CS 6501: Deep Learning for Computer Graphics

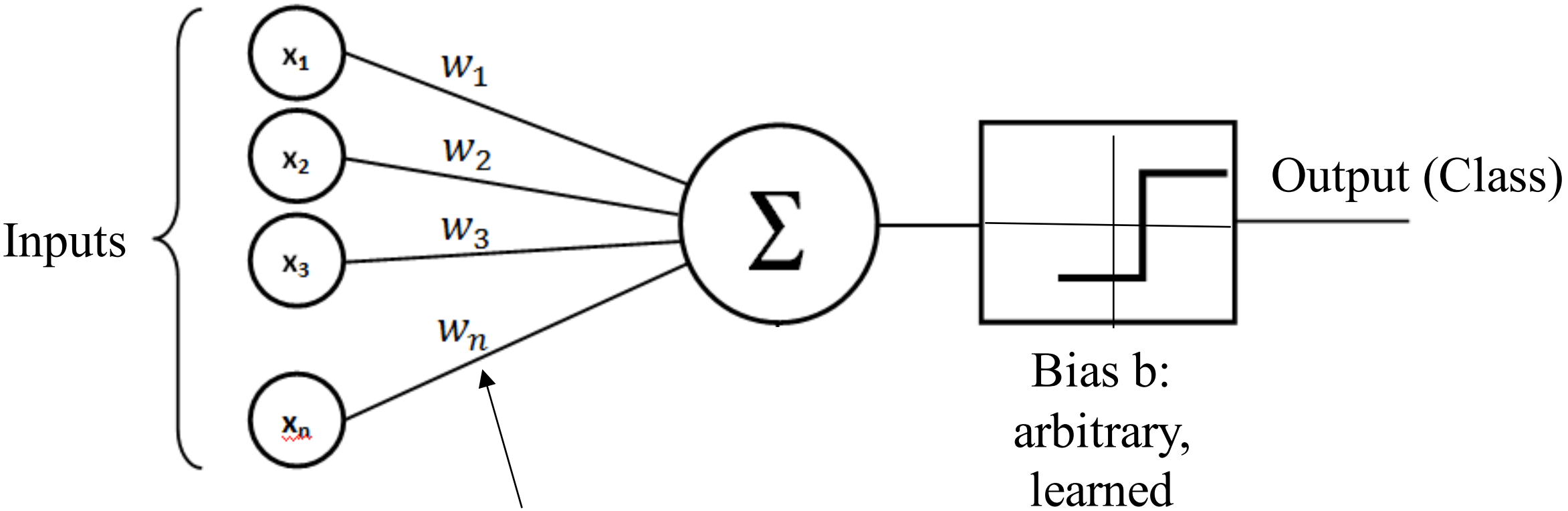
Basics of Neural Networks

Connelly Barnes

Overview

- Simple neural networks
  - **Perceptron**
  - Feedforward neural networks
  - Multilayer perceptron and properties
  - Autoencoders
- How to train neural networks
  - Gradient descent
  - Stochastic gradient descent
  - Automatic differentiation
  - Backpropagation

Perceptron (1957, Cornell)



Perceptron (1957, Cornell)

- Binary classifier, can learn linearly separable patterns.

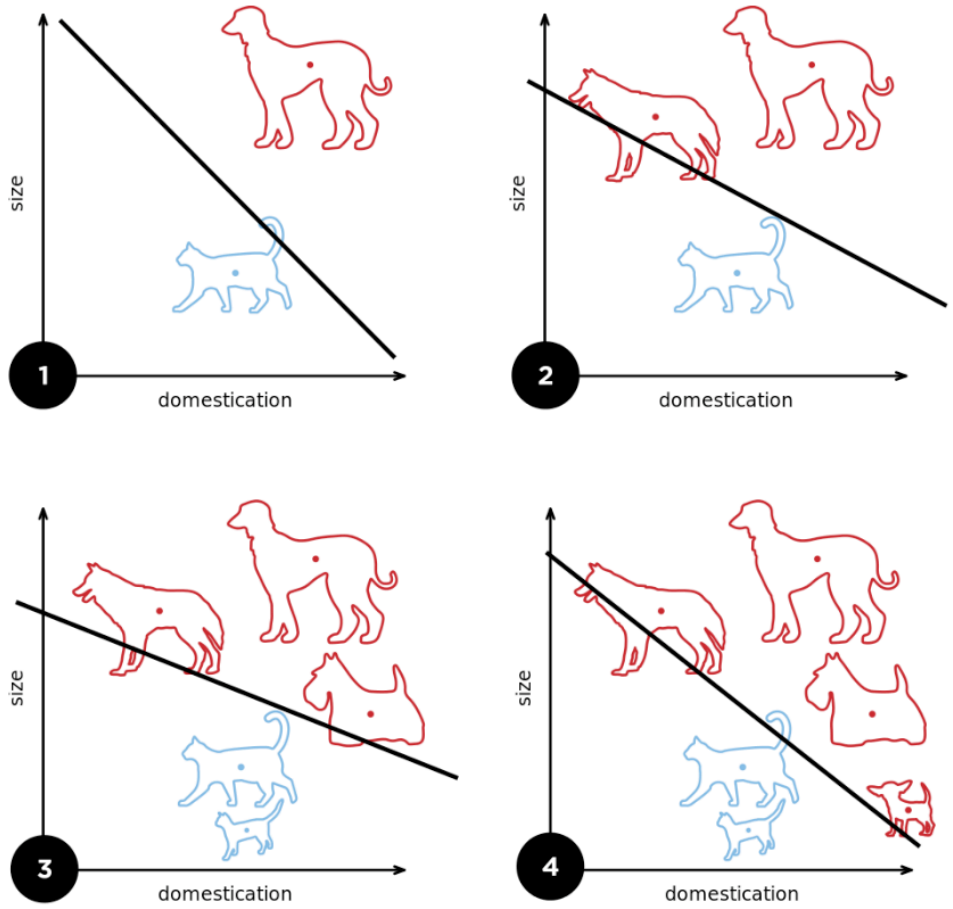
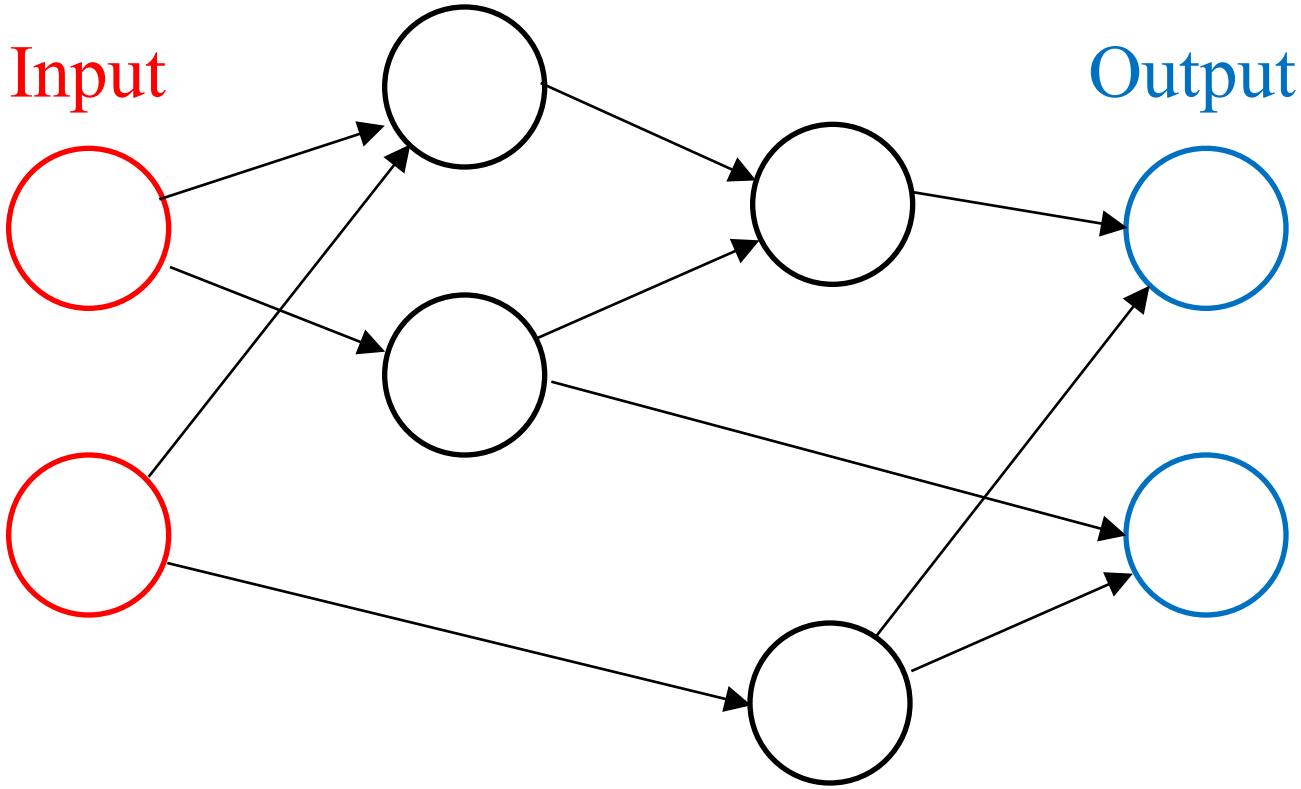


Diagram from Wikipedia

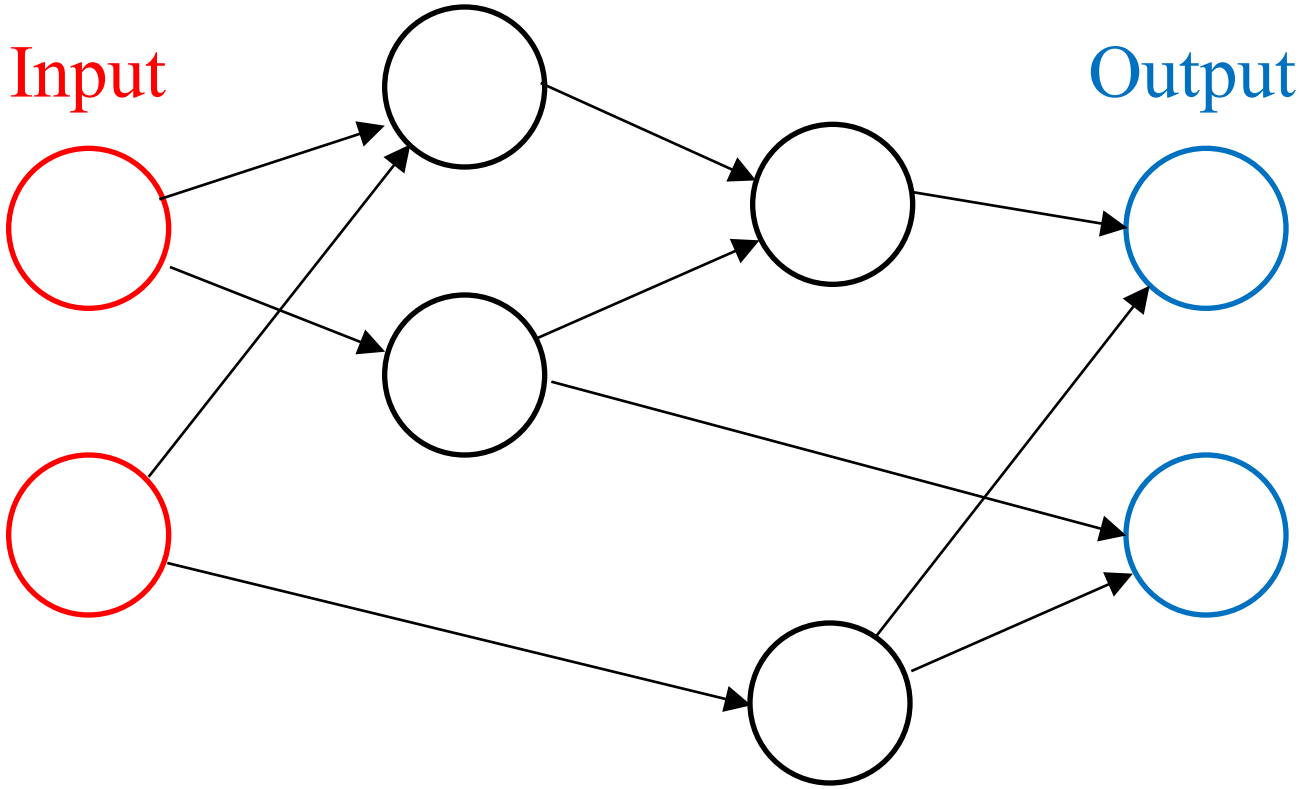
Feedforward neural networks

- We could connect units (neurons) in any arbitrary graph



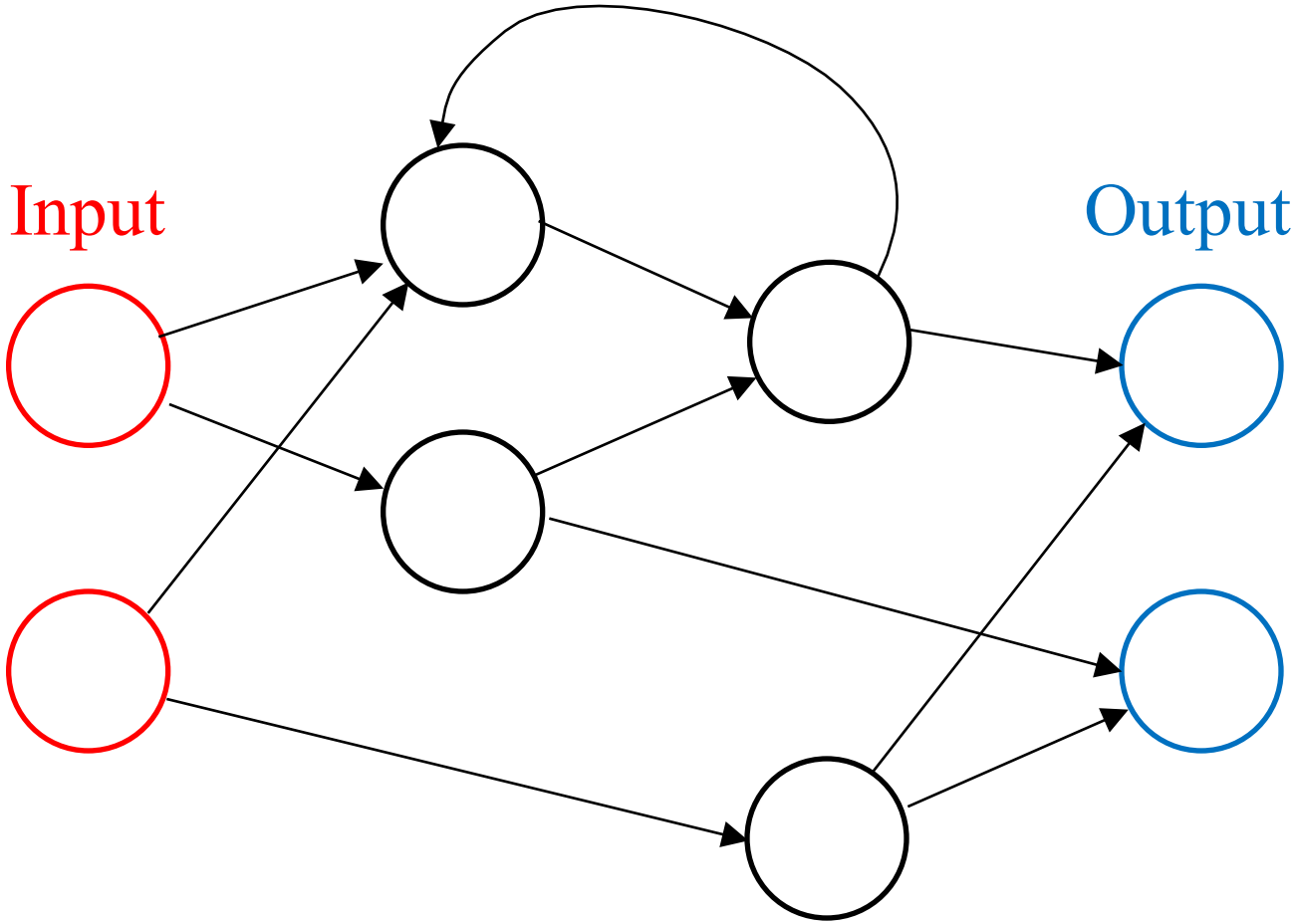
Feedforward neural networks

- We could connect units (neurons) in any arbitrary graph
- If no cycles in the graph we call it a **feedforward neural network**.



Recurrent neural networks (later)

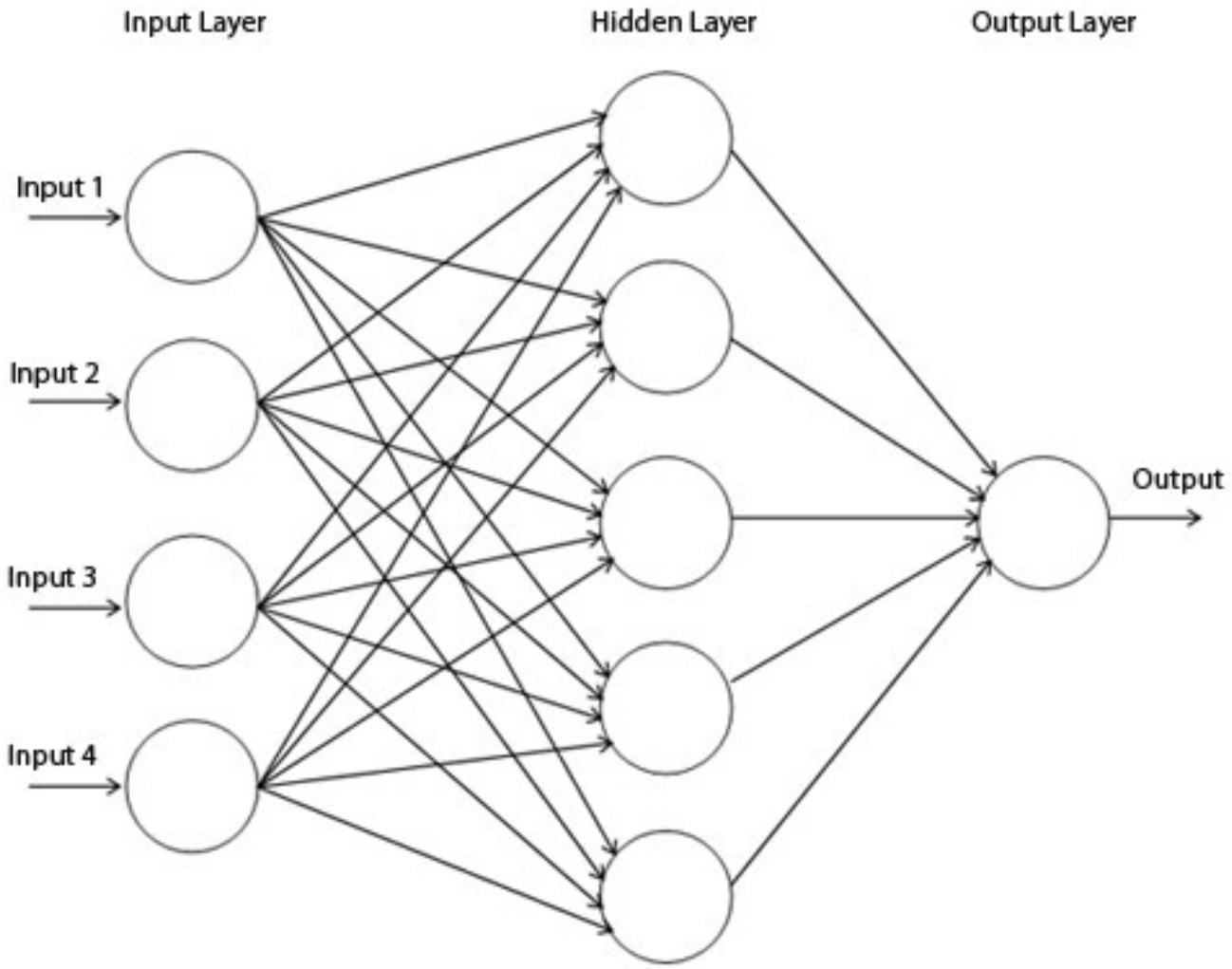
- If cycles in the graph we call it a **recurrent neural network**.



Overview

- Simple neural networks
  - Perceptron
  - Feedforward neural networks
  - **Multilayer perceptron and properties**
  - Autoencoders
- How to train neural networks
  - Gradient descent
  - Stochastic gradient descent
  - Automatic differentiation
  - Backpropagation

Multilayer Perceptron (1960s)



In matrix notation:

$$\mathbf{L}_i = f_i(\mathbf{W}_i\mathbf{L}_{i-1} + \mathbf{b}_i), \quad i \geq 1$$

$\mathbf{L}_1$: Input layer (inputs) vector

$\mathbf{L}_2$: Hidden layer vector

$\mathbf{L}_3$: Output layer vector

$\mathbf{W}_i$: Weight matrix for connections from layer i-1 to layer i

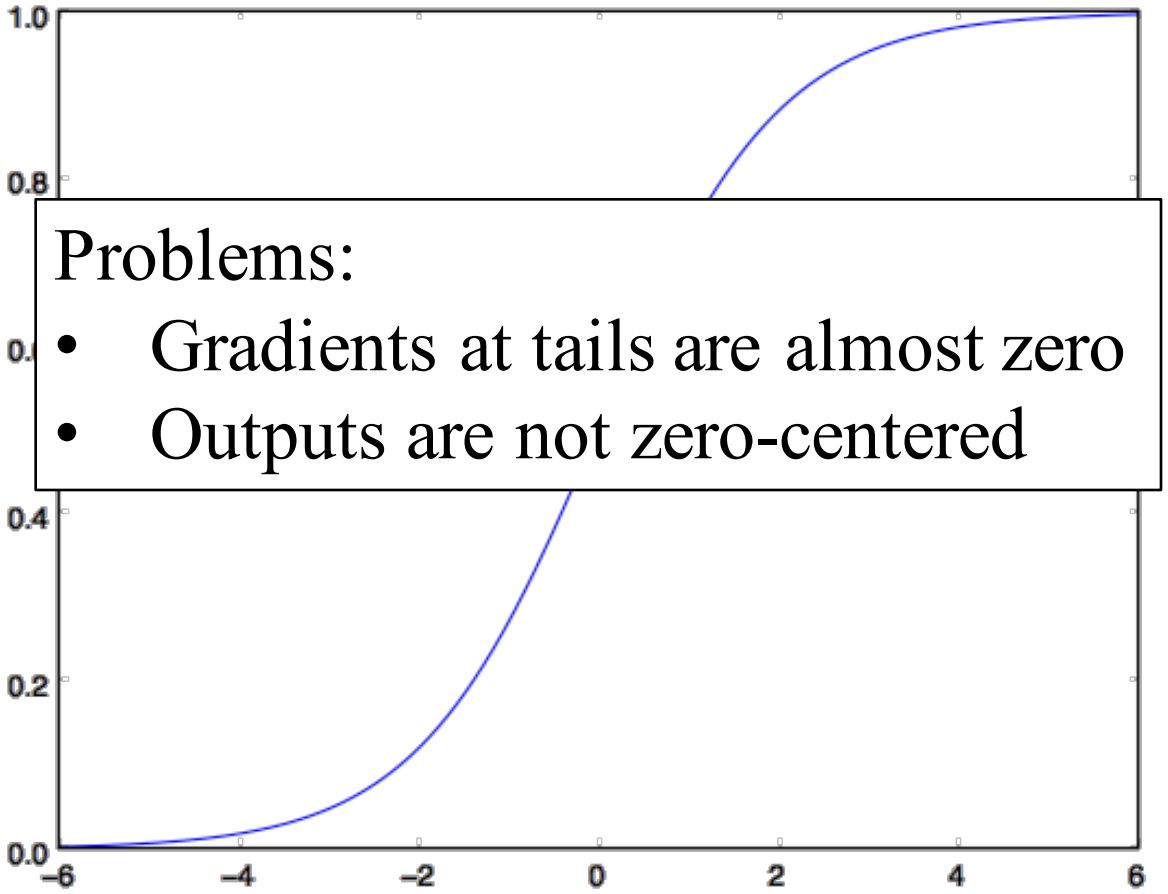
$\mathbf{b}_i$: Biases for neurons in layer i

f_i: Activation function for layer i

Activation Functions: Sigmoid / Logistic

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

$$\frac{df}{dx} = f(x)(1 - f(x))$$



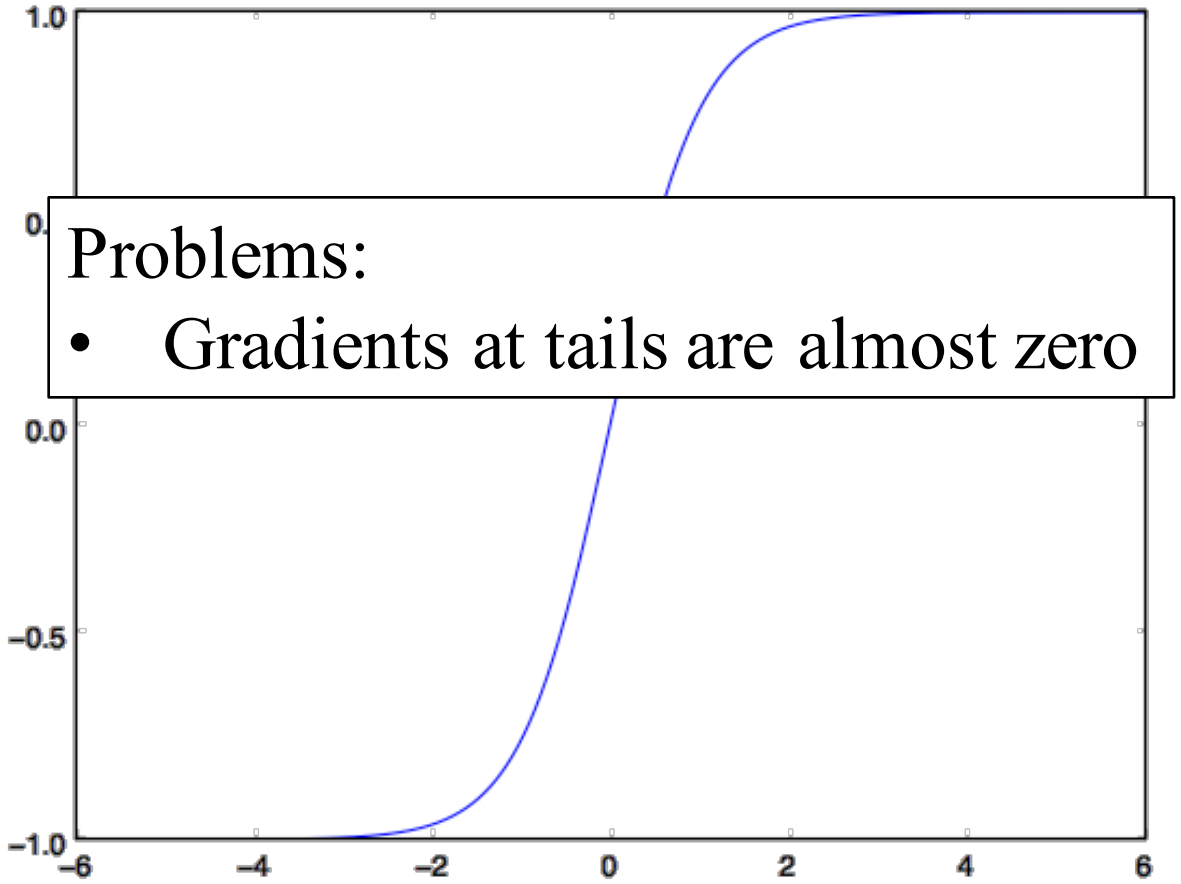
Problems:

- Gradients at tails are almost zero
- Outputs are not zero-centered

Activation Functions: Tanh

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

$$\frac{df}{dx} = 1 - f(x)^2$$



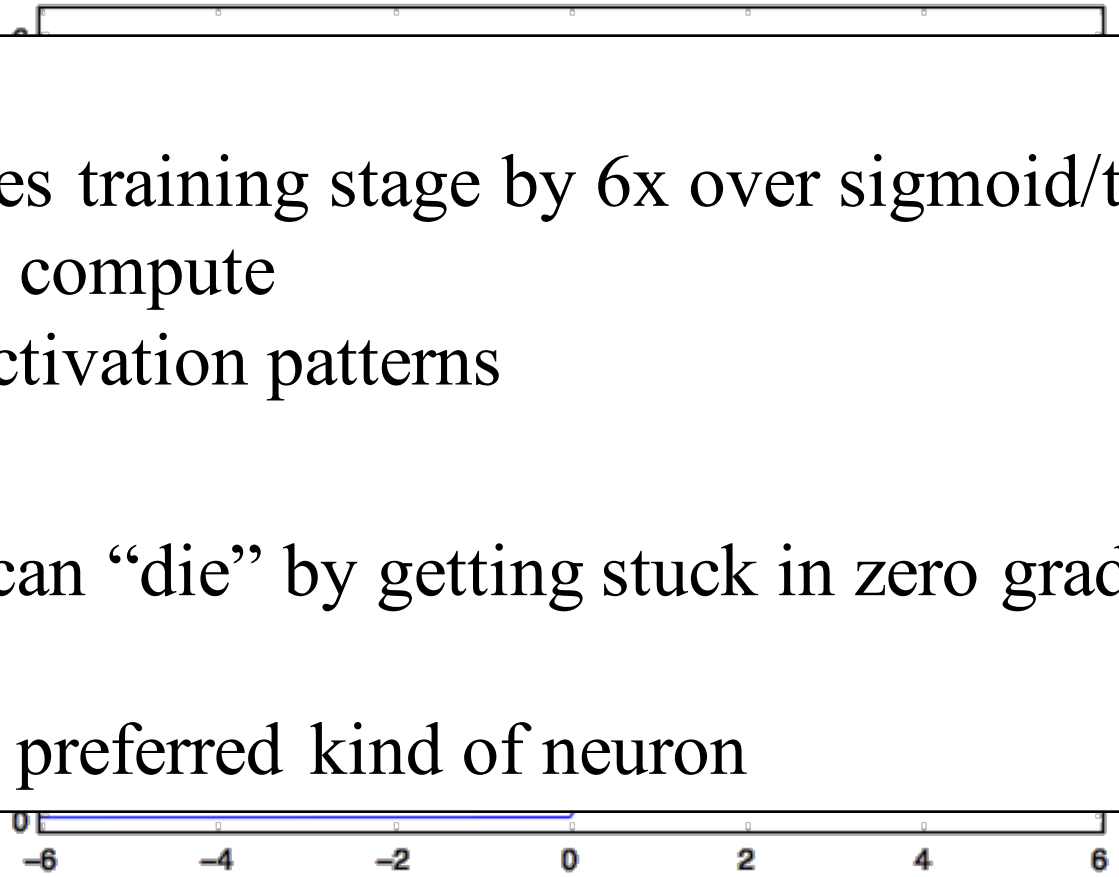
Problems:

- Gradients at tails are almost zero

Activation Functions: ReLU (Rectified Linear Unit)

$$f(x) = \max(x, 0)$$

$$\frac{df}{dx} = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x < 0 \end{cases}$$



Pros:

- Accelerates training stage by 6x over sigmoid/tanh [1]
- Simple to compute
- Sparser activation patterns

Cons:

- Neurons can “die” by getting stuck in zero gradient region

Summary:

- Currently preferred kind of neuron

Universal Approximation Theorem

- Multilayer perceptron with a single hidden layer and linear output layer can approximate any continuous function on a compact subset of $\mathbb{R}^n$ to within any desired degree of accuracy.
- Assumes activation function is bounded, non-constant, monotonically increasing.
- Also applies for ReLU activation function.

Universal Approximation Theorem

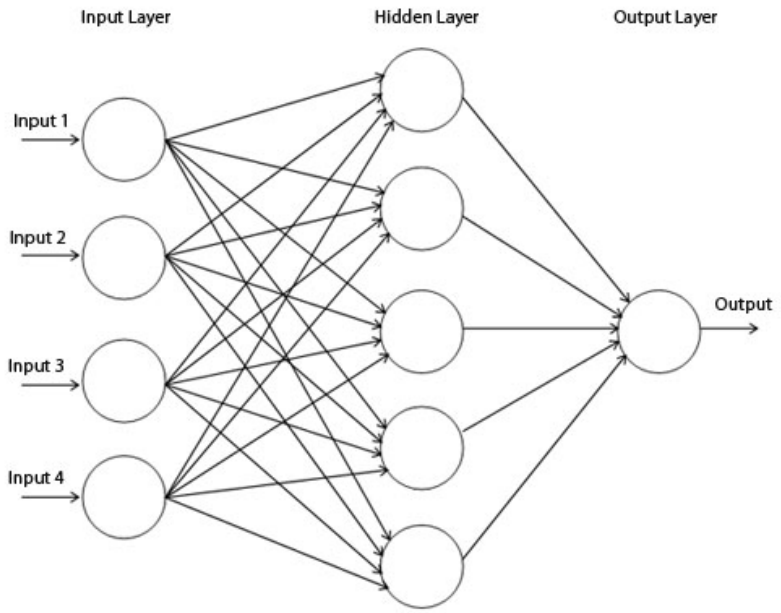
- In the worst case, exponential number of hidden units may be required.
- Can informally show this for binary case:
  - If we have *n* bits input to a binary function, how many possible inputs are there?
  - How many possible binary functions are there?
  - So how many weights do we need to represent a given binary function?

Why Use Deep Networks?

- Functions representable with a deep rectifier network can require an exponential number of hidden units with a shallow (one hidden layer) network (Goodfellow 6.4)
- Piecewise linear networks (e.g. using ReLU) can represent functions that have a number of regions exponential in depth of network.
  - Can capture repeating / mirroring / symmetric patterns in data.
  - Empirically, greater depth often results in better generalization.

Neural Network Architecture

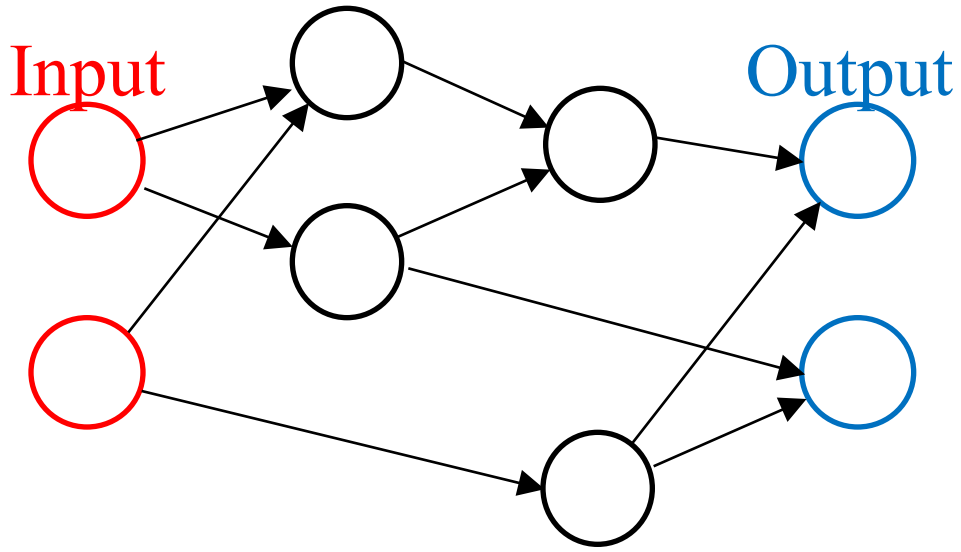
- **Architecture**: refers to which parameters (e.g. weights) are used in the network and their topological connectivity.
- **Fully connected:** A common connectivity pattern for multilayer perceptrons. All possible connections made between layers *i*-1 and *i*.



Is this network fully connected?

Neural Network Architecture

- **Architecture**: refers to which parameters (e.g. weights) are used in the network and their topological connectivity.
- **Fully connected:** A common connectivity pattern for multilayer perceptrons. All possible connections made between layers *i*-1 and *i*.



Is this network fully connected?

How to Choose Network Architecture?

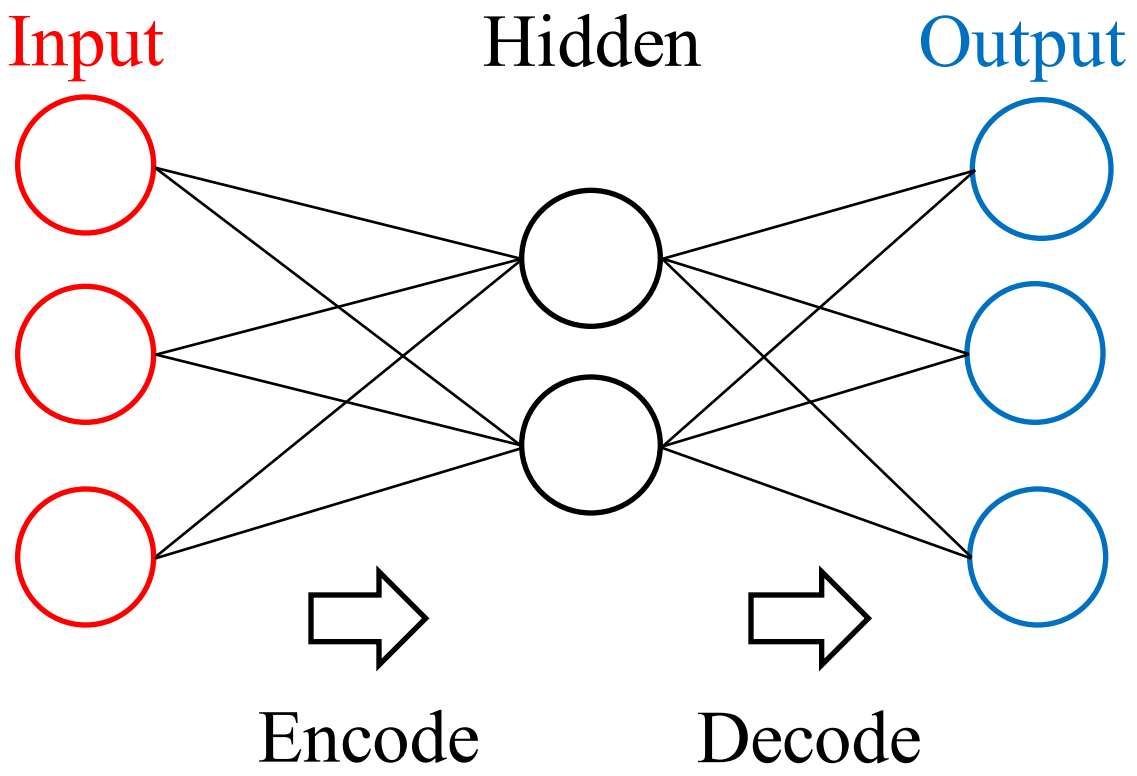
- Long discussion
- Summary:
  - Rules of thumb do not work.
    - “Need 10x [or 30x] more training data than weights.”
    - Not true if very low noise
    - Might need even more training data if high noise
  - Try many networks with different numbers of units and layers
  - Check generalization using validation dataset or cross-validation.

Overview

- Simple neural networks
  - Perceptron
  - Feedforward neural networks
  - Multilayer perceptron and properties
  - **Autoencoders**
- How to train neural networks
  - Gradient descent
  - Stochastic gradient descent
  - Automatic differentiation
  - Backpropagation

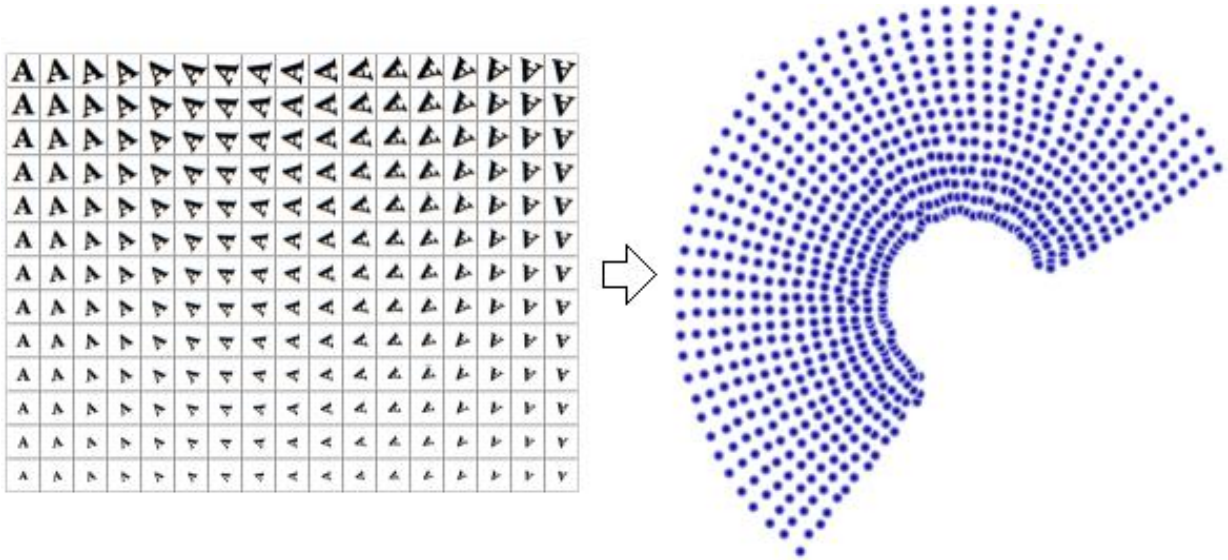
Autoencoders

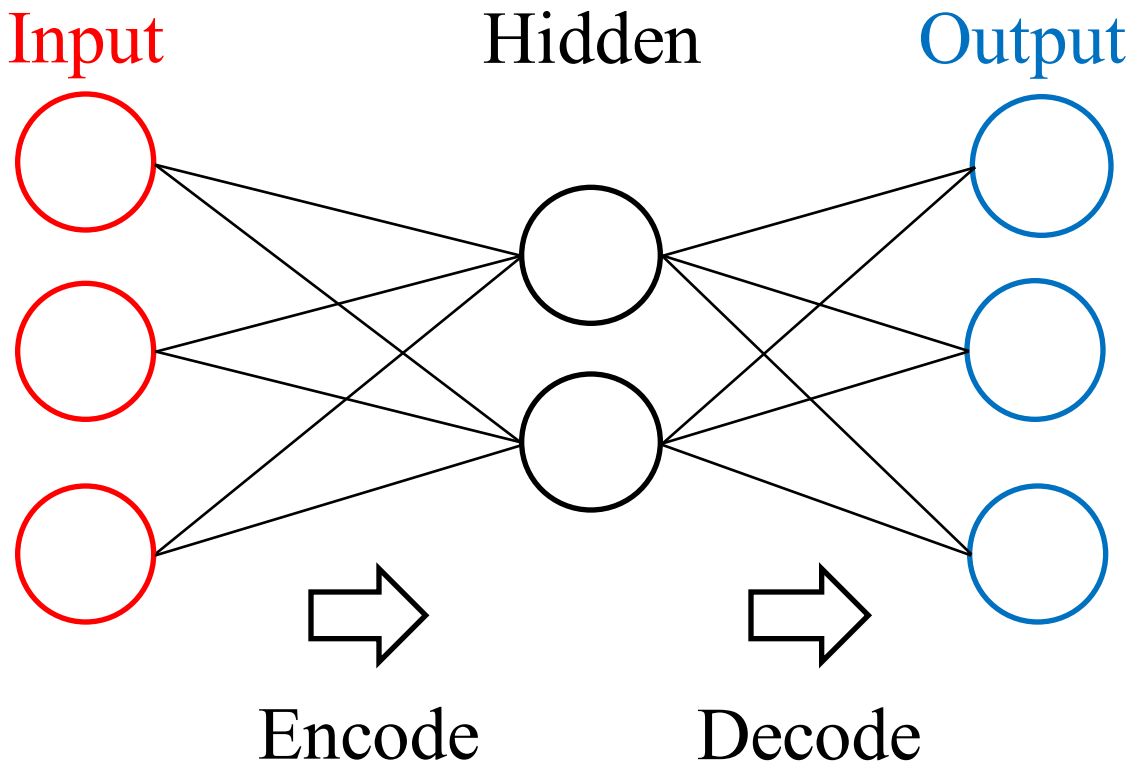
- Learn the identity function $h_w(\mathbf{x}) = \mathbf{x}$
- Is this supervised or unsupervised learning?



Autoencoders

- Applications:
  - Dimensionality reduction
  - Learning manifolds
  - Hashing for search problems





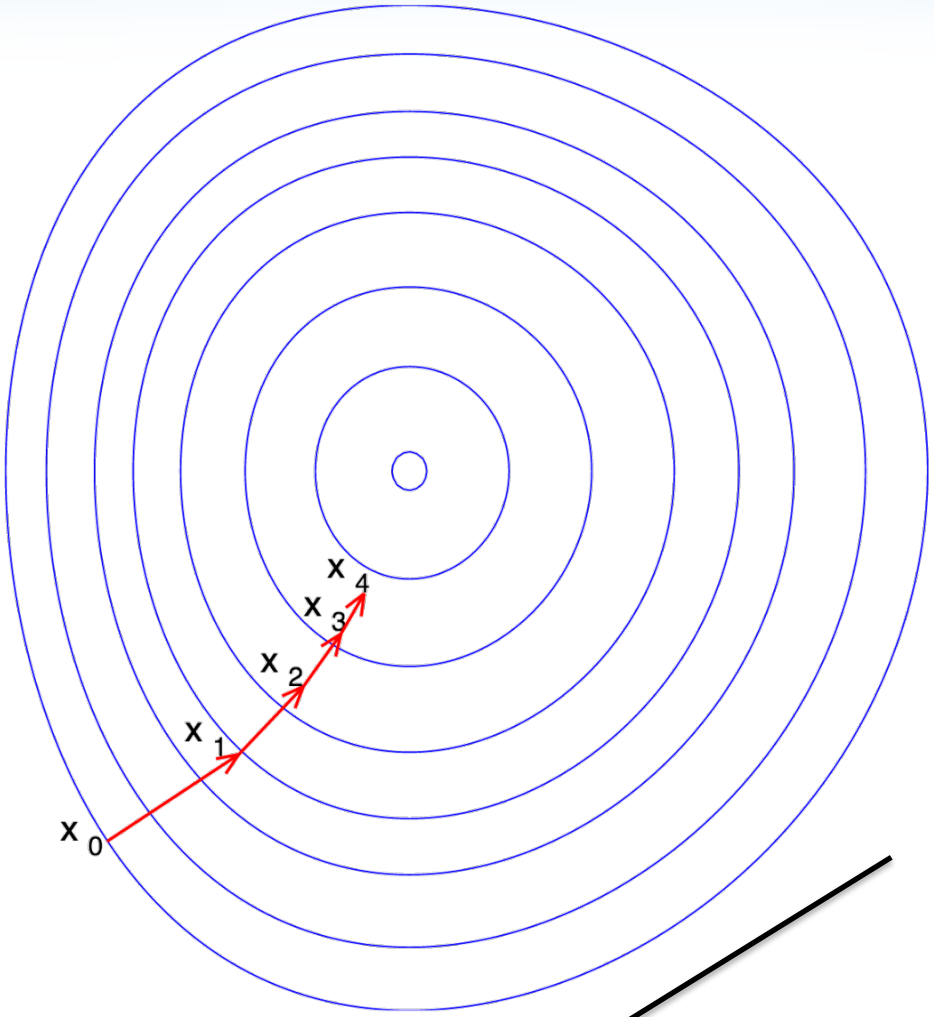
Overview

- Simple neural networks
  - Perceptron
  - Feedforward neural networks
  - Multilayer perceptron and properties
  - Autoencoders
- **How to train neural networks**
  - Gradient descent
  - Stochastic gradient descent
  - Automatic differentiation
  - Backpropagation

Gradient Descent

Discuss amongst students near you:

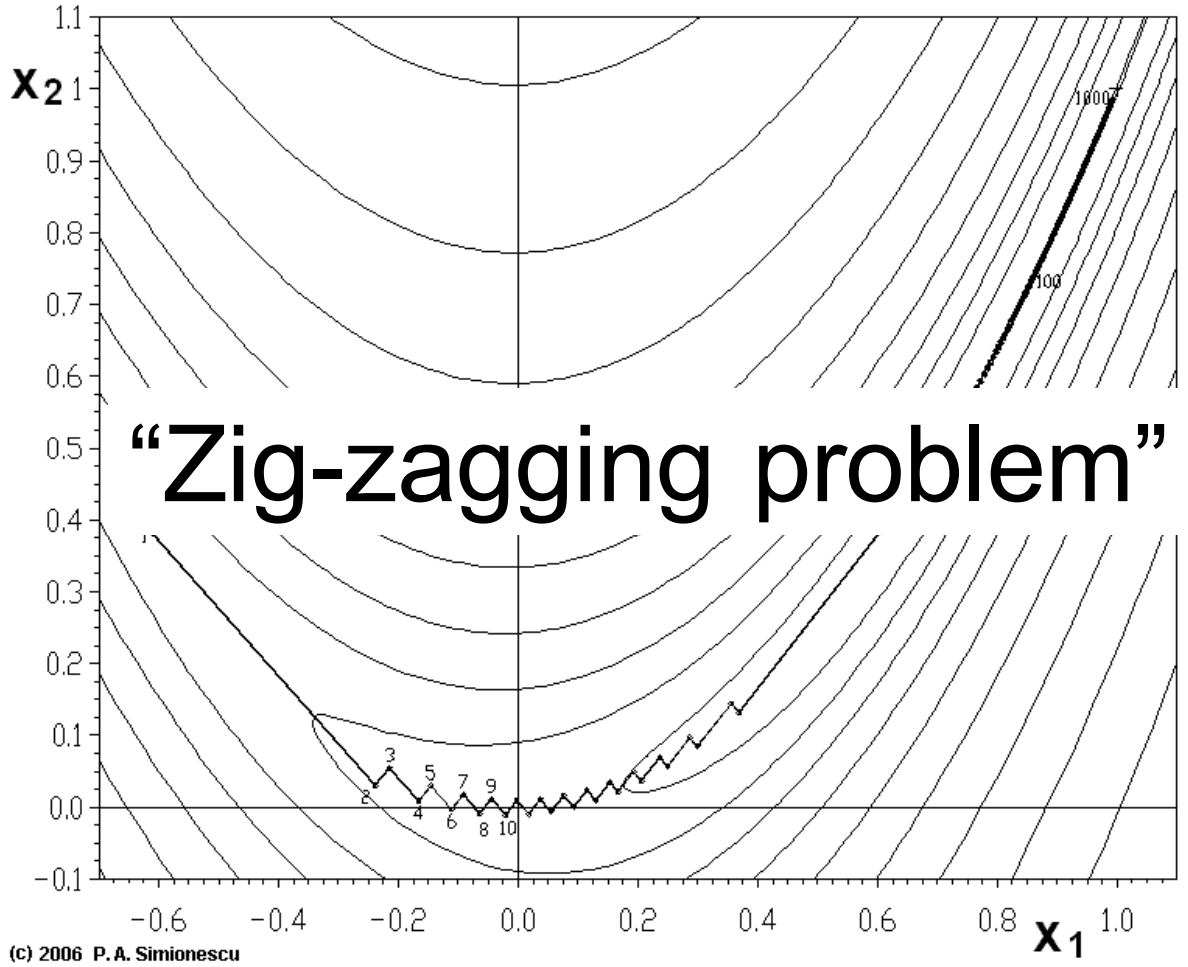
- What are some problems that could be easily optimized with gradient descent?
- Problems where this is difficult?
- Should the learning rate be constant or change?



Step size
or
Learning rate

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma_n \nabla F(\mathbf{x}_n)$$

Gradient Descent with Energy Functions that have Narrow Valleys



"Zig-zagging problem"

Source: "Banana-SteepDesc" by P.A. Simionescu – Wikipedia English

Gradient Descent with Momentum

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \Delta_n$$

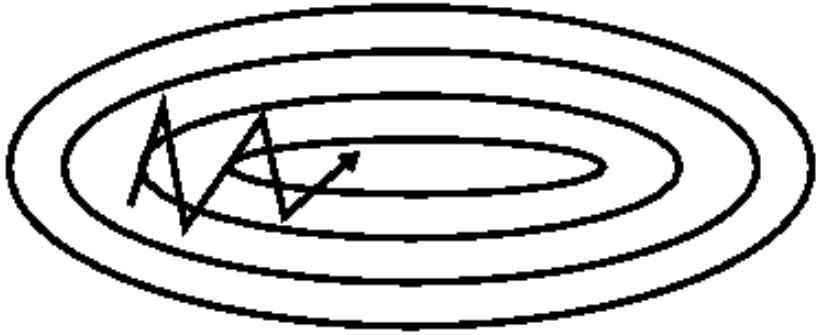
$$\Delta_n = \gamma_n \nabla F(\mathbf{x}_n) + m\Delta_{n-1}$$

Momentum

Could use small value e.g. *m*=0.5 at first
Could use larger value e.g. *m*=0.9 near end of training when there are more oscillations.

Gradient Descent with Momentum

Without Momentum



With Momentum

Figure from Genevieve B. Orr, Willamette.edu

Stochastic gradient descent

- Stochastic gradient descent (Wikipedia)
- Gradient of sum of n terms where n is large
- Sample rather than computing the full sum
  - Sample size *s* is “mini-batch size”
  - Could be 1 (very noisy gradient estimate)
  - Could be 100 (collect photos 100 at a time to find each noisy “next” estimate for the gradient)
- Use same step as in gradient descent to the estimated gradient

Stochastic gradient descent

- Pseudocode:

  - Repeat until an approximate minimum is obtained:
    - Randomly shuffle examples in the training set.
    - For $i = 1, 2, \ldots, n$, do:
      - $w := w - \eta \nabla Q_i(w)$.

From Wikipedia

Problem Statement

- Take the gradient of an arbitrary program or model (e.g. a neural network) with respect to the parameters in the model (e.g. weights).
- If we can do this, we can use gradient descent!

Review: Chain Rule in One Dimension

- Suppose $f: \mathbb{R} \to \mathbb{R}$ and $g: \mathbb{R} \to \mathbb{R}$
- Define

$$h(x) = f(g(x))$$

- Then what is $h'(x) = dh/dx$?

$$h'(x) = f'\big(g(x)\big)g'(x)$$

Chain Rule in Multiple Dimensions

- Suppose $f: \mathbb{R}^m \to \mathbb{R}$ and $g: \mathbb{R}^n \to \mathbb{R}^m$, and $\mathbf{x} \in \mathbb{R}^n$
- Define

$$h(\mathbf{x}) = f(g_1(\mathbf{x}), \ldots, g_m(\mathbf{x}))$$

- Then we can define partial derivatives using the multidimensional chain rule:

$$\frac{\partial f}{\partial x_i} = \sum_{l=1}^{m} \frac{\partial f}{\partial g_j} \frac{\partial g_j}{\partial x_i}$$

The Problem with Symbolic Derivatives

- What if our program takes 5 exponential operations:

$$y = \exp\left(\exp\left(\exp\left(\exp\left(\exp(x)\right)\right)\right)\right)$$

- What is *dy*/*dx*? (blackboard)

- How many exponential operations in the resulting expression?
- What if the program contained *n* exponential operations?

Solution: Automatic Differentiation (1960s, 1970s)

- Write an arbitrary program as consisting of basic operations f_1, …, f_n (e.g. +, -, *, cos, sin, …) that we know how to differentiate.
- Label the inputs of the program as $x_1, \ldots, x_n$, and the output x_N.
- The computation:

For $i = n + 1, \ldots, N$

$$x_i = f_i(\mathrm{x}_{\pi(i)})$$

$\pi(i)$: Sequence of "parent" values
(e.g. if $\pi(3) = (1,2)$, and f_3=+, then x_3=x_1+x_2)

- Reverse mode automatic differentiation: apply the chain rule from the end of the program x_N back towards the beginning.

Explanation from Justin Domke

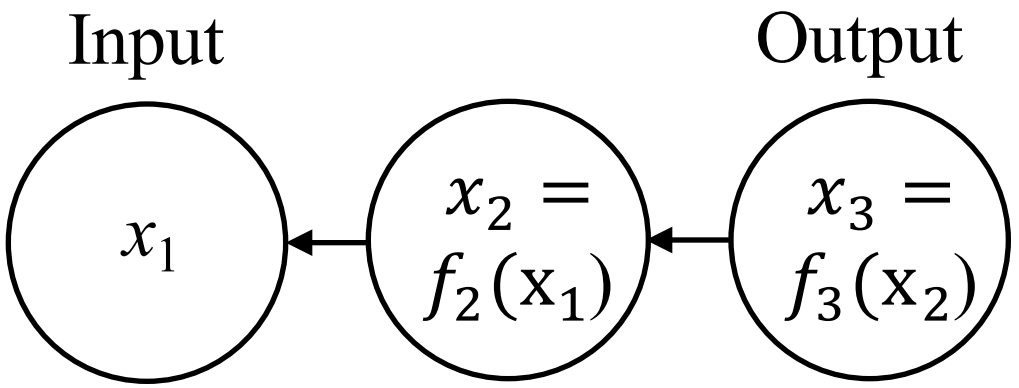
Solution for Simplified Chain of Dependencies

- Suppose $\pi(i) = i - 1$
- The computation:

  For $i = n + 1, \ldots, N$

  $x_i = f_i(\mathrm{x}_{i-1})$

For example:



- What is $\frac{dx_N}{dx_N}$?

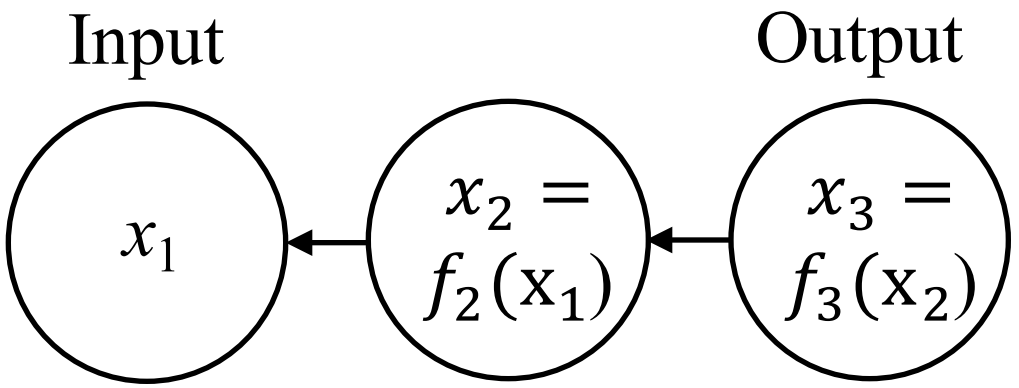
Solution for Simplified Chain of Dependencies

- Suppose $\pi(i) = i - 1$
- The computation:

For $i = n + 1, \ldots, N$

$$x_i = f_i(\mathrm{x}_{i-1})$$

For example:



$$\frac{dx_N}{dx_N} = 1$$

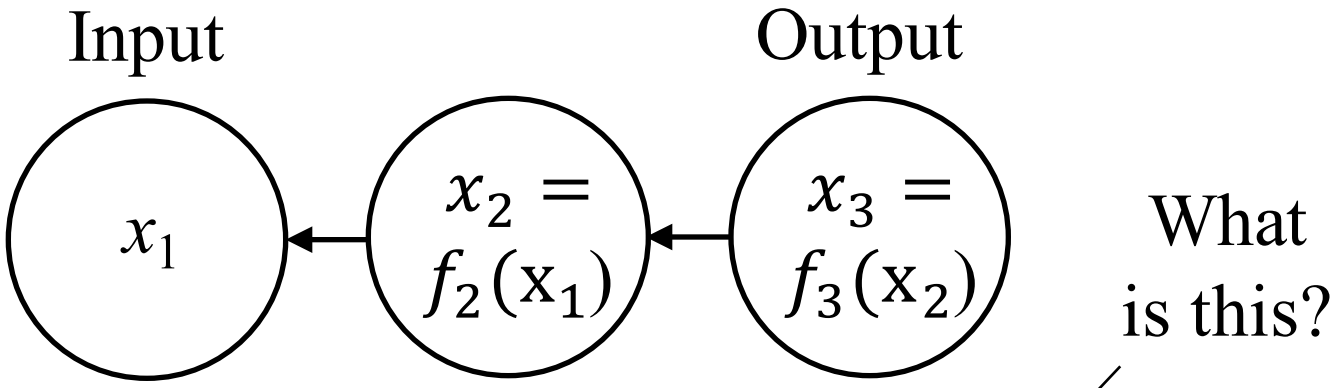
Solution for Simplified Chain of Dependencies

- Suppose $\pi(i) = i - 1$
- The computation:

For $i = n + 1, \ldots, N$

$$x_i = f_i(\mathrm{x}_{i-1})$$

For example:



- What is $\frac{dx_N}{dx_i}$ in terms of $\frac{dx_N}{dx_{i+1}}$?

$$\frac{dx_N}{dx_i} = \frac{dx_N}{dx_{i+1}}\left(\frac{\partial x_{i+1}}{\partial x_i}\right)$$

$$dx_{i+1} = dx_i\left(\frac{\partial x_{i+1}}{\partial x_i}\right) \quad \Rightarrow \quad \frac{dx_{i+1}}{dx_N} = \frac{dx_i}{dx_N}\left(\frac{\partial x_{i+1}}{\partial x_i}\right)$$

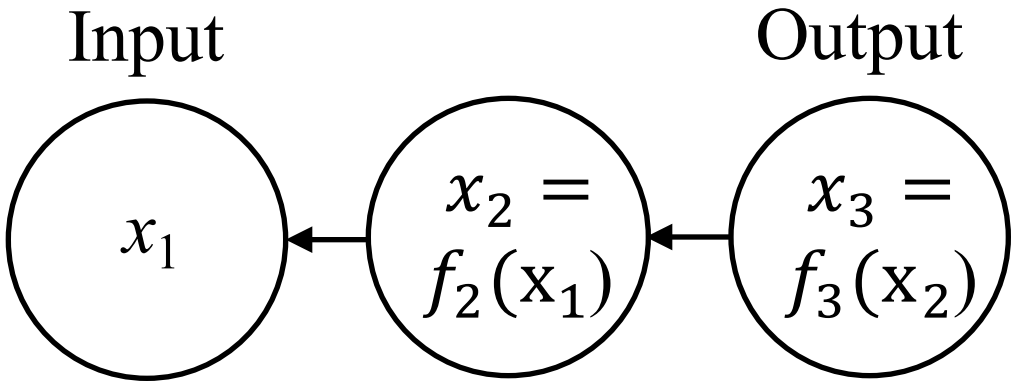
Solution for Simplified Chain of Dependencies

- Suppose $\pi(i) = i - 1$
- The computation:

  For $i = n + 1, \ldots, N$

  $x_i = f_i(\mathrm{x}_{i-1})$

For example:

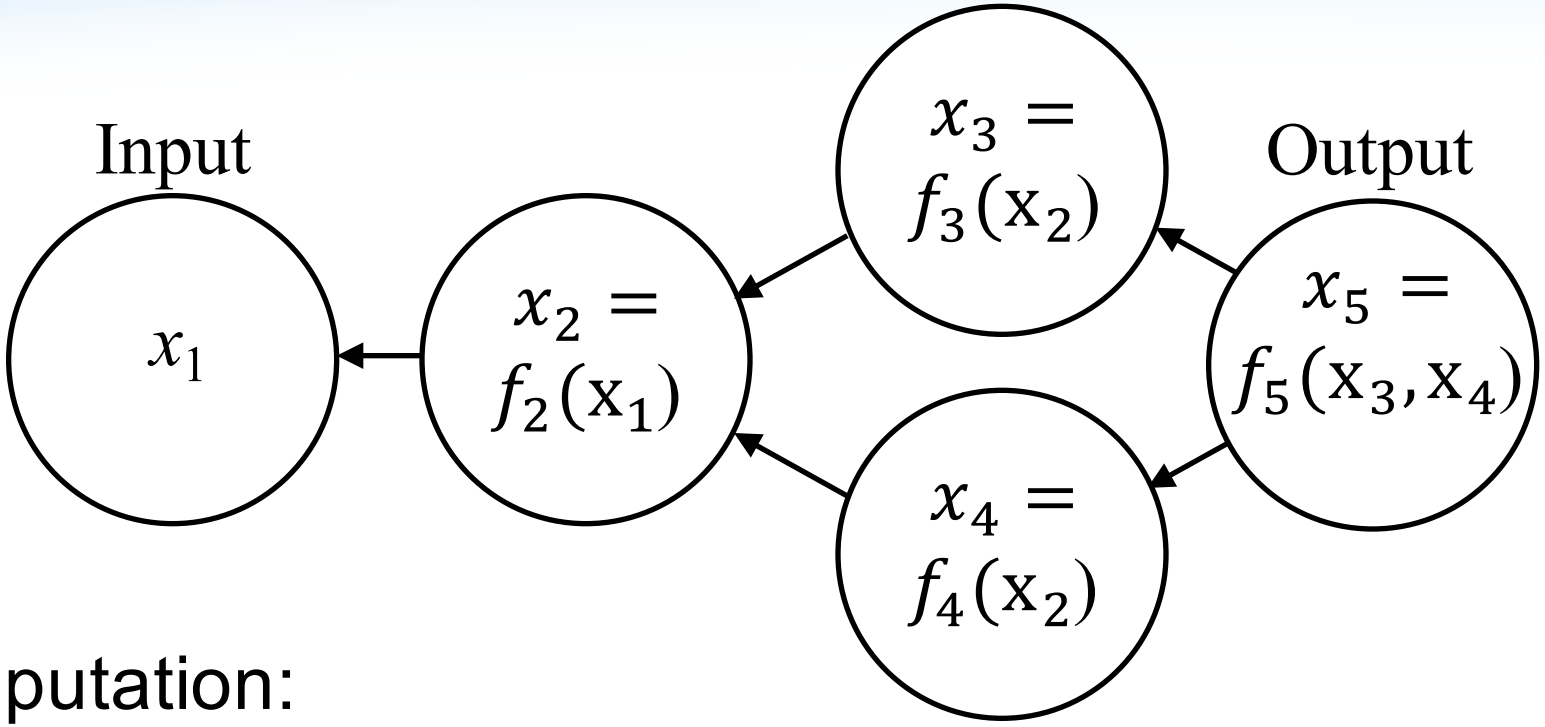


- What is $\frac{dx_N}{dx_i}$ in terms of $\frac{dx_N}{dx_{i+1}}$?

$$\frac{dx_N}{dx_i} = \frac{dx_N}{dx_{i+1}}\left(\frac{\partial x_{i+1}}{\partial x_i}\right)$$

- Conclusion: run the computation forwards. Then initialize $\frac{dx_N}{dx_N} = 1$ and work **backwards** through the computation to find $\frac{dx_N}{dx_i}$ for each *i* from $\frac{dx_N}{dx_{i+1}}$. This gives us the gradient of the output (x_N) with respect to every expression in our compute graph!

What if the Dependency Graph is More Complex?



- The computation:

  For $i = n+1, \ldots, N$

  $x_i = f_i(x_{\pi(i)})$

  $\pi(i)$: Sequence of "parent" values

  (e.g. if $\pi(3) = (1,2)$, and f_3=+, then x_3=x_1+x_2)

- Solution: apply multi-dimensional chain rule.

Solution: Automatic Differentiation (1960s, 1970s)

- Computation:

$$x_i = f_i(\mathrm{x}_{\pi(i)})$$

- Multidimensional chain rule:

$$\frac{\partial f}{\partial x_i} = \sum_{l=1}^{m} \frac{\partial f}{\partial g_j} \frac{\partial g_j}{\partial x_i}$$

$$f(g_1(\mathbf{x}), \ldots, g_m(\mathbf{x}))$$

- Result:

$$\frac{dx_N}{dx_i} = \sum_{j:i \in \pi(j)} \frac{dx_N}{dx_j} \frac{\partial x_j}{\partial x_i}$$

Explanation from Justin Domke

Solution: Automatic Differentiation (1960s, 1970s)

- Back-propagation algorithm: initialize:

$$\frac{dx_N}{dx_N} = 1$$

- For $i = N - 1, N - 2, \ldots, 1$, compute:

$$\frac{dx_N}{dx_i} = \sum_{j:i \in \pi(j)} \frac{dx_N}{dx_j} \frac{\partial x_j}{\partial x_i}$$

Example on blackboard for a program with one addition.

- Now we have differentiated the output of the program x_N with respect to the inputs $x_1, \ldots, x_n$, as well as every computed expression x_i.

Explanation from Justin Domke

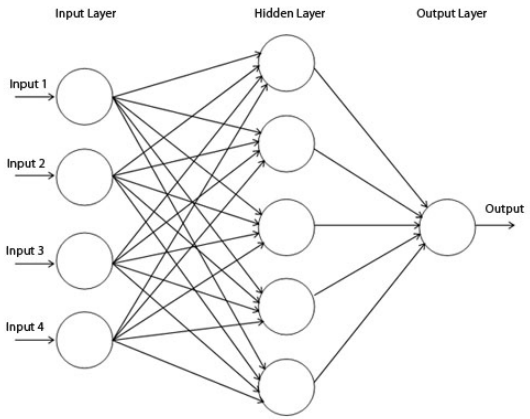
Backpropagation Algorithm (1960s-1980s)

- Apply **reverse mode automatic differentiation** to a neural network's loss function.
- A special case of what we just derived.
- If we have one output neuron, squared error is:

$$E = \frac{1}{2}(t - y)^2$$

t is the target output for a training sample, and

y is the actual output of the output neuron.



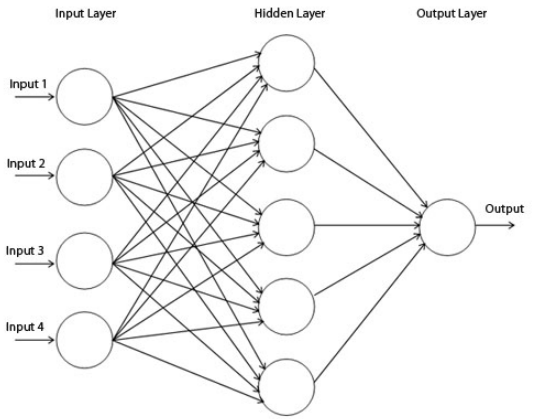
From Wikipedia

Backpropagation Algorithm (1960s-1980s)

For each neuron j, its output o_j is defined as

$$o_j = \varphi(\mathrm{net}_j) = \varphi\left(\sum_{k=1}^{n} w_{kj} o_k\right).$$

The input net_j to a neuron is the weighted sum of outputs o_k of previous neurons. If The variable w_{ij} denotes the weight between neurons i and j.



From Wikipedia

Backpropagation Algorithm (1960s-1980s)

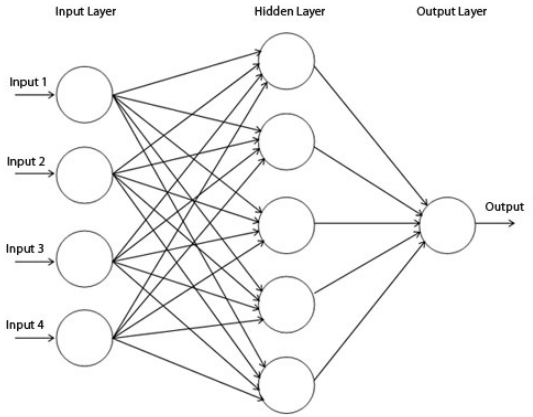
- Apply the chain rule twice:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial \mathrm{net_j}} \frac{\partial \mathrm{net_j}}{\partial w_{ij}}$$

- Last term is easy:

$$o_j = \varphi(\mathrm{net}_j) = \varphi\left(\sum_{k=1}^{n} w_{kj} o_k\right).$$

$$\frac{\partial \mathrm{net_j}}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\left(\sum_{k=1}^{n} w_{kj} o_k\right) = o_i$$



From Wikipedia

Backpropagation Algorithm (1960s-1980s)

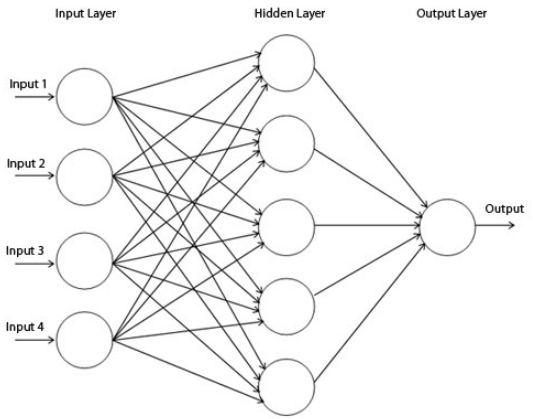
- Apply the chain rule twice:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial \mathrm{net_j}} \frac{\partial \mathrm{net_j}}{\partial w_{ij}}$$

- Second term is easy:

$$o_j = \varphi(\mathrm{net}_j)$$

$$\frac{\partial o_j}{\partial net_j} = \varphi'(\mathrm{net}_j)$$



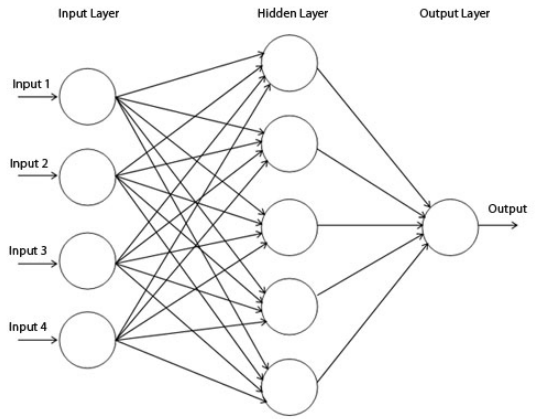
From Wikipedia

Backpropagation Algorithm (1960s-1980s)

$$E = \frac{1}{2}(t - y)^2$$

- Apply the chain rule twice:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial \mathrm{net_j}} \frac{\partial \mathrm{net_j}}{\partial w_{ij}}$$



- If the neuron is in output layer, first term is easy:

$$o_j = y$$

$$\frac{\partial E}{\partial o_j} = \text{(Derivation on board)}$$

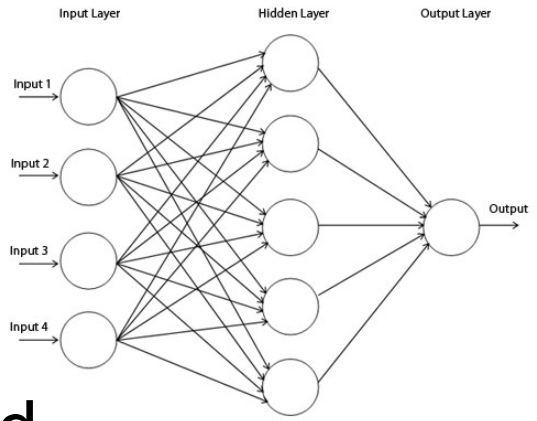
From Wikipedia

Backpropagation Algorithm (1960s-1980s)

$$E = \frac{1}{2}(t - y)^2$$

- Apply the chain rule twice:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial \mathrm{net_j}} \frac{\partial \mathrm{net_j}}{\partial w_{ij}}$$



- If the neuron is interior neuron, we use the chain rule from automatic differentiation.
- To do this, we need to know what expressions depend on the current neuron's output o_j?
- Answer: other neurons input sums, i.e. net_l for all neurons l receiving inputs from the current neuron.

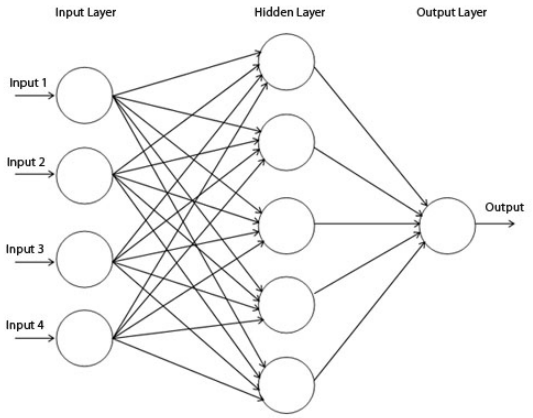
From Wikipedia

Backpropagation Algorithm (1960s-1980s)

$$E = \frac{1}{2}(t - y)^2$$

- Apply the chain rule twice:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial \mathrm{net_j}} \frac{\partial \mathrm{net_j}}{\partial w_{ij}}$$



- If the neuron is an interior neuron, chain rule:

$$\frac{\partial E}{\partial o_j} = \sum_{l \in L} \left(\frac{\partial E}{\partial \mathrm{net}_l} \frac{\partial \mathrm{net}_l}{\partial o_j} \right) = \sum_{l \in L} \left(\frac{\partial E}{\partial o_l} \frac{\partial o_l}{\partial \mathrm{net}_l} w_{jl} \right)$$

All neurons receiving input from the current neuron.

From Wikipedia

Backpropagation Algorithm (1960s-1980s)

- Partial derivative of error E with respect to weight w_{ij}:

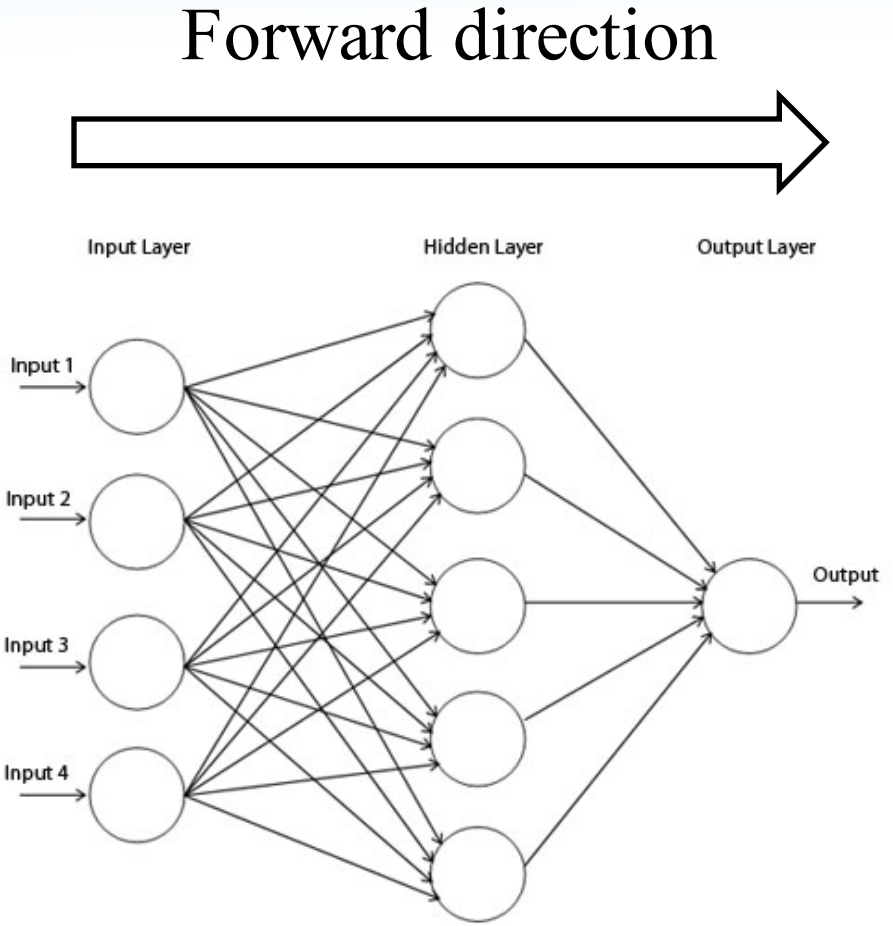
$$\frac{\partial E}{\partial w_{ij}} = \delta_j o_i$$

$$\delta_j = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial \mathrm{net}_j} = \varphi'(o_j)\begin{cases} (o_j - t_j) & \text{if } j \text{ is an output neuron} \\ \sum_{l \in L} \delta_l w_{jl} & \text{if } j \text{ is an interior neuron} \end{cases}$$

All neurons receiving input from the current neuron.

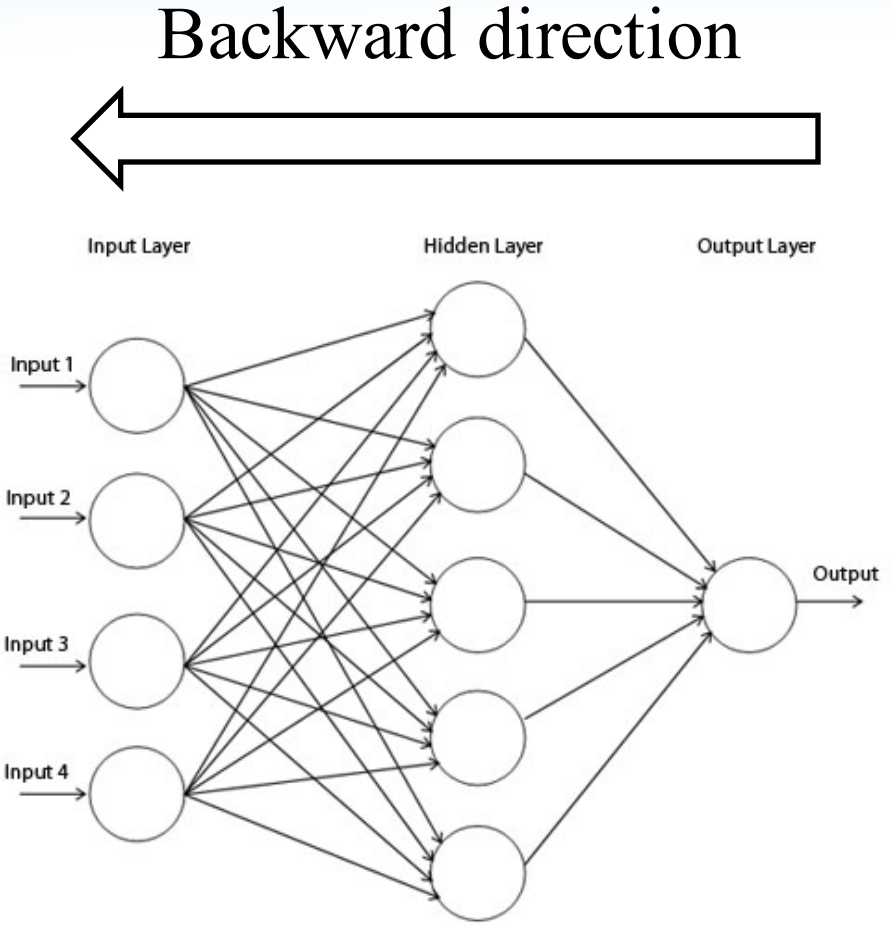
From Wikipedia

Backpropagation Algorithm (1960s-1980s)



- Calculate network and error.

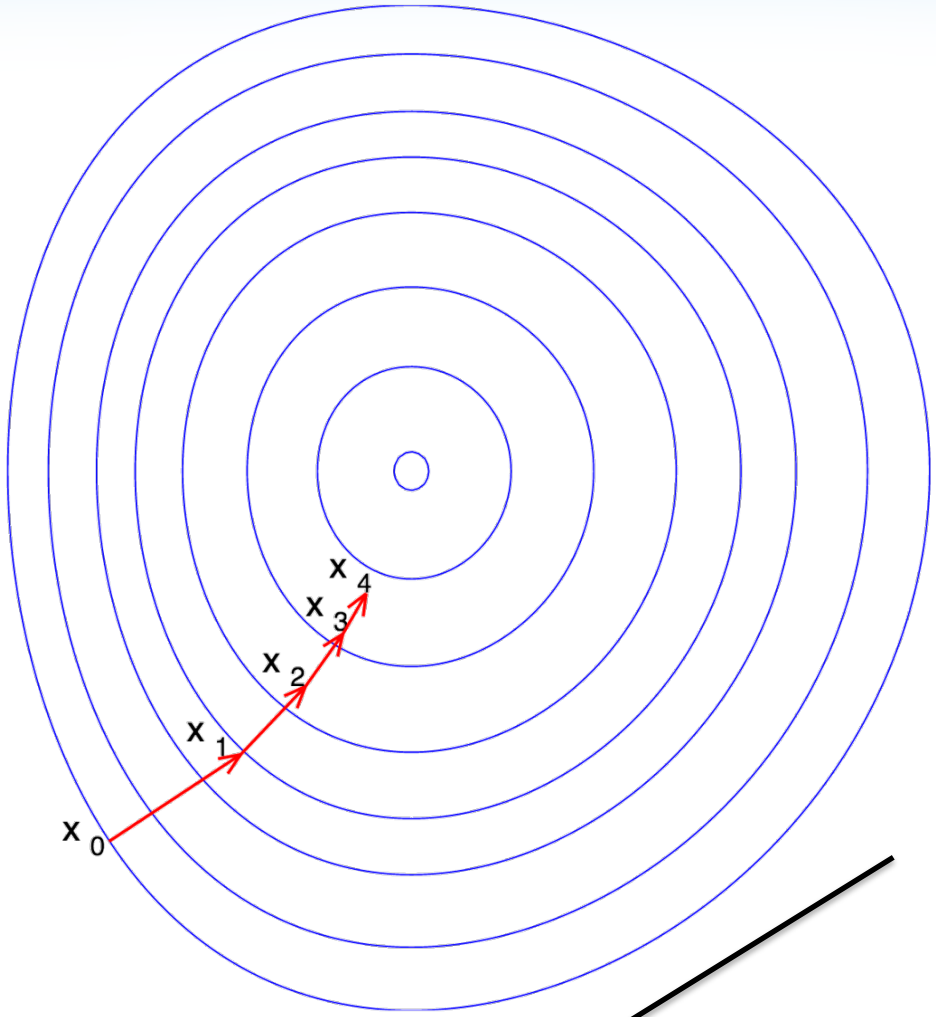
Backpropagation Algorithm (1960s-1980s)



- Backpropagate: from output to input, recursively compute $\frac{\partial E}{\partial w_{ij}} = \nabla_w E$

Gradient Descent with Backpropagation

- Initialize weights at good starting point $\mathbf{w}_0$
- Repeatedly apply gradient descent step (1)
- Continue training until validation error hits a minimum.

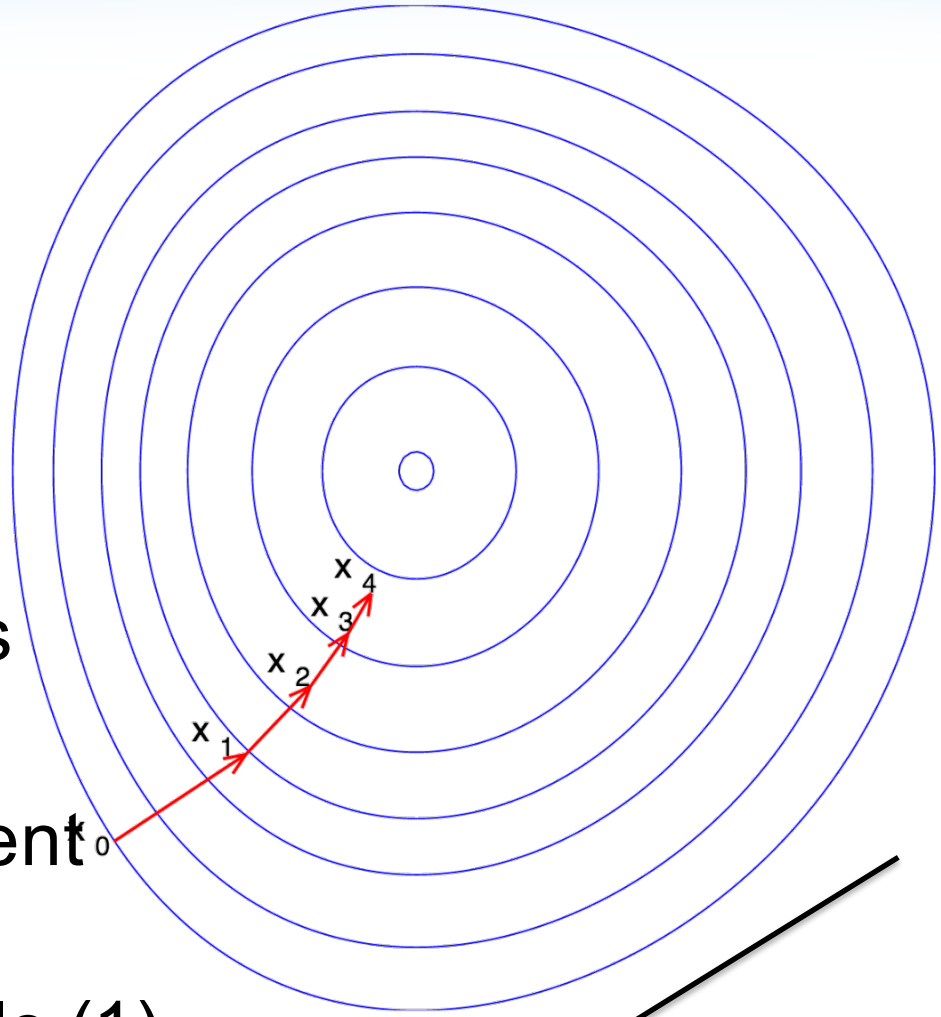


Step size
or
Learning rate

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \gamma_n \nabla_w E(\mathbf{w}_n) \qquad (1)$$

Stochastic Gradient Descent with Backpropagation

- Initialize weights at good starting point $\mathbf{w}_0$
- Repeat until validation error hits a minimum:
  - Randomly shuffle dataset
  - Loop through mini-batches of data, batch index is *i*
  - Calculate stochastic gradient using backpropagation for each, and apply update rule (1)



Step size
or
Learning rate

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \gamma_n \nabla_w E_i\,(\mathbf{w}_n) \qquad (1)$$