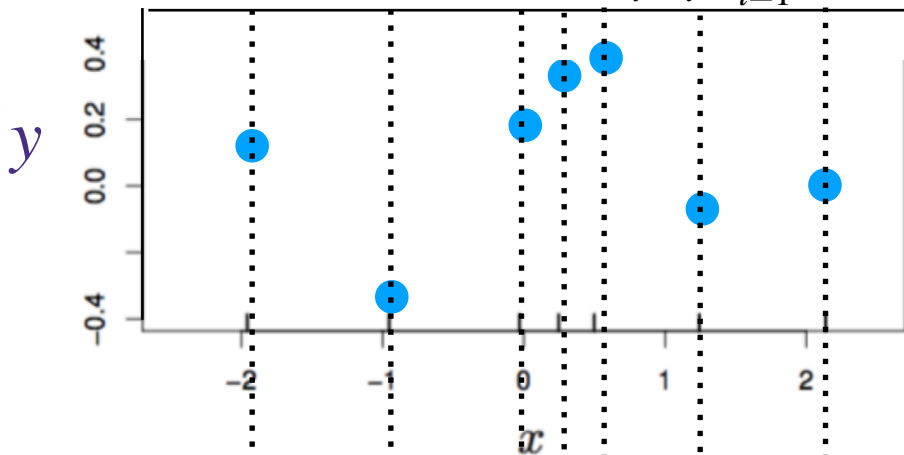
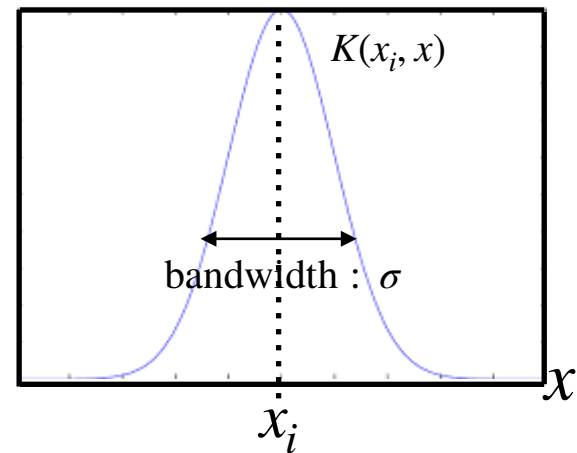


Bootstrap

W

RBF kernel $k(x_i, x) = \exp\left\{-\frac{\|x_i - x\|_2^2}{2\sigma^2}\right\}$

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



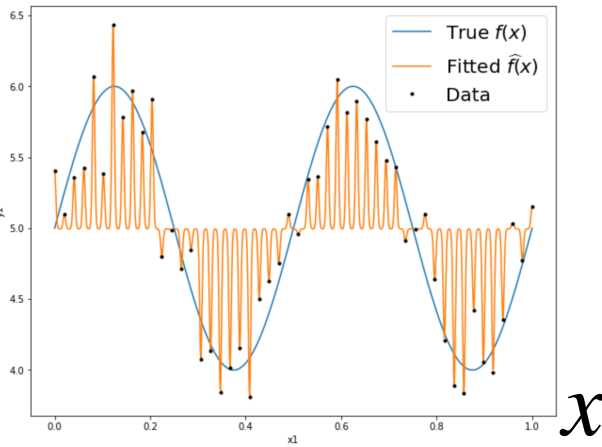
- predictor $f(x) = \sum_{i=1}^n \alpha_i K(x_i, x)$ is taking weighted sum of n kernel functions centered at each sample points

RBF kernel $k(x_i, x) = \exp\left\{-\frac{\|x_i - x\|_2^2}{2\sigma^2}\right\}$

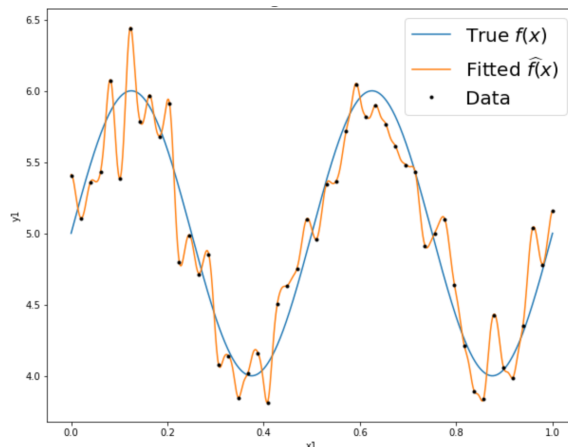
$$\mathbf{P}_{ij} = K(x_i, x_j)$$

- $\mathcal{L}(\alpha) = \|\mathbf{P}\alpha - \mathbf{y}\|_2^2 + \lambda\alpha^T\mathbf{P}\alpha$
- The bandwidth σ^2 of the kernel regularizes the predictor, and the regularization coefficient λ also regularizes the predictor

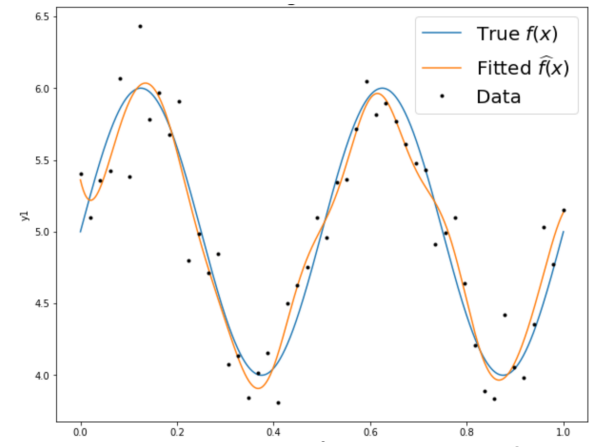
$$\sigma = 10^{-3} \quad \lambda = 10^{-4}$$



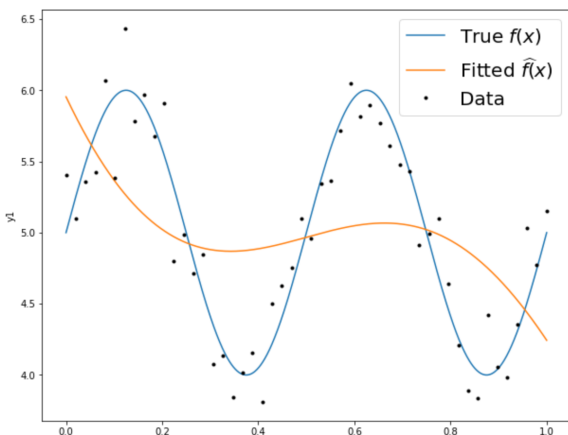
$$\sigma = 10^{-2} \quad \lambda = 10^{-4}$$



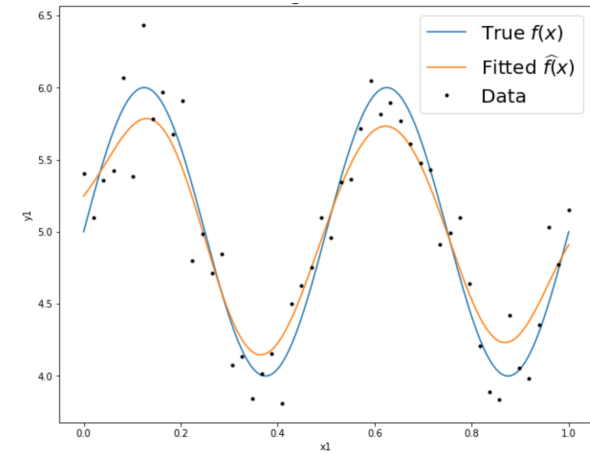
$$\sigma = 10^{-1} \quad \lambda = 10^{-4}$$



$$\sigma = 10^0 \quad \lambda = 10^{-4}$$



$$\sigma = 10^{-1} \quad \lambda = 10^0$$



$$\hat{f}(x) = \sum_{i=1}^n \hat{\alpha}_i K(x_i, x)$$

Confidence intervals

- Suppose you have training data $\{(x_i, y_i)\}_{i=1}^n$ drawn i.i.d. from some true distribution $P_{x,y}$

- We train a kernel ridge regressor, with some choice of a kernel

$$K : \mathbb{R}^{d \times d} \rightarrow \mathbb{R}, \text{ with } \mathbf{P}_{ij} = K(x_i, x_j)$$
$$\text{minimize}_{\alpha} \|\mathbf{P}\alpha - \mathbf{y}\|_2^2 + \lambda \alpha^T \mathbf{P}\alpha$$

- The resulting predictor is

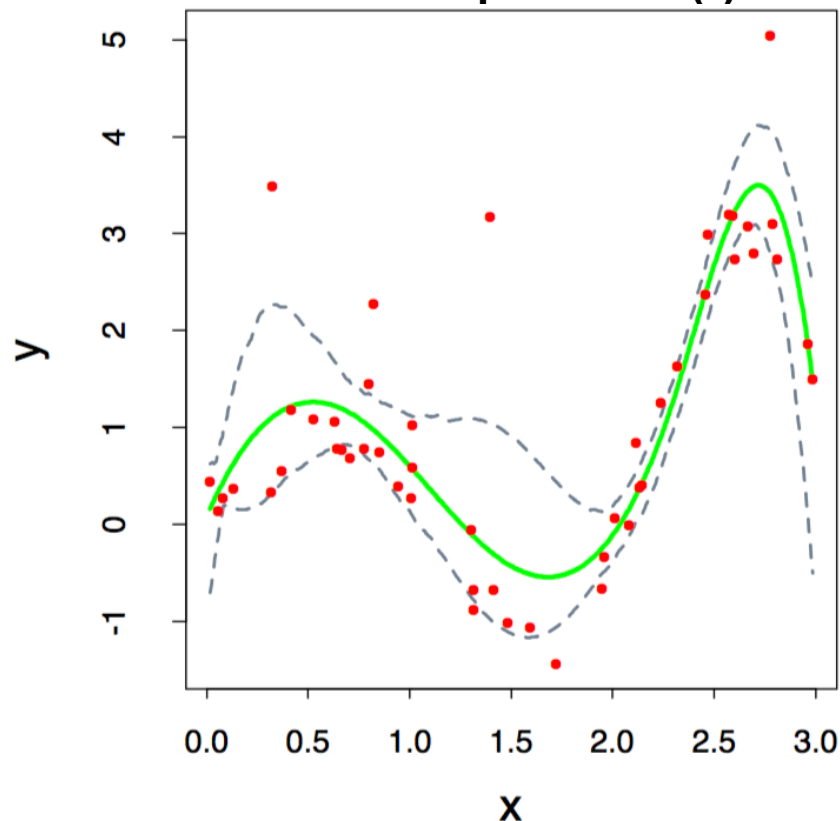
$$f(x) = \sum_{i=1}^n K(x_i, x) \hat{\alpha}_i,$$

where

$$\hat{\alpha} = (\mathbf{P} + \lambda \mathbf{I})^{-1} \mathbf{y} \in \mathbb{R}^n$$

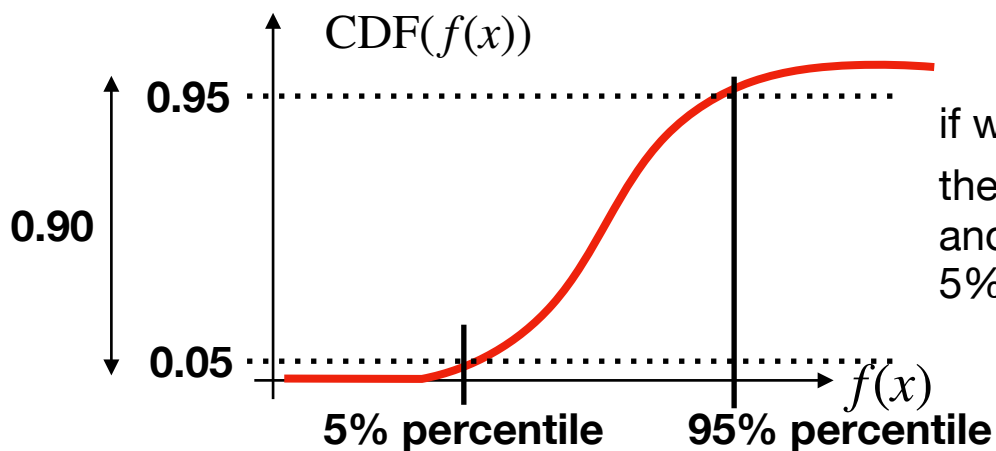
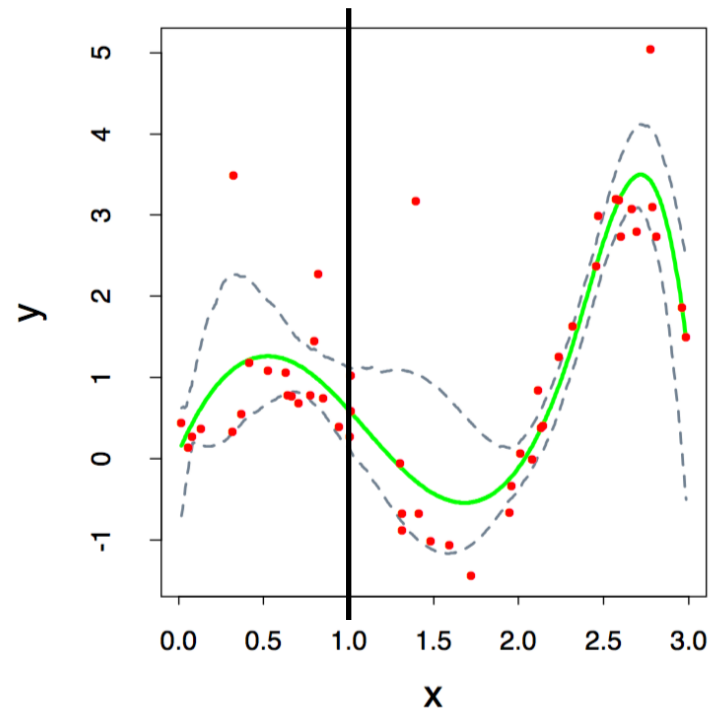
- We wish to build a confidence interval for our predictor $f(x)$, using 5% and 95% percentiles

Example of 5% and 95% percentile curves for predictor $f(x)$



Confidence intervals

- Let's focus on a single $x \in \mathbb{R}^d$
- Note that our predictor $f(x)$ is a random variable, whose randomness comes from the training data $S_{\text{train}} = \{(x_i, y_i)\}_{i=1}^n$
- If we know the statistics (in particular the CDF of the random variable $f(x)$) of the predictor, then the **confidence interval** with **confidence level 90%** is defined as



if we know the distribution of our predictor $f(x)$, the green line is the expectation $\mathbb{E}[f(x)]$ and the black dashed lines are the 5% and 95% percentiles in the figure above

- As we do not have the cumulative distribution function (CDF), we need to approximate them

Confidence intervals

- Hypothetically, if we can sample as many times as we want, then we can train $B \in \mathbb{Z}^+$ i.i.d. predictors, each trained on n fresh samples to get **empirical estimate of the CDF of $\hat{y} = f(x)$**

- For $b=1, \dots, B$

- Draw n fresh samples $\{(x_i^{(b)}, y_i^{(b)})\}_{i=1}^n$

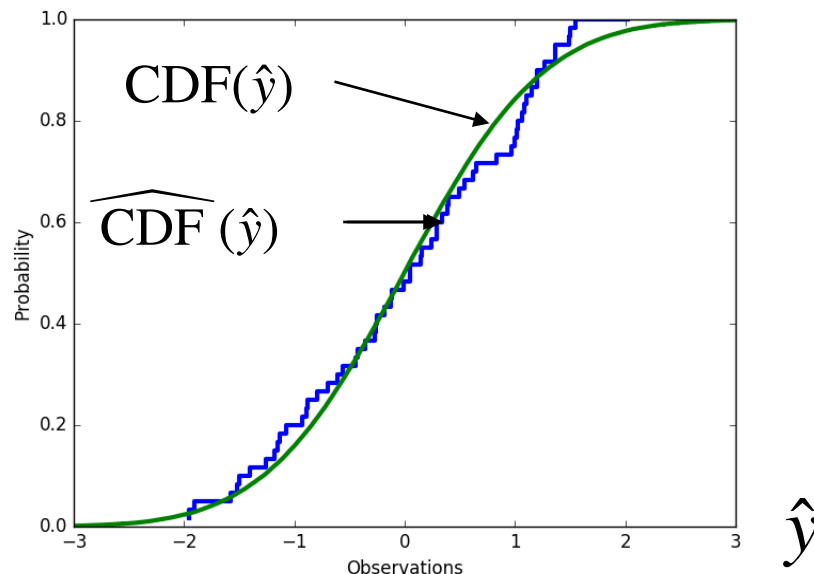
- Train a regularized kernel regression $\alpha^{*(b)}$

- Predict $\hat{y}^{(b)} = \sum_{i=1}^n K(x_i^{(b)}, x) \alpha_i^{*(b)}$

- Let the empirical CDF of those B predictors $\{\hat{y}^{(b)}\}_{b=1}^B$ be $\widehat{\text{CDF}}(\hat{y})$, defined as

$$\widehat{\text{CDF}}(\hat{y}) = \frac{1}{B} \sum_{b=1}^B \mathbf{I}\{\hat{y}^{(b)} \leq \hat{y}\}$$

- Compute the confidence interval using $\widehat{\text{CDF}}(\hat{y})$

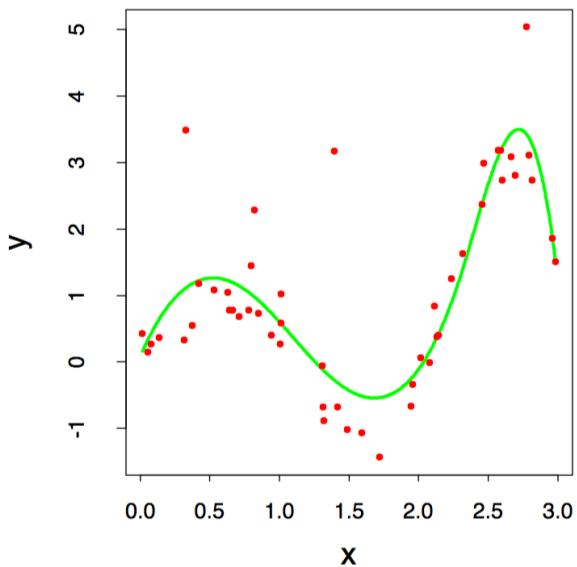


Bootstrap

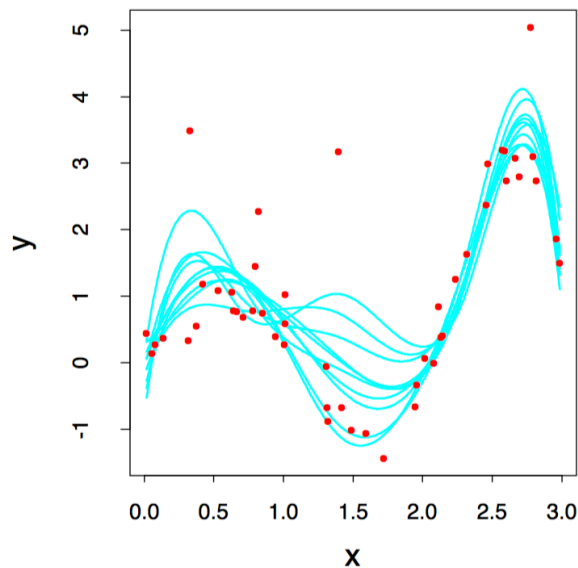
- As we cannot sample repeatedly (in typical cases), we use **bootstrap samples** instead
- Bootstrap is a general tool for assessing statistical accuracy
- We learn it in the context of confidence interval for trained models
- A **bootstrap dataset** is created from the training dataset by taking n (the same size as the training data) examples uniformly at random **with replacement** from the training data $\{(x_i, y_i)\}_{i=1}^n$
- For $b=1, \dots, B$
 - Create a bootstrap dataset $\mathcal{S}_{\text{bootstrap}}^{(b)}$
 - Train a regularized kernel regression $\alpha^{*(b)}$
 - Predict $\hat{y}^{(b)} = \sum_{i=1}^n K(x_i^{(b)}, x) \alpha_i^{*(b)}$
- Compute the empirical CDF from the bootstrap datasets, and compute the confidence interval

Bootstrap

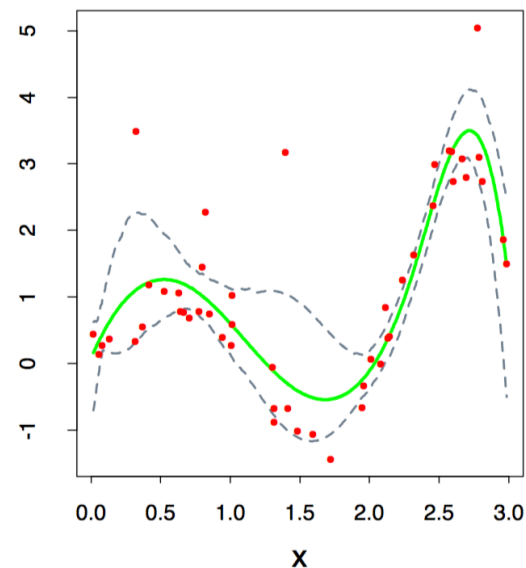
training a single predictor



multiple bootstrapped predictors



90% confidence interval



Figures from Hastie et al

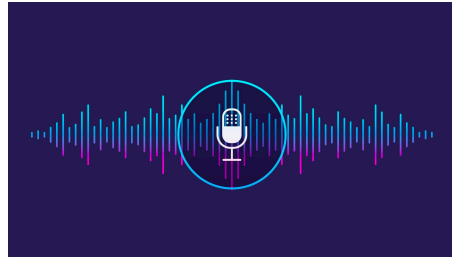
Neural Networks



Applications of Neural Networks



Self-driving cars



Voice assistants



Machine translation

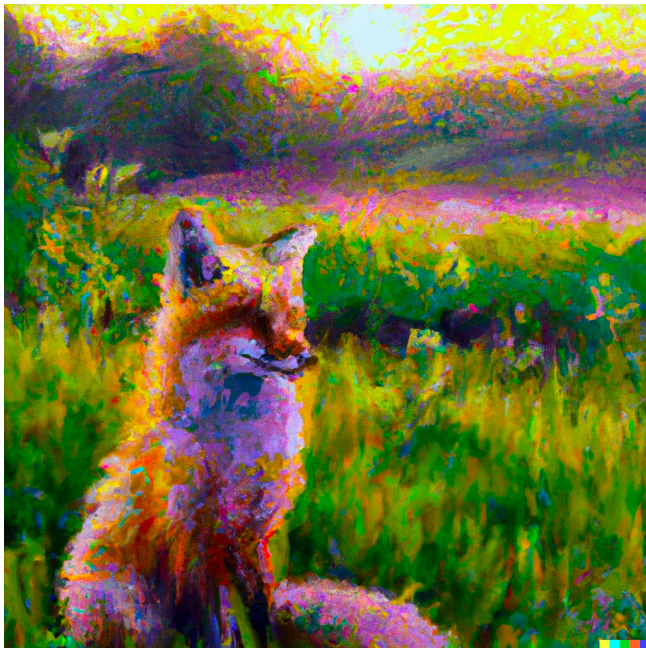


Image generation

“a painting of a fox sitting in a field at sunrise in the style of Claude Monet”

+ many more (images, text, audio)

BUT: Simple methods often still
the best on tabular data.

Neural Networks

- Origins: Algorithms that try to mimic the brain.
- Widely used in 80s and early 90s; popularity diminished in late 90s.
- Recent resurgence from 2010s: state-of-the-art techniques for many applications:
 - Computer Vision (AlexNet 2012)
 - Natural language processing
 - Speech recognition
 - Decision-making / control problems (AlphaGo, Games, robots)
- Limited theory:
 - Why do we find good minima with SGD for Non-convex loss?
 - Why do we not overfit when # of parameters p is much larger than # of samples n ?

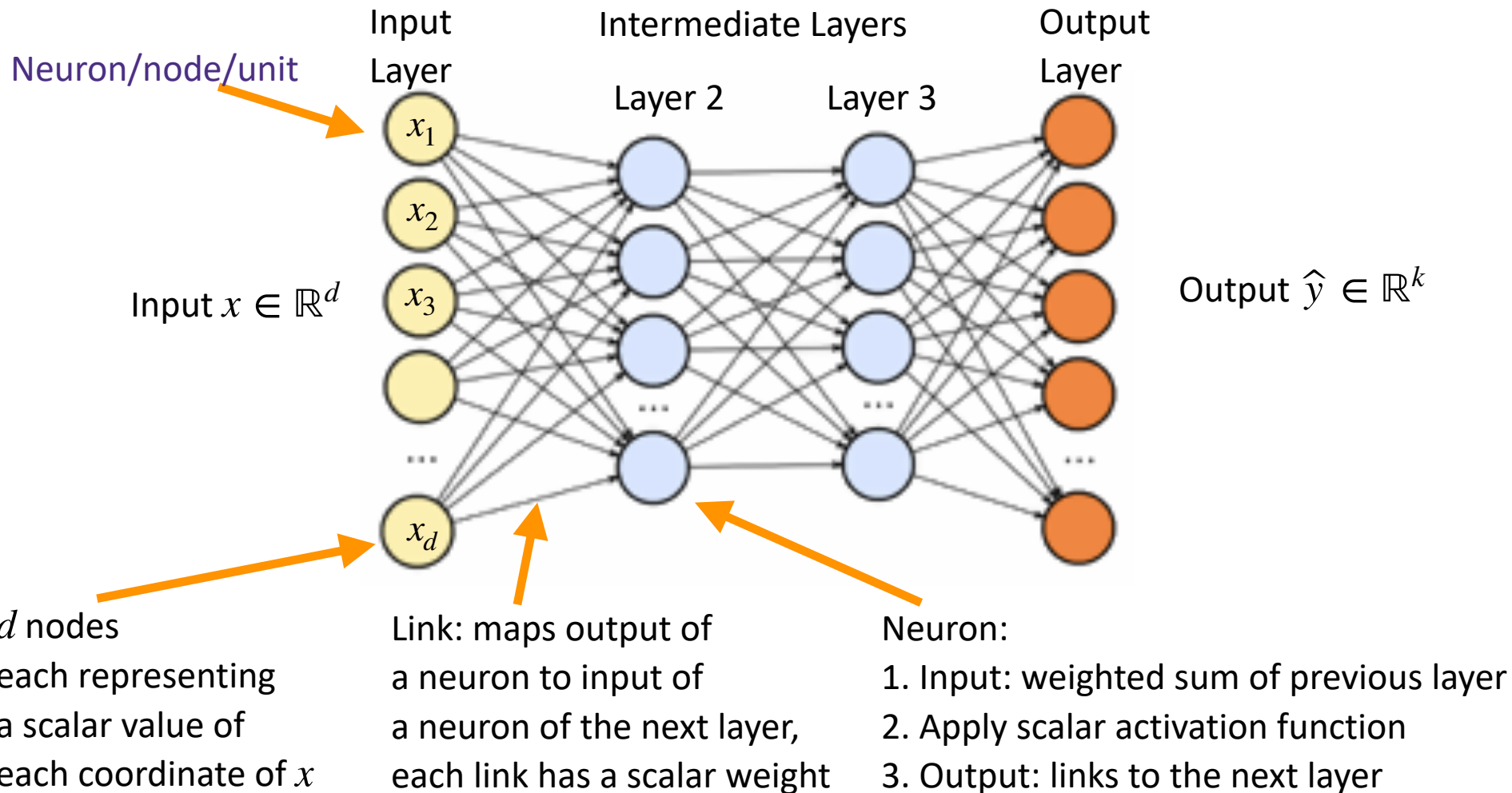
Neural Networks

Agenda:

1. Definitions of neural networks
2. Training neural networks:
 1. Algorithm: back propagation
 2. Putting it to work
3. Neural network architecture design:
 1. Convolutional neural network

Neural Networks

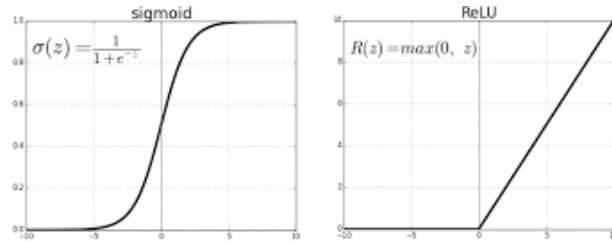
- **Neural Network** is a parametric family of functions from $x \in \mathbb{R}^d$ to $\hat{y} = h_\theta(x) \in \mathbb{R}^k$ with parameter $\theta \in \mathbb{R}^p$
- **Computation graph** illustrates the sequence of operations to be performed by a neural network



Sequence of operations performed at a single node

- For a single node with input $x \in \mathbb{R}^d$, the node is defined by

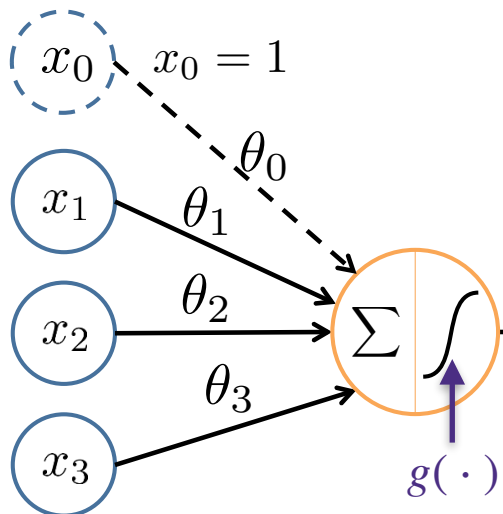
- Parameters $\theta \in \mathbb{R}^{d+1}$ (including the intercept/bias)
- Activation function $g : \mathbb{R} \rightarrow \mathbb{R}$



- A common choice is sigmoid function: $g(z) = \frac{1}{1 + e^{-z}}$
- Another popular choice is Rectified Linear Unit (ReLU): $g(z) = \max\{0, z\}$

- The node performs $h_{\theta}(x) = g\left(\sum_{i=0}^d \theta_i x_i\right) = g(\theta^T x)$

“bias unit”

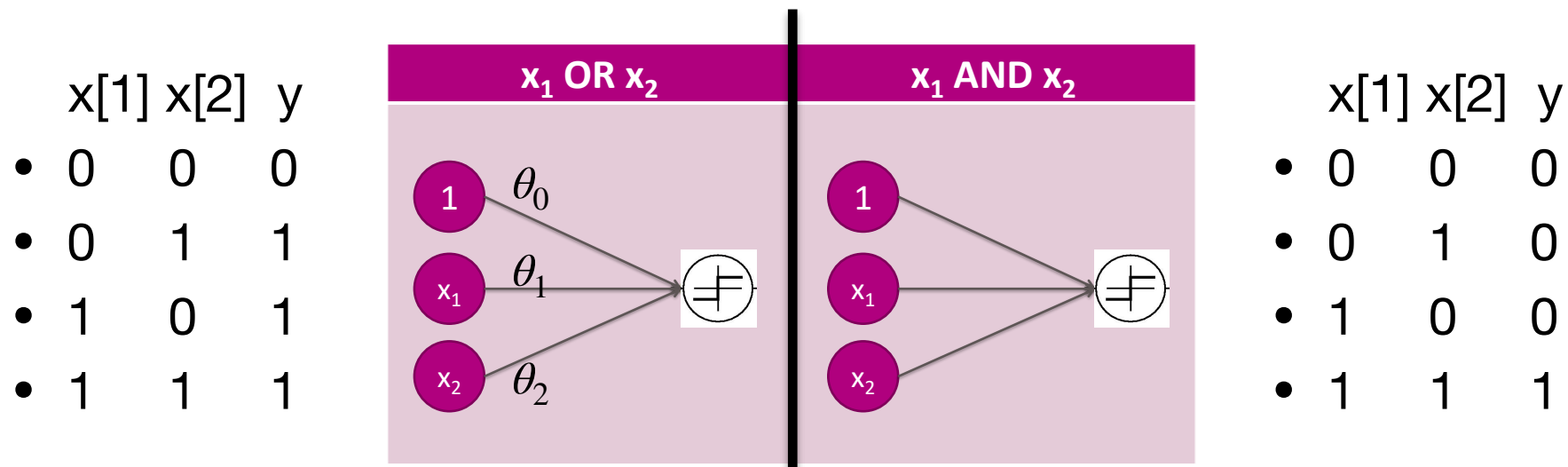


$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$

$$h_{\theta}(\mathbf{x}) = g(\theta^T \mathbf{x})$$

(= $\frac{1}{1 + e^{-\theta^T \mathbf{x}}}$ for sigmoid)

Toy example: What can be represented by a single node with $g(z) = \text{sign}(z)$?



What should be the weights?

$$f_{\theta}(x) = \text{sign}(\theta_0 + \theta_1 x[1] + \theta_2 x[2])$$

$$f_{\theta}(x) = \text{sign}(\theta_0 + \theta_1 x[1] + \theta_2 x[2])$$

Note that there is a one-to-one correspondence between a **linear classifier** and a **neural network with a single node** of the above form

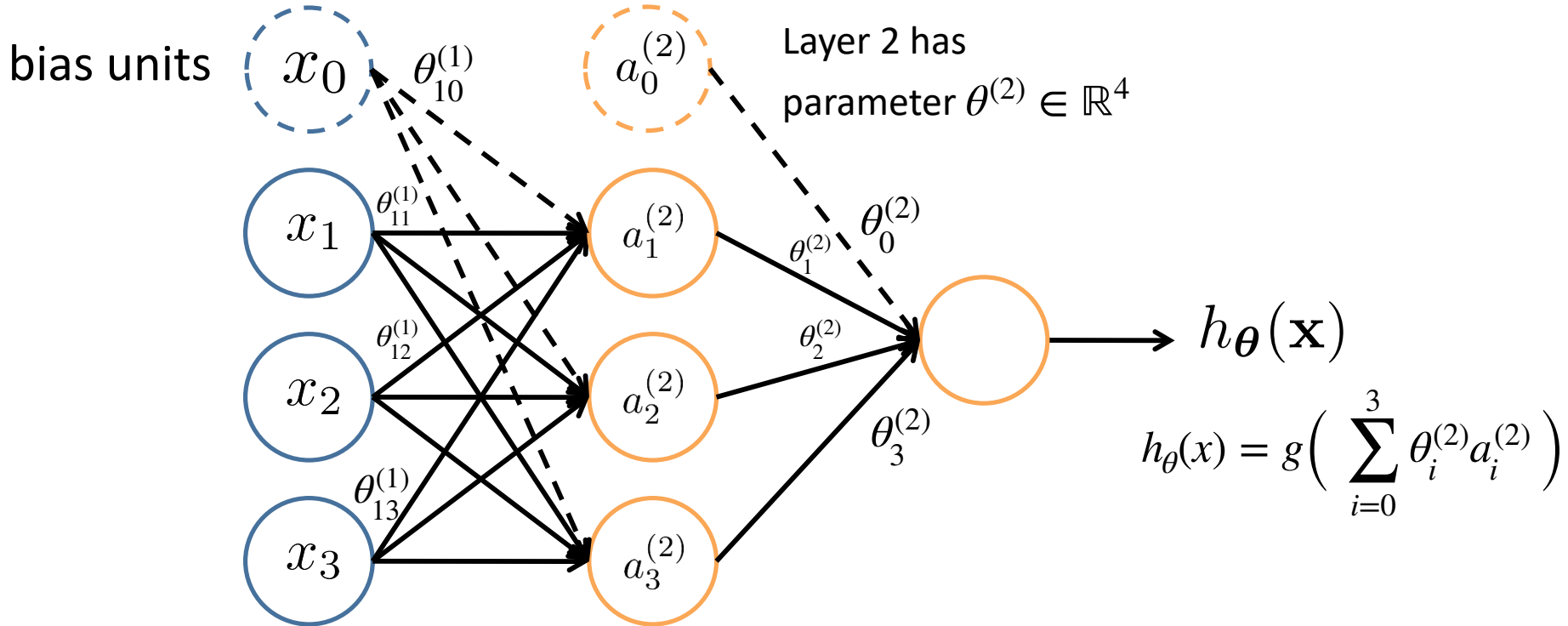
What cannot be learned?

Neural Network composes simple functions to make complex functions

- Each layer performs simple operations
- Neural Network (with parameter $\theta = (\theta^{(1)}, \theta^{(2)})$) composes multiple layers of operations

Layer 1 has parameter $\theta^{(1)} \in \mathbb{R}^{3 \times 4}$

$$a_1^{(2)} = g\left(\sum_{i=0}^3 \theta_{1i}^{(1)} x_i\right)$$



Layer 1

(Input Layer)

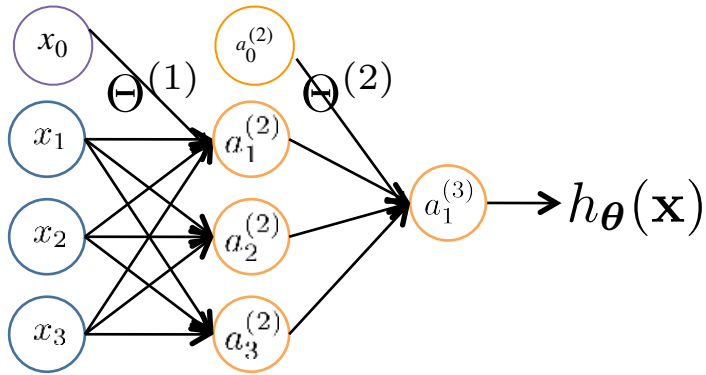
Layer 2

(Hidden Layer)

Layer 3

(Output Layer)

This is called
a 2-layer Neural Network



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

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

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

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

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

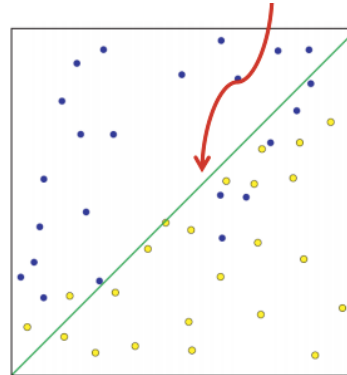
If network has s_j units in layer j and s_{j+1} units in layer $j+1$, then $\Theta^{(j)}$ has dimension $s_{j+1} \times (s_j+1)$.

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

Example of 2-layer neural network in action

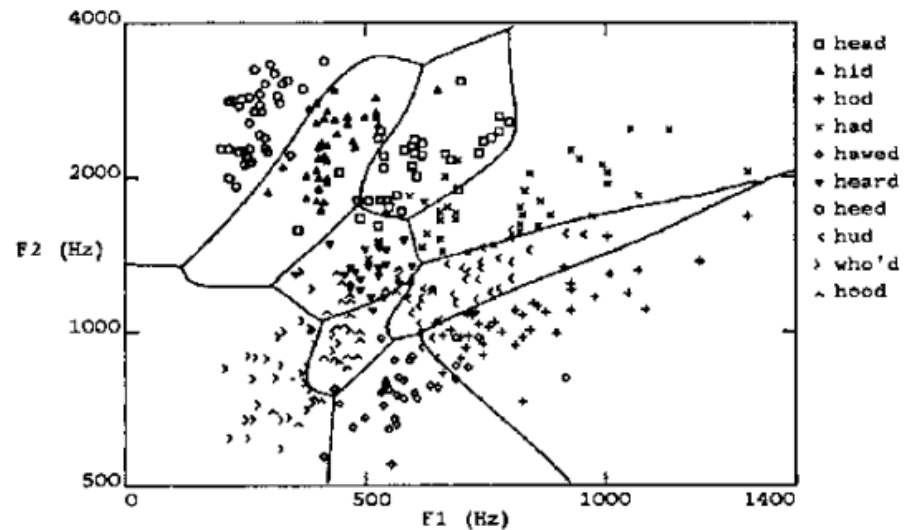
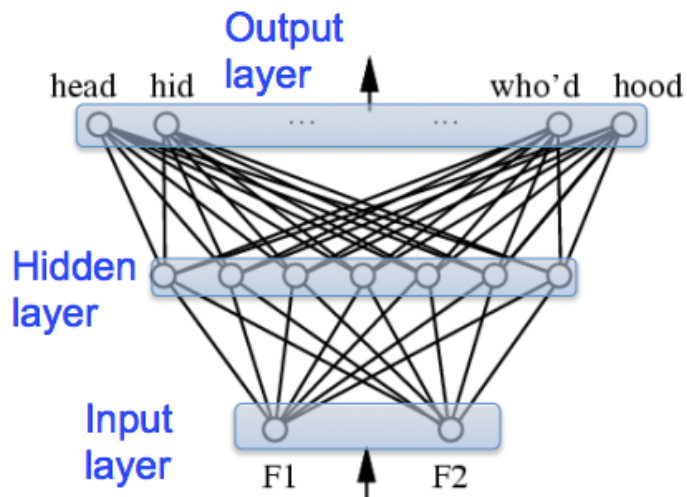
Linear decision boundary

1-layer neural networks
only represents linear classifiers



Example: 2-layer neural network trained to distinguish vowel sounds using 2 formants (features)

A highly non-linear decision boundary can be learned from 2-layer neural networks



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)

But Deep Neural Networks have many powerful properties not yet understood theoretically.

Multi-layer Neural Network - Binary Classification in $\{0,1\}$

L -th layer plays the role of features, but trained instead of pre-determined

This is a 5-dimensional vector

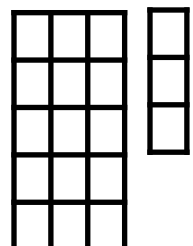
$$a^{(1)} = x$$

↓

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

↗

Scalar function g is applied coordinate-wise

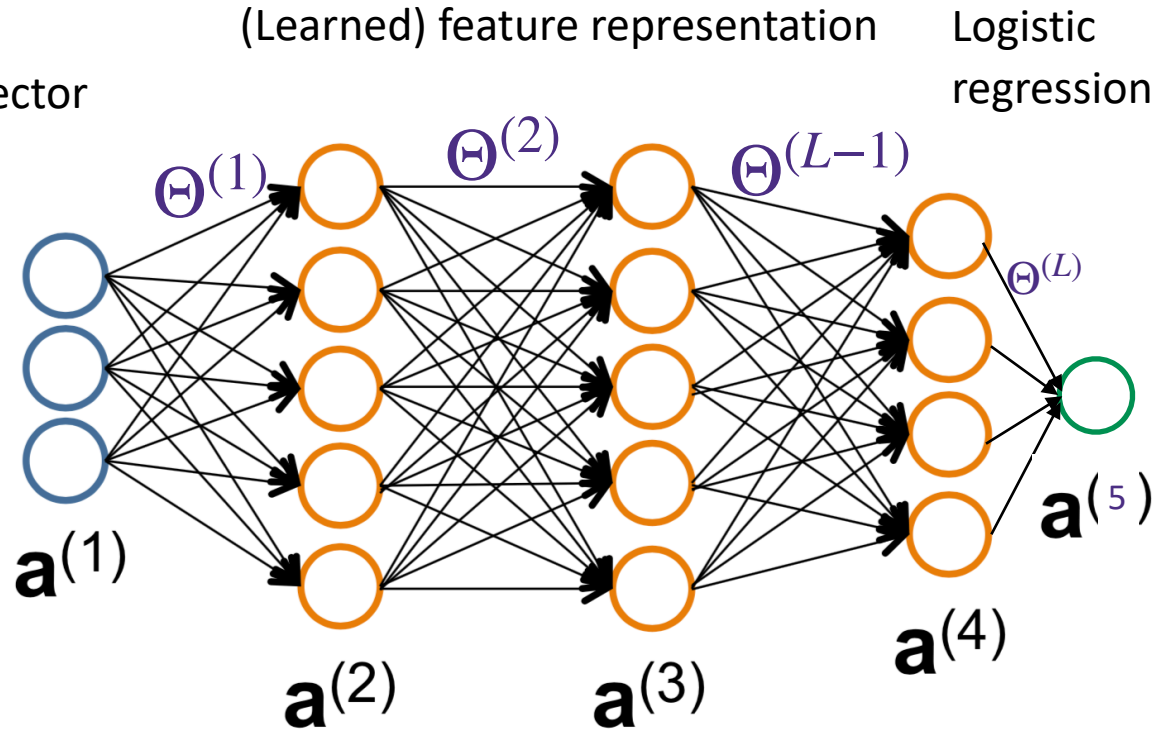


⋮

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

⋮

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



Cross entropy loss:

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

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

**Binary Logistic Regression
with learned feature $a^{(4)}$**

Multi-layer Neural Network - Binary Classification

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

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

ReLU

⋮

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

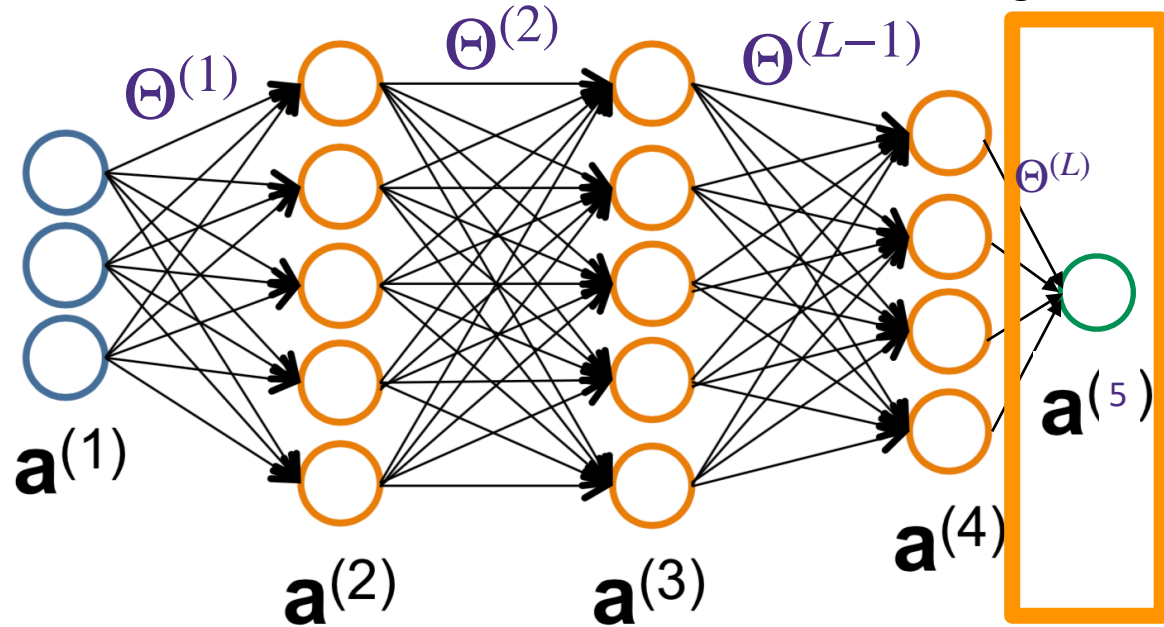
⋮

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

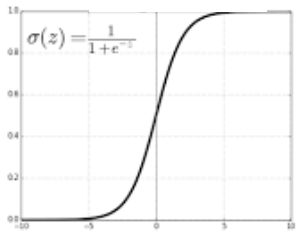
Sigmoid

(Learned) feature representation

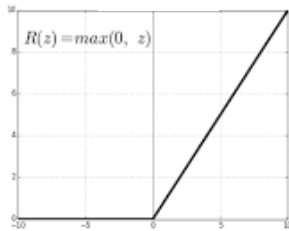
Logistic regression



Sigmoid



ReLU



- Why is ReLU better than sigmoid?

Cross entropy loss:

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

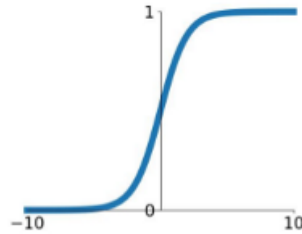
$$\sigma(z) = \max\{0, z\} \quad g(z) = \frac{1}{1 + e^{-z}} \quad \begin{array}{l} \text{Binary} \\ \text{Logistic} \\ \text{Regression} \end{array}$$

Nonlinear activation function

- popular choices of activation function includes

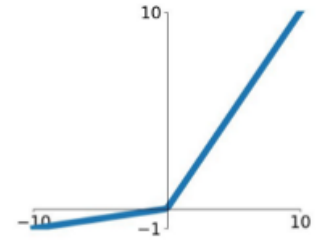
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



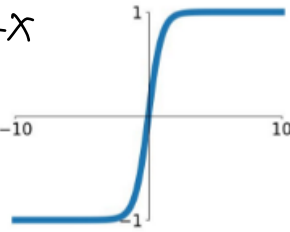
Leaky ReLU

$$\max(0.1x, x)$$



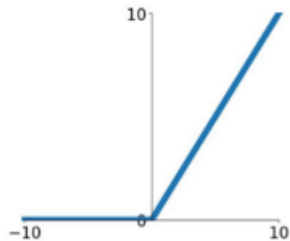
tanh

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



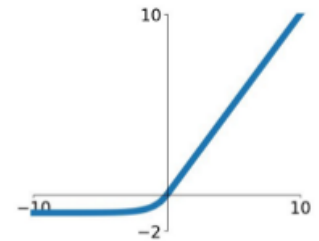
ReLU

$$\max(0, x)$$



ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



- Why is ReLU better than Sigmoid?
- Why is ELU better than ReLU?

K -class Classification: multiple output units



Pedestrian



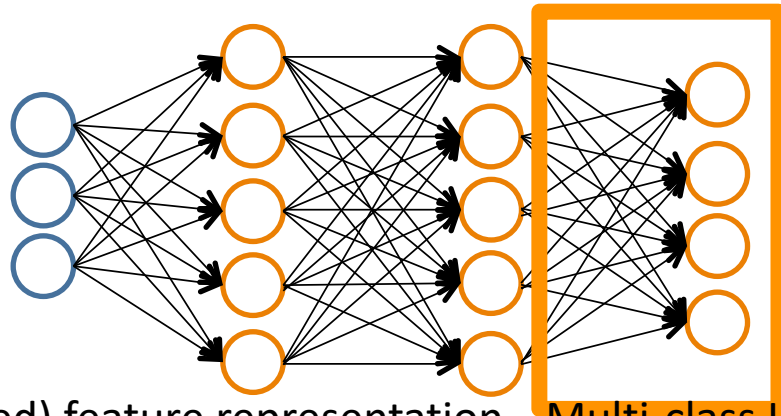
Car



Motorcycle



Truck



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

Multi-class
Logistic
Regression

(Learned) feature representation

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

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

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

⋮

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

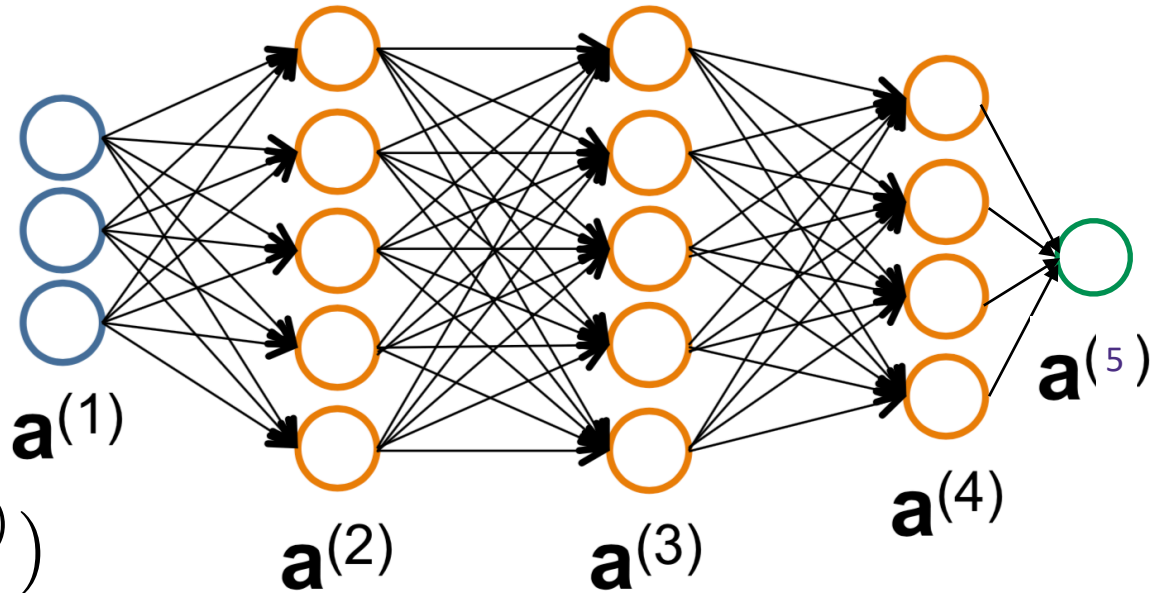
⋮

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

Linear model

(Learned) feature representation

Logistic
regression



Square loss:

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

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

Training Neural Networks

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

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

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

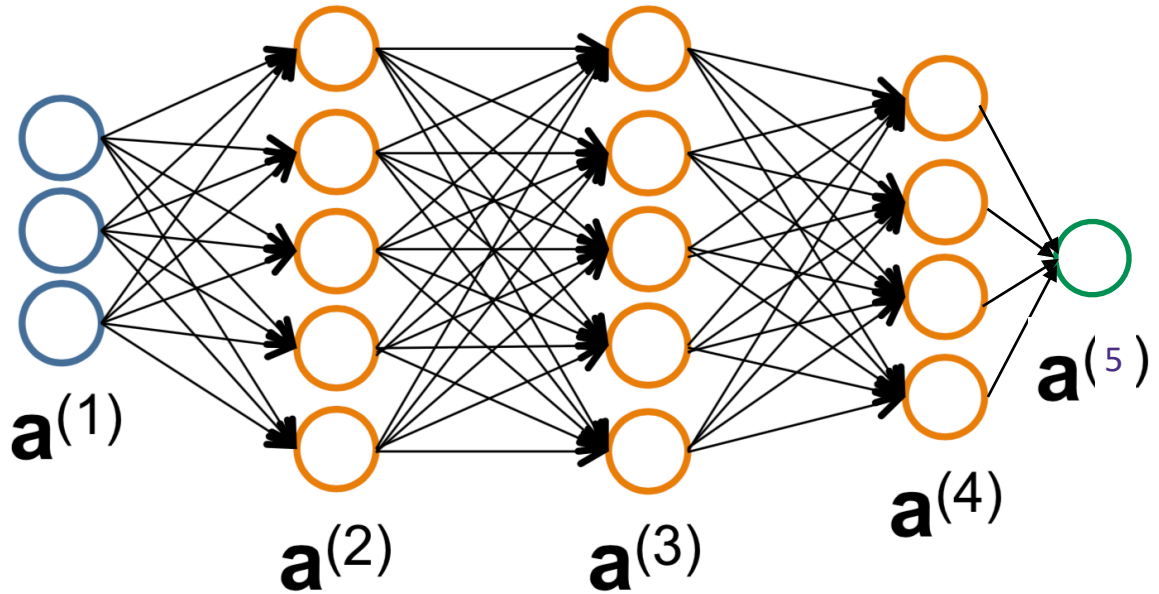
$$\vdots$$

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

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

$$\vdots$$

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



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

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

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

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

Seems simple enough - what do packages like PyTorch, Tensorflow, Jax, Theano, Caffe, MxNet provide?

1. Automatic differentiation
 1. Given a NN, compute the gradient automatically
 2. Compute the gradient efficiently
2. Convenient libraries
 1. Set-up NN
 2. Choose algorithms (SGD,Adam,etc.) for training
 3. Hyper-parameter tuning
3. GPU support
 1. Linear algebraic operations

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
```


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** and **batchsize** 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!*
- Start from a code that someone else has tried and tested

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 advanced 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

Back Propagation

W

Forward Propagation

- We are not writing the intercept at each layer for simplicity
- To compute gradients, we first run forward pass to get the intermediate representations $\{a^{(2)}, \dots, a^{(L)}\}$

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

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

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

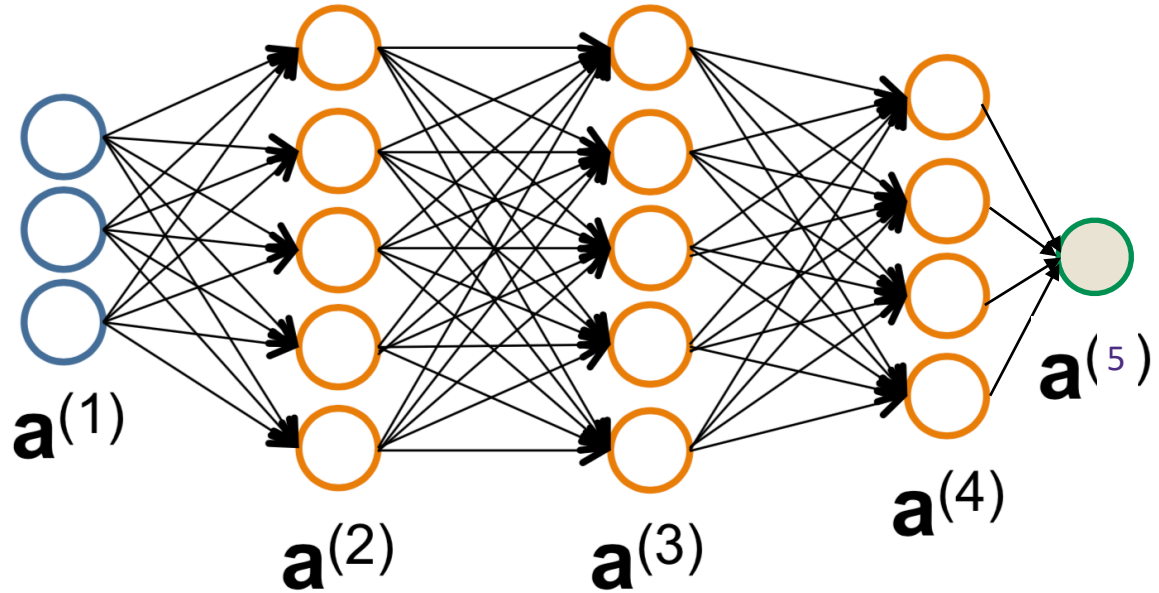
$$\vdots$$
$$a^{(l)} = g(z^{(l)})$$

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

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

$$\vdots$$

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



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

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

Backprop

- Parameters: $\Theta^{(1)} \in \mathbb{R}^{m \times d}$, $\Theta^{(2)}, \dots, \Theta^{(L-1)} \in \mathbb{R}^{m \times m}$
- Naive implementation takes $O(L^2)$ time, as each layer requires a full forward pass (with $O(L)$ operations) and some backward pass
- Backprop requires only $O(L)$ operations

$$a^{(1)} = x \in \mathbb{R}^d$$

$$z^{(2)} = \Theta^{(1)} a^{(1)} \in \mathbb{R}^m$$

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

⋮

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

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

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

⋮

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

Train by Stochastic Gradient Descent:

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

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

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

Backprop

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

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

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

⋮

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

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

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

⋮

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

Recursively
computed in
one backward pass

Computed
in the
forward pass

- Chain rule with $z_i^{(l+1)} = \Theta_{i,j}^{(l)} a_j^{(l)}$

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

Train by Stochastic Gradient Descent:

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

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

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

$$\delta_i^{(l+1)} \triangleq \frac{\partial L(y, \hat{y})}{\partial z_i^{(l+1)}}$$

Backprop

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

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

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

$$\vdots$$

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

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

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

$$\vdots$$

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

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

$$\delta_i^{(l)} = \frac{\partial L(y, \hat{y})}{\partial z_i^{(l)}} = \sum_k \underbrace{\frac{\partial L(y, \hat{y})}{\partial z_k^{(l+1)}}}_{\delta_k^{(l+1)}} \cdot \underbrace{\frac{\partial z_k^{(l+1)}}{\partial z_i^{(l)}}}_{\Theta_{k,i}^{(l)} g'(z_i^{(l)})}$$

$$z_k^{(l+1)} = \sum_{i=1}^m \Theta_{k,i}^{(l)} g(z_i^{(l)})$$

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

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

Backprop

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

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

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

⋮

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

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

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

⋮

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

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

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

$$= \sum_k \delta_k^{(l+1)} \cdot \Theta_{k,i}^{(l)} g'(z_i^{(l)})$$

Computed
in the
forward pass

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

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

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

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

$$g'(z) = g(z)(1 - g(z))$$

Backprop

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

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

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

⋮

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

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

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

⋮

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

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

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

- We can recursively compute all $\delta^{(\ell)}$'s in a single backward pass
- And compute all gradients via

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

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

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

Backprop

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

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

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

⋮

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

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

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

⋮

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

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

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

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

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

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

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

$$g'(z) = g(z)(1 - g(z))$$

Backprop

Recursive Algorithm!

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

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

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

⋮

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

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

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

⋮

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

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

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

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

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

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

Backpropagation

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

For each training instance (x_k, y_k)

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

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

Intercept do not
have regularizer

Average loss + ℓ_2 regularizer

$$\frac{1}{n} \sum_{k=1}^n L(y_k, \hat{y}) + \lambda \|\Theta\|_2^2$$