Neural Networks

Computer Vision I

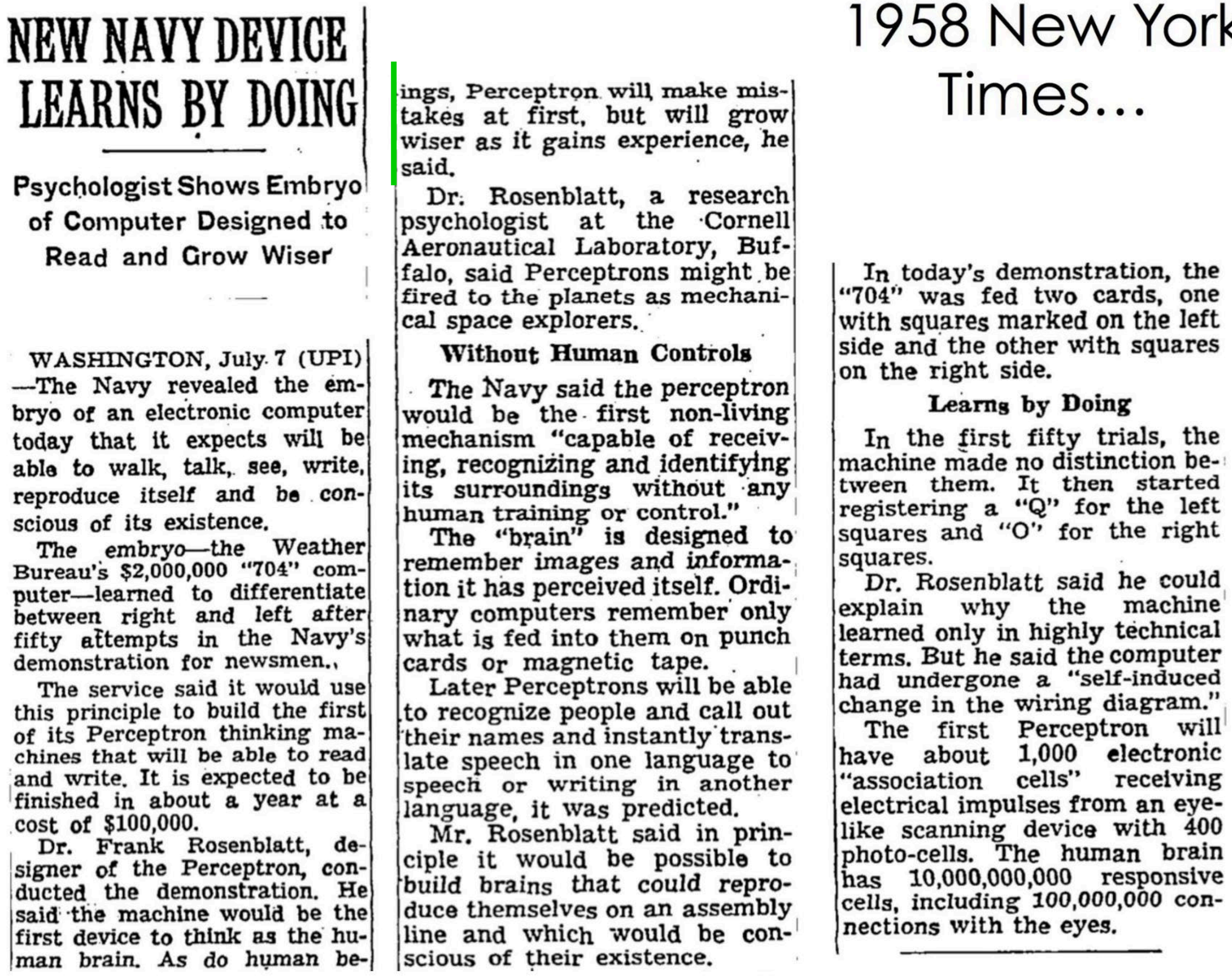
CSE 252A

Lecture 15

Announcements

- Assignment 3 is due Nov 22, 11:59 PM
- Assignment 4 will be released Nov 22
  - Due Dec 6, 11:59 PM

1958 New York Times…



NEW NAVY DEVICE LEARNS BY DOING

Psychologist Shows Embryo of Computer Designed to Read and Grow Wiser

WASHINGTON, July 7 (UPI)—The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.

The embryo—the Weather Bureau's $2,000,000 "704" computer—learned to differentiate between right and left after fifty attempts in the Navy's demonstration for newsmen.,

The service said it would use this principle to build the first of its Perceptron thinking machines that will be able to read and write. It is expected to be finished in about a year at a cost of $100,000.

Dr. Frank Rosenblatt, designer of the Perceptron, conducted the demonstration. He said the machine would be the first device to think as the human brain. As do human beings, Perceptron will make mistakes at first, but will grow wiser as it gains experience, he said.

Dr. Rosenblatt, a research psychologist at the Cornell Aeronautical Laboratory, Buffalo, said Perceptrons might be fired to the planets as mechanical space explorers.

Without Human Controls

The Navy said the perceptron would be the first non-living mechanism "capable of receiving, recognizing and identifying its surroundings without any human training or control."

The "brain" is designed to remember images and information it has perceived itself. Ordinary computers remember only what is fed into them on punch cards or magnetic tape.

Later Perceptrons will be able to recognize people and call out their names and instantly translate speech in one language to speech or writing in another language, it was predicted.

Mr. Rosenblatt said in principle it would be possible to build brains that could reproduce themselves on an assembly line and which would be conscious of their existence.

In today's demonstration, the "704" was fed two cards, one with squares marked on the left side and the other with squares on the right side.

Learns by Doing

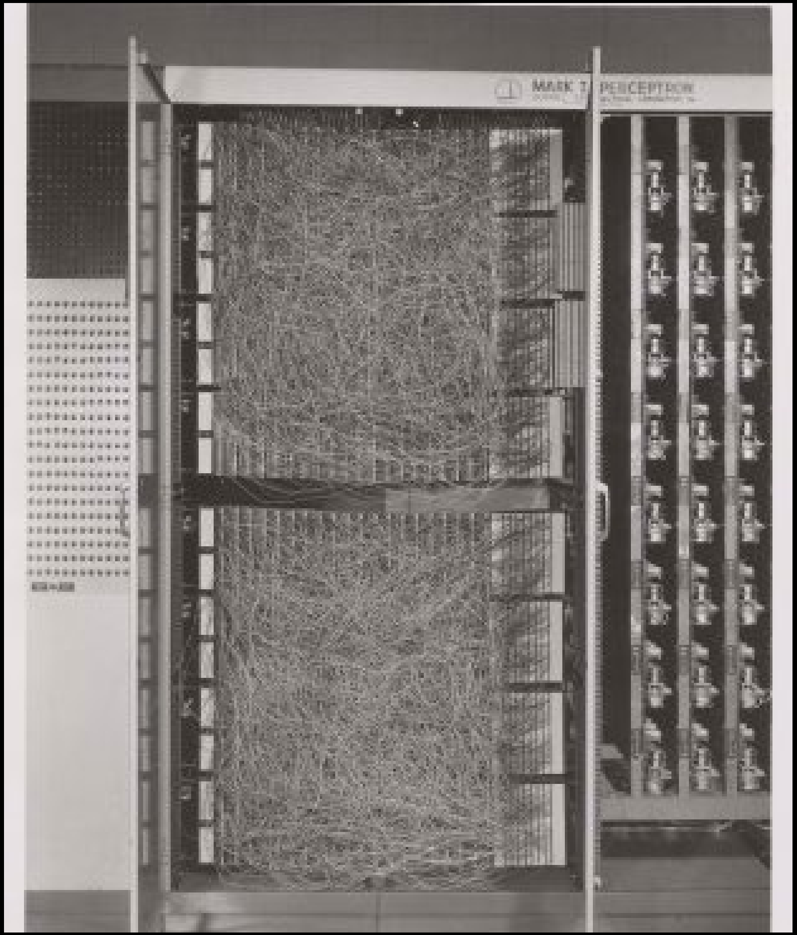
In the first fifty trials, the machine made no distinction between them. It then started registering a "Q" for the left squares and "O" for the right squares.

Dr. Rosenblatt said he could explain why the machine learned only in highly technical terms. But he said the computer had undergone a "self-induced change in the wiring diagram."

The first Perceptron will have about 1,000 electronic "association cells" receiving electrical impulses from an eye-like scanning device with 400 photo-cells. The human brain has 10,000,000,000 responsive cells, including 100,000,000 connections with the eyes.

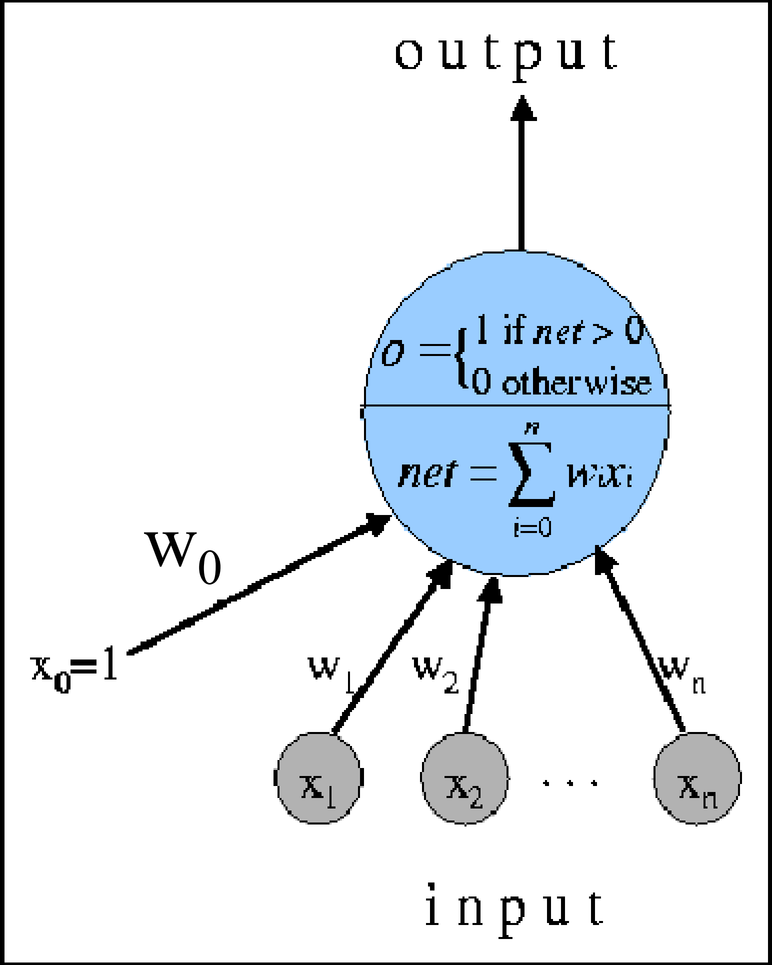
Mark I Perceptron machine

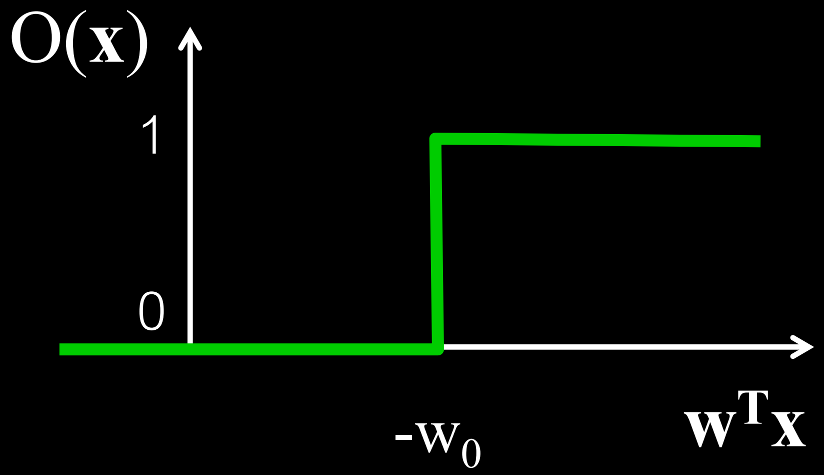
- The Mark I Perceptron machine was the first implementation of the perceptron algorithm. The machine was connected to a camera that used **20×20 cadmium sulfide photocells to produce a 400-pixel image**. The main visible feature is a patchboard that allowed experimentation with different combinations of input features. To the right of that are arrays of potentiometers that implemented the adaptive weights



[From Wikipedia]

Perceptron


output
$o = \begin{cases} 1 \text{ if } net > 0 \\ 0 \text{ otherwise} \end{cases}$
$net = \sum_{i=0}^{n} w_i x_i$
w_0
$x_0=1$
w_1
w_2
w_n
x_1
x_2
...
x_n
input



Note: For $x=(x_1,\ldots,x_2)$, x_i can be binary or a real number

$$o(x_1,\ldots,x_n) = \begin{cases} 1 & \text{if } w_0 + w_1x_1 + \cdots + w_nx_n > 0 \\ 0 & \text{otherwise.} \end{cases}$$

Questions

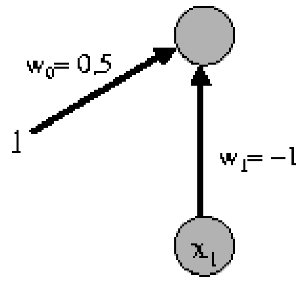
For a Network, even as simple as a single perceptron, we an ask questions:

1. What can be represented with it?
2. How do we evaluate it?
3. How do we train it?

How powerful is a perceptron?

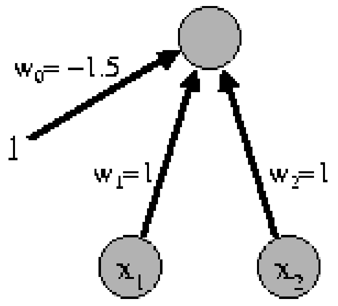
Inverter

input x1	output
0	1
1	0



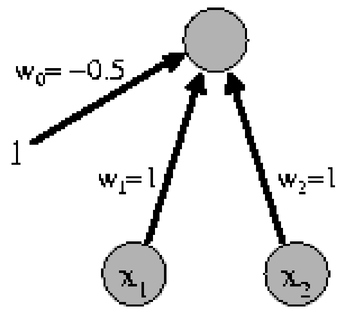
Boolean AND

input x1	input x2	ouput
0	0	0
0	1	0
1	0	0
1	1	1



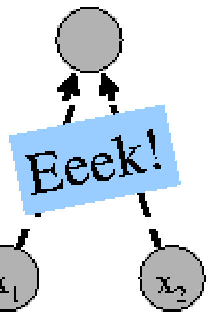
Boolean OR

input x1	input x2	ouput
0	0	0
0	1	1
1	0	1
1	1	1

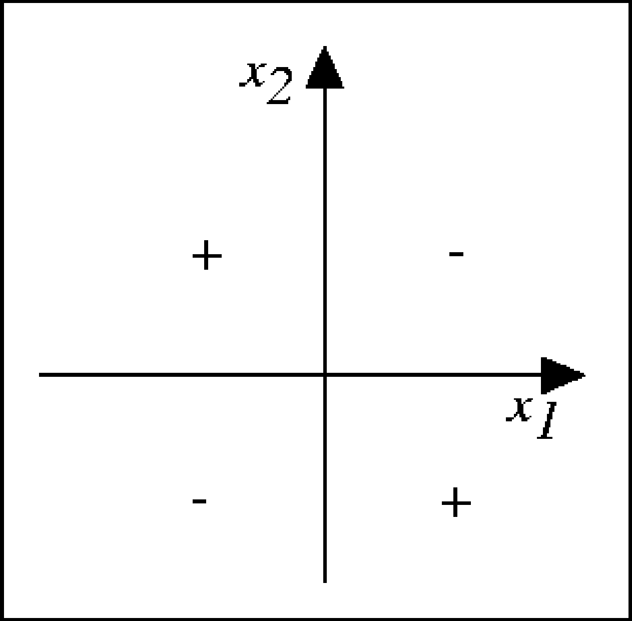


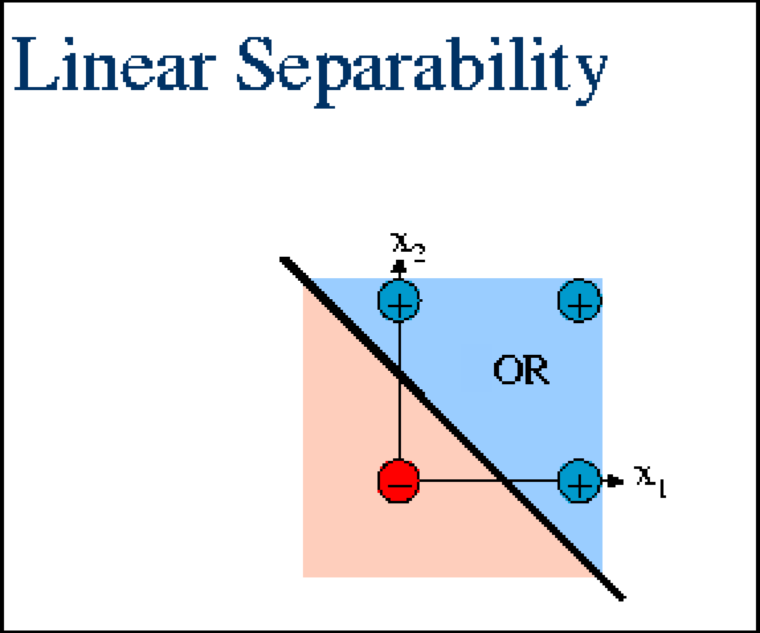
Boolean XOR

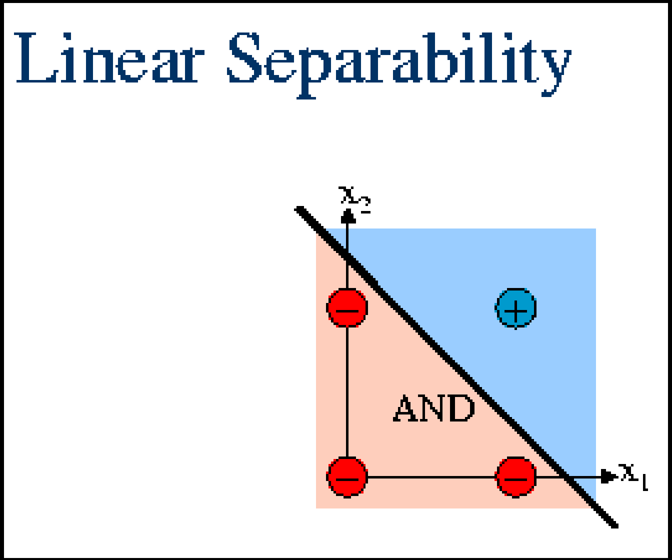
input x1	input x2	ouput
0	0	0
0	1	1
1	0	1
1	1	0

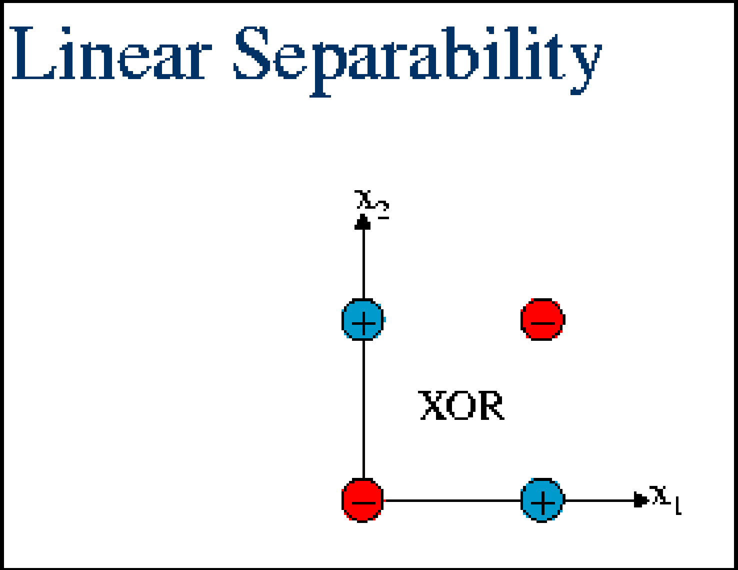


Concept Space & Linear Separability




Linear Separability

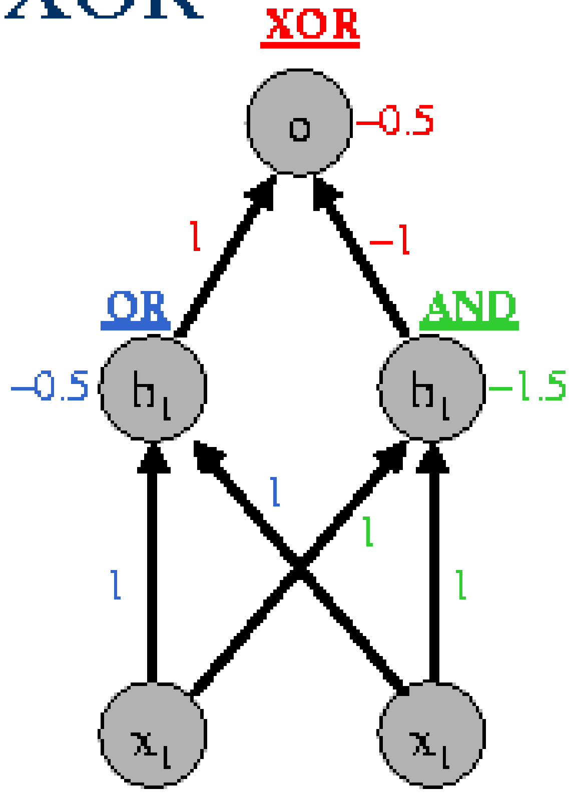

Linear Separability


Linear Separability

Increasing Expressiveness: Multi-Layer Neural Networks

Boolean XOR

input x1	input x2	ouput
0	0	0
0	1	1
1	0	1
1	1	0



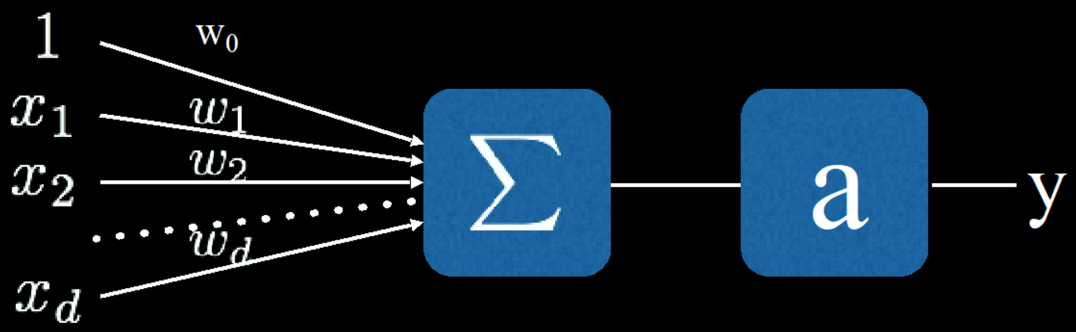
2-layer Perceptron Net

Any Boolean function can be represented by a two layer network!

But where did those weights come from?

Stay tuned

The nodes of multilayered network



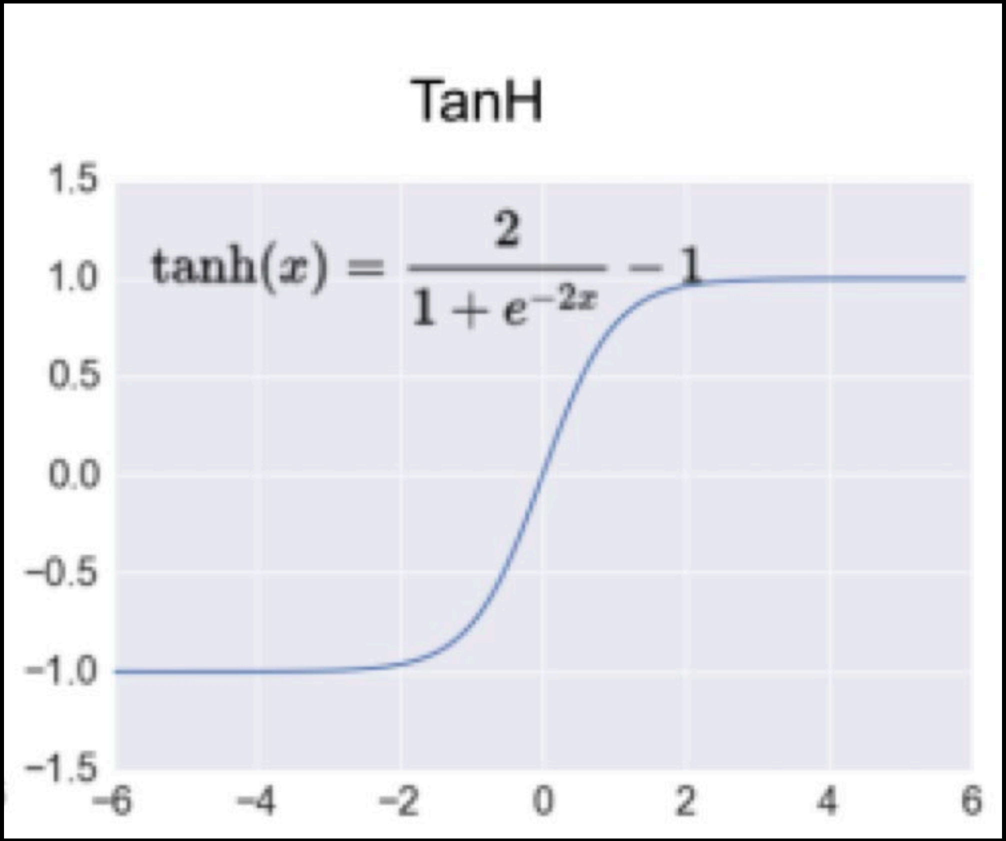
$y(\mathbf{x}; \mathbf{w}) = a(\mathbf{w}^T\mathbf{x}+w_0)$

x: input vector
w: weights
w_0: bias term
a: activation function

$y(\mathbf{x}; \mathbf{w}) = a(\mathbf{w}^T\mathbf{x})$

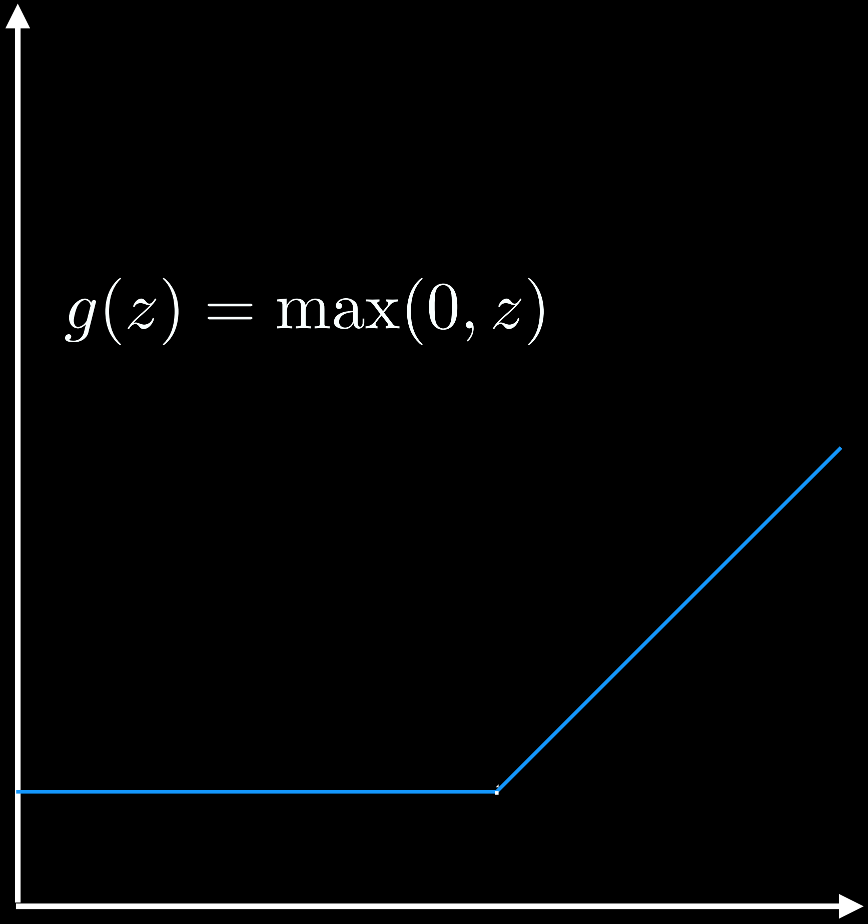
x: input vector padded with 1
w: weights including bias
a: activation function

Activation Function: Tanh



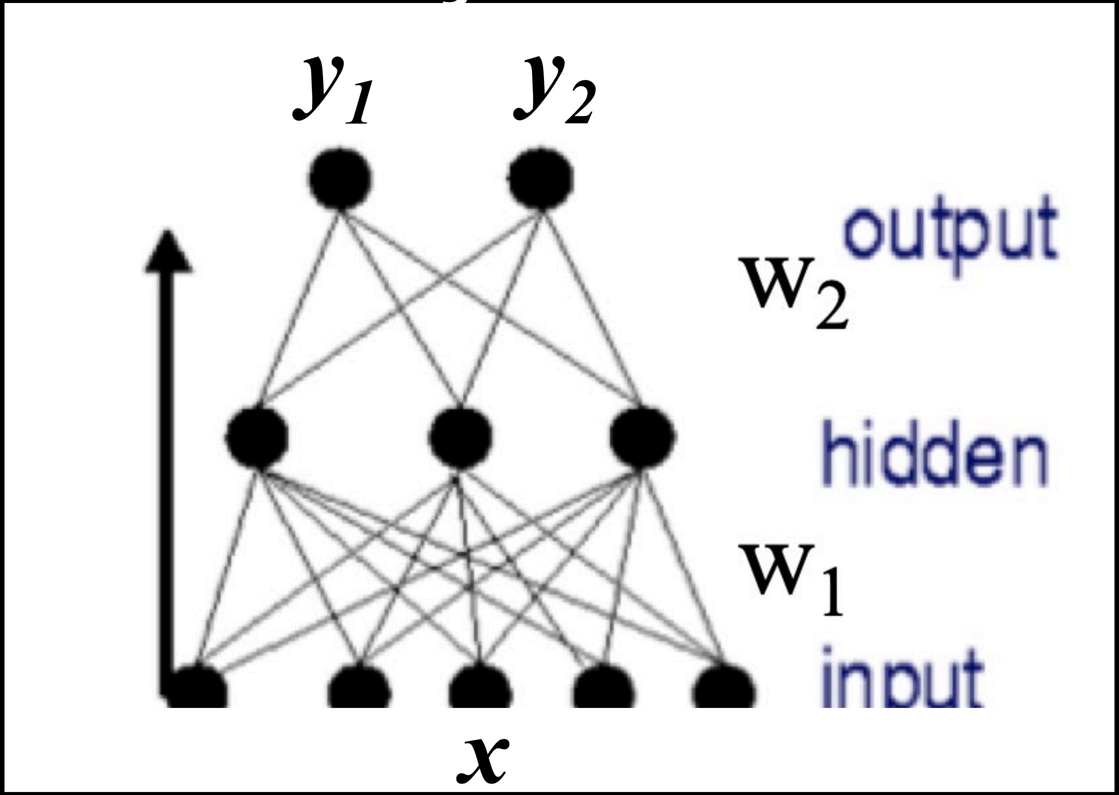
- As x goes from -∞ to ∞, tanh(x) goes from -1 to 1
- It has a "sigmoid" or S-like shape
- tanh(0) = 0

Activation Function Rectified Linear Unit ReLU



$$g(z) = \max(0, z)$$

Two Layer Network



$$y(\boldsymbol{x}, \boldsymbol{w}) = \boldsymbol{a_2}(W_2 a_1(W_1 \boldsymbol{x} + w_{1,0\ d}) + w_{2,0})$$

- Two sets of weights W_1 and W_2
- Two activation functions $\mathbf{a}_1$ and a_2

Feedforward Networks

- These networks are composed of functions represented as **“layers"**

$$y(\mathbf{x}) = a_3(a_2(a_1(\mathbf{x}; w_1); w_2); w_3)$$

with weights w_i associated with layer i and a_i is the activation function for layer i.

- y(***x***) can be a scalar or a vector function.

Classification Networks and Softmax

- To classify the input **x** into one of *c* classes, we have *c* outputs.
- Output *i* can be viewed as $p(\omega_i \mid \mathbf{x})$. That is the posterior probability of the class, given the input. Recognition decision is arg max $p(\omega_i \mid \mathbf{x})$.
- If the network were certain about the class, one output would be 1 and the rest would be zero.
- More generally $\sum_{i=1}^{c} p(\omega_i \mid \mathbf{x}) = 1$, the *c* outputs must sum to 1.
- This can be implemented with a softmax layer $O_i = \frac{e^{z_i}}{\sum_{j=1}^{c} e^{z_j}}$

Feedforward Networks

- The functions defining the layers have been influenced by neuroscience
- Our training dictates the values to be produced output layer and the weights are chosen accordingly
- The weights for intermediate or **“hidden”** layers are learned and not specified directly
- You can think of the network as mapping the raw input space $\mathbf{x}$ to some transformed feature space $\phi(\mathbf{x})$ where the samples are ideally linearly separable

Universal Approximation Theorem

- **Universal Approximation Theorem**: A feedforward neural network with a linear output layer and one or more hidden layers with ReLU **[Leshno et al. ’93]**, or sigmoid or some other “squashing” activation function **[Hornik et al. ’89, Cybenko ’89]** can approximate any continuous function on a closed and bounded subset of $\mathbb{R}^n$ This holds for functions mapping finite dimensional discrete spaces as well.

- If we have enough hidden units we can approximate “any” function! … but we may not be able to train it.

Universal Approximation Theorem: Caveats

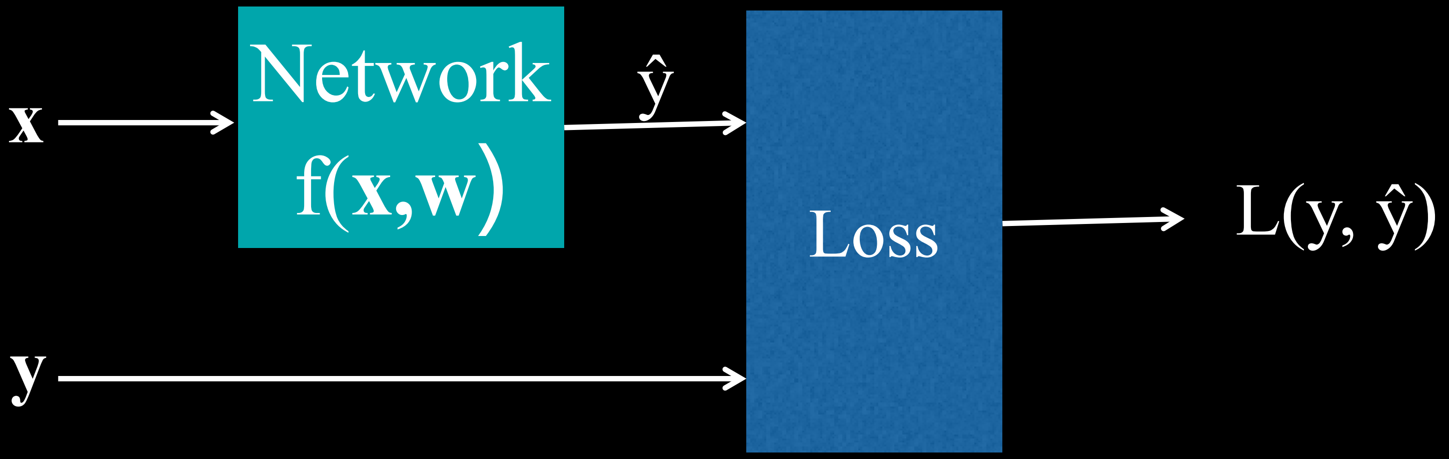
- Optimization may fail to find the parameters needed to represent the desired function.
- Training might choose the wrong function due to overfitting.
- The network required to approximate this function might be so large as to be infeasible.

Universal Approximation Theorem: Caveats

- So even though “any” function can be approximated with a network as described with single hidden layer, the network may fail to train, fail to generalize, or require so many hidden units as to be infeasible.
- This is both encouraging and discouraging!
- However, **[Montufar et al. 2014]** showed that **deeper networks are more efficient** in that a deep rectified net can represent functions that would require an exponential number of hidden units in a shallow one hidden layer network.
- Deep networks composed on many rectified hidden layers are good at approximating functions that can be composed from simpler functions. And lots of tasks such as image classification may fit nicely into this space.

High level view of evaluation and training

- Training data: $\{ < \mathbf{x}^{(i)}, \mathbf{y}^{(i)} > : 1 \leq i \leq n \}$



- Total Loss: $TL(\mathbf{w}) = \sum_{i=1}^{n} L(f(\mathbf{x}^{(i)}; \mathbf{w}), \mathbf{y}^{(i)})$
- Training: Find $\mathbf{w}$ that minimizes the total loss.

The loss function

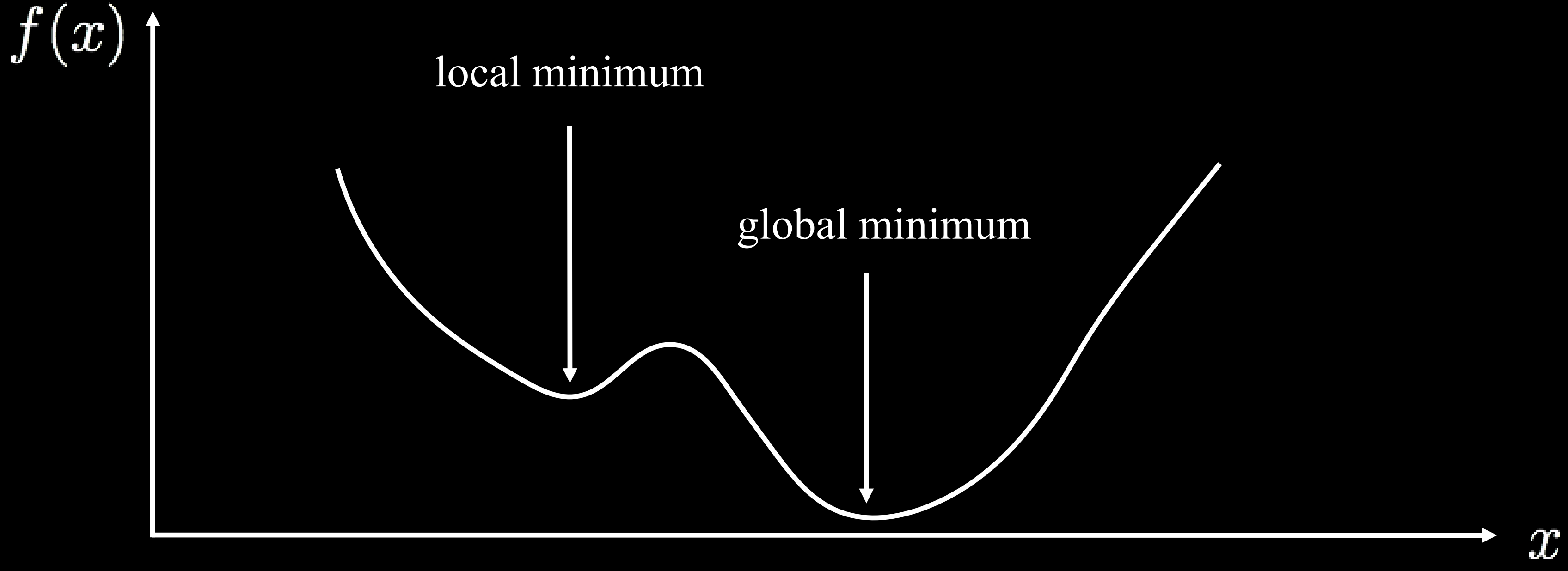
- The loss function is really important. It's how we compare the network output to the training labels.
- Common loss functions:
  - Regression problems:
    - Distance : $L(y, \hat{y}) = ||y - \hat{y}||^p$, usually p = 1 or 2
  - Classification: Softmax + cross entropy
    - Softmax: $\hat{y}_i(\mathbf{z}) = \frac{e^{z_i}}{\sum_{j=1}^{c} e^{z_j}}$
    - Cross entropy between **y** and ŷ is $H(y, \hat{y}) = \sum_{i=1}^{n} y_i log \frac{1}{\hat{y}_i} = - \sum_{i=1}^{n} y_i log \hat{y}_i$
    - where: **y** is a vector with one 1 and the rest 0's.

      ŷ is a vector with positive floats that sum to 1

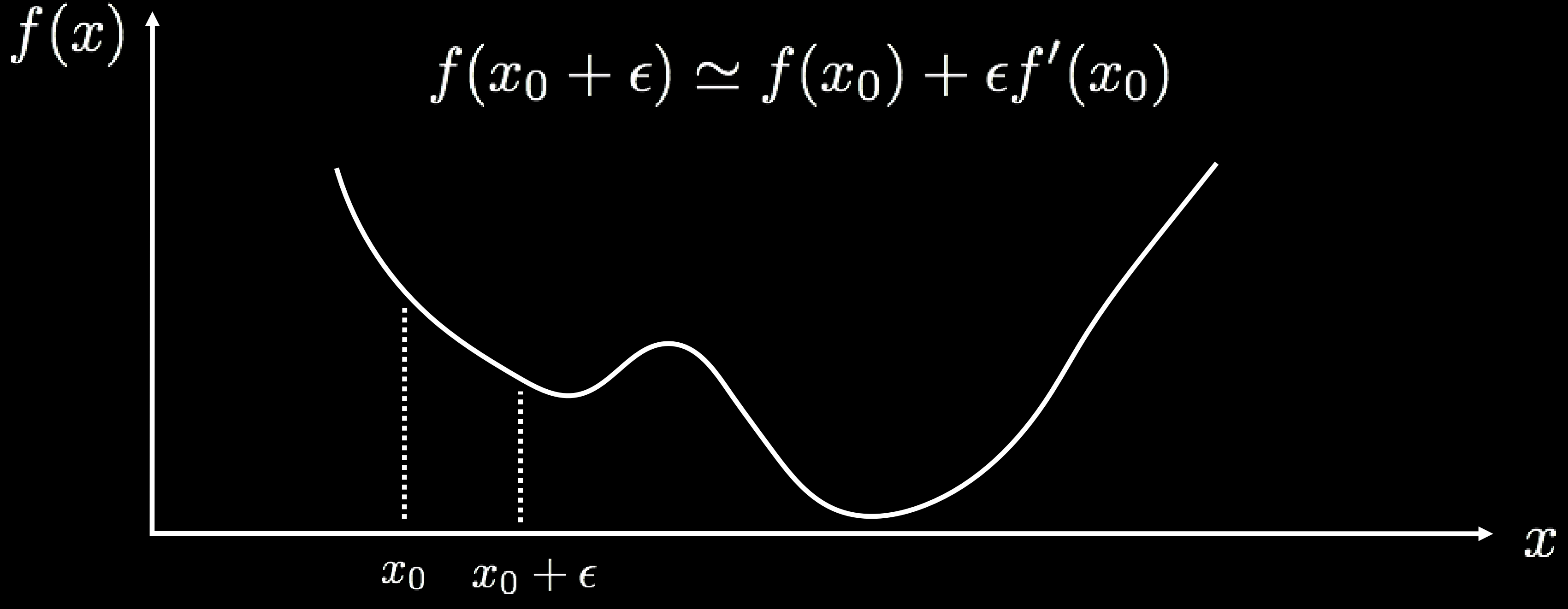
Training Feed Forward Networks

- Given a training set $\{<x^{(1)}, y^{(1)}>, <x^{(2)}, y^{(2)}>, \ldots, <x^{(n)}, y^{(n)}>\}$, estimate (learn) **w** by making $TL(\mathbf{w}) = \sum_{i=1}^{n} L(f(\mathbf{x}^{(i)}; \mathbf{w}), \mathbf{y}^{(i)})$ small.
- Back propagation using Stochastic Gradient Descent
- Adagrad, RMSprop, ADAM
- Regularization: Dropout, Batch/Group/Instance Normalization
- Early Stopping

Gradient-Based Optimization



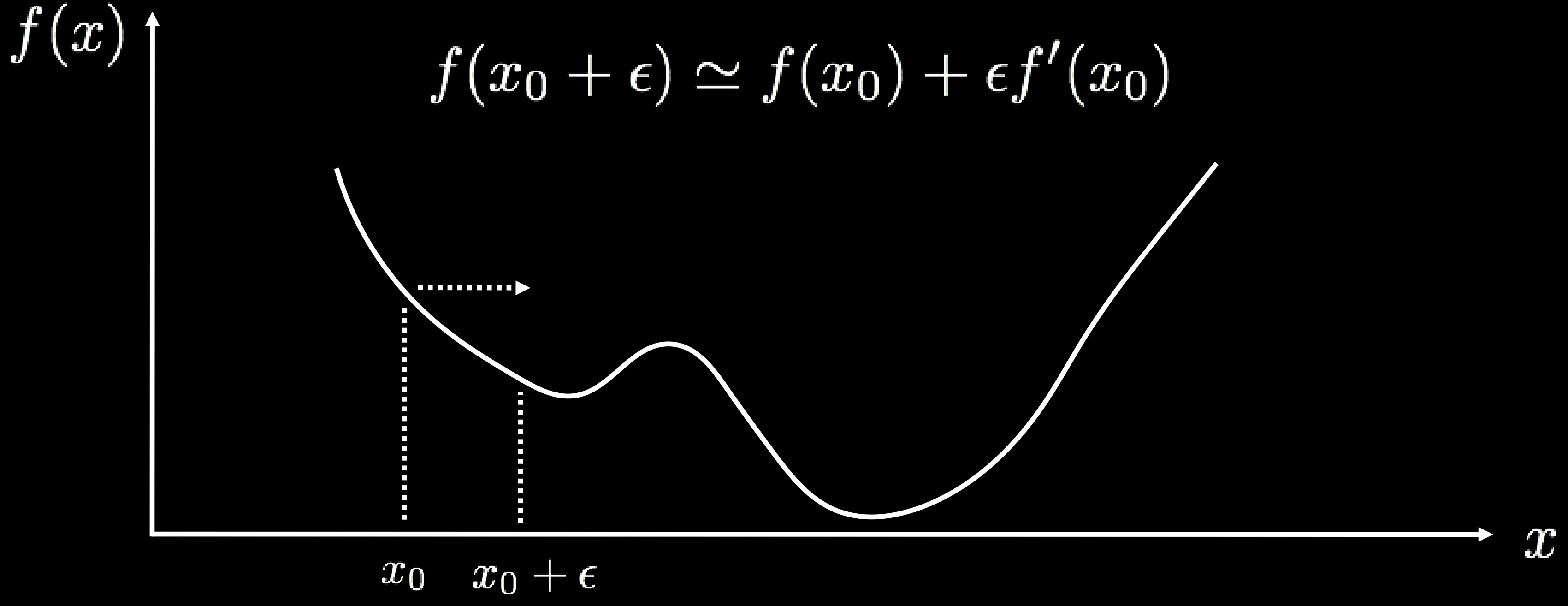
Gradient-Based Optimization



$$f(x_0 + \epsilon) \simeq f(x_0) + \epsilon f'(x_0)$$

$$f' = \frac{df}{dx}$$

Gradient Descent



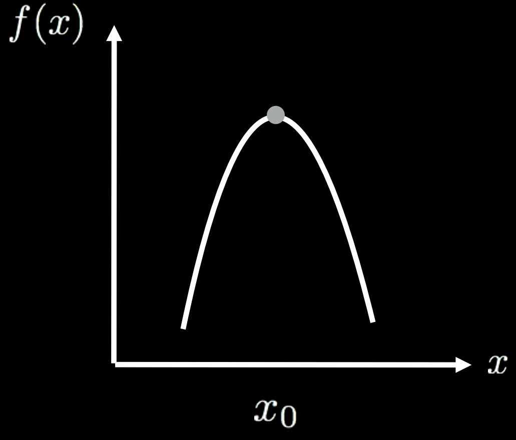
$$f(x_0 + \epsilon) \simeq f(x_0) + \epsilon f'(x_0)$$

Note that f' is negative, so going in positive direction decreases the function.

Critical Points

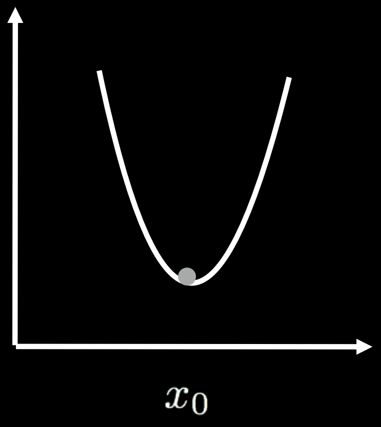
$$f'(x_0) = 0$$

Maximum



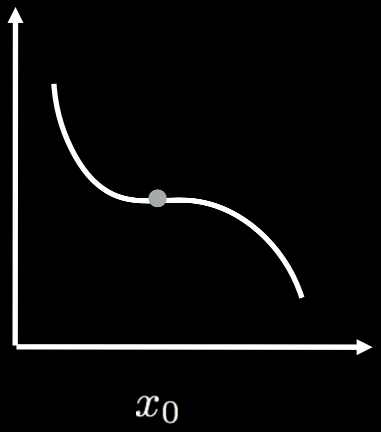
$$f''(x_0) < 0$$

Minimum



$$f''(x_0) > 0$$

Saddle Point



$$f''(x_0) = 0$$

When **x** is a vector?

- Use gradient

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \cdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$$

- and make step in opposite direction

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \eta \nabla_{\mathbf{x}} f(\mathbf{x_t})$$

Optimization for Deep Nets

- Deep learning optimization is usually expressed as a loss summed over all the training samples.
- Our goal is not so much find the weights that **globally** minimize the loss but rather to find parameters that produce a network with the desired behavior.
- Note that there are LOTS of solutions to which our optimization could converge to—with very different values for the weights—but each producing a model with very similar behavior on our sample data.

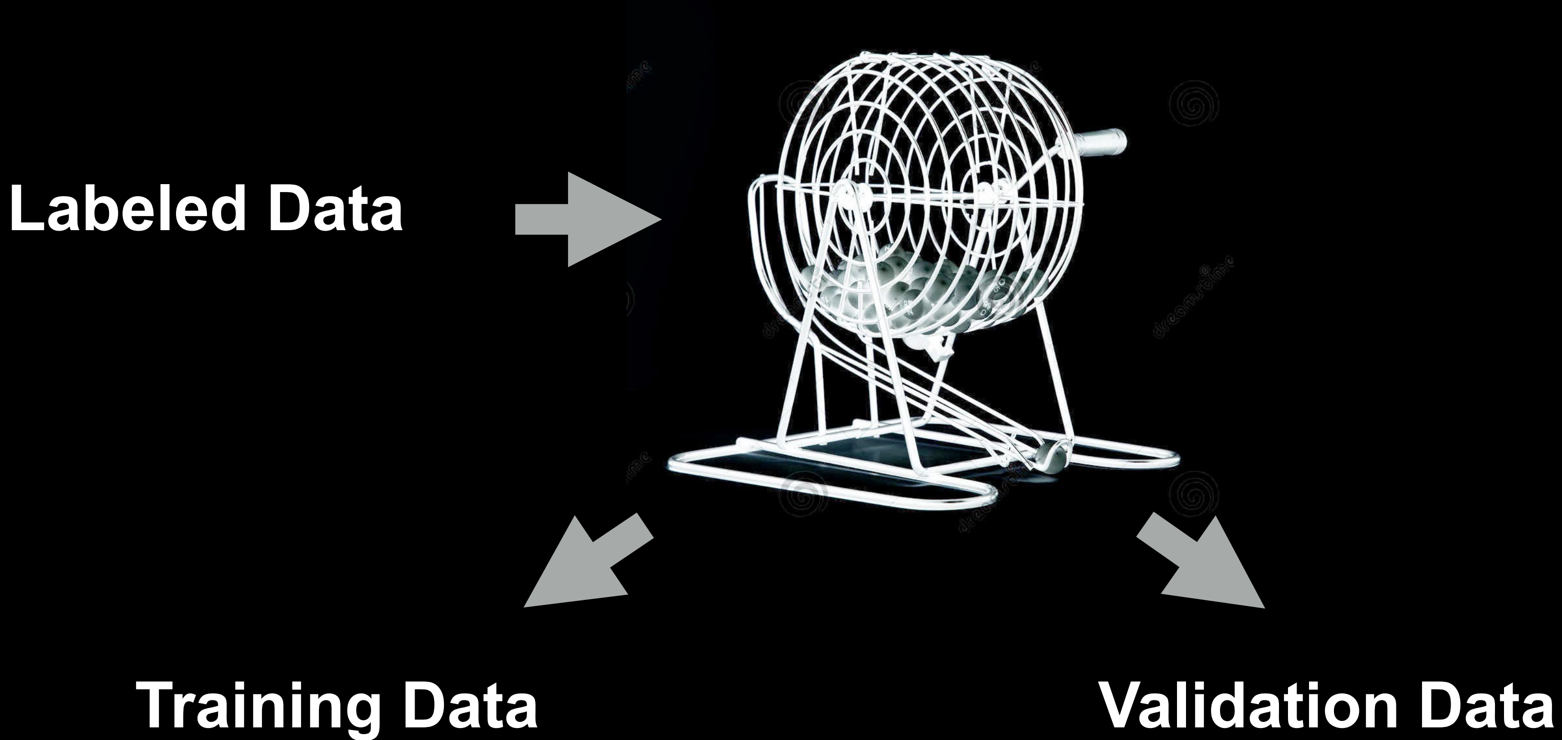
Optimization for Deep Nets

- Although there is a large literature on global optimization, gradient descent-based methods are used in practice.
- Our optimizations for deep learning are typically done in very high dimensional spaces, where the number of weights can run into the millions.
- And for these optimizations, when starting the training from scratch (i.e., some random initialization of the weights), we will need LOTS of labeled training data.

Back propagation

- Basically another name for gradient descent
- Because of nature of network $a_3(a_2(a_1(\mathbf{x};w_1);w_2);w_3)$, gradients with respect to w_i are determined by chain rule
- Can be thought of as “propagating” from loss function to input.
- Adaptive step size methods (e.g., ADAM).

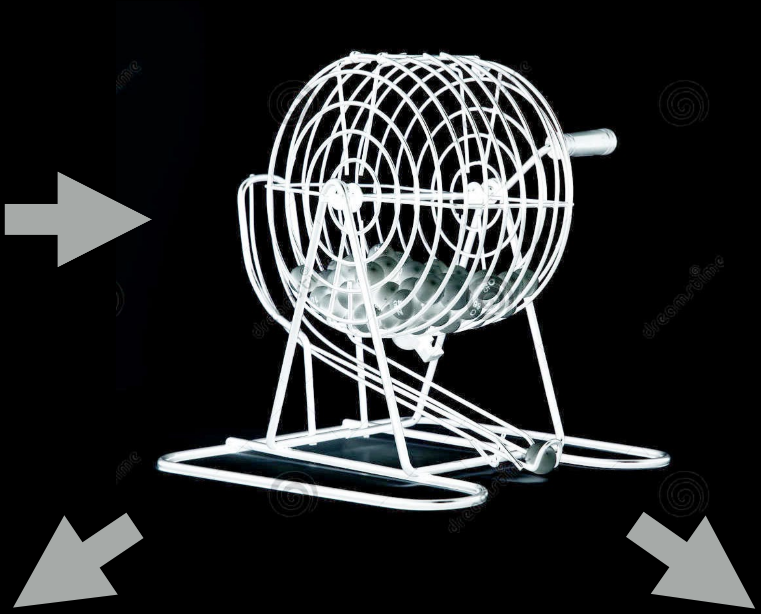
Training and Validation Sets



- Given a bunch of labelled data, divide into Training Set and Validation Set.
- Often 90%-10% or 80%-20% split.
- Often shuffle data before splitting

Training and Validation Sets

Labeled Data



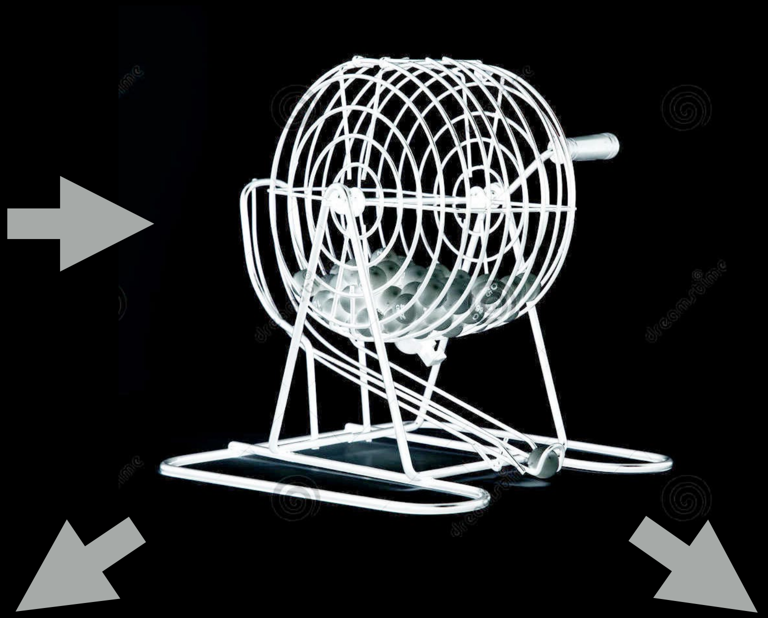
Training Data

Validation Data

NEVER TRAIN ON YOUR VALIDATION SET!

Training and Validation Sets

Labeled Data



Training Data

Validation Data

NEVER TRAIN ON YOUR VALIDATION SET!

Training and Validation Sets

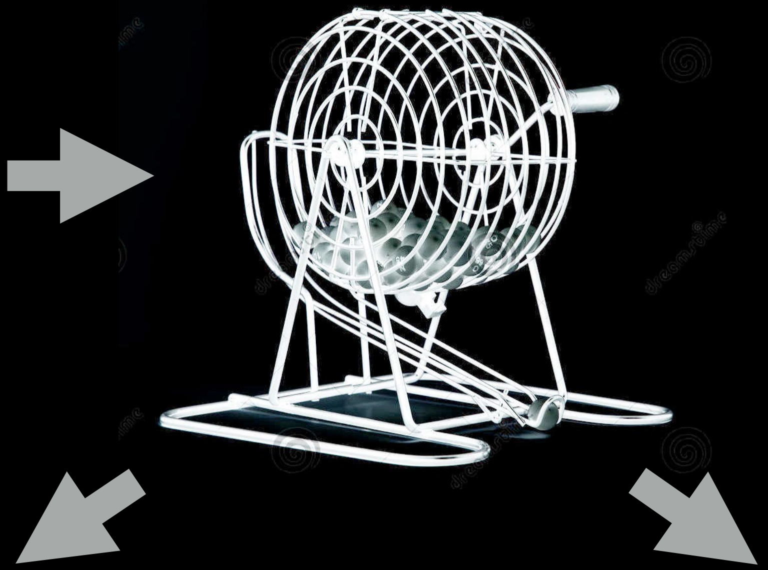
Labeled Data

Training Data

Validation Data

NEVER TRAIN ON YOUR VALIDATION SET!

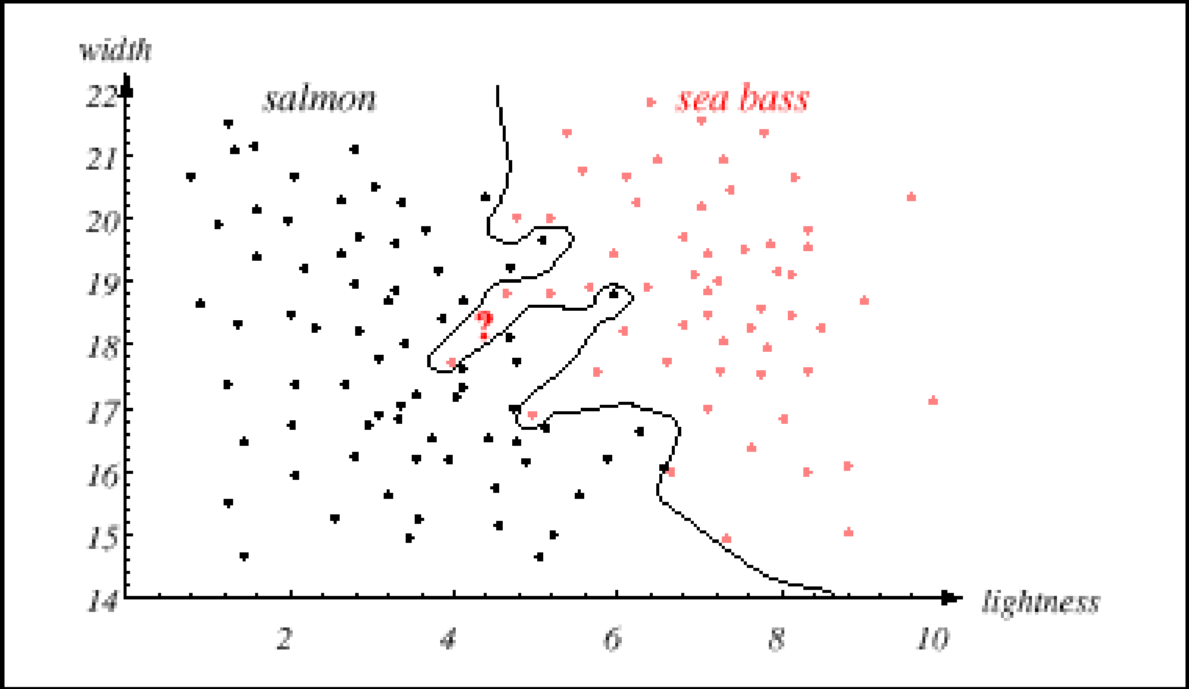
Training and Validation Sets



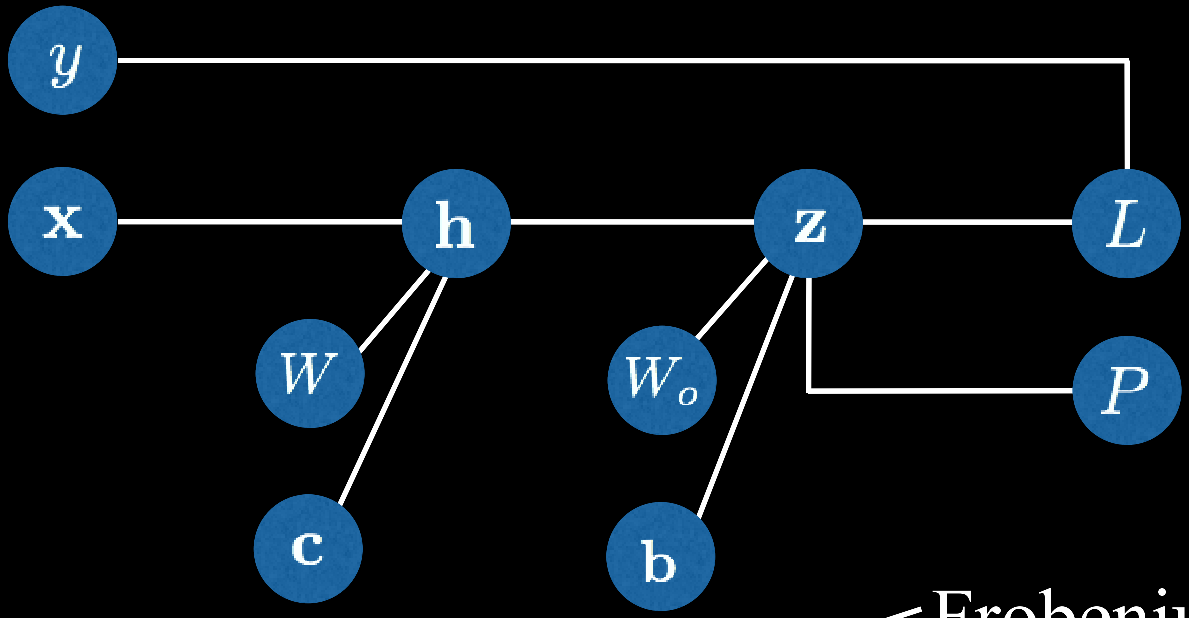
NEVER TRAIN ON YOUR VALIDATION SET!

Regularization

The goal of ***regularization*** is to prevent **overfitting** the training data with the hope that this improves **generalization**, *i.e.*, the ability to correctly handle data that the network has not trained on.



Regularization for MLP



Frobenius norm

$$L = -z_i + \log \sum_j e^{z_j} + \frac{\lambda}{2}(||W||^2 + ||W_o||^2)$$

loss

regularization

Dropout [Srivastava et al., 2014]

- For every training batch through the network, ***dropout*** 0.5 of hidden units and 0.2 of input units. You can choose the probabilities as you like…

- Train as you normally would using SGD, but each time you impose a random ***dropout*** that essentially trains for that batch on a random sub-network.

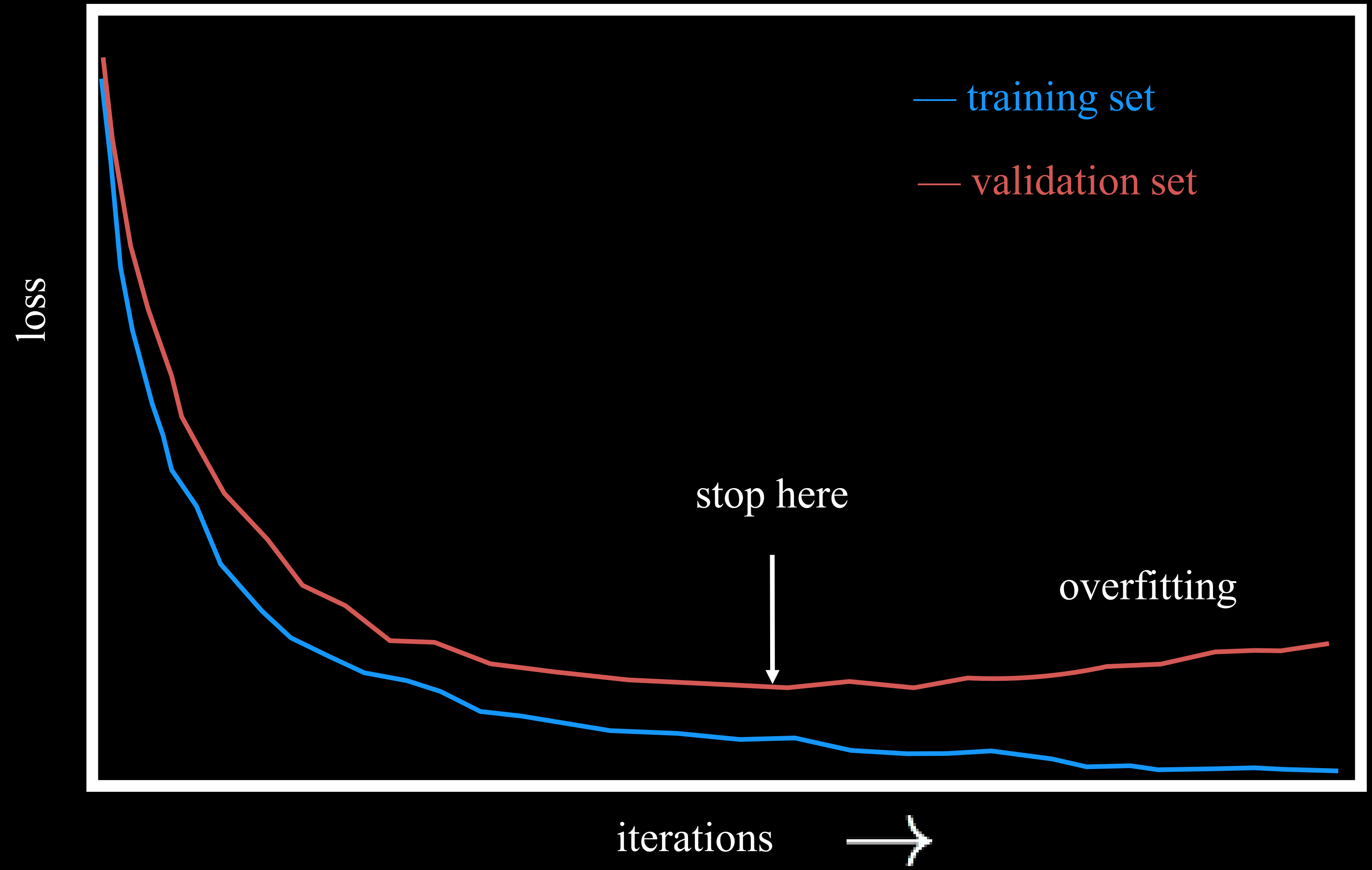
- When you are done training, you use for your model the complete network with all its learned weights, except multiply the weight by the probability of including its parent unit.

- This is called the ***weight scaling inference rule***. [Hinton et al., 2012]

Early Stopping

- Typical deep neural networks have millions and millions of weights!
- With so many parameters at their disposal how to we prevent them from overfitting?
- Clearly we can use some of the other regularization techniques that have been mentioned…
- …but given enough training time, our network will eventually start to overfit the data.

Early Stopping



More Data / Data Augmentation

- Possibly the best way to prevent overfitting is to get more training data!
- When this isn't possible, you can often perform ***data augmentation*** where training samples are randomly perturbed or ***jittered*** to produce more training samples.
- This is usually easy to do when the samples are images and one can crop, rotate, etc. the samples to produce new samples.
- And if the data can be generated synthetically using computer graphics, we can produce an endless supply.

Deterministic vs. Stochastic Methods

- If we performed our gradient descent optimization using <u>all</u> the training samples to compute each step in our parameter updates, then our optimization would be ***deterministic***.
- Confusingly, ***deterministic*** gradient descent algorithms are sometimes referred to as ***batch*** algorithms
- In contrast, when we use a <u>subset</u> of randomly selected training samples to compute each update, we call this ***stochastic gradient descent*** and refer to the subset of samples as a ***mini-batch***.
- And even more confusingly, we often call this mini-batch the “batch” and refer to its size as the “batch size.”

Stochastic Gradient Descent

The SGD algorithm could not be any simpler:

1. Choose a learning rate schedule η_t.
2. Choose stopping criterion.
3. Choose batch size m.
4. Randomly select mini-batch $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(m)}\}$
5. Forward and backpropagation
6. Update $\theta_{t+1} = \theta_t - \eta_t\, g \qquad \mathbf{g} = \frac{1}{m}\sum_i^m \nabla_\theta L(\mathbf{x}^{(i)}, y^i)$
7. Repeat 4, 5, 6 until the stopping criterion is satisfied.

SGD with Momentum

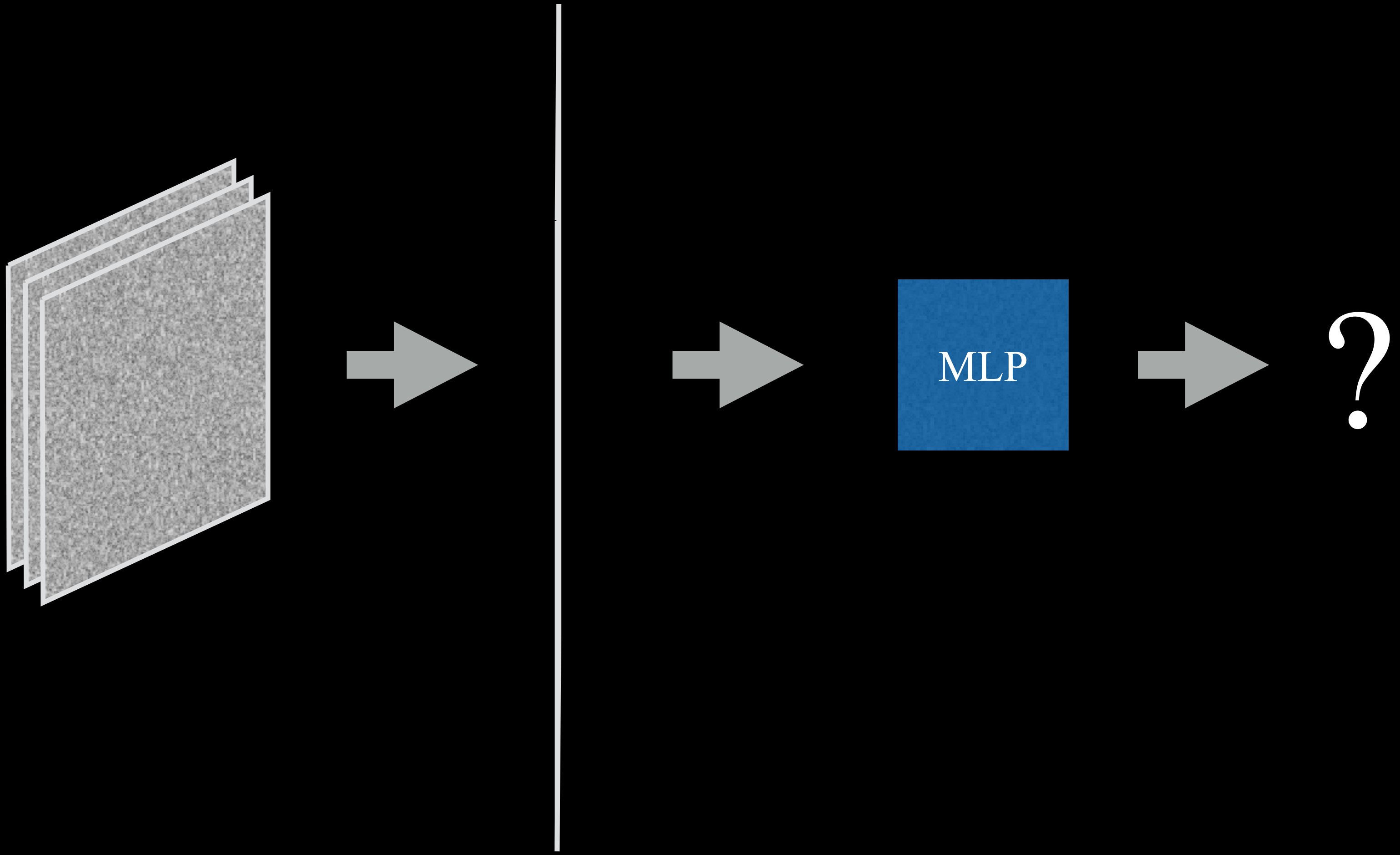
Update rule with momentum:

1. Compute the gradient: $\mathbf{g} = \frac{1}{m} \sum_{i}^{m} \nabla_\theta L(\mathbf{x}^{(i)}, y^{i})$
2. Compute the velocity: $v_t = \alpha_t \, v_{t-1} - \eta_t \, g$
3. Update: $\theta_{t+1} = \theta_t + v_t$

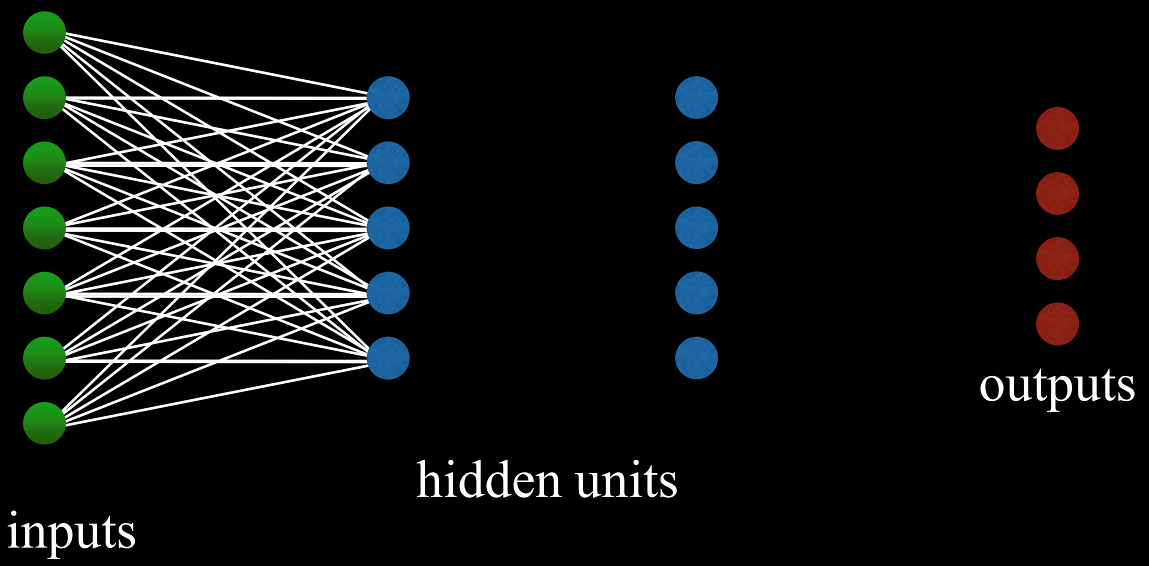
Note: α_t starts small and increases with time (typically)
η_t starts large and decreases with time

Finally, we get to images…

What if we just vectorized images and stuffed these into a MLP?

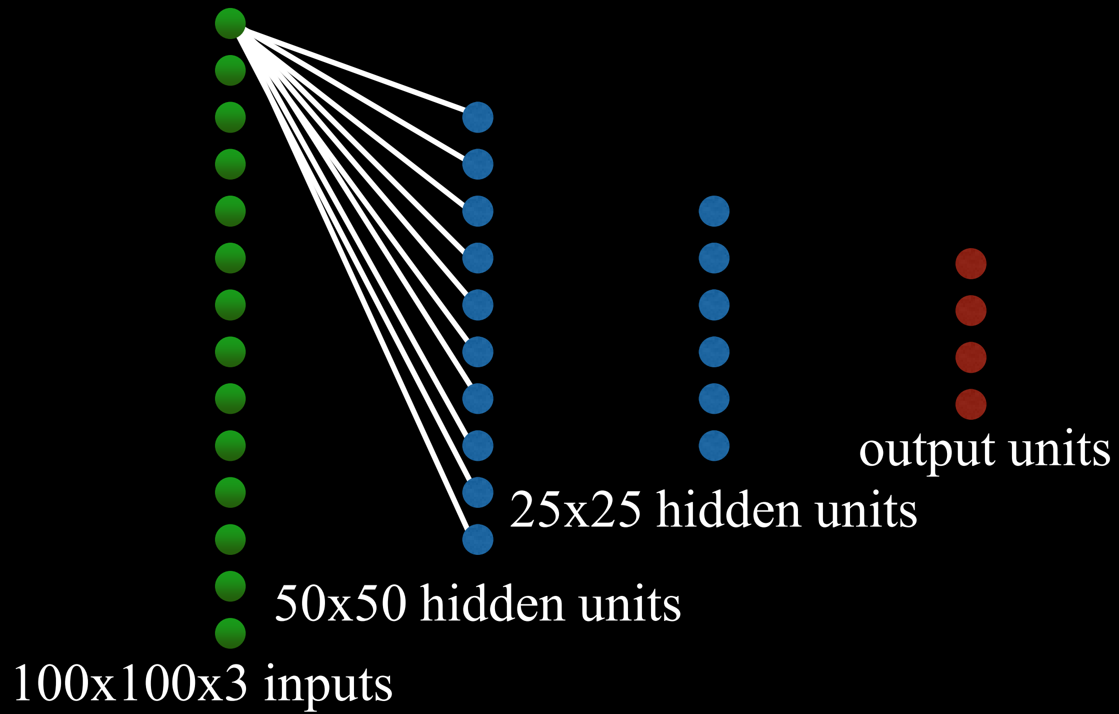


Fully Connected (FC) Layer



Every input unit is connected to every output unit.

Too many weights and connections!



- This fully connected hidden layer might have 75 million weights!
- And this is just for a thumbnail image and a two layer net.

Convolutional Neural Networks

Next Lecture

- Convolutional neural networks