

Today: Outline

- **Overfitting**
- **Neural networks:** artificial neuron, MLP, sigmoid units; neuroscience inspiration, output vs hidden layers; linear vs nonlinear networks;
- **Feed-forward networks**
- **Reminders:** *PS1, will be posted on Friday*



Supervised Learning

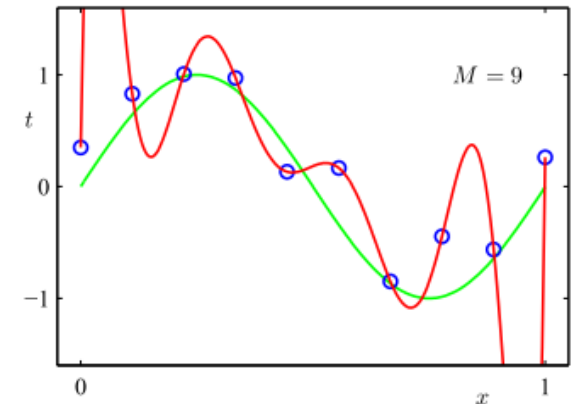
Overfitting

Overfitting

Parameters for higher-order polynomials are very large

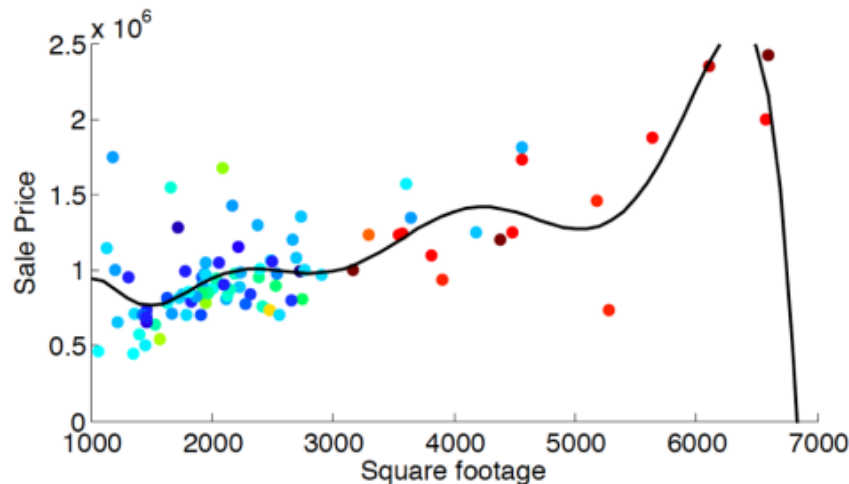
	M = 0	M = 1	M = 3	M = 9
θ_0	0.19	0.82	0.31	0.35
θ_1		-1.27	7.99	232.37
θ_2			-25.43	-5321.83
θ_3			17.37	48568.31
θ_4				-231639.30
θ_5				640042.26
θ_6				-1061800.52
θ_7				1042400.18
θ_8				-557682.99
θ_9				125201.43

M=9: *overfitting*



Overfitting disaster

Fitting the housing price data with $M = 3$

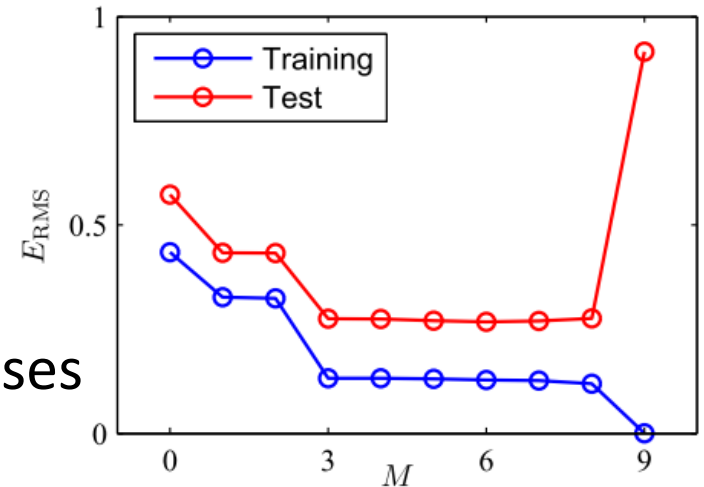


Note that the price would go to zero (or negative) if you buy bigger houses!
This is called poor generalization/overfitting.

Detecting overfitting

Plot model complexity versus objective function on test/train data

As model becomes more complex, performance on training keeps improving while on test data error increases



Horizontal axis: measure of model complexity

In this example, we use the maximum order of the polynomial basis functions.

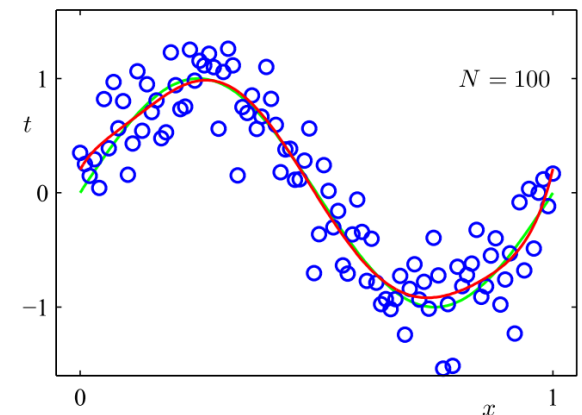
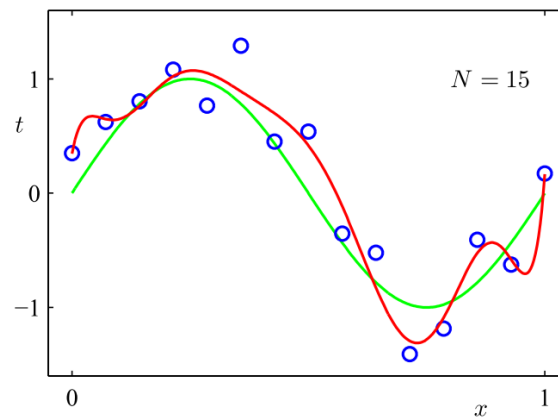
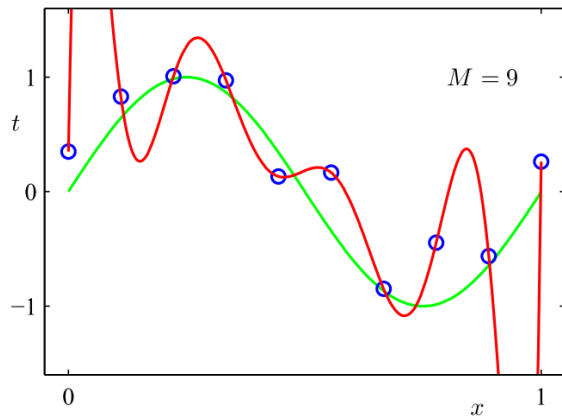
Vertical axis: For regression, it would be SSE or mean SE (MSE)
For classification, the vertical axis would be classification error rate or cross-entropy error function

Overcoming overfitting

- Basic ideas
 - Use more training data
 - Regularization methods
 - Cross-validation

Solution: use more data

$M=9$, increase N



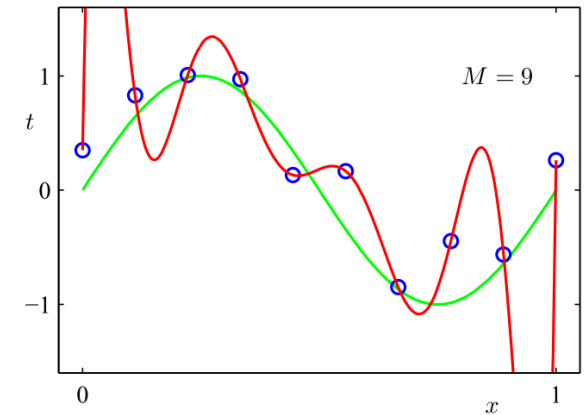
What if we do not have a lot of data?

Overcoming overfitting

- Basic ideas
 - Use more training data
 - Regularization methods
 - Cross-validation

Solution: Regularization

- Use regularization:
 - Add $\lambda \|\theta\|_2^2$ term to SSE cost function
 - “L-2” norm squared, *i.e.* sum of sq. elements $\sum \theta_j^2$
 - Penalizes large θ
 - λ controls amount of regularization

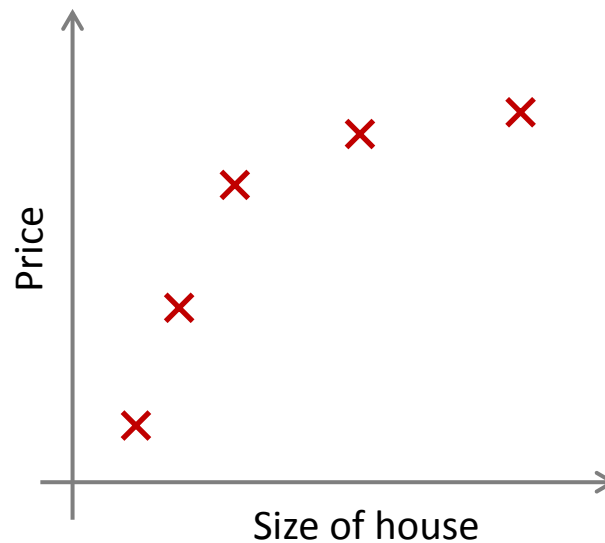


	M = 9
θ_0	0.35
θ_1	232.37
θ_2	-5321.83
θ_3	48568.31
θ_4	-231639.30
θ_5	640042.26
θ_6	-1061800.52
θ_7	1042400.18
θ_8	-557682.99
θ_9	125201.43

Regularized Linear Regression

$$J(\theta) = \frac{1}{2m} \left[\sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^n \theta_j^2 \right]$$

$$\min_{\theta} J(\theta)$$



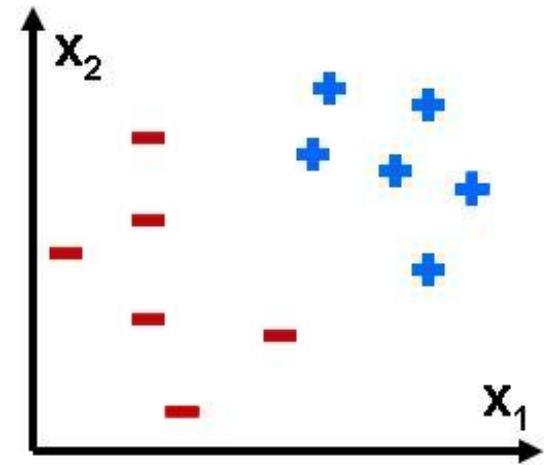
Regularized Logistic Regression

Hypothesis:

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

θ : parameters

$D = (x^{(i)}, y^{(i)})$: data

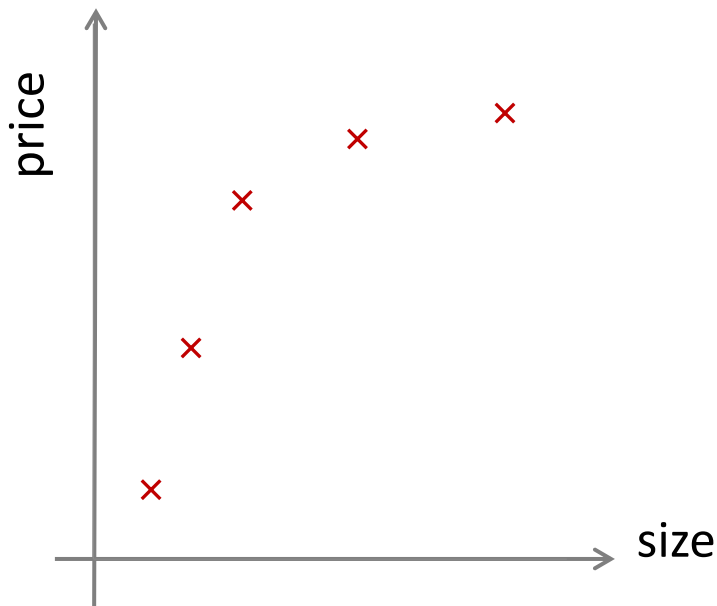


Cost Function:

$$J(\theta) = -\frac{1}{m} \left[\sum_{i=1}^m y^{(i)} \log h_{\theta}(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_{\theta}(x^{(i)})) \right] + \lambda \|\theta\|_2^2$$

Goal: minimize cost $\min_{\theta} J(\theta)$

Overfitting example



$$h_{\theta}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$$

Once parameters $(\theta_0, \theta_1, \dots, \theta_4)$ were fit to some set of data (training set), the error of the parameters as measured on that data (the training error $J(\theta)$) is likely to be lower than the actual generalization error.

One solution is to regularize, but how can we choose the regularization weight λ ?

Choosing weight λ

$$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$$



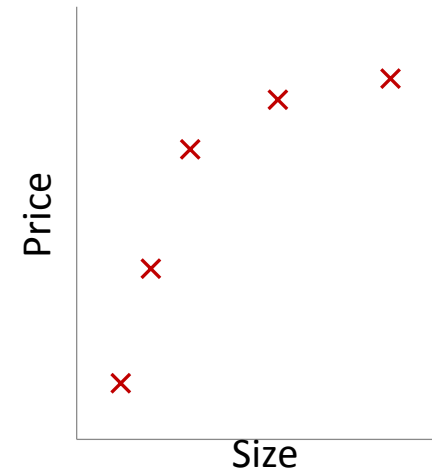
$$\lambda = 100$$

High bias
(underfit)



$$\lambda = 1$$

“Just right”



$$\lambda = 0.01$$

High variance
(overfit)

Overcoming overfitting

- Basic ideas
 - Use more training data
 - Regularization methods
 - Cross-validation

Train/Validation/Test Sets

	Size	Price
train	2104	400
	1600	330
	2400	369
	1416	232
	3000	540
validation	1985	300
	1534	315
	1427	199
test	1380	212
	1494	243

Solution: split data into three sets.

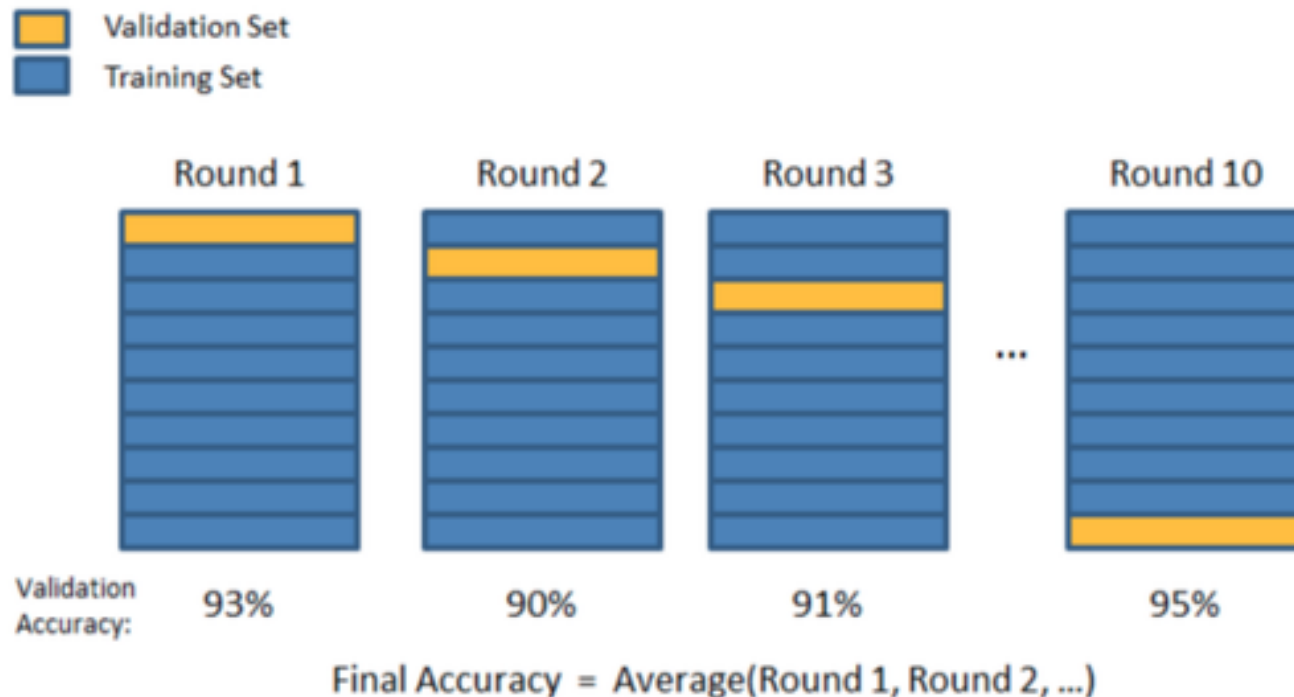
For each value of a hyperparameter, train on the train set, evaluate learned parameters on the validation set.

Pick the model with the hyperparameter that achieved the lowest validation error.

Report this model's test set error.

N-Fold Cross Validation

- What if we don't have enough data for train/test/validation sets?
- Solution: use N-fold cross validation.
- Split training set into train/validation sets N times.
- Report average predictions over N val sets, *e.g.* N=10:





Introduction to Neural Networks

Motivation

Recall: Logistic Regression

$$0 \leq h_{\theta}(x) \leq 1$$

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

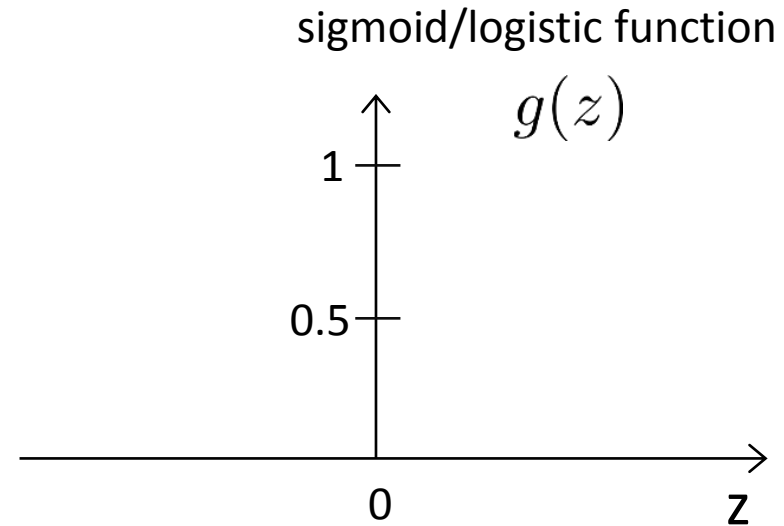
$$g(z) = \frac{1}{1 + e^{-z}}$$

Output is probability of label 1 given input

$$p(y = 1|x) = \frac{1}{1 + e^{-\theta^T x}}$$

predict “ $y = 1$ ” if $h_{\theta}(x) \geq 0.5$

predict “ $y = 0$ ” if $h_{\theta}(x) < 0.5$



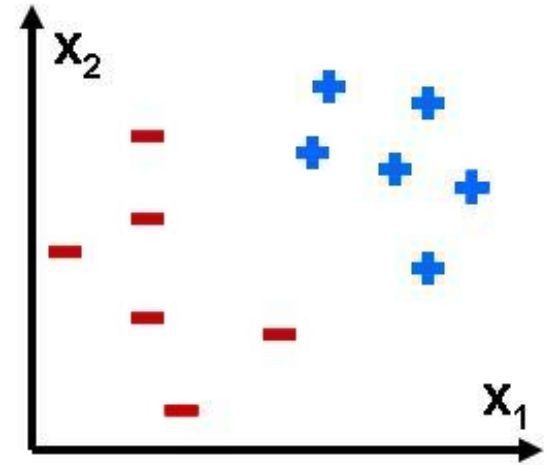
Recall: Logistic Regression Cost

Logistic Regression Hypothesis:

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

θ : parameters

$D = \{x^i, y^i\}$: data

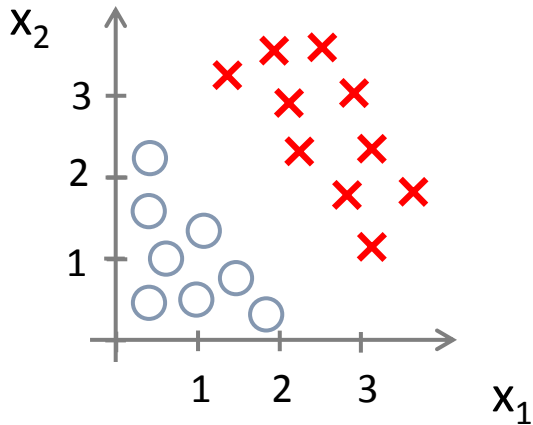


Logistic Regression Cost Function:

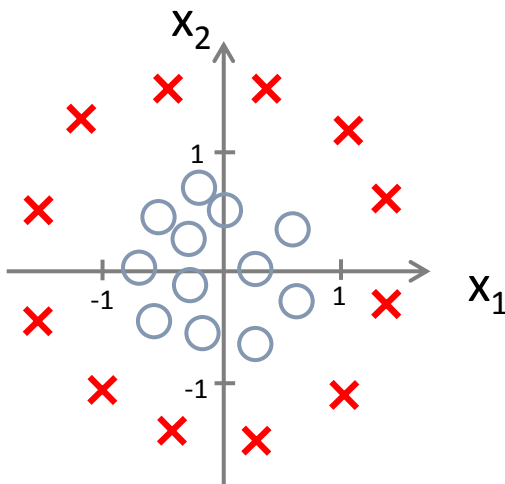
$$\begin{aligned} J(\theta) &= \frac{1}{m} \sum_{i=1}^m \text{Cost}(h_{\theta}(x^{(i)}), y^{(i)}) \\ &= -\frac{1}{m} \left[\sum_{i=1}^m y^{(i)} \log h_{\theta}(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_{\theta}(x^{(i)})) \right] \end{aligned}$$

Goal: minimize cost $\min_{\theta} J(\theta)$

Decision boundary



Non-linear decision boundaries

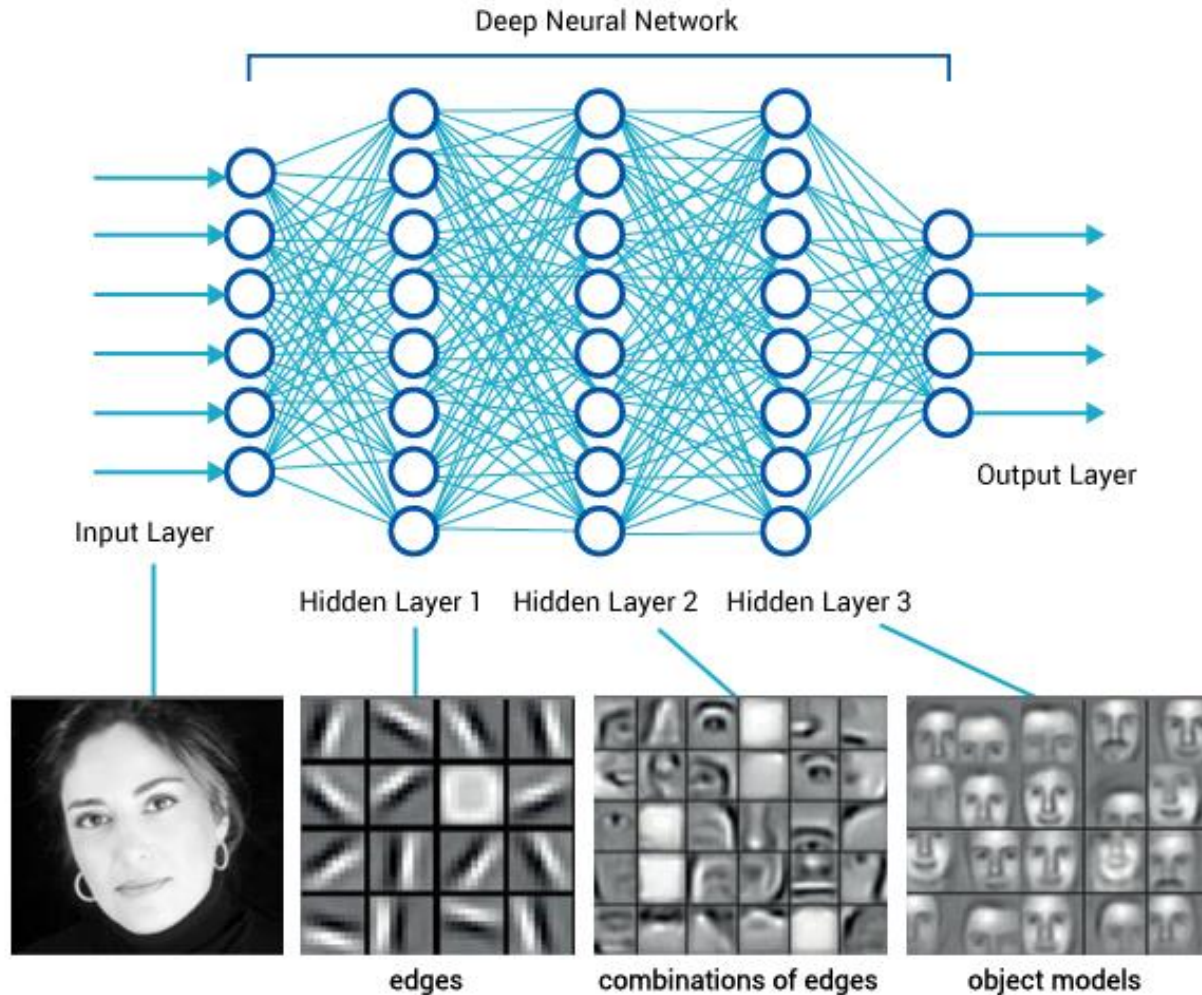


Replace features with non-linear functions
e.g. log, cosine, or **polynomial**

Limitations of linear models

- Logistic regression and other linear models cannot handle nonlinear decision boundaries
 - Must use non-linear feature transformations
 - Up to designer to specify which one
- Can we instead **learn** the transformation?
 - Yes, this is what neural networks do!
- A **Neural network** chains together many layers of “neurons” such as logistic units (logistic regression functions)

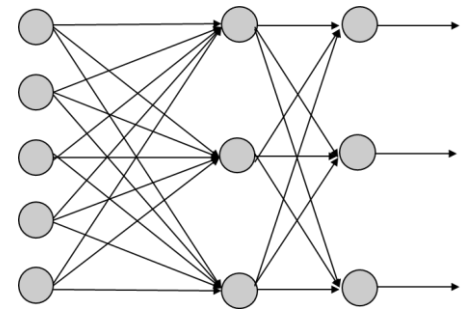
Neural Networks learn features



Neurons in the Brain

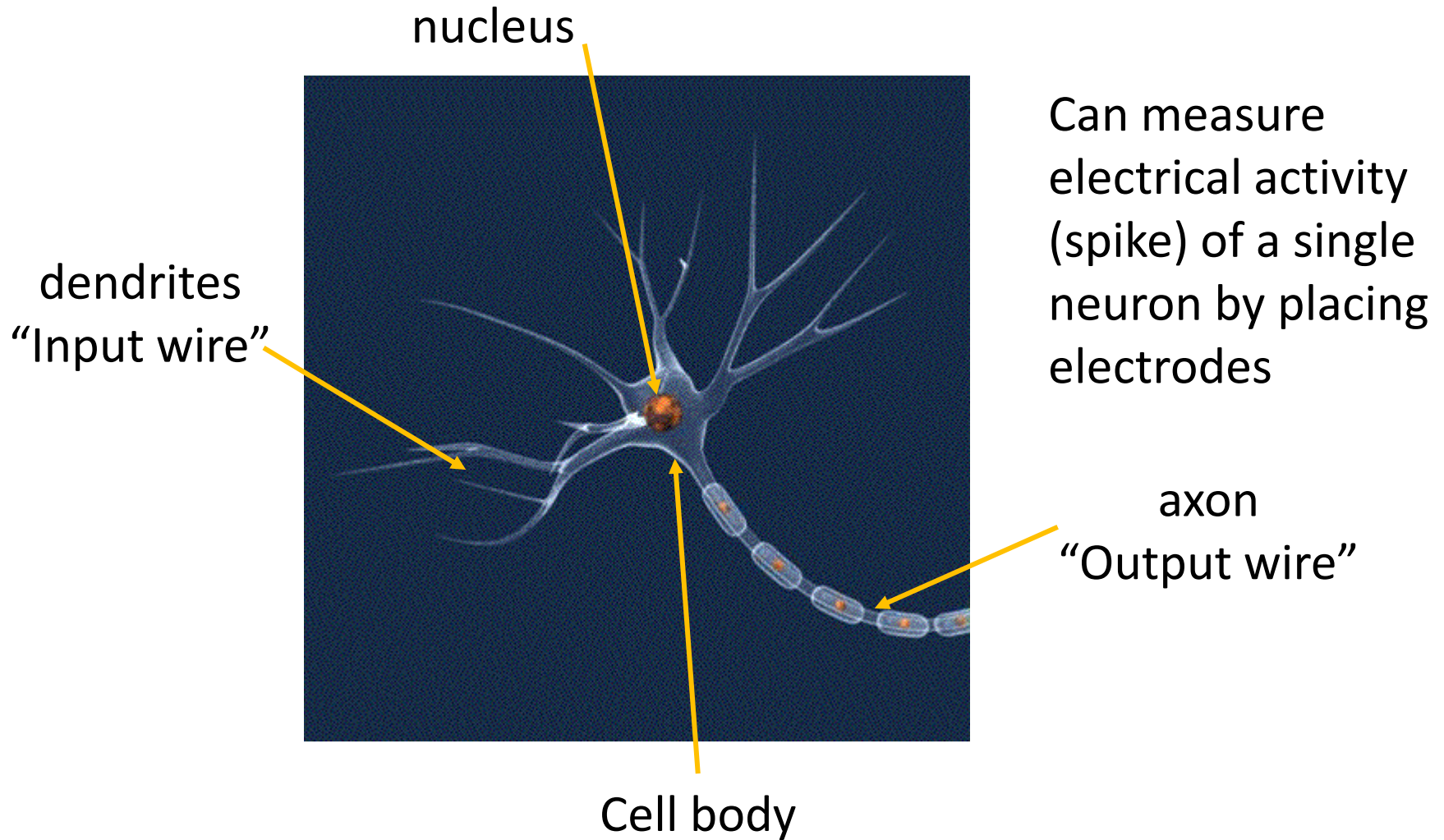


Inspired “Artificial Neural Networks”

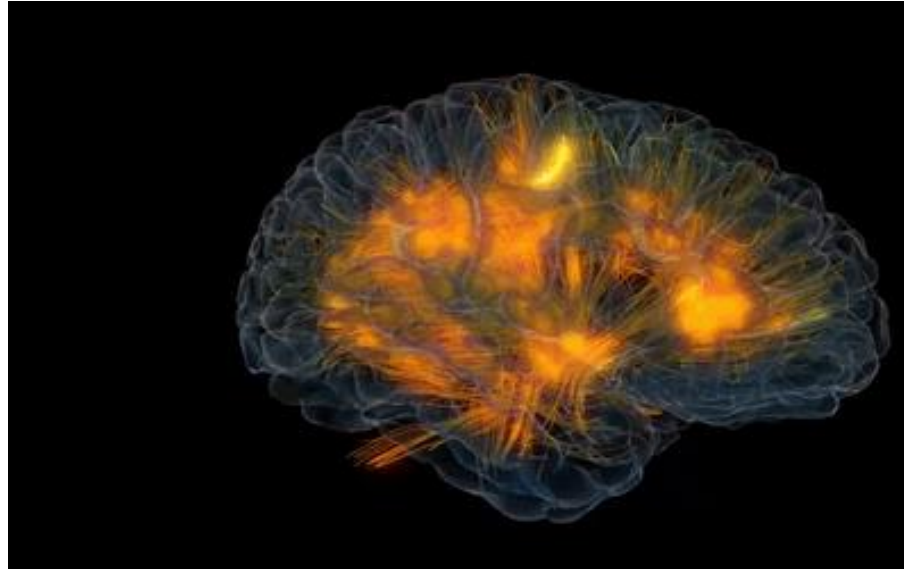


Neurons are cells that process chemical and electrical signals and transmit these signals to neurons and other types of cells

Neuron in the brain

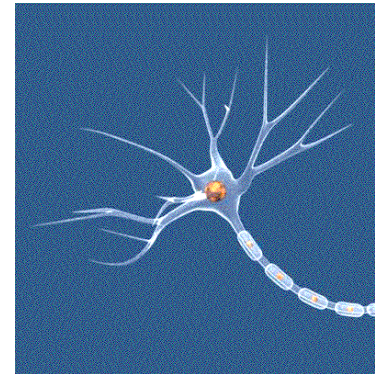
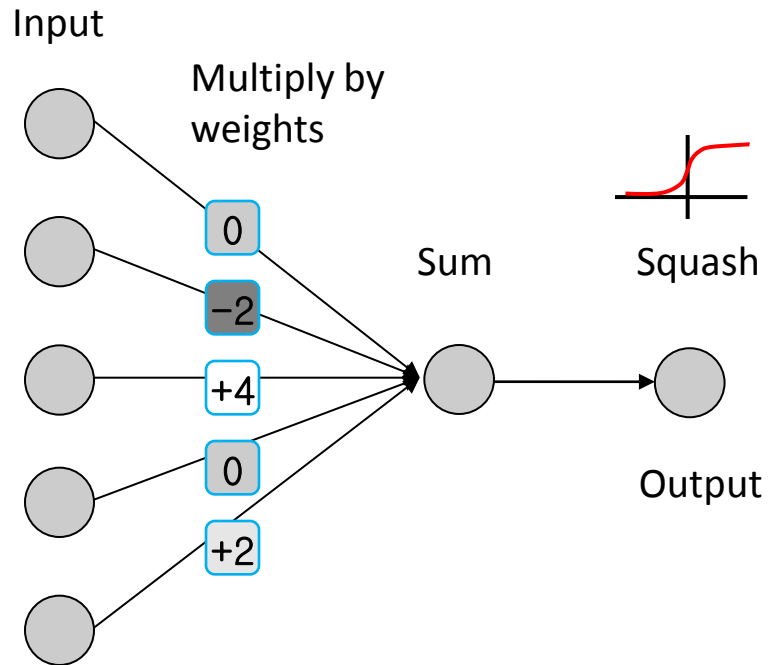


Neural network in the brain

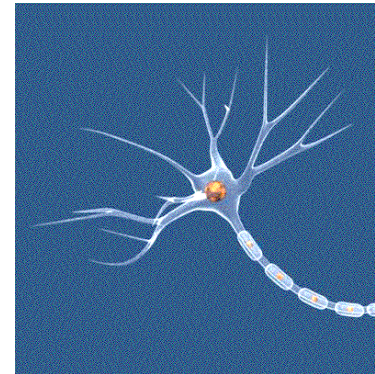
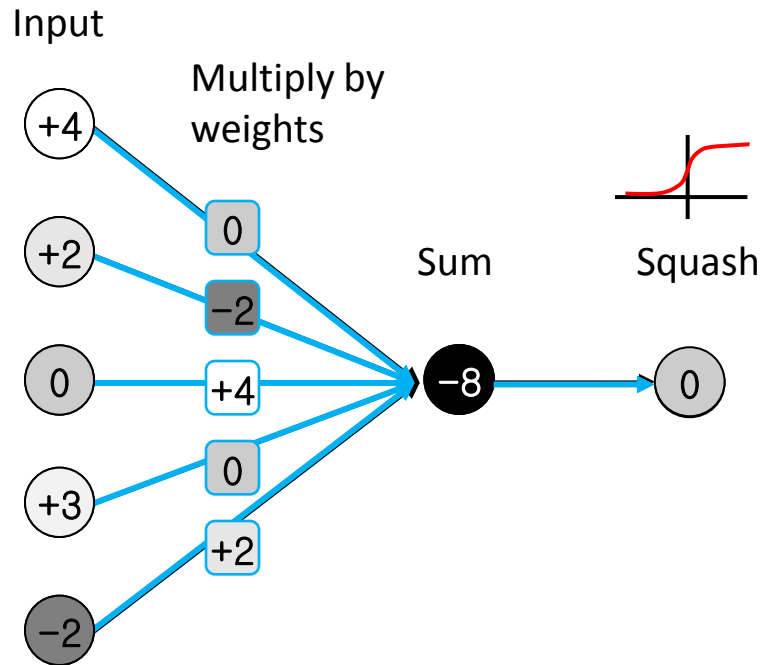


- **Micro networks:** several connected neurons perform sophisticated tasks: mediate reflexes, process sensory information, generate locomotion and mediate learning and memory.
- **Macro networks:** perform higher brain functions such as object recognition and cognition.

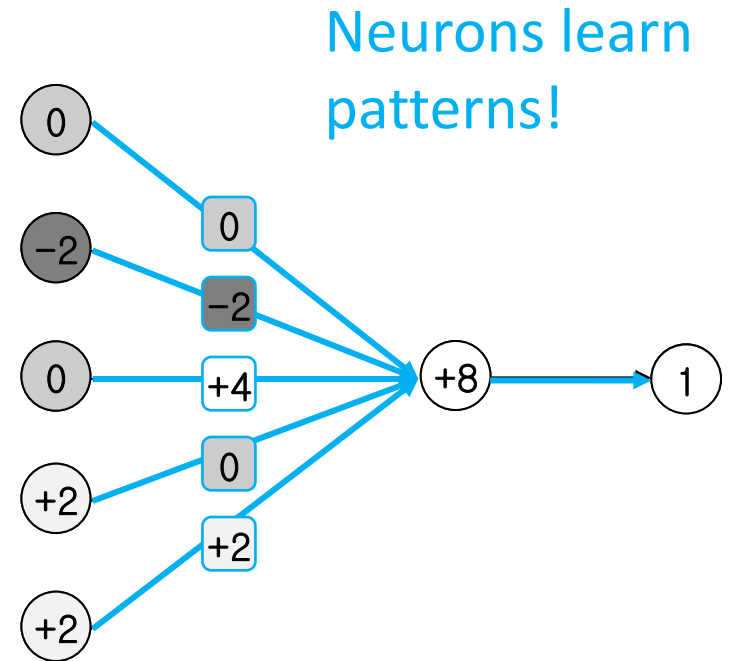
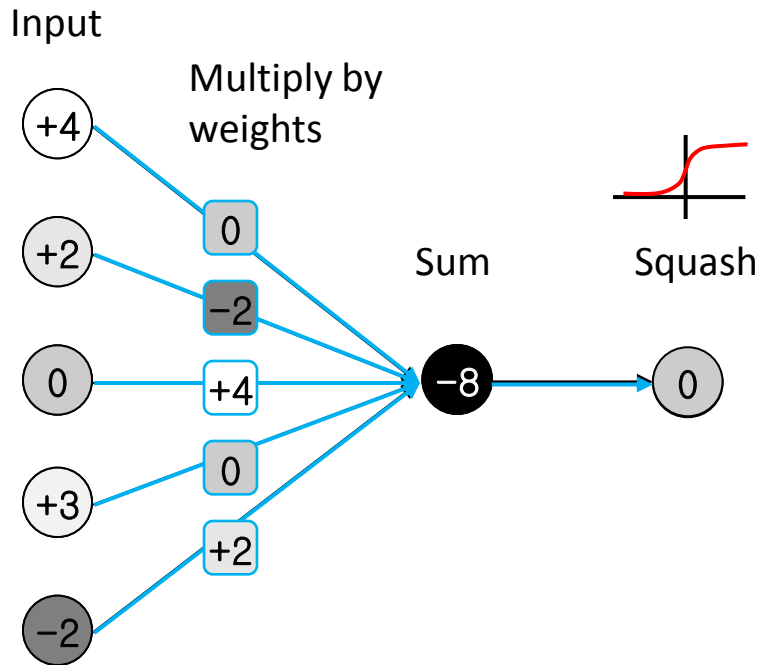
Logistic Unit as Artificial Neuron



Logistic Unit as Artificial Neuron

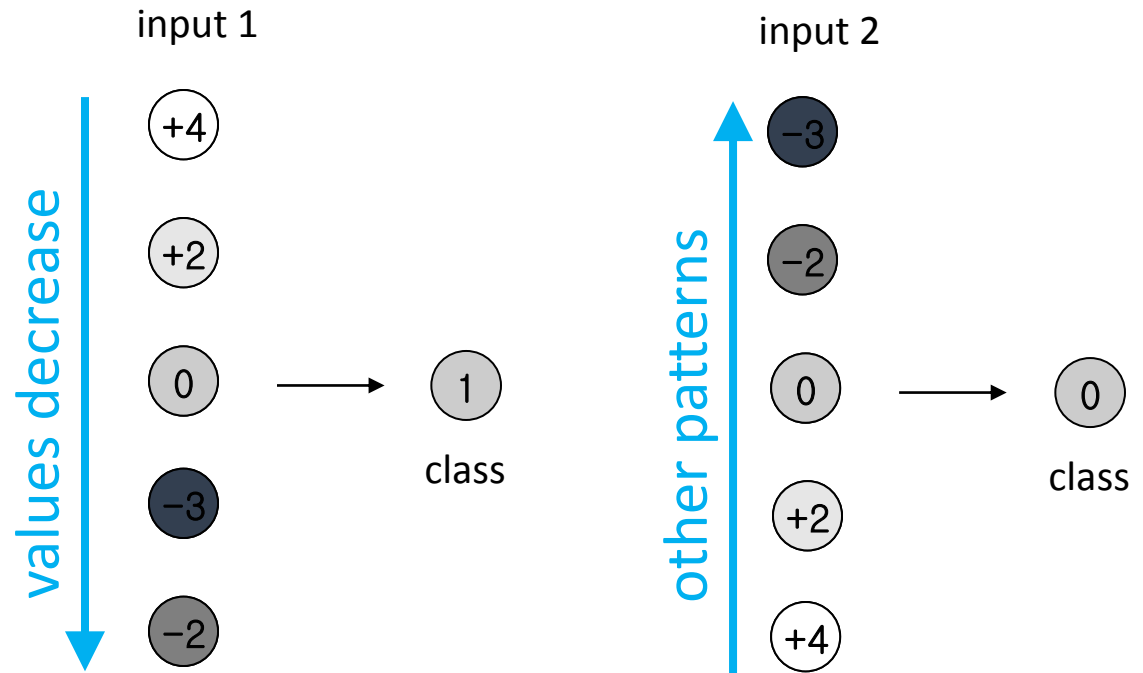


Logistic Unit as Artificial Neuron



Artificial Neuron Learns Patterns

- Classify input into class 0 or 1
- Teach neuron to predict correct class label
- Detect presence of a simple “feature”



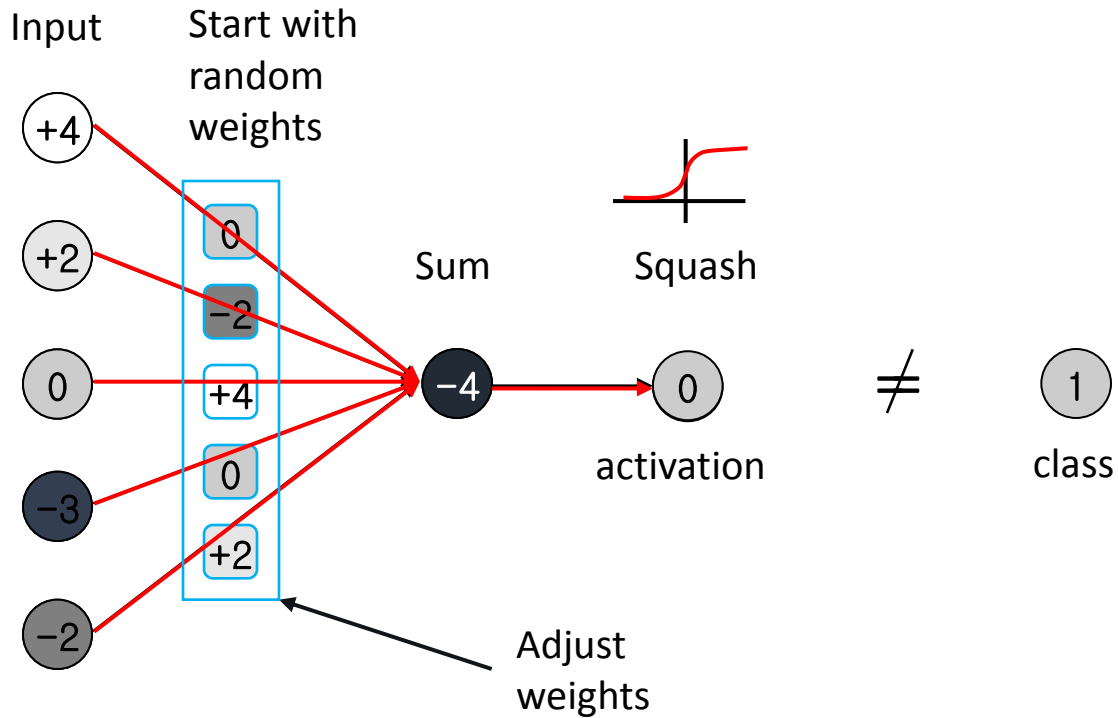
Example



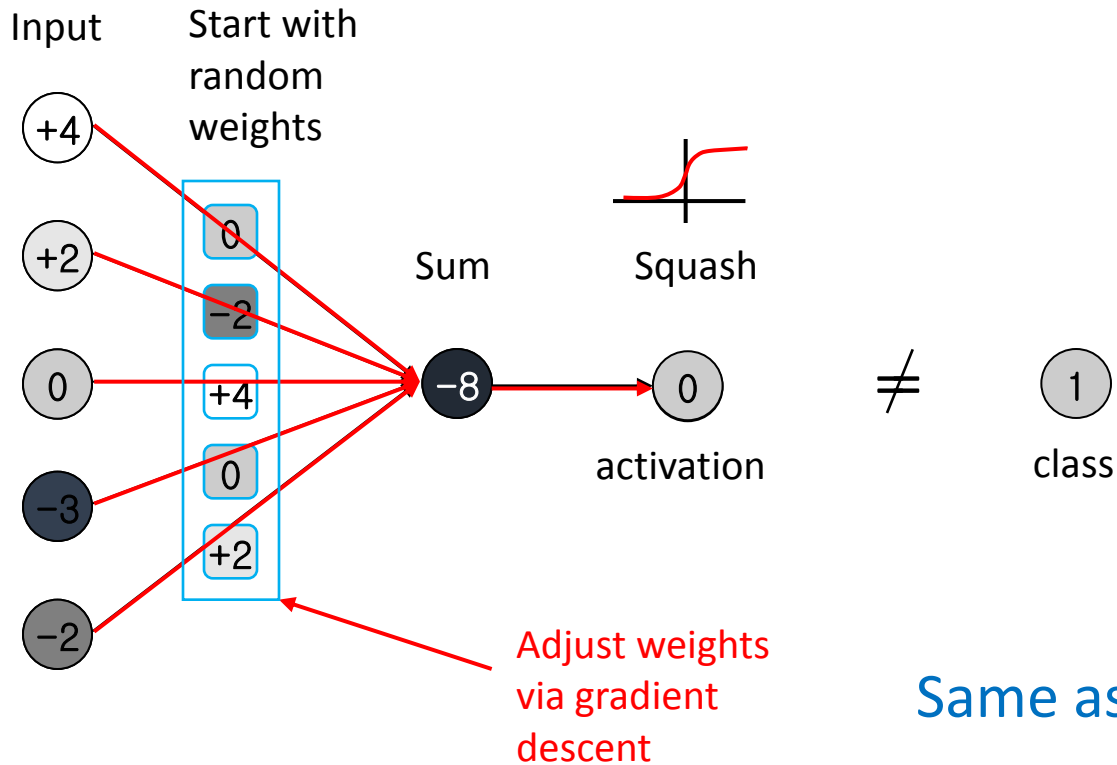
Neural Networks: Learning

Intuition

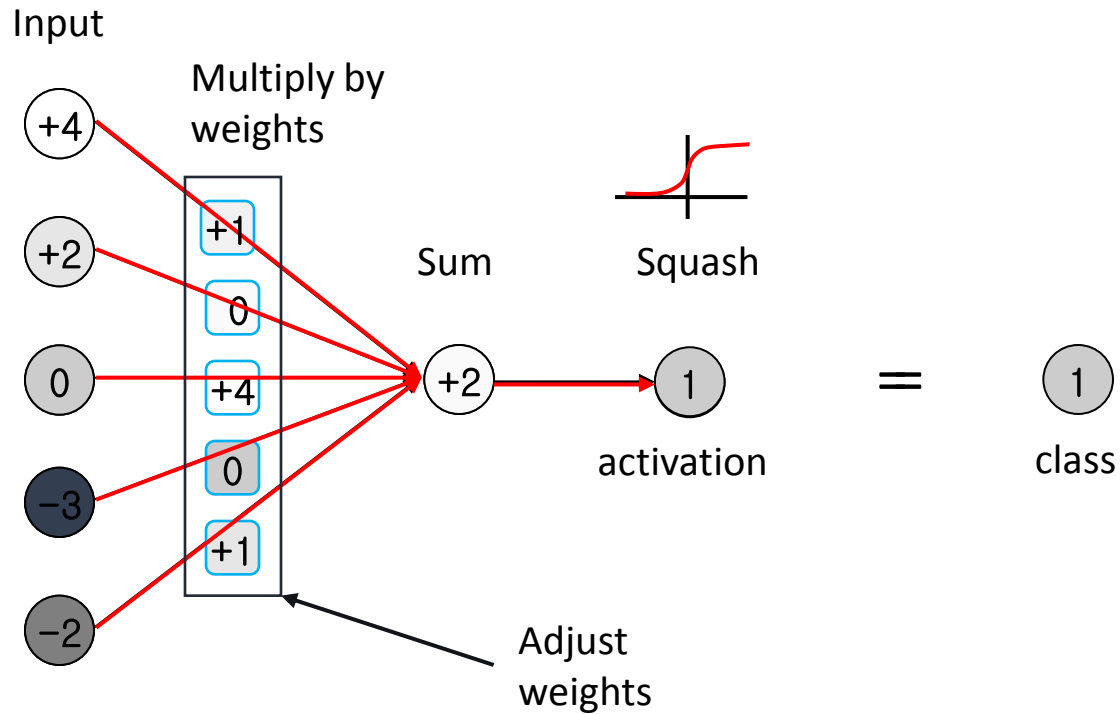
Artificial Neuron: Learning



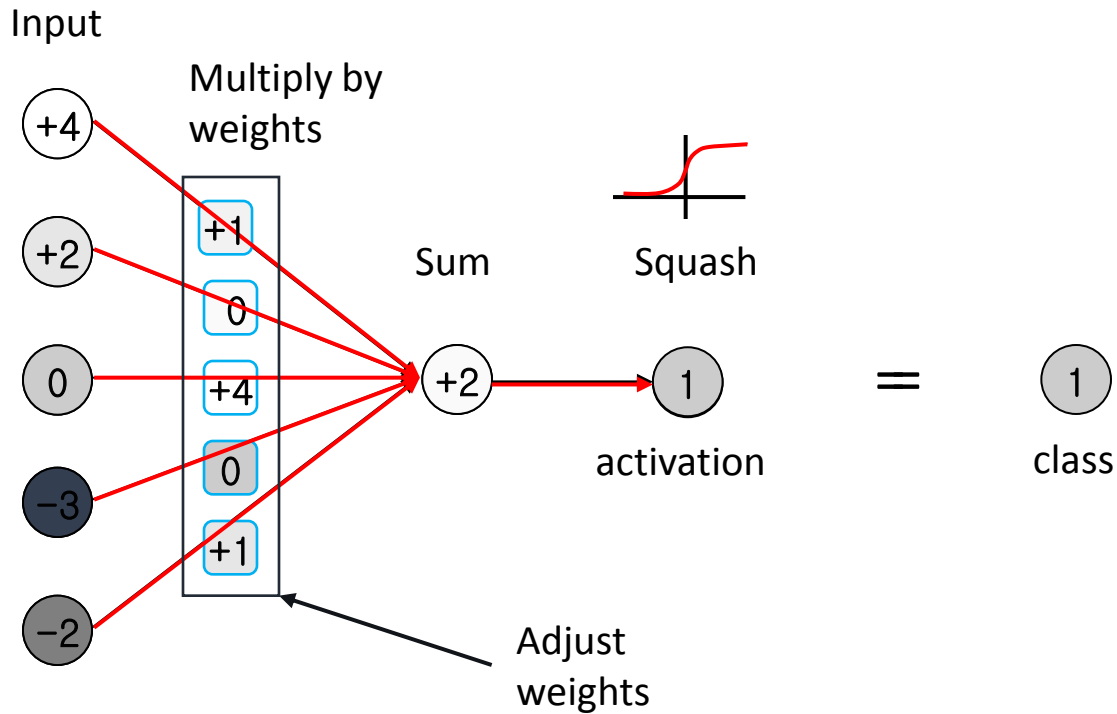
Artificial Neuron: Learning



Artificial Neuron: Learning



Artificial Neuron: Learning



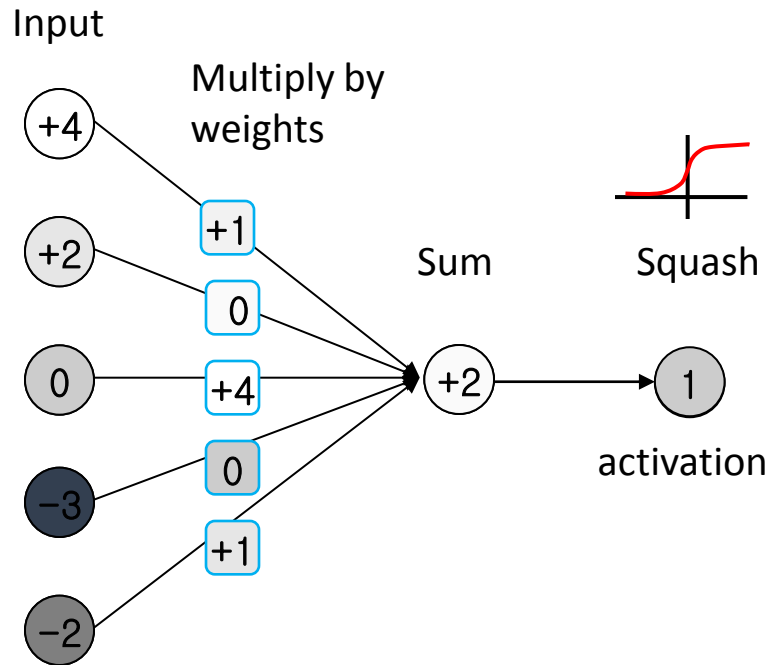
Forward propagation of information through a neuron



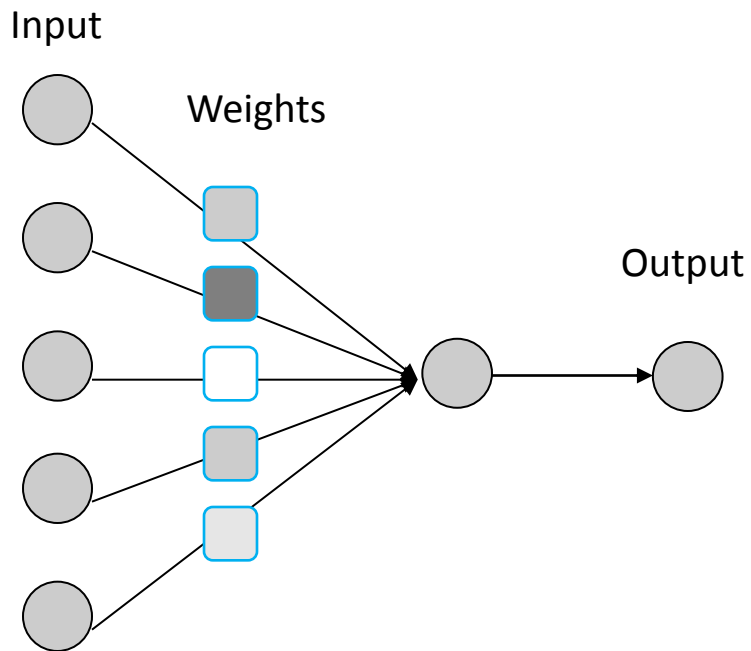
Neural Networks: Learning

Multi-layer network

Artificial Neuron: simplify

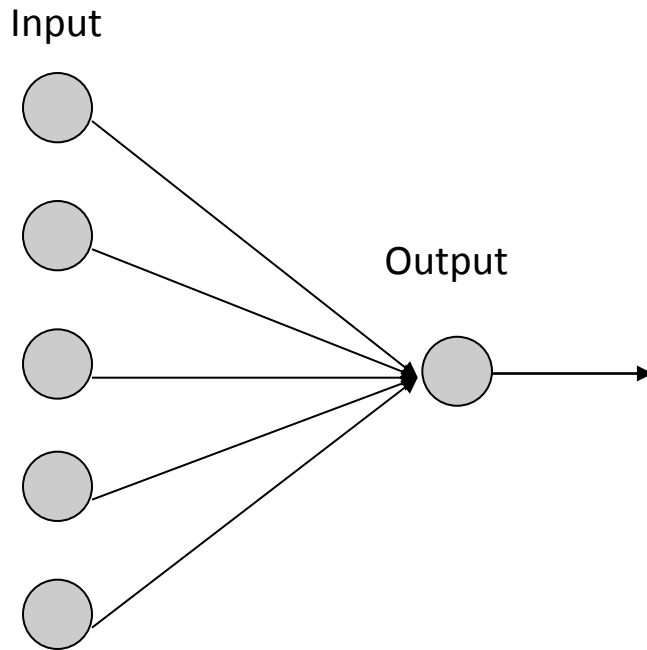


Artificial Neuron: simplify

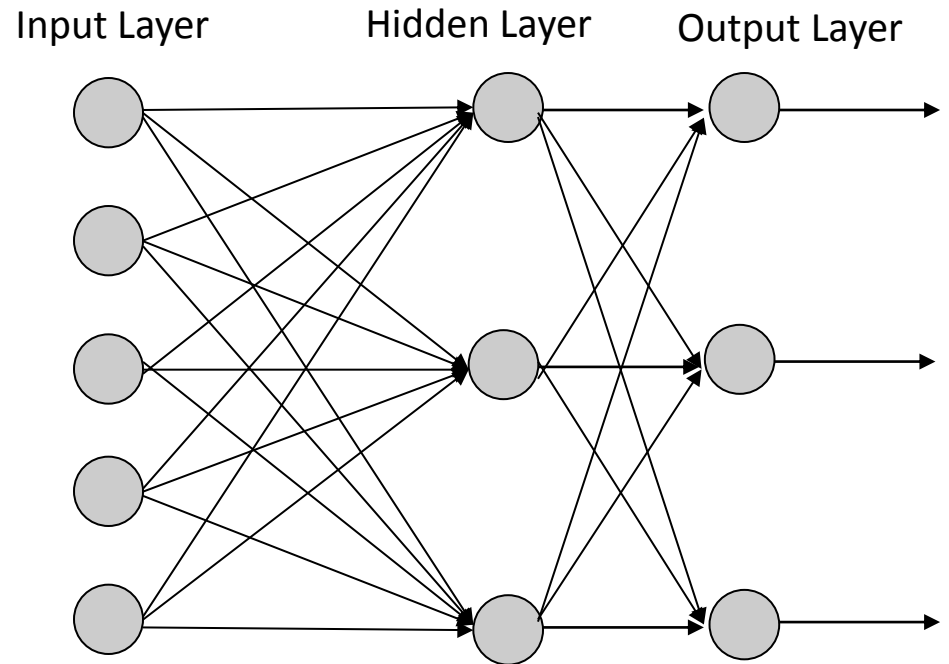


A single neuron is also called a perceptron

Artificial Neural Network



Single Neuron

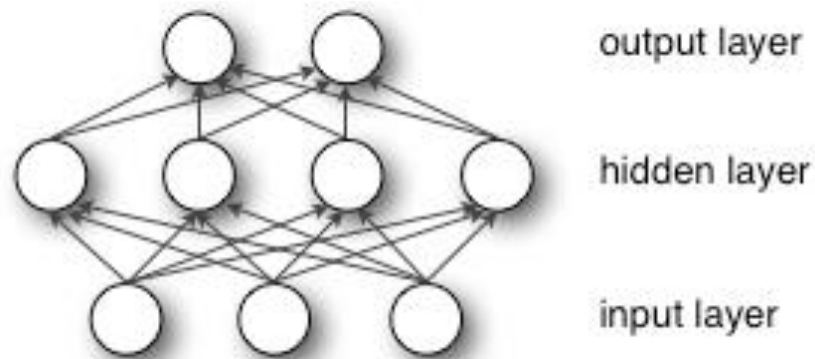


Neural Network (fully connected)

Deep Network: many hidden layers

Multi-layer perceptron (MLP)

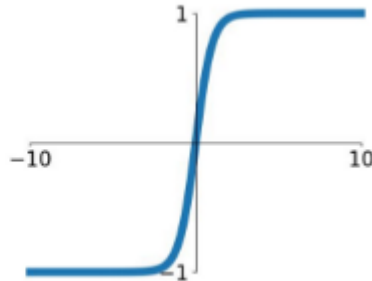
- Just another name for a feed-forward neural network
- Logistic regression is a special case of the MLP with no hidden layer and sigmoid output.



Other Non-linearities

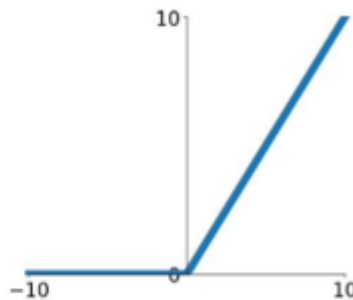
- Also called activation functions

tanh
 $\tanh(x)$



$$\tanh(x) = \frac{2}{1+e^{-2x}} - 1$$

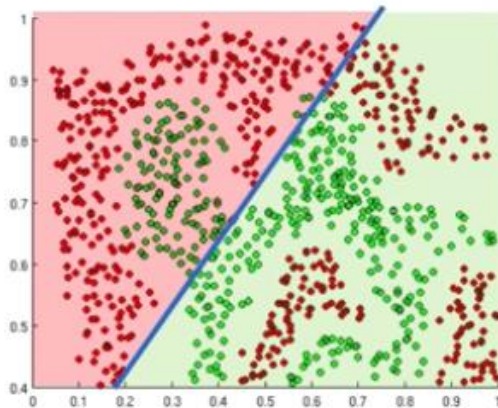
ReLU
 $\max(0, x)$



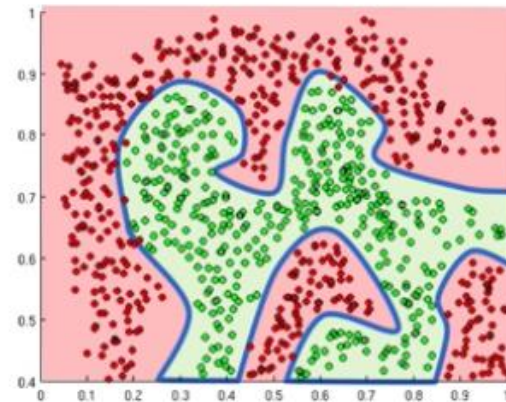
$$ReLU(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

Importance of Non-linearities

*The purpose of activation functions is to **introduce non-linearities** into the network*



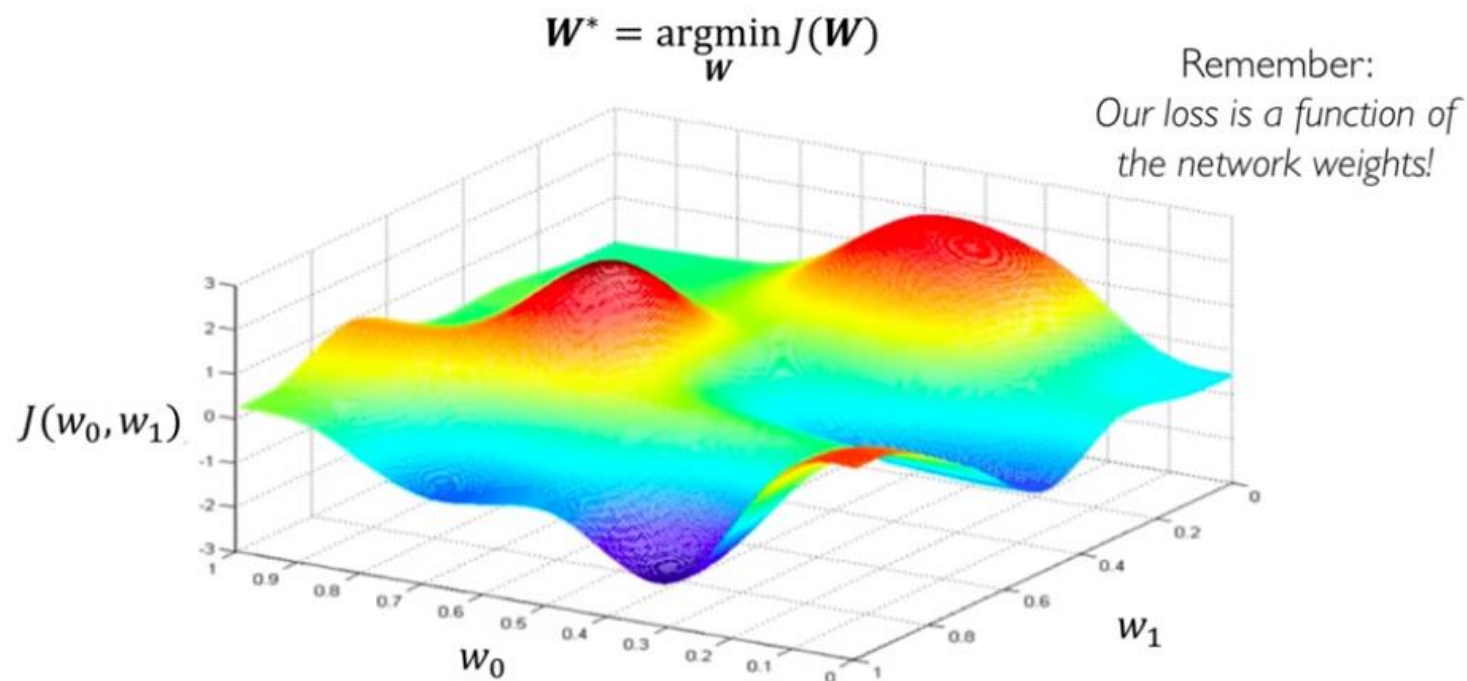
Linear activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

Loss Optimization

- Neural network parameters θ are often referred to as weights W .



Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
4. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
5. Return weights


Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
4. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
5. Return weights

Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(W)}{\partial W}$ 
4. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(W)}{\partial W}$
5. Return weights

Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(W)}{\partial W}$ *Not feasible to compute over all dataset*
4. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(W)}{\partial W}$
5. Return weights

Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(W)}{\partial W}$ *Compute over a mini-batch*
4. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(W)}{\partial W}$
5. Return weights

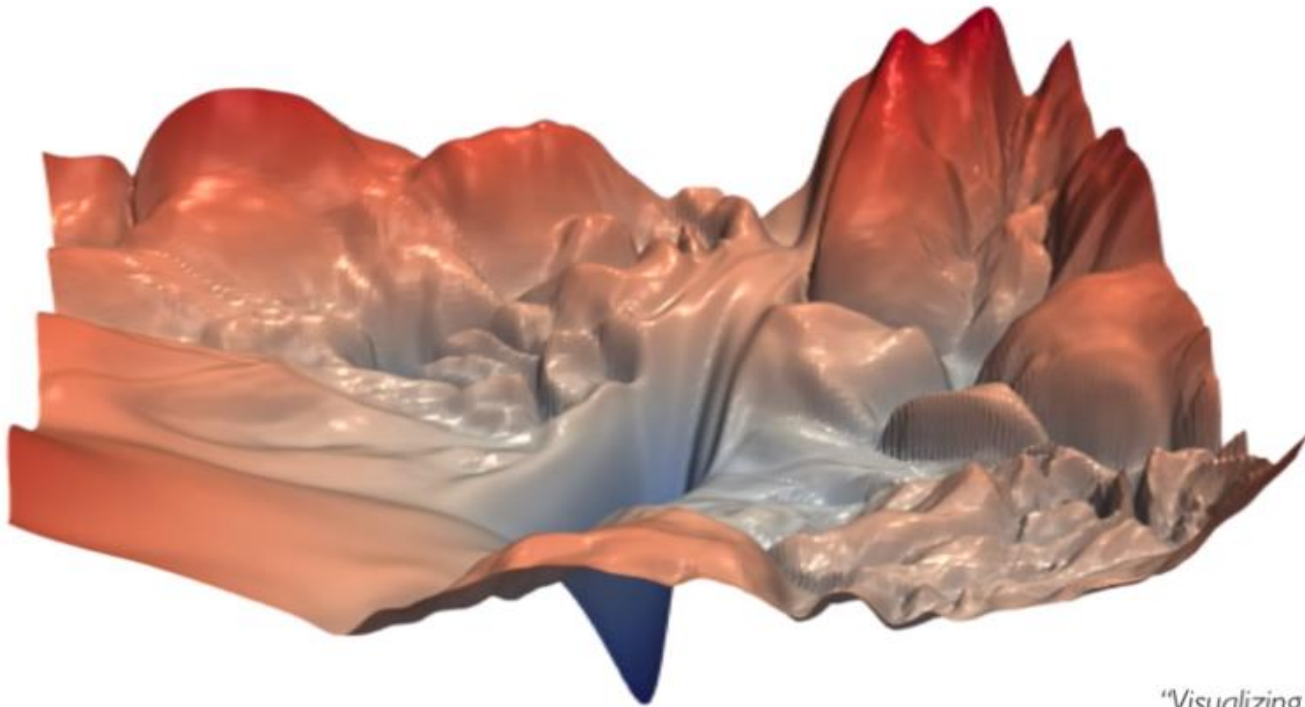
Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(W)}{\partial W}$ *Compute over a mini-batch*
4. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(W)}{\partial W}$
5. Return weights

Parallelization: Batches can be split onto multiple GPUs

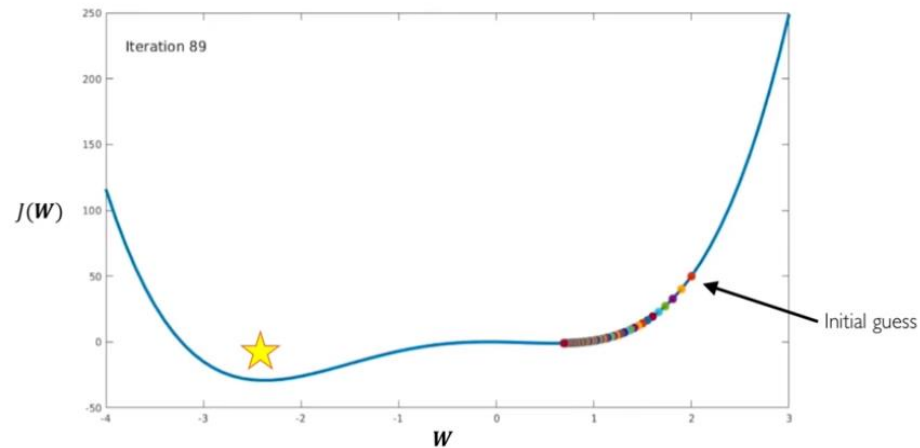
Landscape Visualization



*"Visualizing the loss landscape
of neural nets". Dec 2017.*

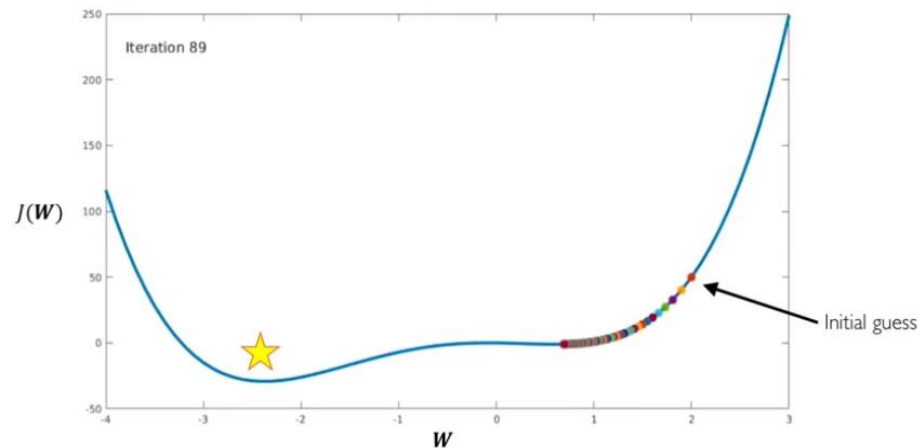
Setting the Learning Rate

Small learning rate converges slowly and gets stuck in false local minima

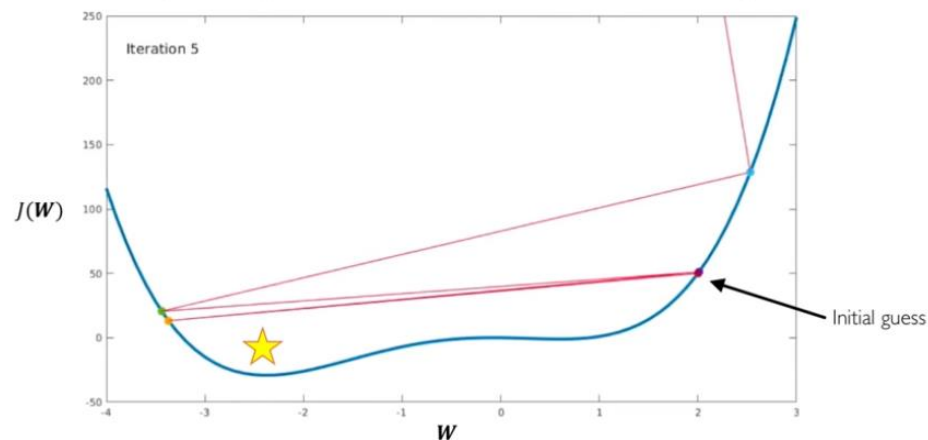


Setting the Learning Rate

Small learning rate converges slowly and gets stuck in false local minima



Large learning rates overshoot, become unstable and diverge



Setting the Learning Rate

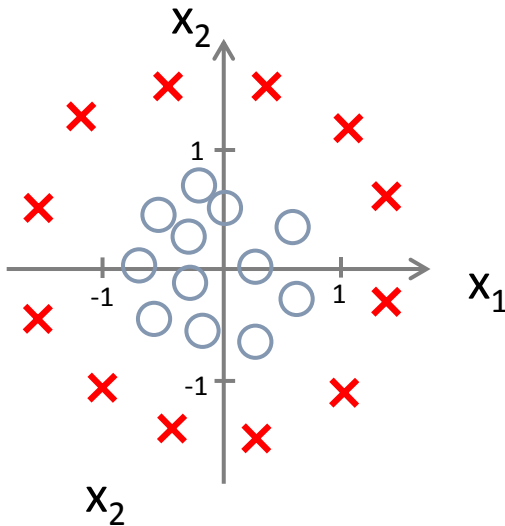
- How to select the learning Rate?
 - Try several, and see which works best
 - Start with a learning rate, and change it ***adaptively*** as the model trains
 - Many are implemented in Neural Network Tools

Neural Networks Learn Features

logistic regression unit == **artificial neuron**

chain several units together == **neural network**

“earlier” units learn non-linear feature transformation

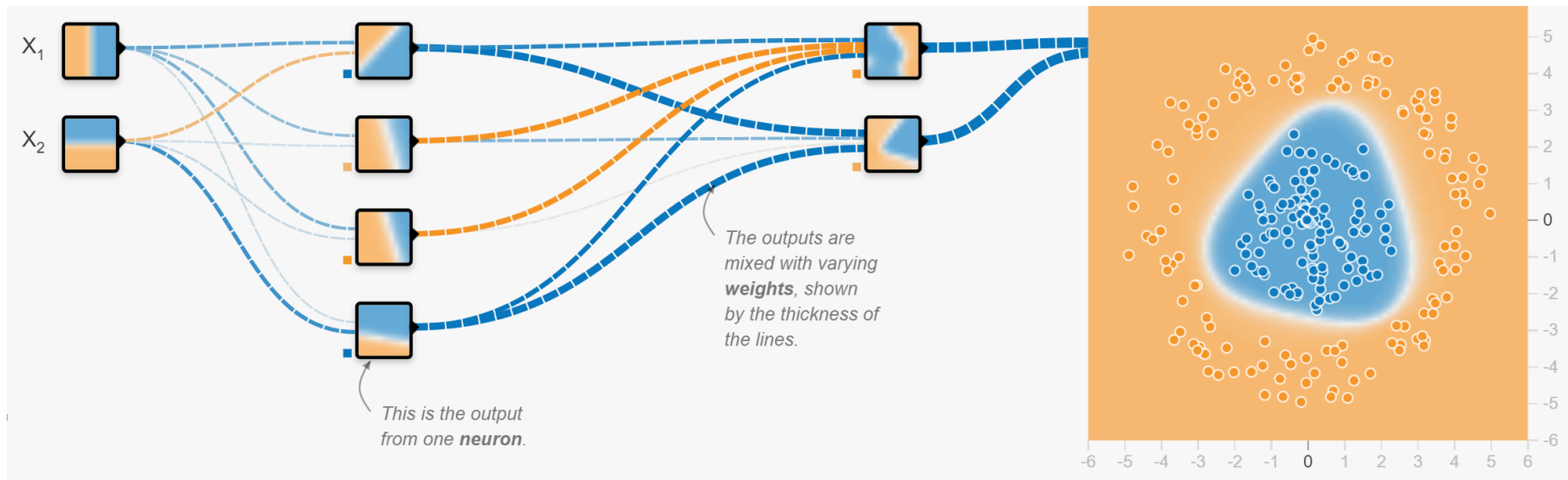


$$h_{\theta}(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$$

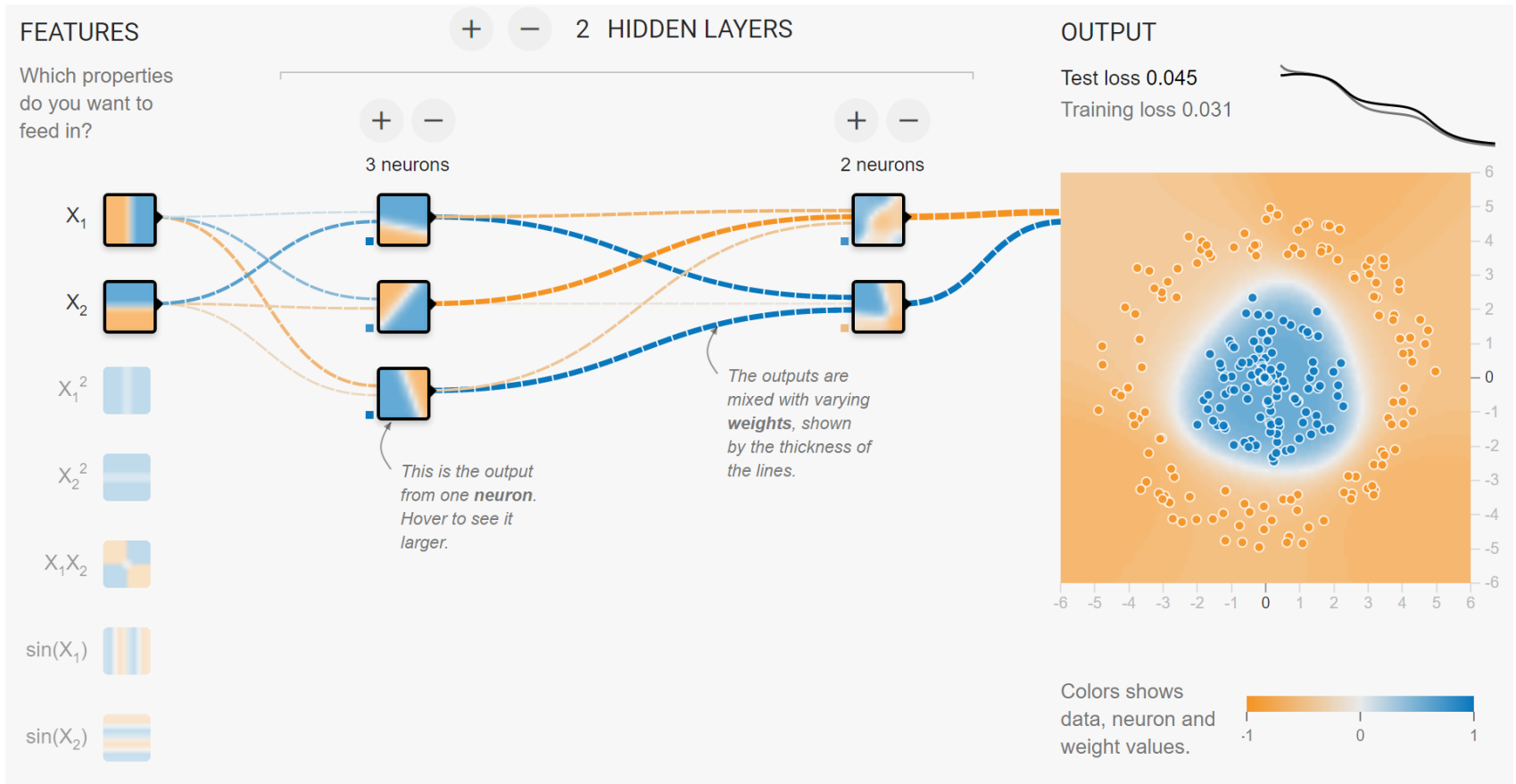
simple neural network

$$h(x) = g(\theta + \theta_1 h^{(1)}(x) + \theta_2 h^{(2)}(x) + \theta_3 h^{(3)}(x))$$

Example



Training a neural net: Demo



[Tensorflow playground](#)

Artificial Neural Network:

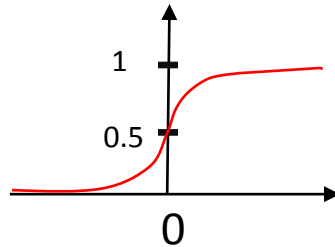
general notation

input $x = \begin{bmatrix} x_1 \\ \dots \\ x_5 \end{bmatrix}$

hidden layer activations

$$h^i = g(\Theta^{(i)}x)$$

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



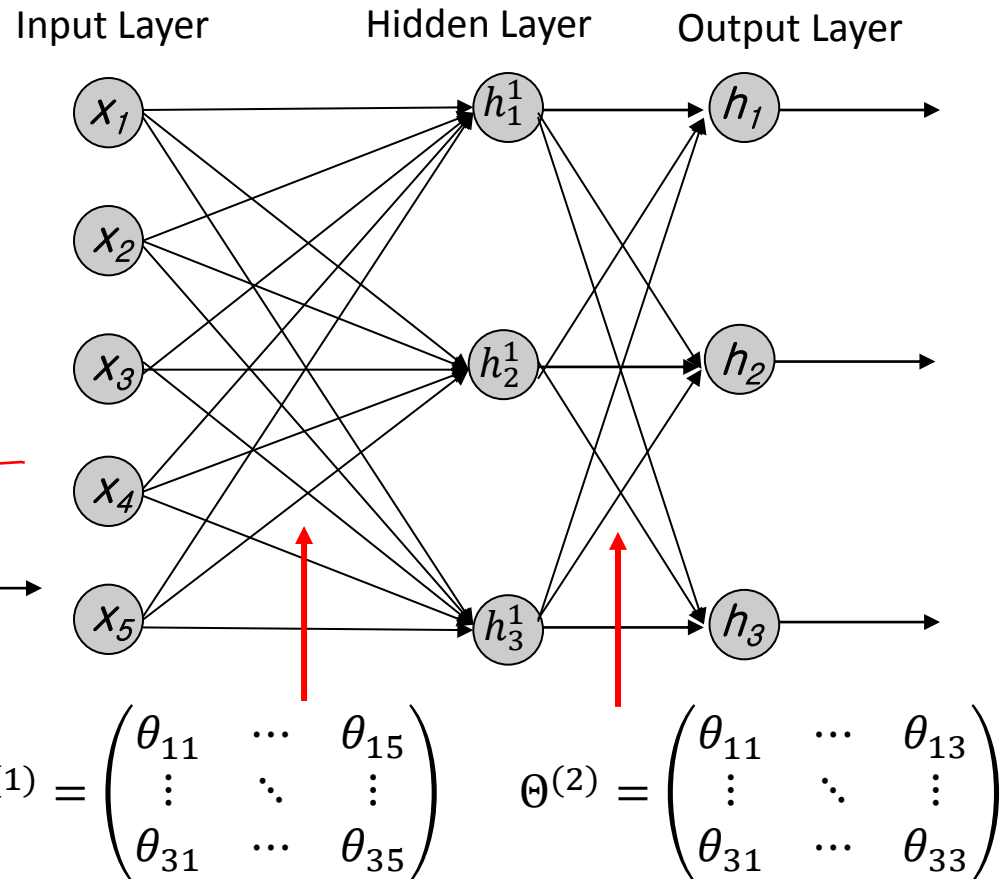
output

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

weights

$$\Theta^{(1)} = \begin{pmatrix} \theta_{11} & \dots & \theta_{15} \\ \vdots & \ddots & \vdots \\ \theta_{31} & \dots & \theta_{35} \end{pmatrix}$$

$$\Theta^{(2)} = \begin{pmatrix} \theta_{11} & \dots & \theta_{13} \\ \vdots & \ddots & \vdots \\ \theta_{31} & \dots & \theta_{33} \end{pmatrix}$$





Neural Networks II

General Notation

Artificial Neural Network:

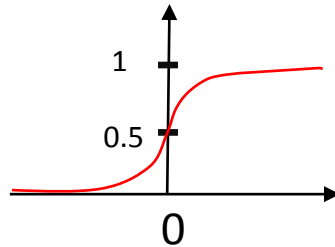
general notation

input $x = \begin{bmatrix} x_1 \\ \dots \\ x_5 \end{bmatrix}$

hidden layer activations

$$h^i = g(\Theta^{(i)}x)$$

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

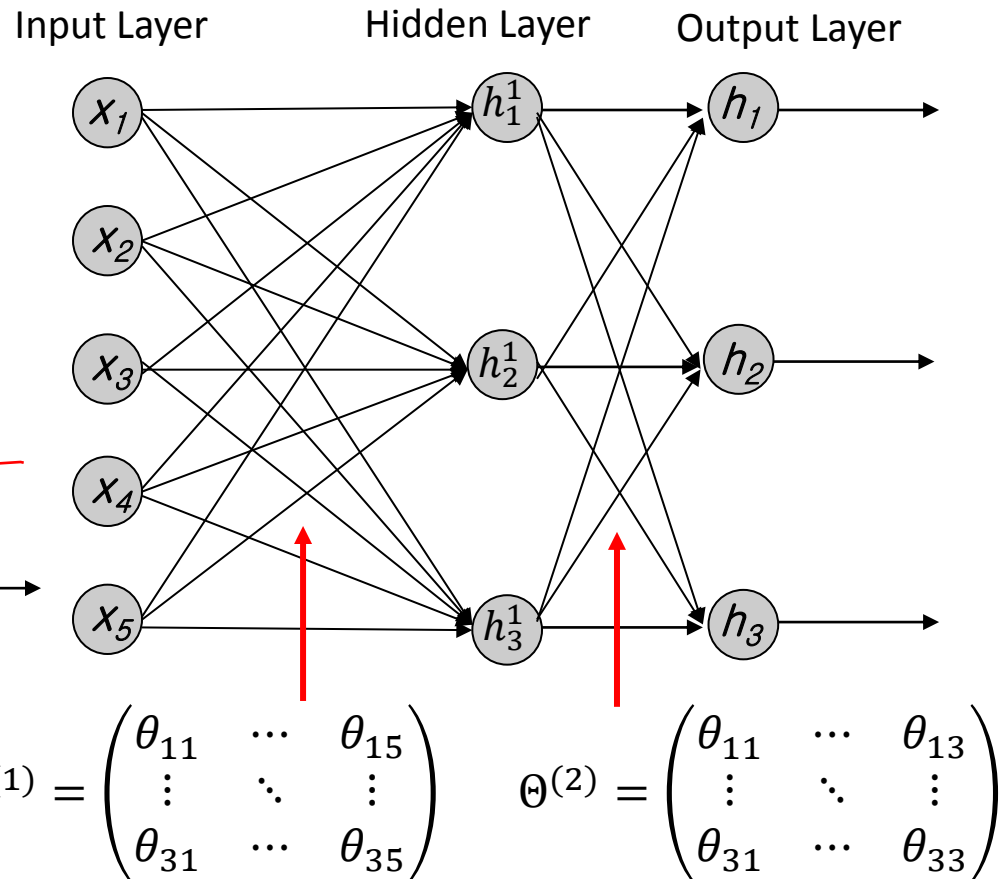


output

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

weights

$$\Theta^{(1)} = \begin{pmatrix} \theta_{11} & \dots & \theta_{15} \\ \vdots & \ddots & \vdots \\ \theta_{31} & \dots & \theta_{35} \end{pmatrix} \quad \Theta^{(2)} = \begin{pmatrix} \theta_{11} & \dots & \theta_{13} \\ \vdots & \ddots & \vdots \\ \theta_{31} & \dots & \theta_{33} \end{pmatrix}$$



Cost function

Neural network: $h_{\Theta}(x) \in \mathbb{R}^K$ $(h_{\Theta}(x))_i = i^{th}$ output

training error

$$J(\Theta) = -\frac{1}{m} \left[\sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log(h_{\Theta}(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_{\Theta}(x^{(i)}))_k) \right]$$

$$+ \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\Theta_{ji}^{(l)})^2$$

regularization

Gradient computation

$$J(\Theta) = -\frac{1}{m} \left[\sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log h_{\theta}(x^{(i)})_k + (1 - y_k^{(i)}) \log(1 - h_{\theta}(x^{(i)})_k) \right] \\ + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\Theta_{ji}^{(l)})^2$$

$$\min_{\Theta} J(\Theta)$$

Need code to compute:

- $J(\Theta)$
- $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$

Gradient computation

$$J(\Theta) = -\frac{1}{m} \left[\sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log h_{\theta}(x^{(i)})_k + (1 - y_k^{(i)}) \log(1 - h_{\theta}(x^{(i)})_k) \right] \\ + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\Theta_j^{(l)})^2$$

$$\min_{\Theta} J(\Theta)$$

Need code to compute:

$$- J(\Theta)$$

$$- \frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$$

Backpropagation

Gradient computation

$$J(\Theta) = -\frac{1}{m} \left[\sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log h_{\theta}(x^{(i)})_k + (1 - y_k^{(i)}) \log(1 - h_{\theta}(x^{(i)})_k) \right] \\ + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\Theta_{ji}^{(l)})^2$$

$$\min_{\Theta} J(\Theta)$$

Need code to compute:

- $J(\Theta)$
- $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$

Use “**Backpropagation algorithm**”

- Efficient way to compute $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$
- Computes gradient incrementally by “propagating” backwards through the network