Introduction to Machine Learning CS182

Lu Sun

School of Information Science and Technology

ShanghaiTech University

December 14, 2023

Today:

- Artificial neuron
- Single layer neural networks
- Multi-layer neural networks

Readings:

- Deep Learning (DL), Chapter 6

Artificial Neuron

- Mathematical Model
- Activation Function

Motivation

So far we've mostly tackled simpler forms of data

- e.g. linearly separable data

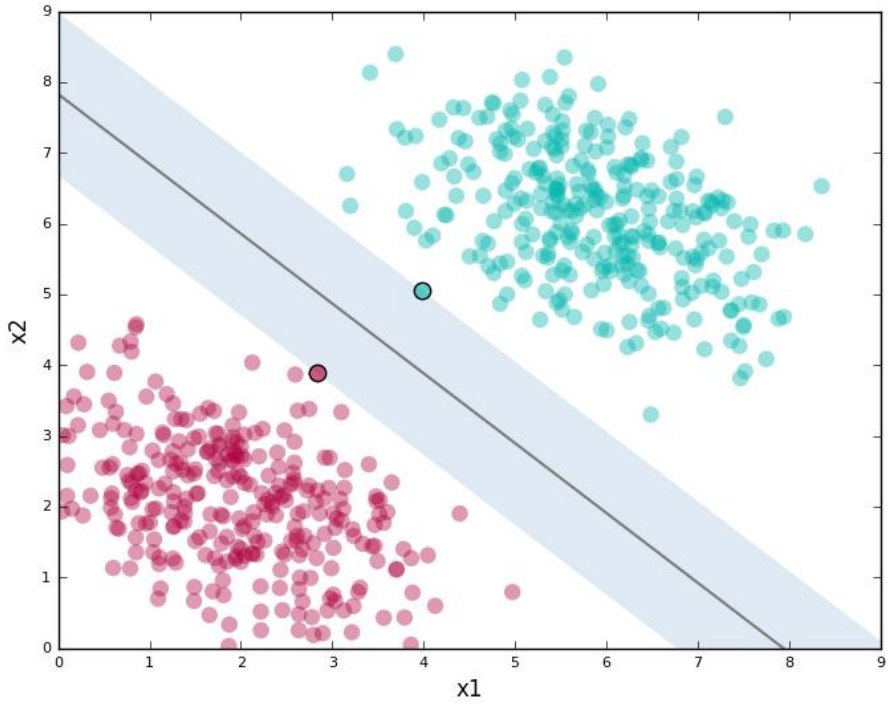


Image Courtesy

What about more complex data?

- Neural networks can learn complex, non-linear functions
- **Non-parametric** model (versus linear regression, which assumes a set of parameters that fully specifies a Gaussian distribution, e.g. mean variance)
- Does not impose fixed relationships in data (does not assume inherent distribution data is sampled from)
  - e.g. better at modeling non-constant variance

What I see

What a computer sees

What do neural networks do?

Today, we'll be tackling **feed forward** neural networks - the fundamental basis of all neural networks

Other names: deep feed forward neural networks, multilayer perceptrons (MLP)

Basis of many modern commercial applications

- Stepping stone in understanding CNNs, RNNs

Goal: simply put, neural networks are used to approximate some function *f*

Example: classification, regression

What do neural networks do? (Ct'd)

Example: classification

$y = f^*(x)$ is a function that maps input x to a discrete output variable (category) y

Feedforward NNs will then define a mapping: $y = f(x;\theta)$

→ the neural network then **learns** the value of the set of parameters $\boldsymbol{\theta}$ that result in the best function approximation

What does this function approximation look like? What does $\boldsymbol{\theta}$ look like?

Easier to understand given visual graph of neural network, biological inspiration.

Biological Inspiration

- Our brain has $\sim 10^{11}$ neurons, each of which communicates (is connected) to $\sim 10^{4}$ other neurons

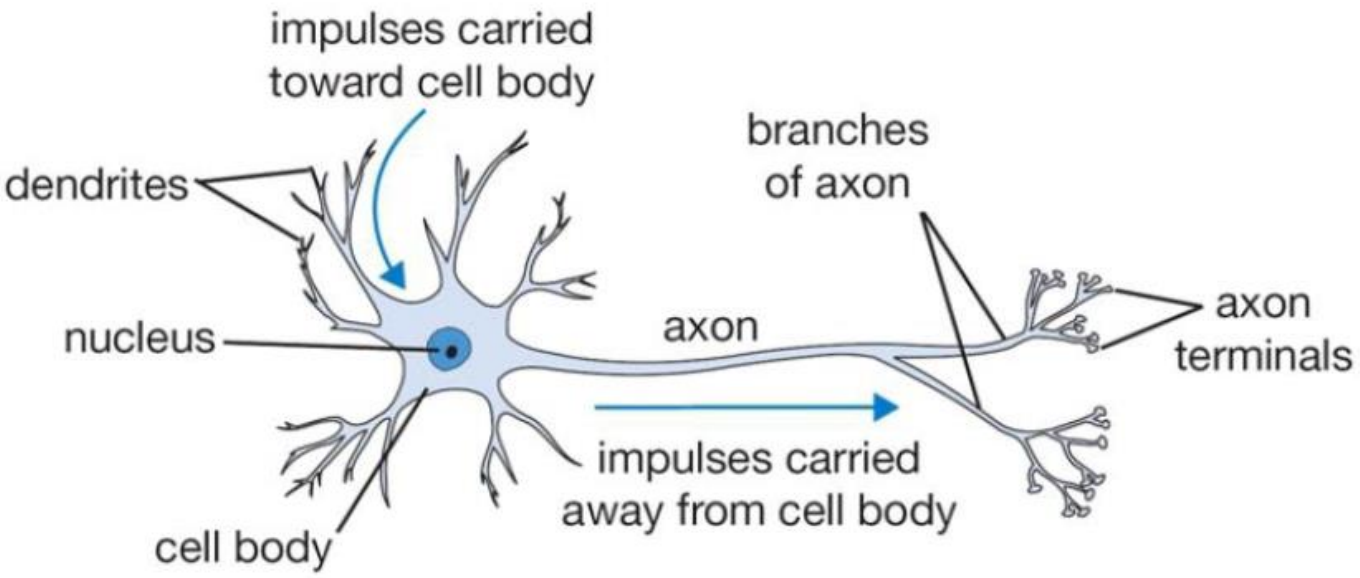
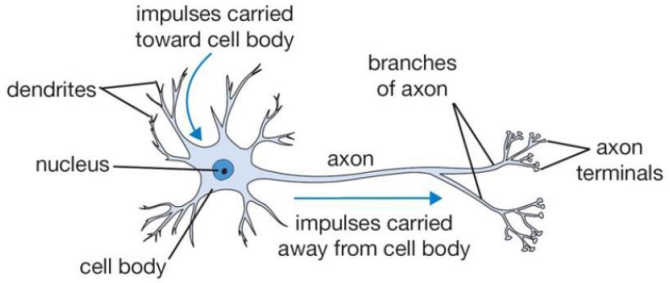
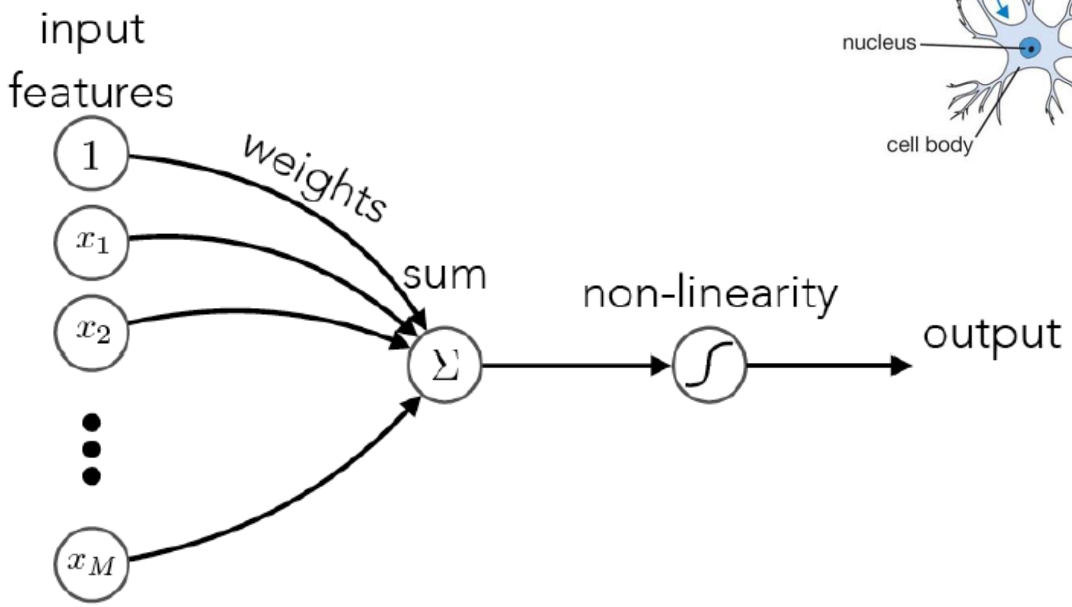


Figure : The basic computational unit of the brain: Neuron

Mathematical Model of A Neuron





artificial neuron: *weighted sum and non-linearity*

bias — input features

$$s = b + w_1 x_1 + w_2 x_2 + \cdots + w_M x_M = \mathbf{w}^\intercal \mathbf{x}$$

sum — weights

$$h = g(s)$$

output — non-linearity — sum

Biological vs Artificial

Consider humans:

- Neuron switching time ~ .001 second
- Number of neurons ~ 10^{10}
- Connections per neuron ~ 10^{4-5}
- Scene recognition time ~ .1 second
- 100 inference steps doesn't seem like enough

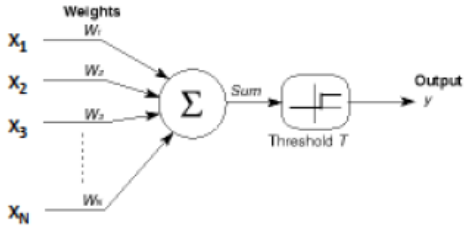
$\rightarrow$ much parallel computation

Properties of artificial neural nets (ANN's):

- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process

What A Single Neuron Does?

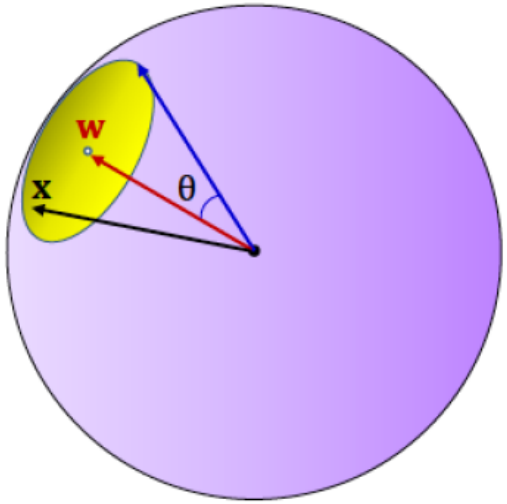
- A neuron (perceptron) fires if its input is within a specific angle of its weight
  - If the input pattern matches the weight pattern closely enough



$$\mathbf{x}^T\mathbf{w} > T$$

$$\Rightarrow \cos\theta > \frac{T}{|\mathbf{x}||\mathbf{w}|}$$

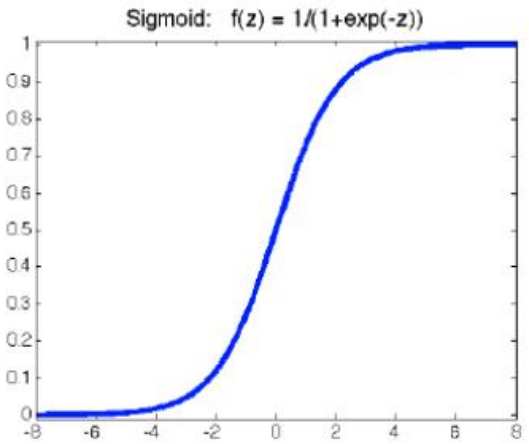
$$\Rightarrow \theta < \cos^{-1}\left(\frac{T}{|\mathbf{x}||\mathbf{w}|}\right)$$

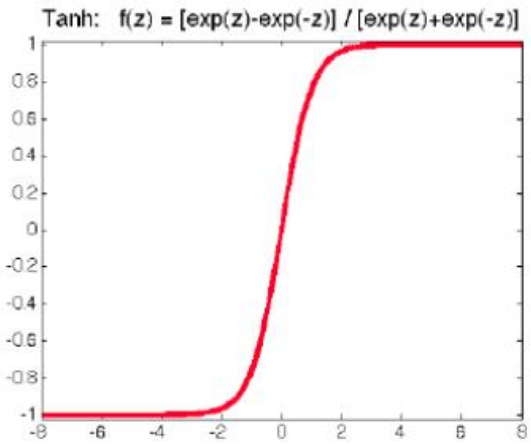


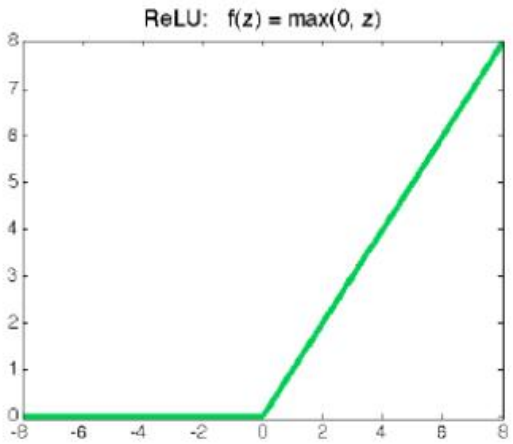
Activation Functions

Most commonly used activation functions:

- Sigmoid: $\sigma(z) = \frac{1}{1+\exp(-z)}$
- Tanh: $\tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$
- ReLU (Rectified Linear Unit): $\mathrm{ReLU}(z) = \max(0, z)$

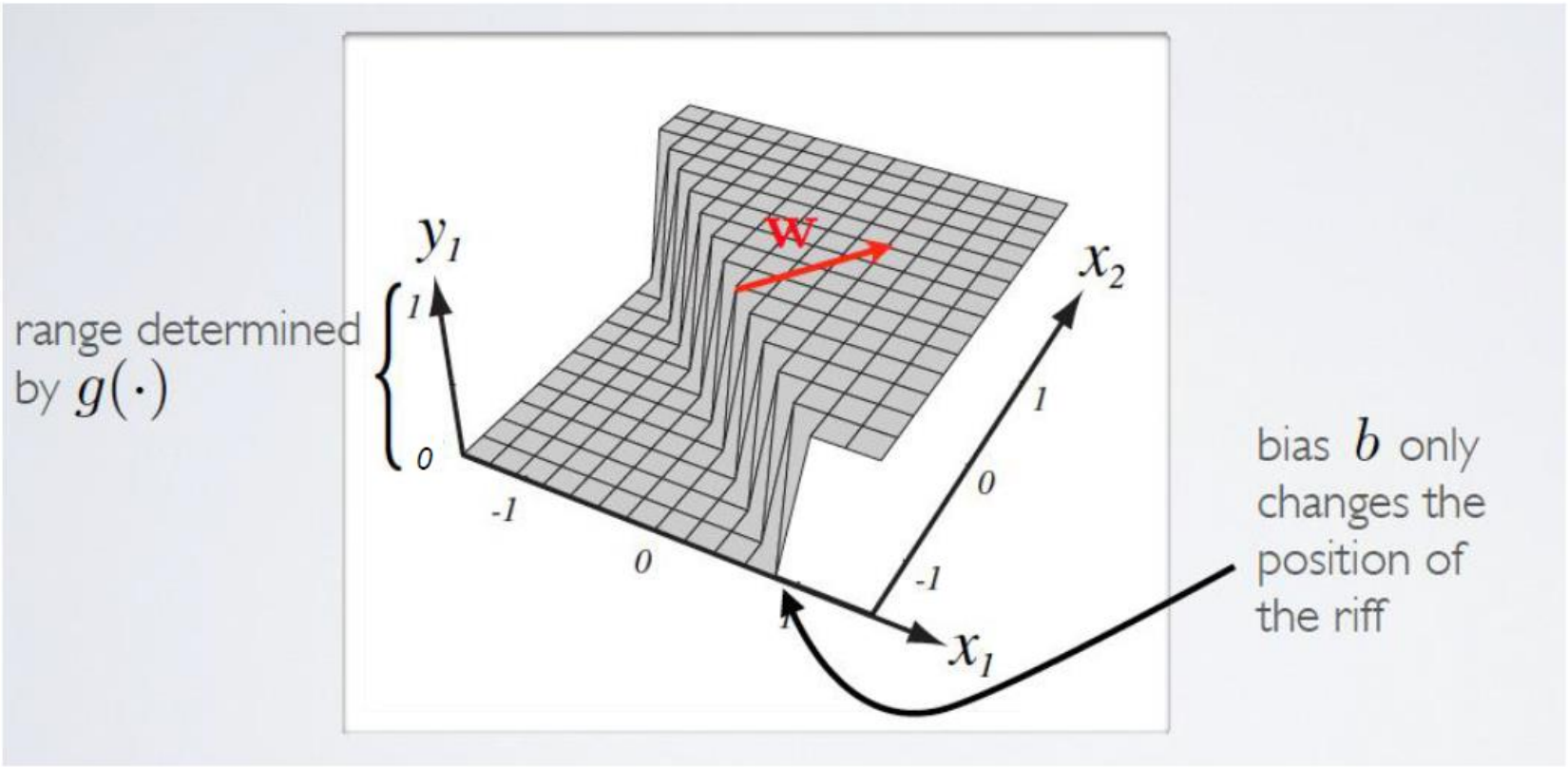






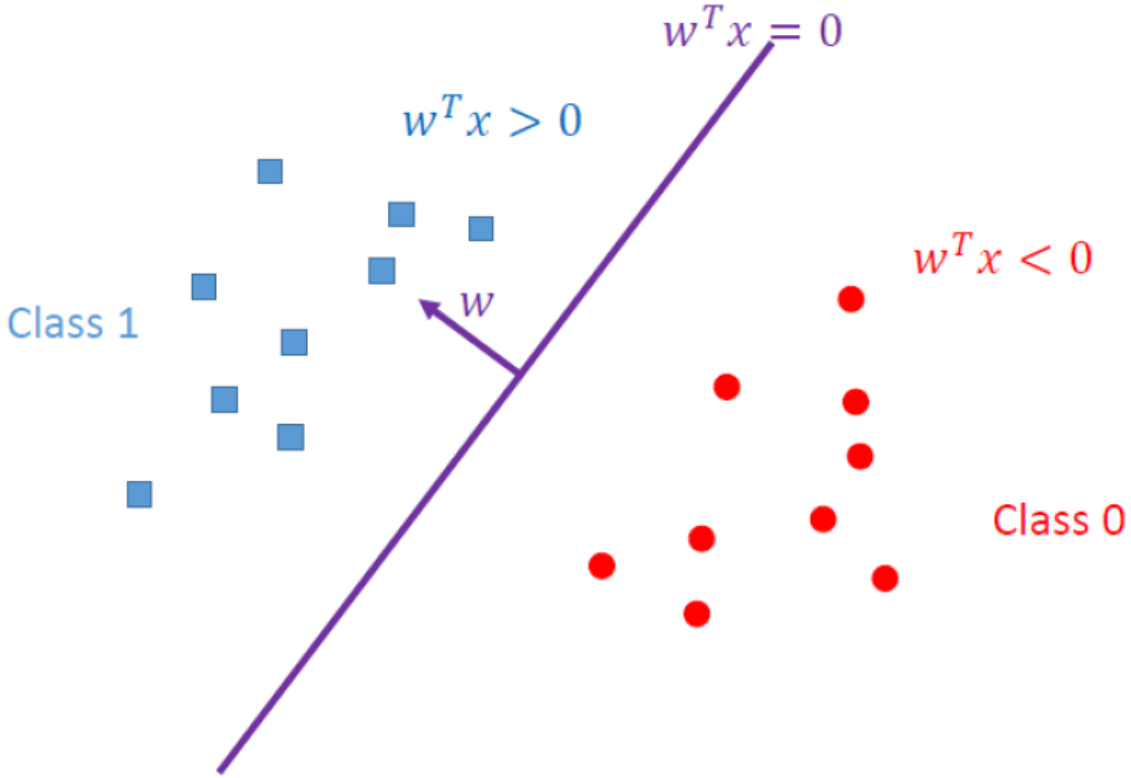
Capacity of Single Neuron

- Sigmoid activation function



Capacity of Single Neuron

- Binary classification

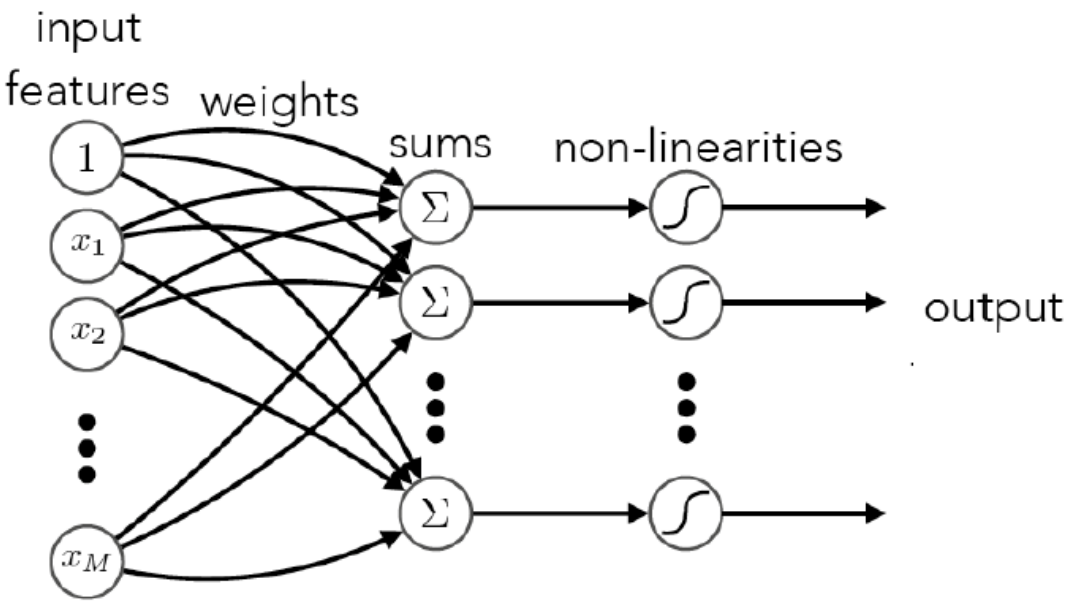


Single neuron as a linear classifier.

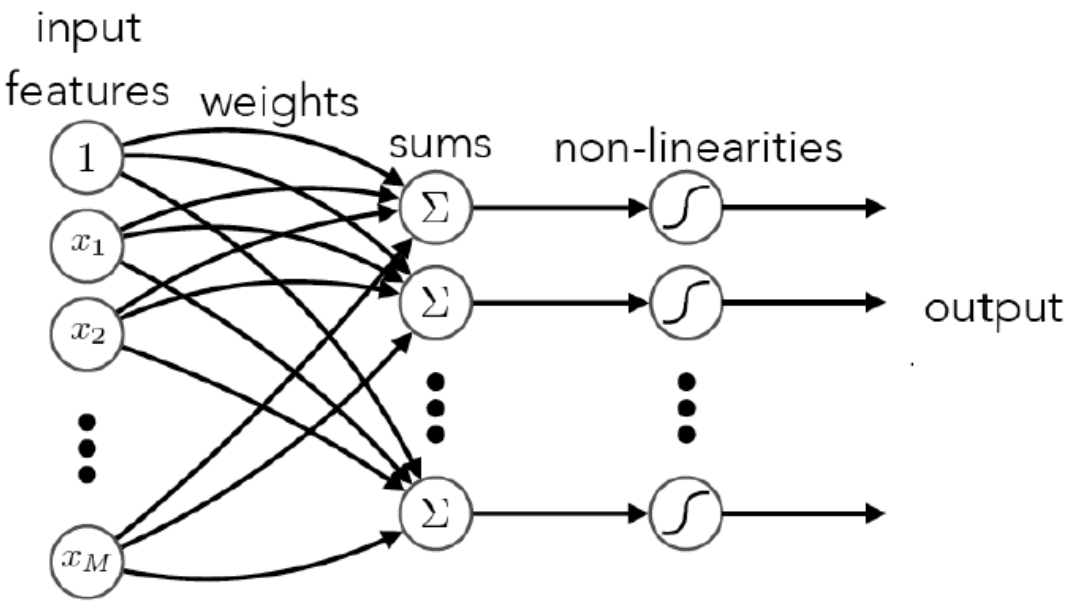
Single-Layer Neural Network

- Network Structure
- Parameter Learning

Single Layer Neural Network



Single Layer Neural Network

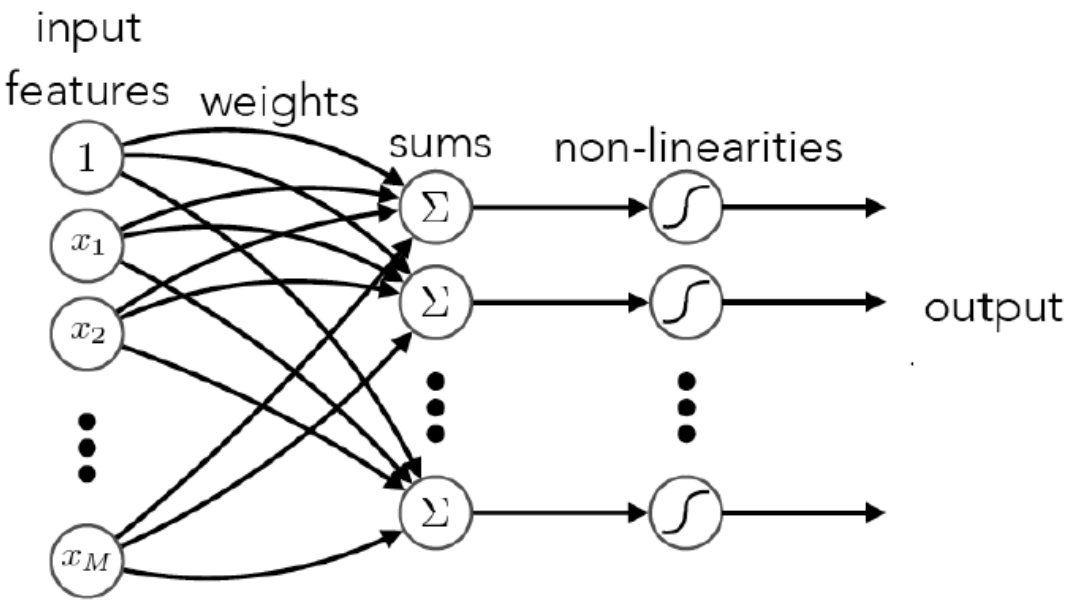


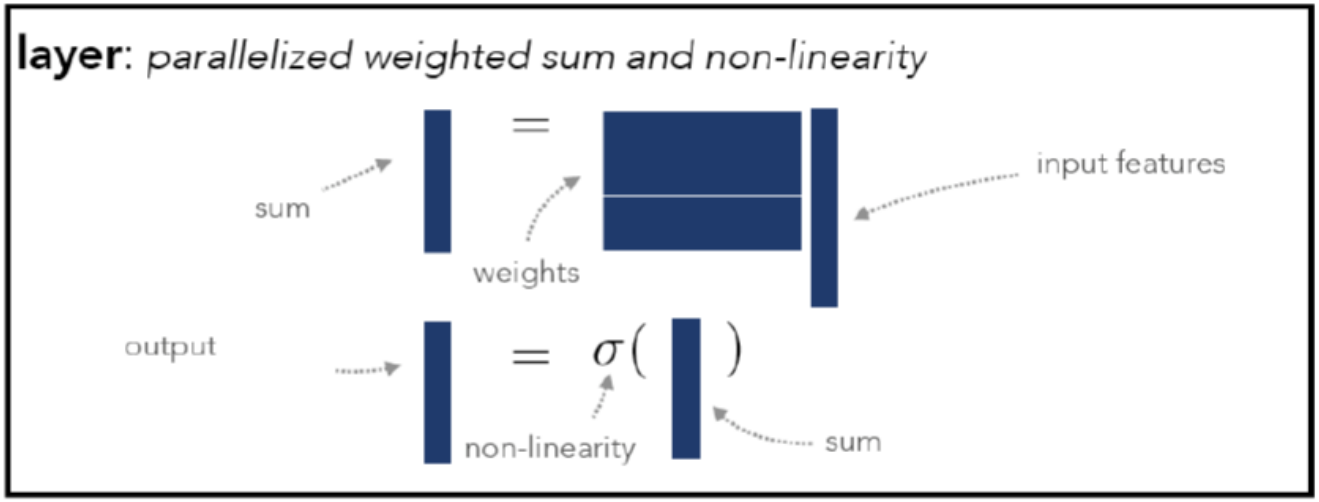
layer: *parallelized weighted sum and non-linearity*

one sum per weight *vector* $s_j = \mathbf{w}_j^\intercal \mathbf{x} \longrightarrow \mathbf{s} = \mathbf{W}^\intercal \mathbf{x}$ vector of sums from weight *matrix*

$$\mathbf{h} = \sigma(\mathbf{s})$$

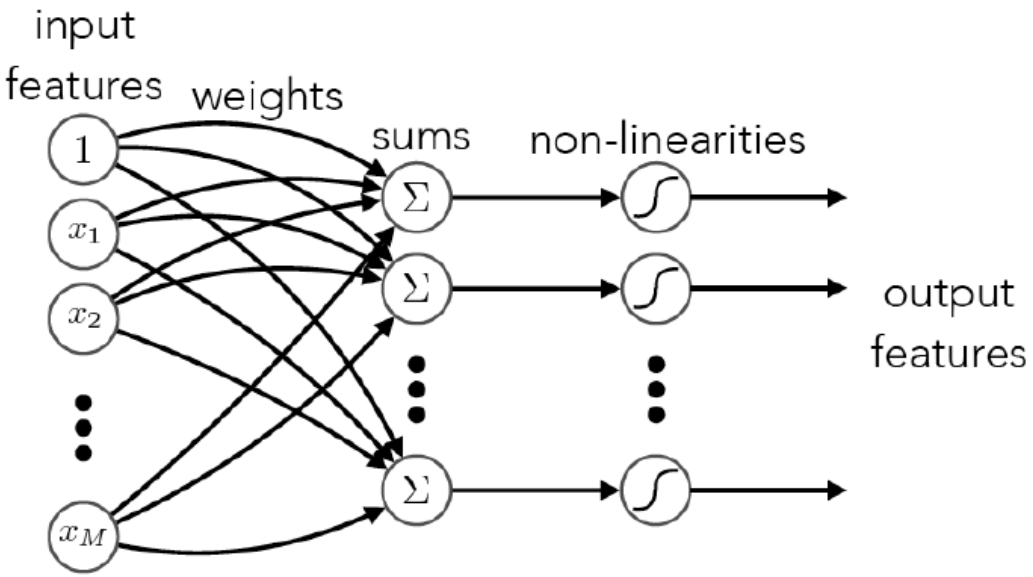
Single Layer Neural Network





What Is the Output?

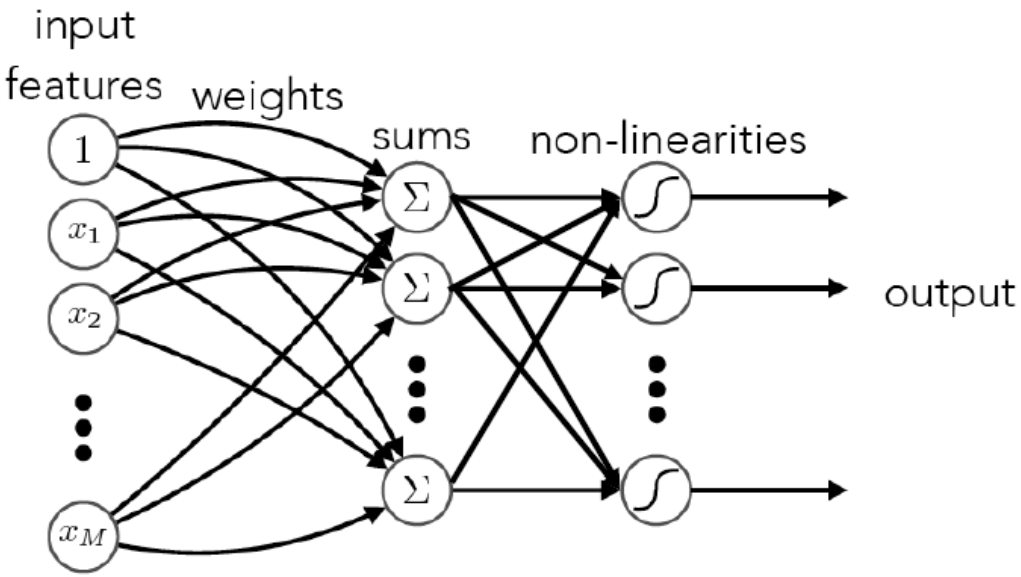
- Element-wise nonlinear functions
  - Independent feature/attribute detectors



$$\mathbf{h} = [h_j] \qquad h_j = \sigma(s_j) = \sigma(\mathbf{w}_j^{\mathsf{T}}\mathbf{x})$$

What Is the Output?

- Nonlinear functions with vector input
  - Competition between neurons

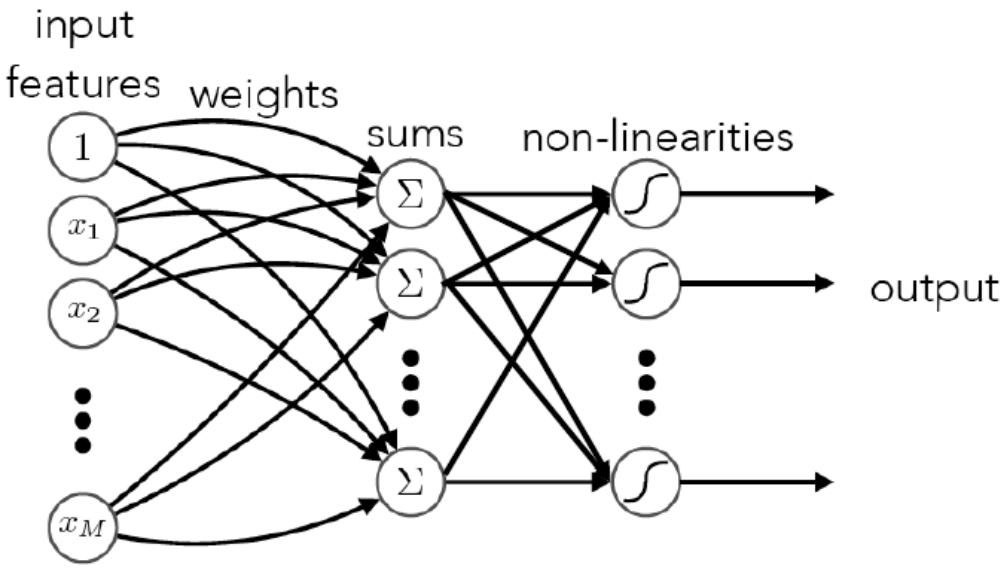


$$\mathbf{h} = [h_j]$$

$$h_j = g(\mathbf{s}) = g(\mathbf{w}_1^\intercal \mathbf{x}, \cdots, \mathbf{w}_m^\intercal \mathbf{x})$$

What Is the Output?

- Nonlinear functions with vector input
  - Example: Winner-Take-All (WTA)

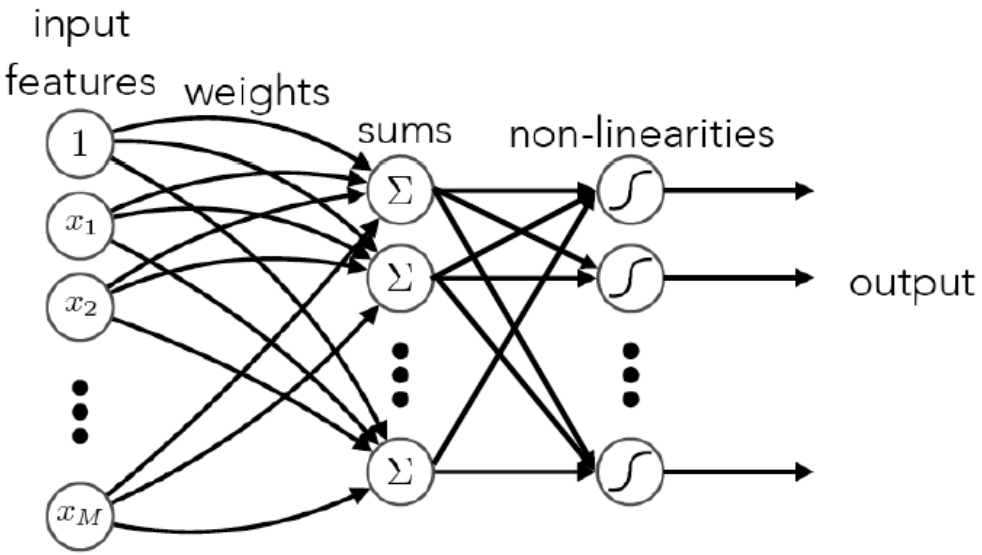


$$\mathbf{h} = [h_j]$$

$$h_j = g(\mathbf{s}) = \begin{cases} 1 & \text{if } j = \arg\max_i \mathbf{w}_i^\mathsf{T}\mathbf{x} \\ 0 & \text{if otherwise} \end{cases}$$

A Probabilistic Perspective

- Change the output nonlinearity



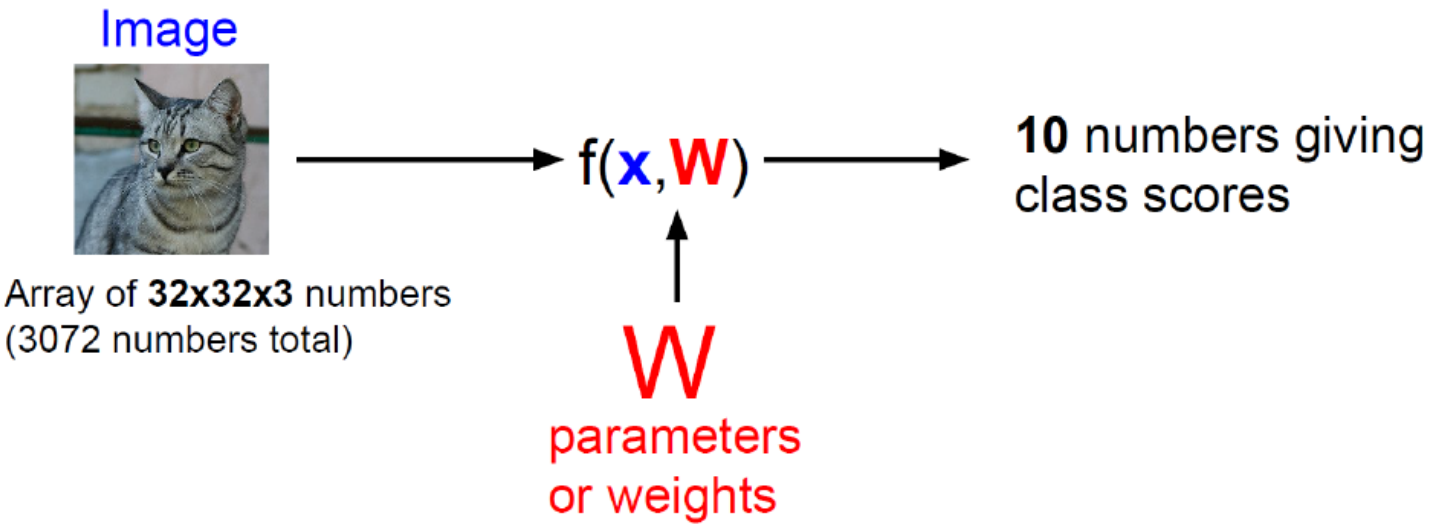
  - From WTA to Softmax function

scores = unnormalized log probabilities of the classes.

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}} \quad \text{where} \quad s = f(x_i; W)$$

Example: Multiclass classification

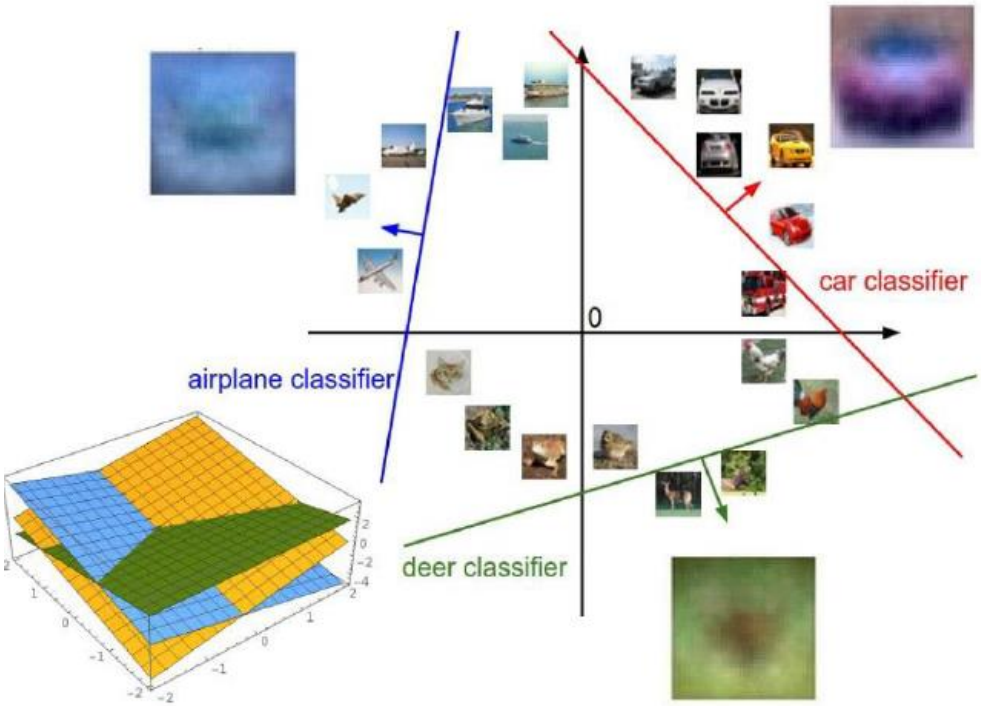
- CIFAR10 as an example



- The output/prediction: WTA

Example: Multiclass classification

- What are those weights?

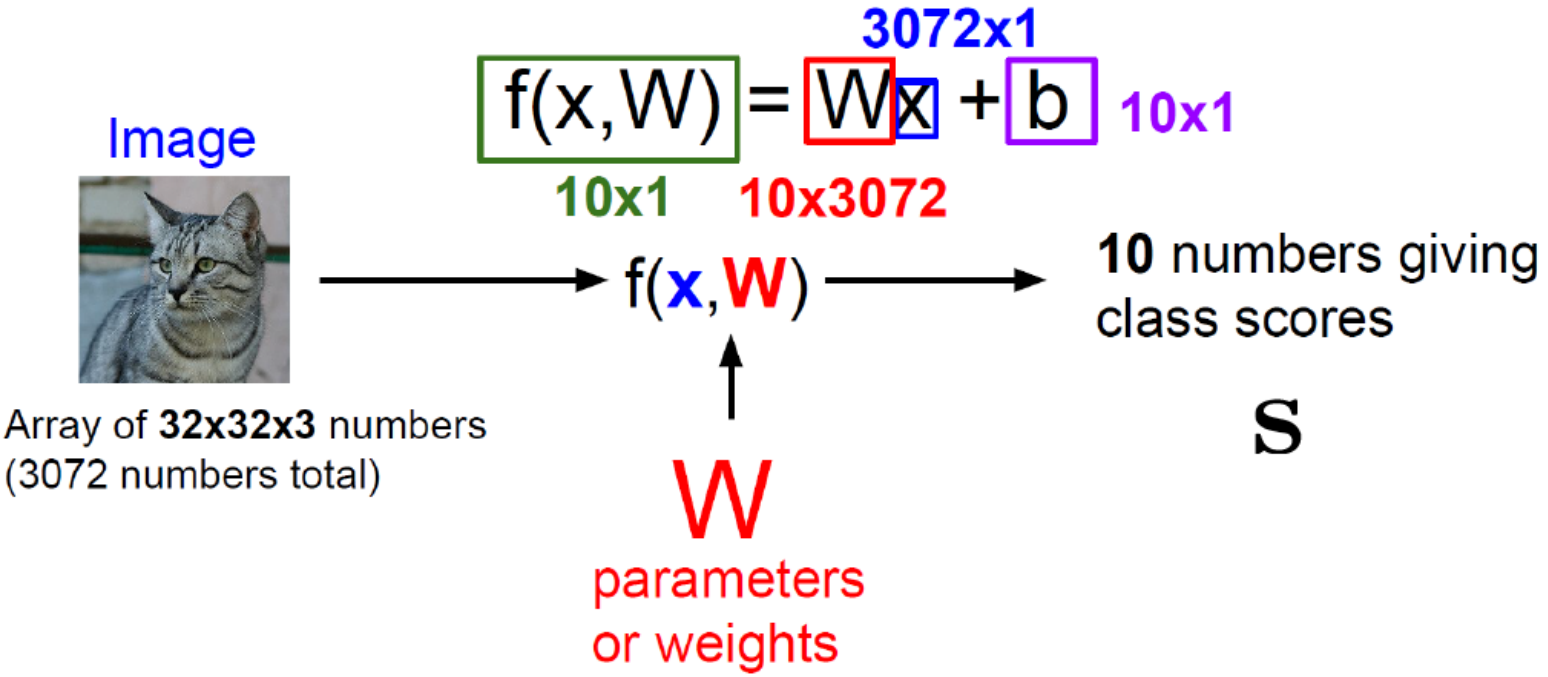


$$f(x,W) = Wx + b$$

Array of **32x32x3** numbers
(3072 numbers total)

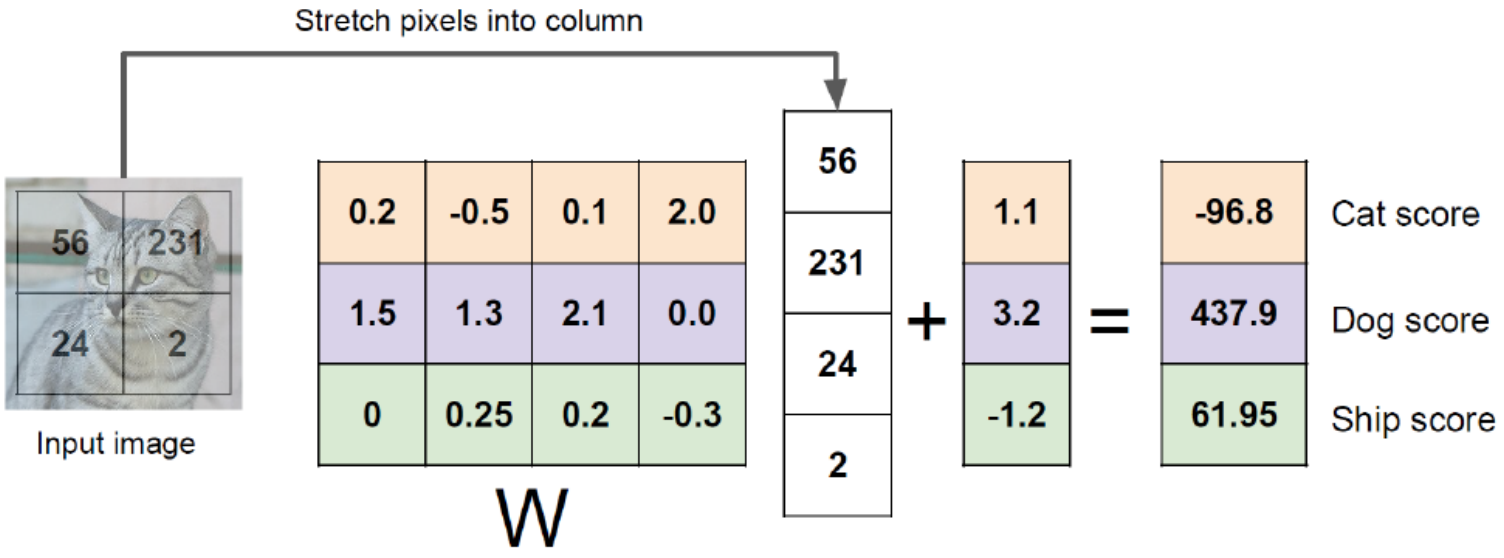
Multiclass Linear Classifiers

- Extending linear classifier in binary case



Multiclass Linear Classifiers

- Example with an image with 4 pixels, and 3 classes (cat/dog/ship)



- The WTA prediction: one-hot encoding of its predicted label

$$y = 1 \Leftrightarrow y = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad y = 2 \Leftrightarrow y = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \qquad y = 3 \Leftrightarrow y = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Probabilistic Outputs

- Softmax layer/function

scores = unnormalized log probabilities of the classes.

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$ where $$s = f(x_i; W)$$

unnormalized probabilities

cat	**3.2**	exp	**24.5**	normalize **0.13**
car	5.1		164.0	0.87
frog	-1.7		0.18	0.00

unnormalized log probabilities — probabilities

Single-Layer Neural Network

- Network Structure
- Parameter Learning

Learning A Single Layer Network

- Design a loss function for multiclass classifiers
  - Hinge loss
    - The SVM and max-margin
  - Probabilistic formulation
    - Log loss and logistic regression
- Generalization issue
  - Avoid overfitting by regularization

Example: Logistic Regression

- Learning loss: negative log likelihood

scores = unnormalized log probabilities of the classes.

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$ where $$s = f(x_i; W)$$

Want to maximize the log likelihood, or (for a loss function) to minimize the negative log likelihood of the correct class:

$$L_i = -\log P(Y = y_i|X = x_i)$$

Example: Logistic Regression

- Learning loss: example

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

unnormalized probabilities

	unnormalized log probabilities	exp	unnormalized probabilities	normalize	probabilities	
cat	**3.2**		**24.5**		**0.13**	→ L_i = -log(0.13) = **0.89**
car	5.1		164.0		0.87	
frog	-1.7		0.18		0.00	

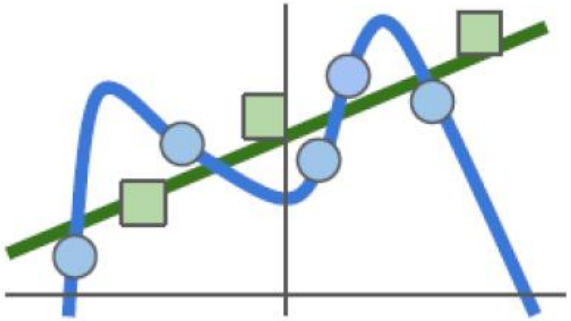
Learning with Regularization

- Constraints on hypothesis space
  - Typically, penalizing large weights

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

Data loss: Model predictions should match training data

Regularization: Model should be "simple", so it works on test data



Learning with Regularization

- Regularization terms

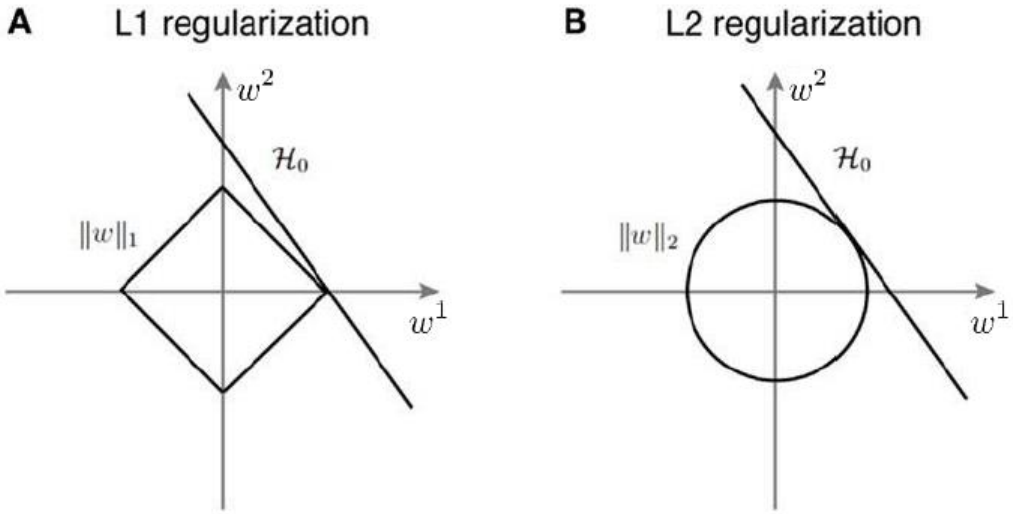
In common use:

L2 regularization $R(W) = \sum_k \sum_l W_{k,l}^2$

L1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2) $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

Max norm regularization (might see later)



Optimization of Loss Functions

- Recall gradient descent
  - choose initial $w^{(0)}$, repeat

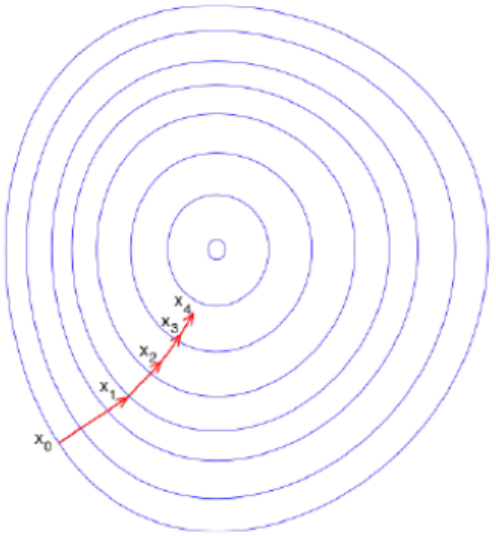
$$w^{(t+1)} = w^{(t)} - \eta_t \cdot \nabla \mathrm{L}\left(w^{(t)}\right)$$

    until stop
  - η_t is the learning rate, and

$$\nabla \mathrm{L}\left(w^{(t)}\right) = \frac{1}{n} \sum_i \nabla_w \mathrm{L_i}(w^{(t)}; y_i, x_i)$$

  - How to stop? $\|w^{(t+1)} - w^{(t)}\| \leq \epsilon$ or $\|\nabla \mathrm{L}(w^{(t)})\| \leq \epsilon$

Two dimensional example:



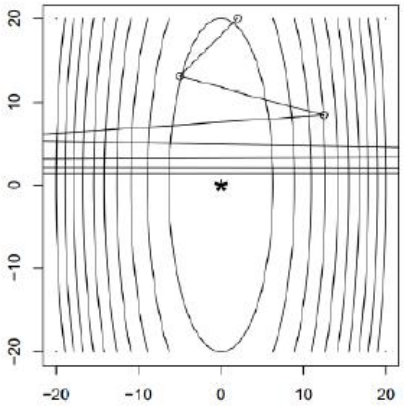
Optimization: Gradient Descent

- Gradient descent

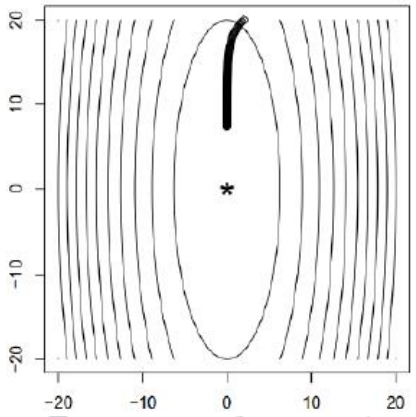
```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

- Learning rate matters

$\eta_t = t$, it is too big



too small η_t, after 100 iterations



Optimization: Gradient Descent

- Stochastic gradient descent

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W) + \lambda R(W)$$

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$

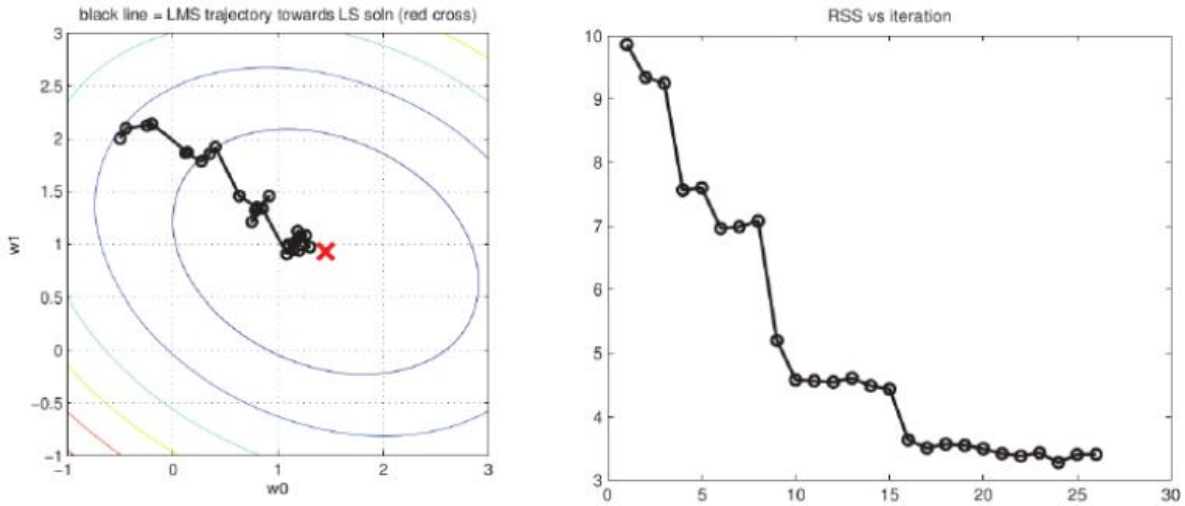
Full sum expensive when N is large!

Approximate sum using a **minibatch** of examples
32 / 64 / 128 common

```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Optimization: Gradient Descent

- Stochastic gradient descent



  - the objective does not always decrease for each step
  - comparing to GD, SGD needs more steps, but each step is cheaper
  - mini-batch, say pick up 100 samples and do average, may accelerate the convergence

Multi-Layer Neural Network

- Limitations of Single-Layer NN
- Generalization Ability
- Inference and Learning
- Practice

Capacity of Single Neuron

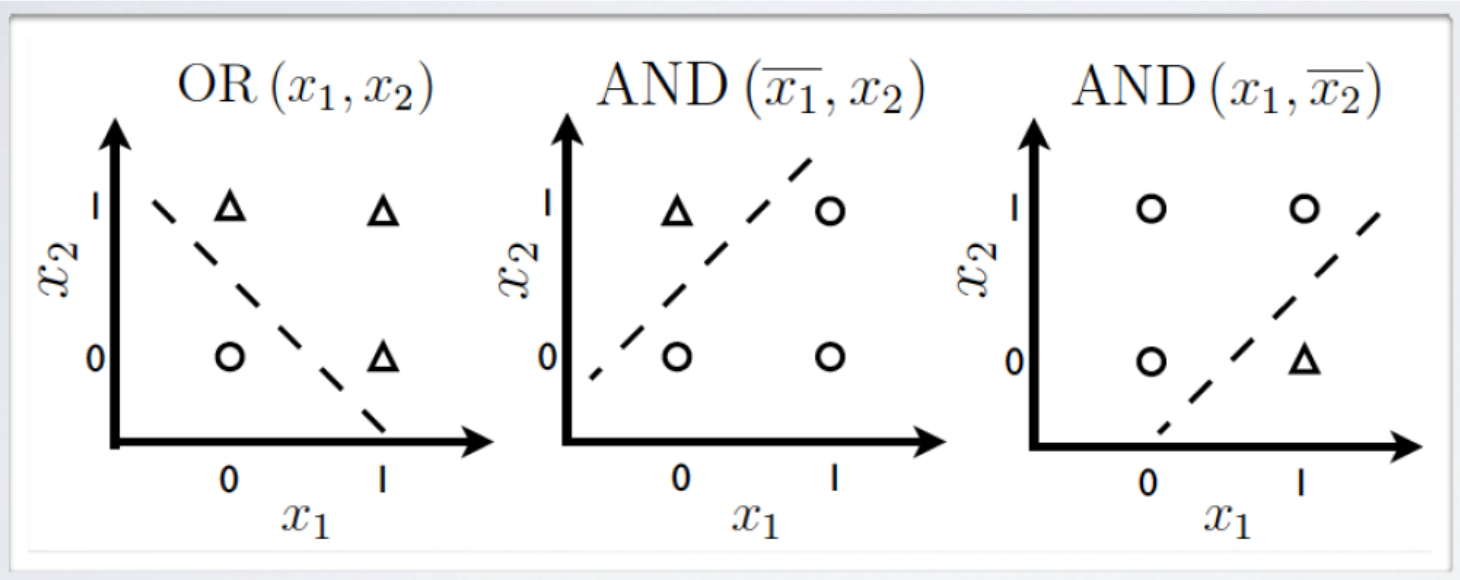
- Can solve linearly separable problems

$$\mathcal{D} = \mathcal{D}^+ \cup \mathcal{D}^-$$

$$\exists\, \mathbf{w}^*, \mathbf{w}^{*\mathsf{T}}\mathbf{x} > 0,\ \forall \mathbf{x} \in \mathcal{D}^+$$

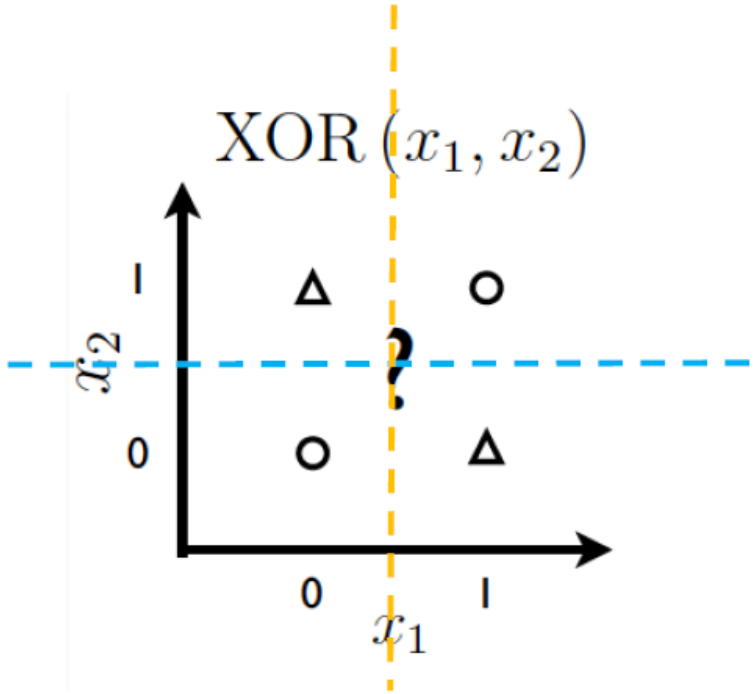
$$\mathbf{w}^{*\mathsf{T}}\mathbf{x} < 0,\ \forall \mathbf{x} \in \mathcal{D}^-$$

  - Examples



Capacity of Single Neuron

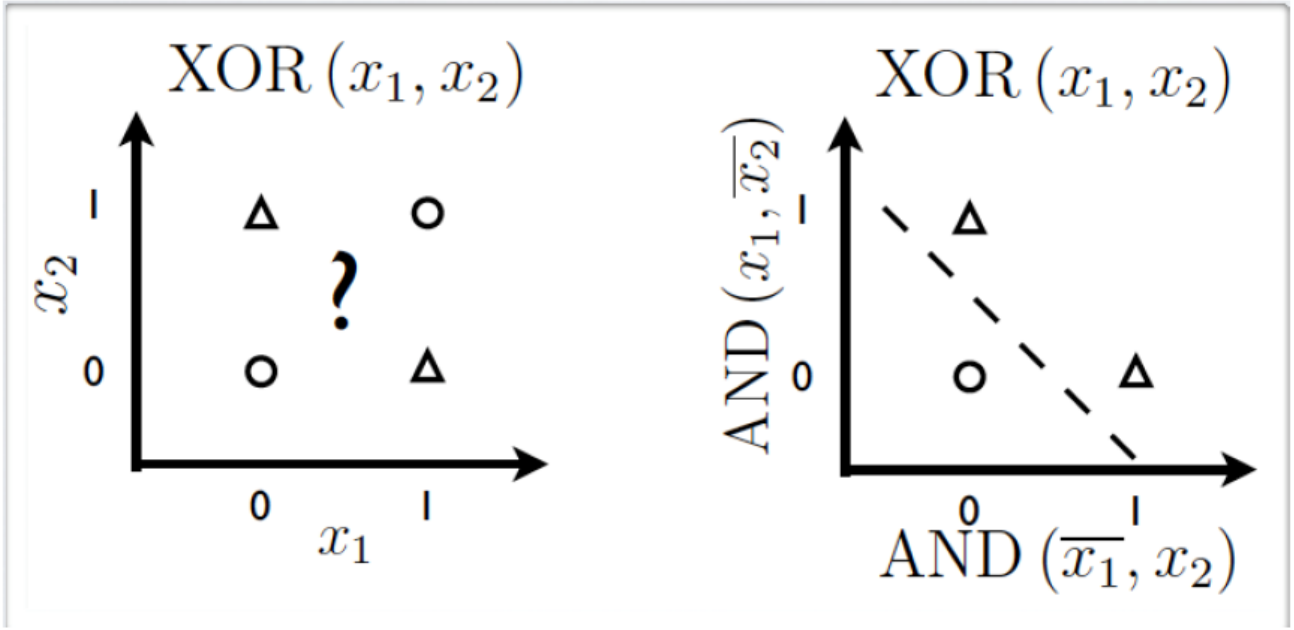
- Can't solve non linearly separable problems



- Can we use multiple neurons to achieve this?

Capacity of Single Neuron

- Can't solve non linearly separable problems
- Unless the input is transformed in a better representation



Adding One More Layer

- Single hidden layer neural network
  - 2-layer neural network: ignoring input units

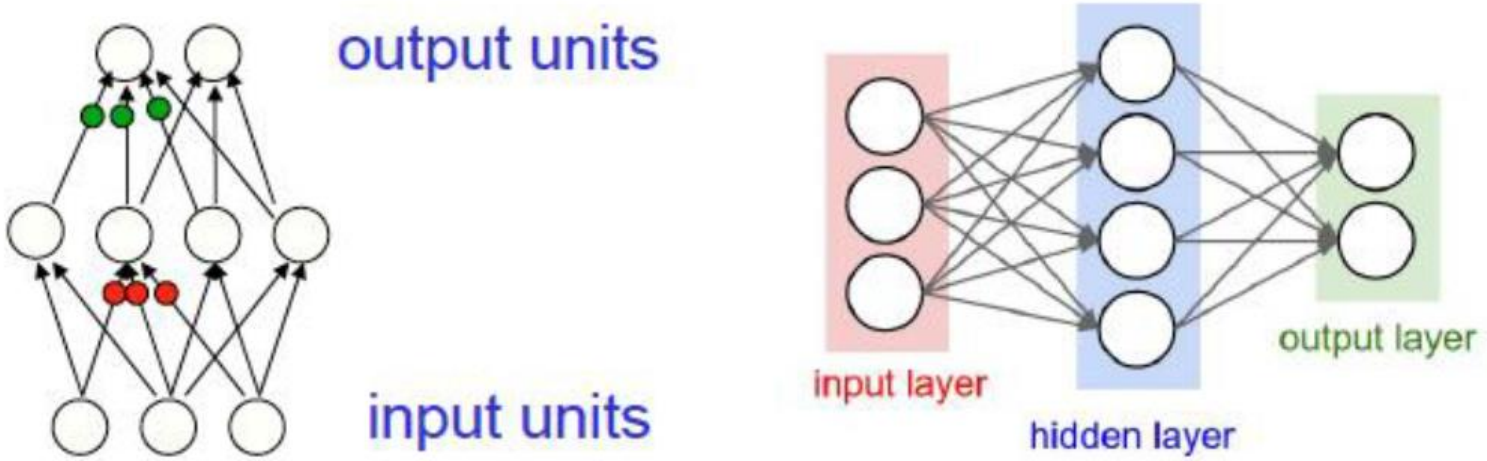


Figure : Two different visualizations of a 2-layer neural network. In this example: 3 input units, 4 hidden units and 2 output units

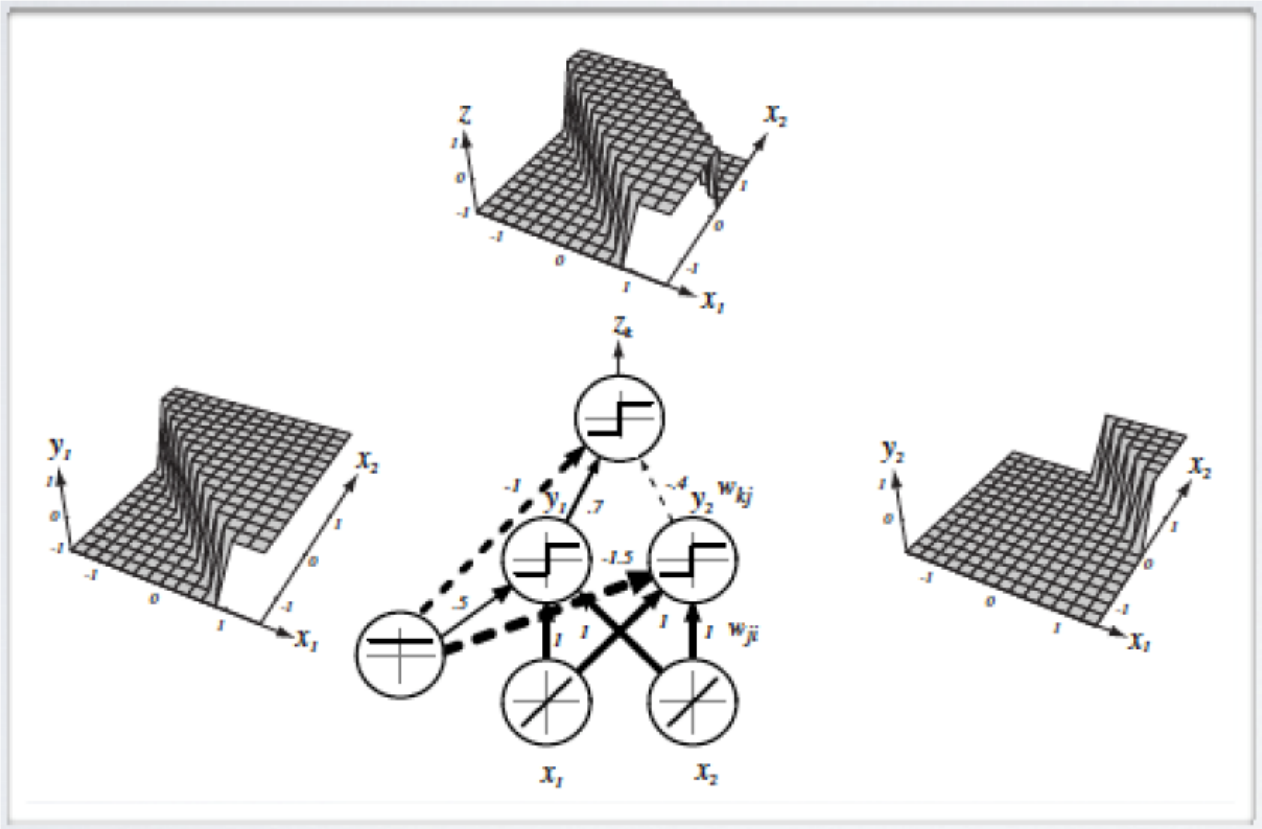
- Q: What if using linear activation in hidden layer?

Multi-Layer Neural Network

- Limitations of Single-Layer NN
- Generalization Ability
- Inference and Learning
- Practice

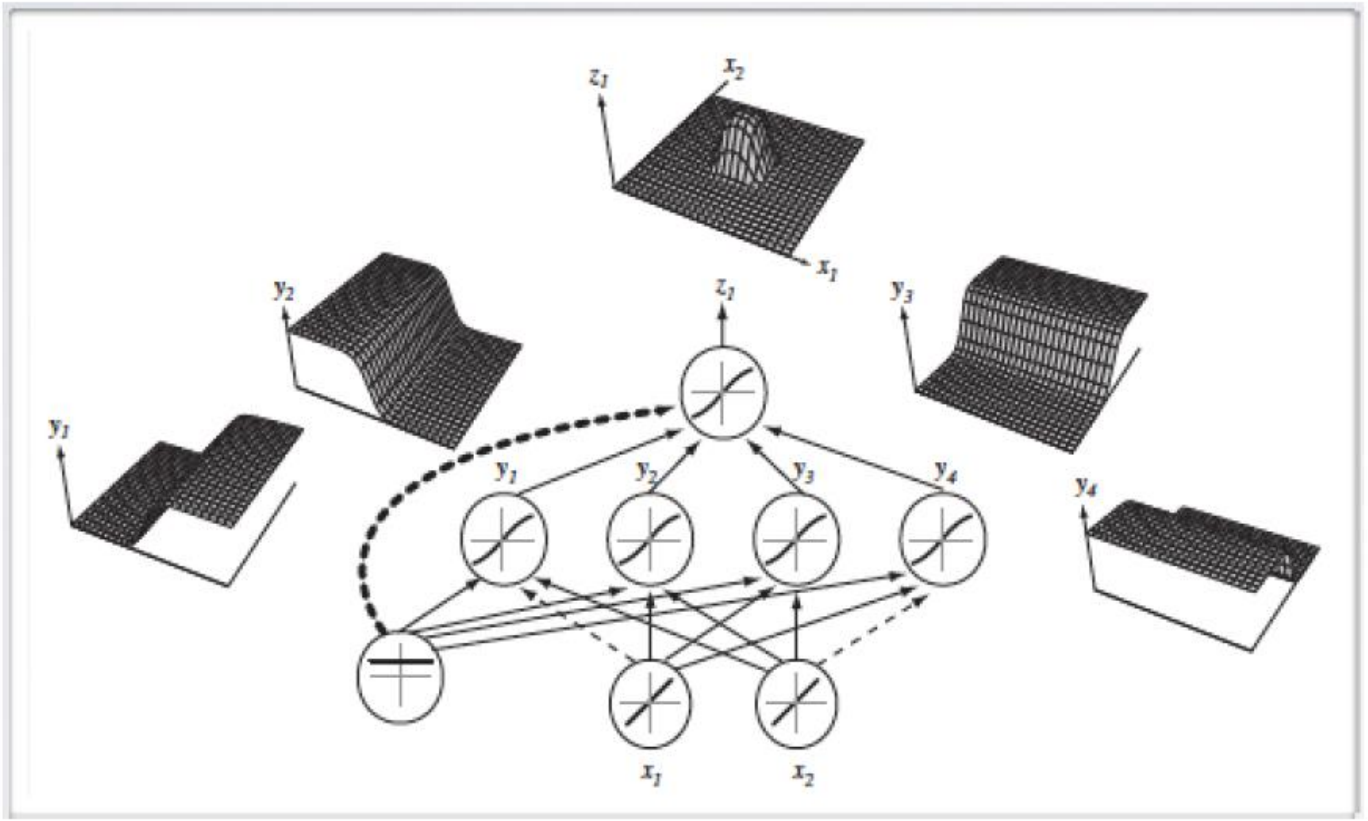
Capacity of Neural Network

- Single hidden layer neural network
  - Partition the input space into regions



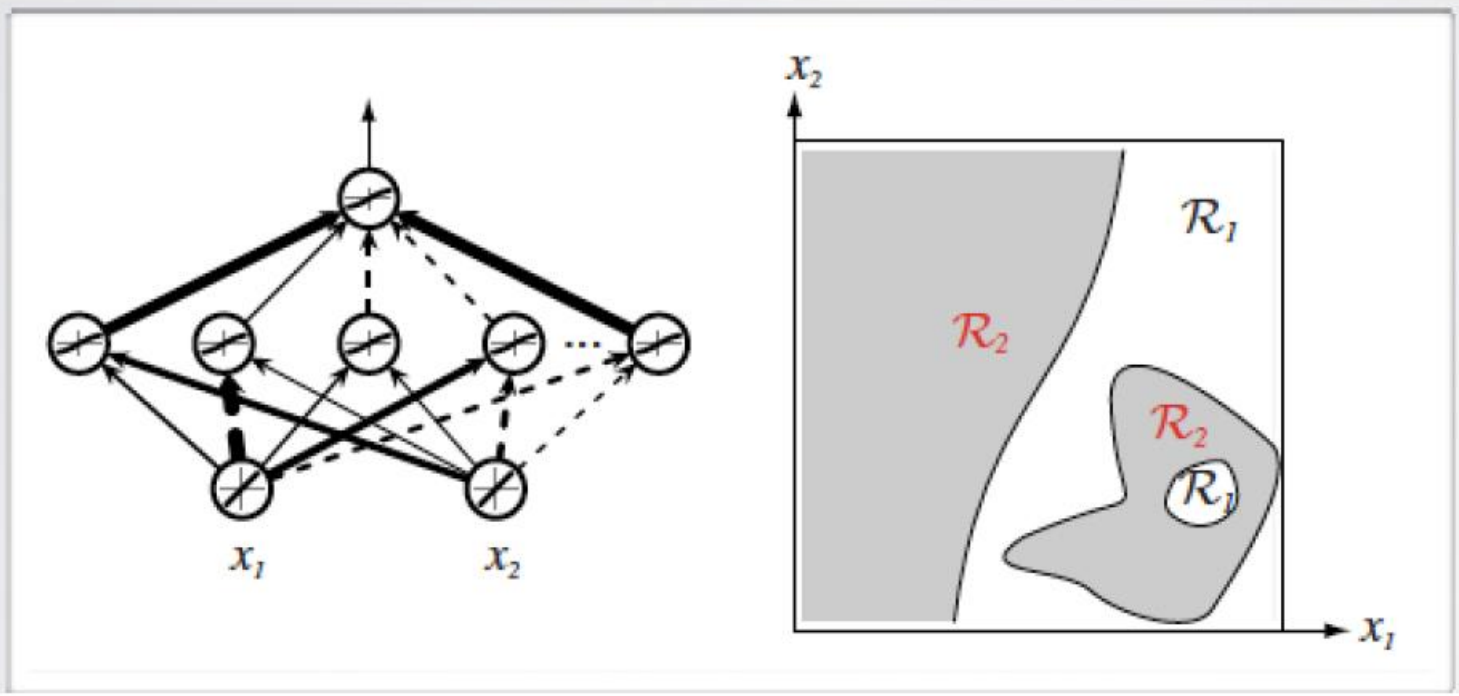
Capacity of Neural Network

- Single hidden layer neural network
  - Form a stump/delta function



Capacity of Neural Network

- Single hidden layer neural network



Capacity of Neural Network

- Universal approximation
  - Theorem (Hornik, 1991)

    A single hidden layer neural network with a linear output unit can approximate any continuous function arbitrarily well, given enough hidden units.
  - The result applies for sigmoid, tanh and many other hidden layer activation functions

- Caveat: good result but not useful in practice
  - How many hidden units?
  - How to find the parameters by a learning algorithm?

Multi-Layer Neural Network

- Limitations of Single-Layer NN
- Generalization Ability
- Inference and Learning
- Practice

General Neural Network

- Multi-layer neural network

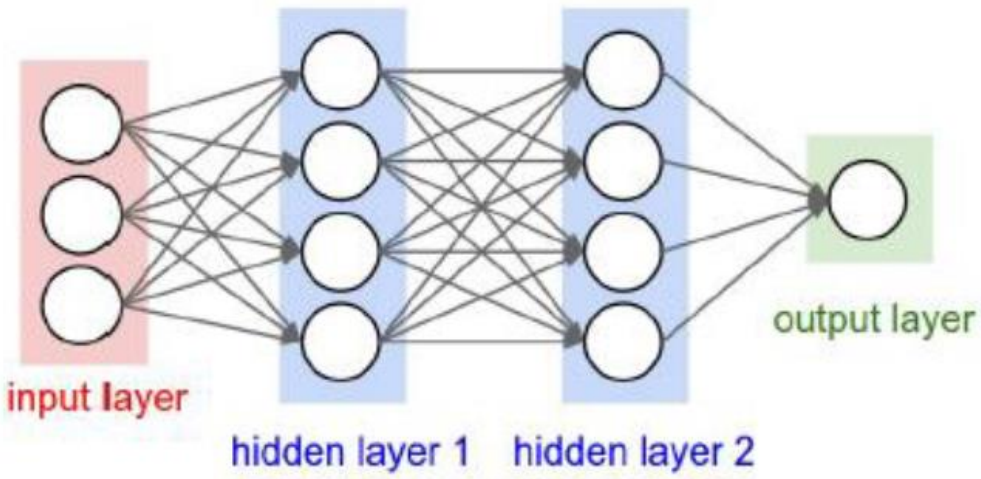
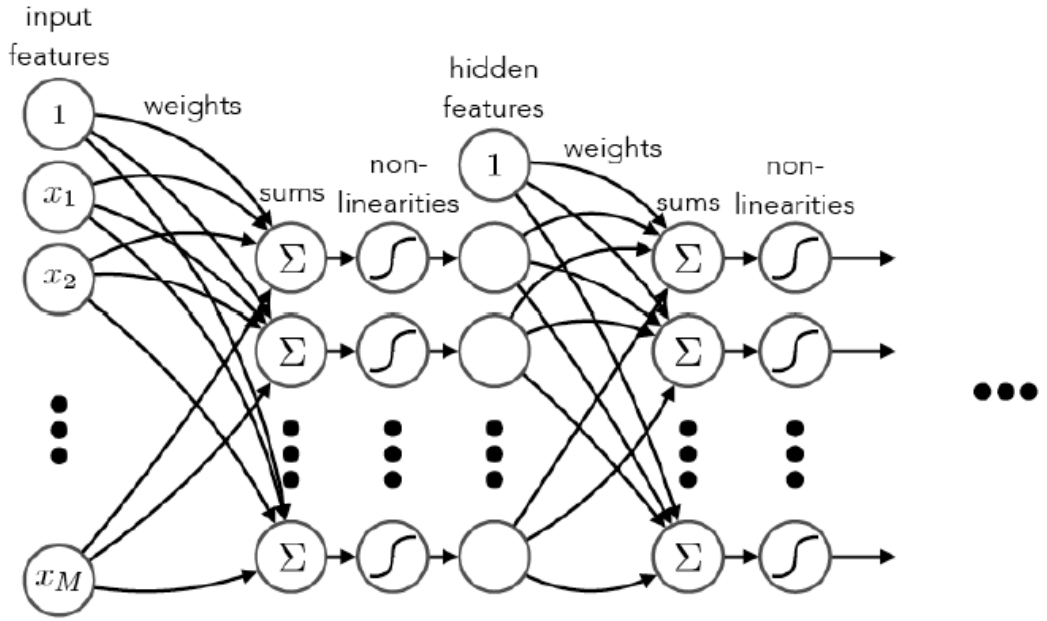


Figure : A 3-layer neural net with 3 input units, 4 hidden units in the first and second hidden layer and 1 output unit

- Naming conventions; a N-layer neural network:
  - $N-1$ layers of hidden units
  - One output layer

Multilayer Networks



network: *sequence of parallelized weighted sums and non-linearities*

DEFINE $\mathbf{x}^{(0)} \equiv \mathbf{x}$, $\mathbf{x}^{(1)} \equiv \mathbf{h}$, ETC.

1st layer

$$\mathbf{s}^{(1)} = \mathbf{W}^{(1)\intercal}\mathbf{x}^{(0)}$$

$$\mathbf{x}^{(1)} = \sigma(\mathbf{s}^{(1)})$$

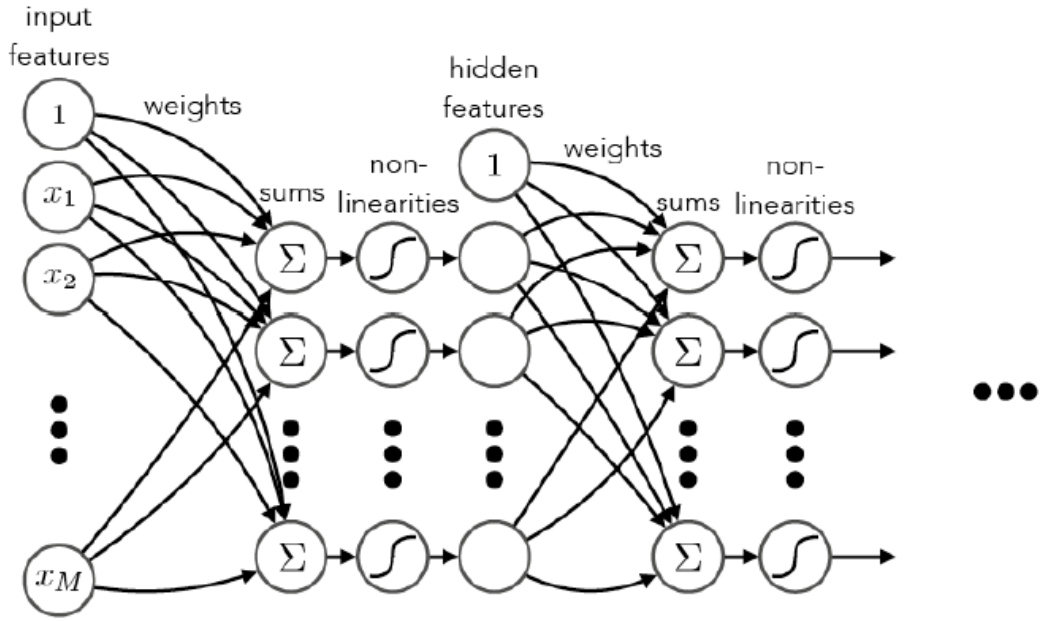
2nd layer

$$\mathbf{s}^{(2)} = \mathbf{W}^{(2)\intercal}\mathbf{x}^{(1)}$$

$$\mathbf{x}^{(2)} = \sigma(\mathbf{s}^{(2)})$$

...

Multilayer Networks

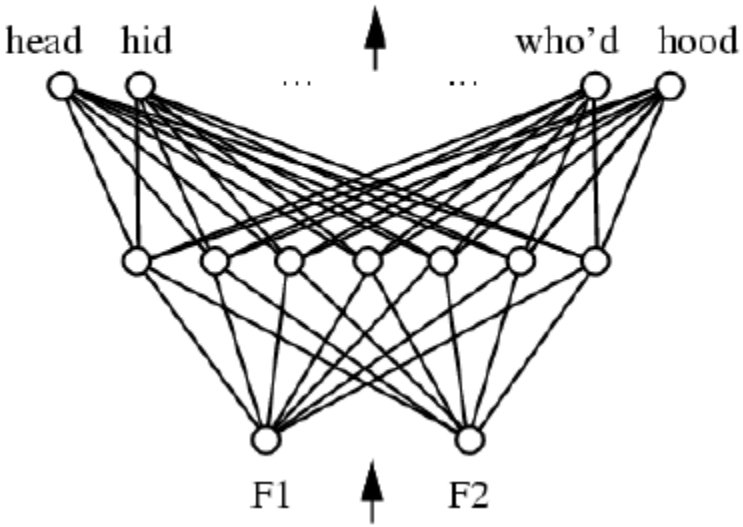


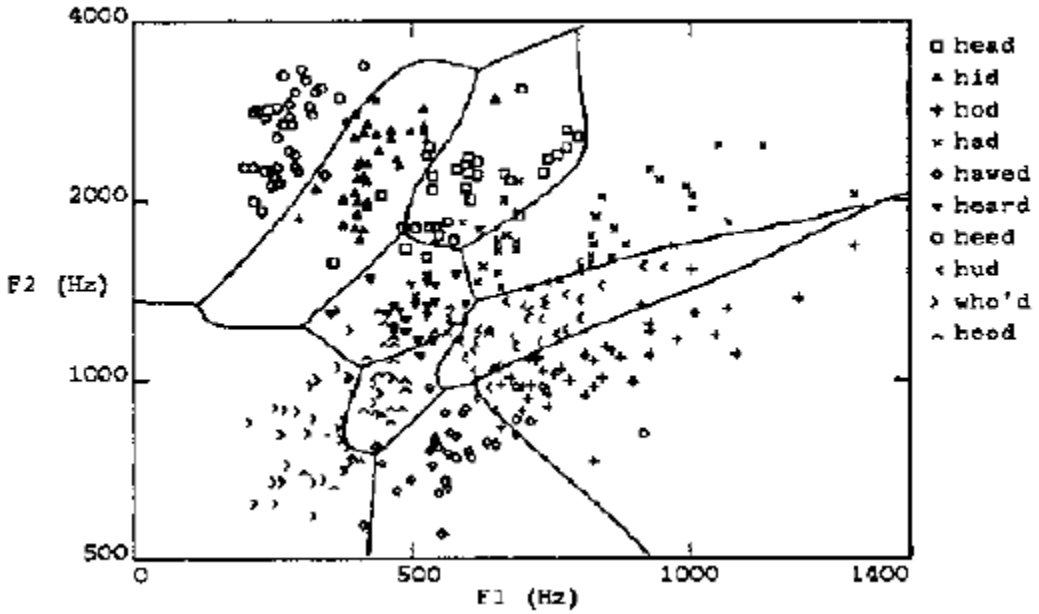
network: *sequence of parallelized weighted sums and non-linearities*

$$= \sigma(\ \ldots \ \sigma(\quad \sigma(\quad) \ ) \ \ldots)$$

output 2nd weights 1st weights input

Neural Networks - Example

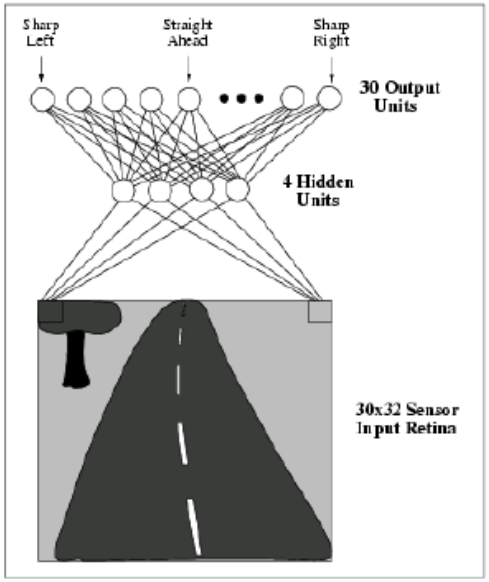


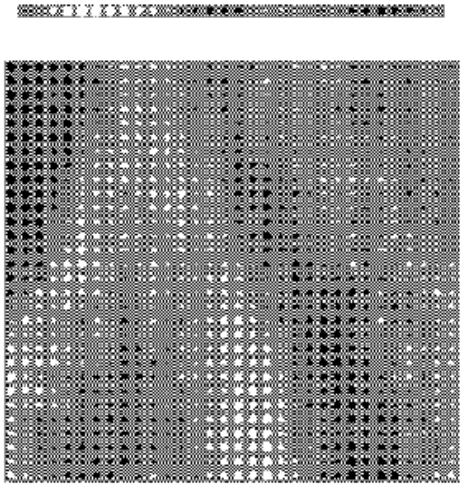


Neural Networks - Example

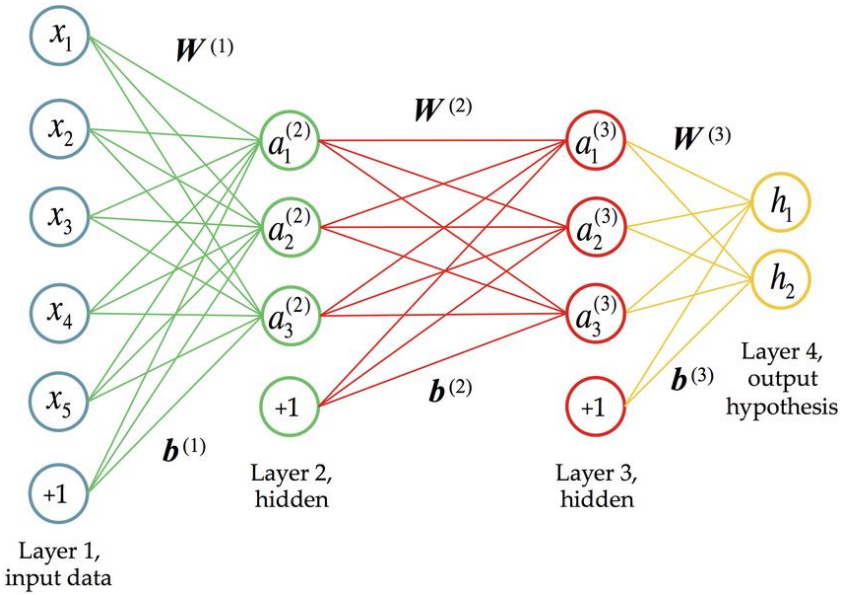
ALVINN

[Pomerleau 1993]





Biological Inspiration - A Layered Architecture

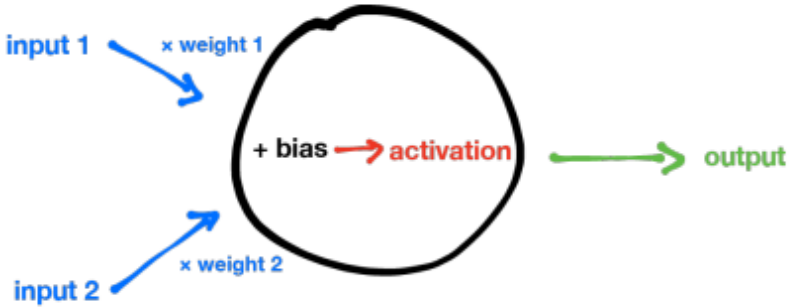


- Outputs of layer k would serve as the inputs to layers k+1 → **"feedforward"** (no feedback connections)
- **"Networks"** : typically represented by composing a chain of different functions:

$$f(x) = f^{(3)}f^{(2)}f^{(1)}(x)$$

Input Layer: takes in input data (size corresponds to input space)
Hidden Layer: neurons hidden from view (this is where the magic happens)
Output Layer: neurons in this layer provide the output of the network

The Hidden Unit



Given a vector of inputs **x**...

- Compute an affine transformation on **x** → z = $\mathbf{w}^T$ **x** +b
  - Inputs **x** to the neuron are multiplied by weights **w**
  - Bias **b** then summed with weighted inputs
- Non-linear activation function **g**(z) applied on z

For most hidden units, the sole difference lies within the form of the "activation function" **g**(z)!

Activation Functions

For gradient descent to work, we need activation functions that are:

- Continuous
- Differentiable: to compute gradients for each activation function *
- Monotonic: the error surface associated with a single-layer model is guaranteed to be convex

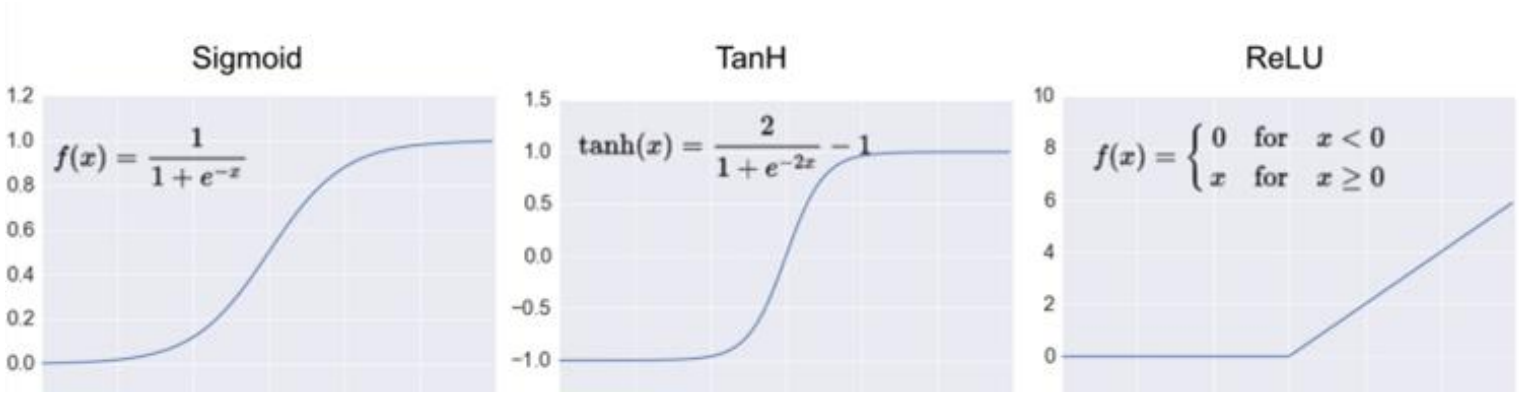


Image Credits

Activation Functions

- The set of weights from all the hidden units form the set of parameters **θ** the network learns
- As with parametric learning algorithms, neural networks also learn via **gradient descent** (why we need differentiable activation functions in the hidden layers)

NOTE: In practice, the activation function just needs to be differentiable at most points, e.g. ReLU's gradient is undefined at x = 0

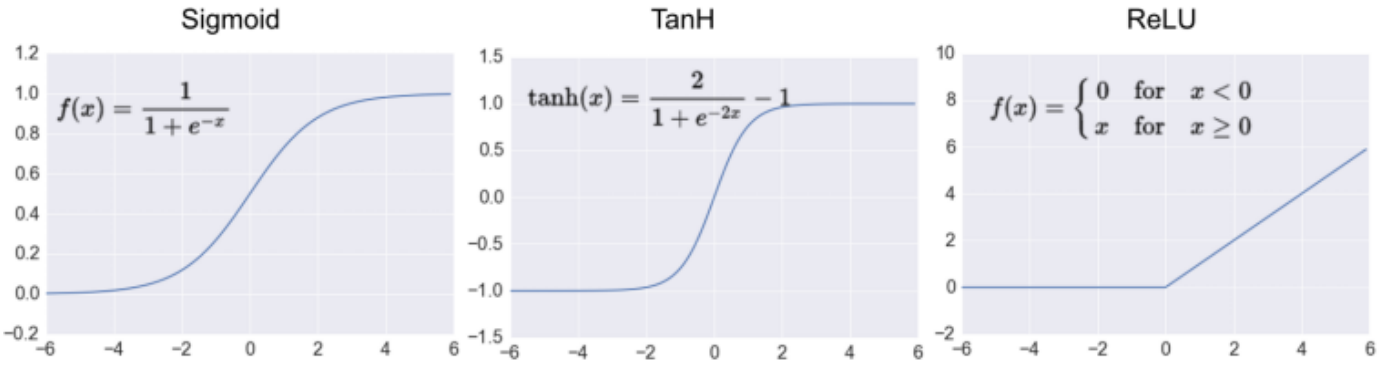


Image Credits

A Note On Choosing Activation Functions

Still an experimental field, no definitive rules by default

- ReLU is typically a good choice, does not experience vanishing gradient problem sigmoid and TanH functions have (also faster)
- Predicting which activation function work best in advance is **impossible** in practice

Design Protocol - Trial and Error

1. Pick a hidden unit you think will work well for neural network.
2. Train network with that type of hidden unit.
3. Evaluate network performance on validation set.

A Brief Note on Output Units

So far we've only talked about activation functions in **hidden units**. Depending on the task at hand, we define an appropriate output unit activation function, and corresponding cost function.

Can imagine the output of the hidden layers as a set of **hidden features** to the **output layer**:

$$h = f(x; \theta)$$

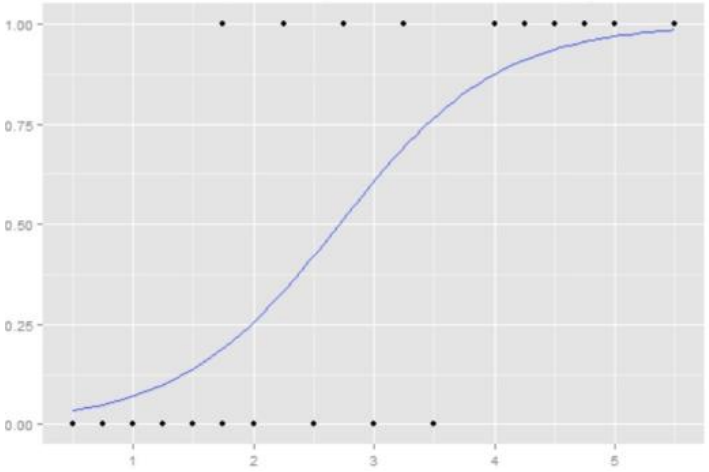
Then, the role of the output layer is to transform the set of hidden features to complete the task the network must perform

A Brief Note on Output Units

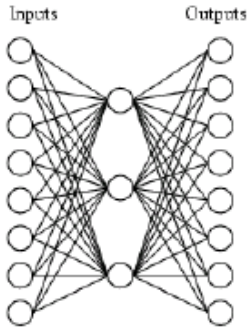
For regression, can use linear unit → **hidden** features are the inputs of a linear model

For binary classification, can use sigmoid → **hidden features** are input of logistic regression

Many other activation functions possible



Learning Hidden Layer Representations



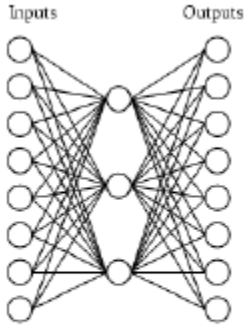
A target function:

Input		Output
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001

Can this be learned??

Learning Hidden Layer Representations

A network:



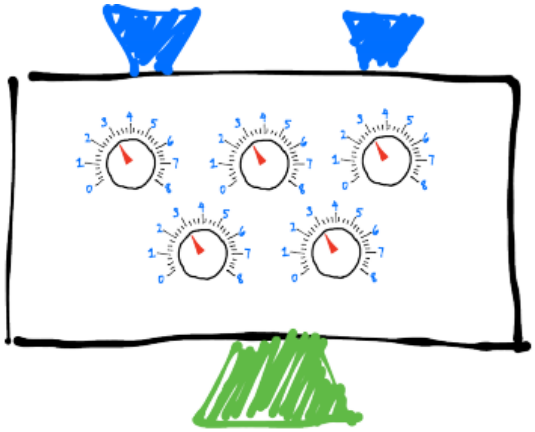
Learned hidden layer representation:

Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

Multi-Layer Neural Network

- Limitations of Single-Layer NN
- Generalization Ability
- Inference and Learning
- Practice

Learning in Neural Networks



- From an interpretability point of view, the neural network seems like a **black box**
- Neural networks as a black box: Modern networks often times have hundreds of millions of parameters, sometimes even billions
- Since we have so many parameters to learn, need **lots of data**

How do we learn these parameters?

Computations in Neural Network

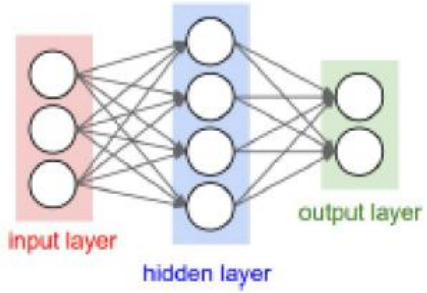
- We only need to know two algorithms
  - Inference/prediction: simply forward pass
  - Parameter learning: needs backward pass
- Basic fact:
  - A neural network is a function of composed operations

$$f_L(\mathbf{w}_L, f_{L-1}(\mathbf{w}_{L-1}, \dots f_1(\mathbf{w}_1, \mathbf{x}) \dots))$$

  - All the f functions are linear + (simple) nonlinear (differentiable a.e.) operators

Inference Example: Forward Pass

- What does the network compute?



- Output of the network can be written as:

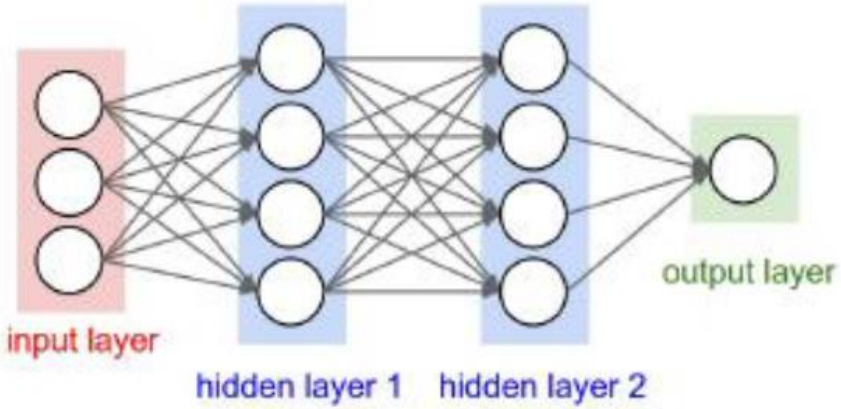
$$h_j(\mathbf{x}) = f(v_{j0} + \sum_{i=1}^{D} x_i v_{ji})$$

$$o_k(\mathbf{x}) = g(w_{k0} + \sum_{j=1}^{J} h_j(\mathbf{x}) w_{kj})$$

(j indexing hidden units, k indexing the output units, D number of inputs)

Forward Pass in Python

- Example code for a forward pass for a 3-layer network in Python:




```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

- Can be implemented efficiently using matrix operations

Parameter Learning: Backward Pass

- Supervised learning framework
  - Find weights:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{n=1}^{N} \operatorname{loss}(\mathbf{o}^{(n)}, \mathbf{t}^{(n)})$$

    where $\mathbf{o} = f(\mathbf{x}; \mathbf{w})$ is the output of a neural network
  - Define a loss function, eg:
    - Squared loss: $\sum_k \frac{1}{2}(o_k^{(n)} - t_k^{(n)})^2$
    - Cross-entropy loss: $-\sum_k t_k^{(n)} \log o_k^{(n)}$
  - Gradient descent:

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \frac{\partial E}{\partial \mathbf{w}^t}$$

    where η is the learning rate (and E is error/loss)

Backward Pass

- Backpropagation
  - An efficient method for computing gradients in NNs
  - A neural network as a function of composed operations

$$f_L(\mathbf{w}_L, f_{L-1}(\mathbf{w}_{L-1}, \dots f_1(\mathbf{w}_1, \mathbf{x}) \dots))$$

and the loss $\mathcal{L}$ is a function of the network output

→ use chain rule to calculate gradients

Review: Chain Rule

- Formal definition

For any nested function $y = f(g(x))$

$$\boxed{\frac{dy}{dx} = \frac{\partial y}{\partial g(x)} \frac{dg(x)}{dx}}$$

Check – we can confirm that : $\Delta y = \frac{dy}{dx} \Delta x$

$z = g(x) \Rightarrow \Delta z = \frac{dg(x)}{dx} \Delta x$

$y = f(z) \Rightarrow \Delta y = \frac{dy}{dz} \Delta z = \frac{dy}{dz} \frac{dg(x)}{dx} \Delta x$

Gradient Descent Iteration

- Forward pass

1st layer

$$\mathbf{s}^{(1)} = \mathbf{W}^{(1)\intercal}\mathbf{x}^{(0)}$$
$$\mathbf{x}^{(1)} = \sigma(\mathbf{s}^{(1)})$$

2nd layer

$$\mathbf{s}^{(2)} = \mathbf{W}^{(2)\intercal}\mathbf{x}^{(1)}$$
$$\mathbf{x}^{(2)} = \sigma(\mathbf{s}^{(2)})$$

$$\cdots$$

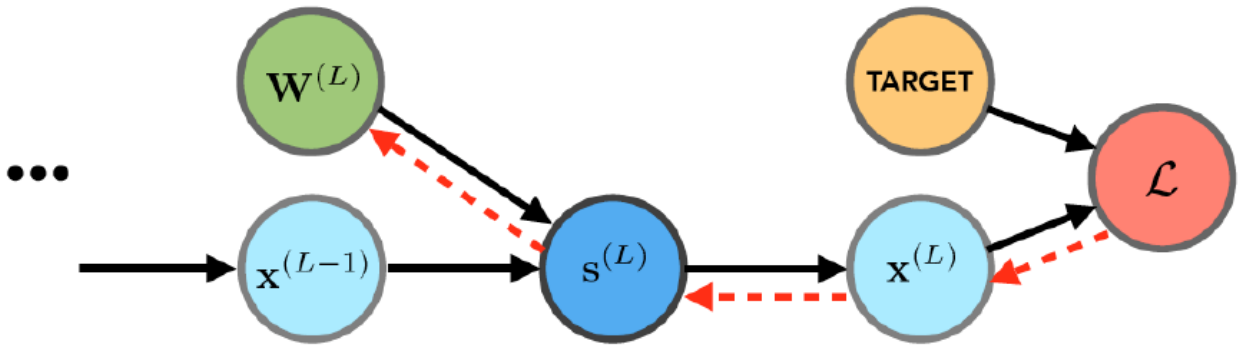
Loss

$$\mathcal{L}$$

- Backward pass

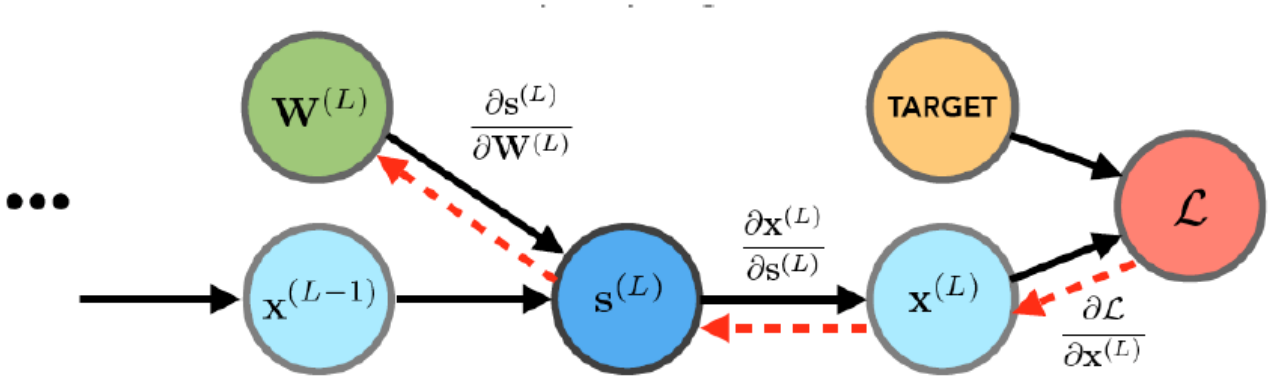
calculate $\nabla_{W^{(1)}}\mathcal{L}, \nabla_{W^{(2)}}\mathcal{L}, \ldots$ let's start with the final layer: $\nabla_{W^{(L)}}\mathcal{L}$

to determine the chain rule ordering, we'll draw the dependency graph



Gradient Descent Iteration

- Backward pass



$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{s}^{(L)}} \frac{\partial \mathbf{s}^{(L)}}{\partial \mathbf{W}^{(L)}}$$

depends on the form of the loss

derivative of the non-linearity

$$\frac{\partial}{\partial \mathbf{W}^{(L)}} (\mathbf{W}^{(L)\intercal} \mathbf{x}^{(L-1)}) = \mathbf{x}^{(L-1)\intercal}$$

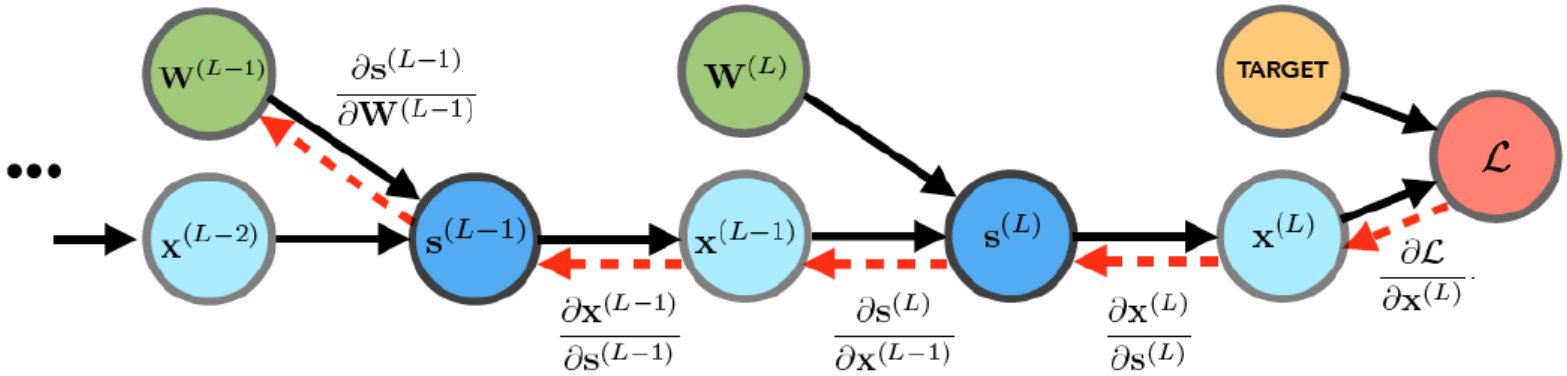
note $\nabla_{\mathbf{W}^{(L)}} \mathcal{L} \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}}$ is notational convention

Gradient Descent Iteration

- Backward pass

now let's go back one more layer…

again we'll draw the dependency graph:

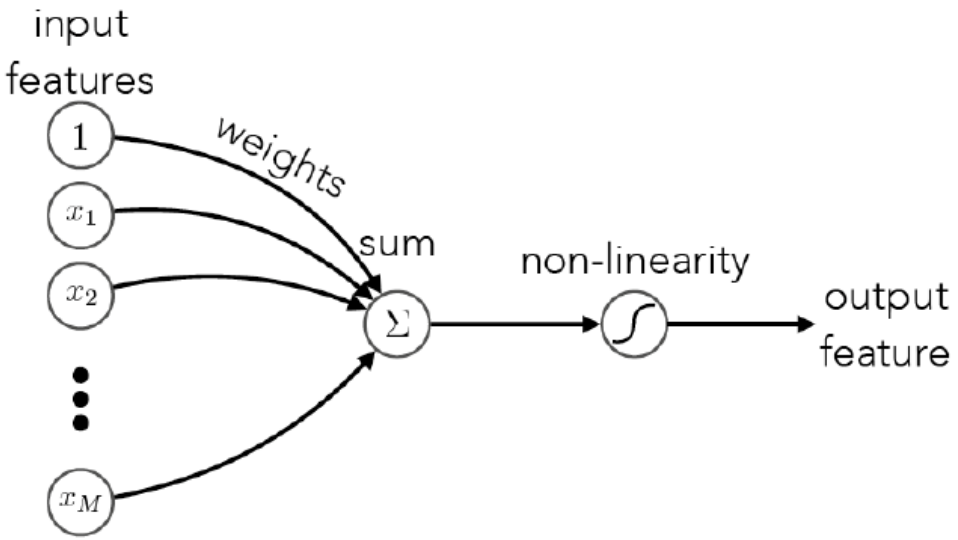


$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{s}^{(L)}} \frac{\partial \mathbf{s}^{(L)}}{\partial \mathbf{x}^{(L-1)}} \frac{\partial \mathbf{x}^{(L-1)}}{\partial \mathbf{s}^{(L-1)}} \frac{\partial \mathbf{s}^{(L-1)}}{\partial \mathbf{W}^{(L-1)}}$$

An Implementation Perspective

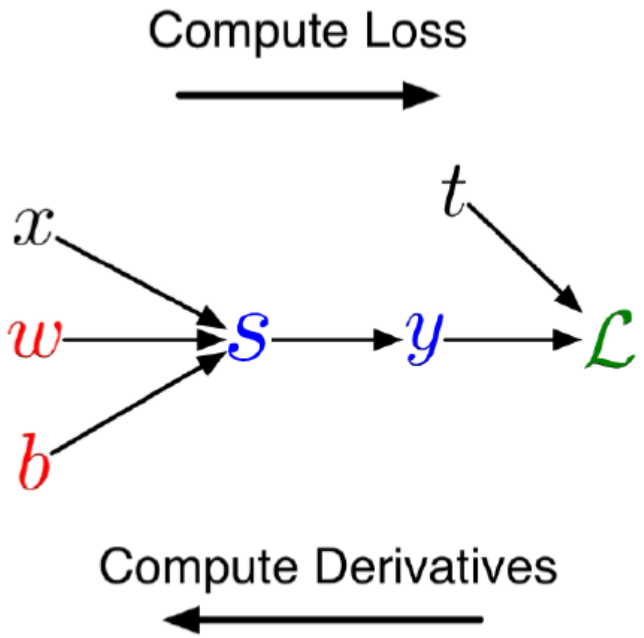
- Example: Univariate logistic least square model

$$s = wx + b$$
$$y = \sigma(s)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$



Computation Graph

- Represent the computations using a computation graph
  - Nodes: inputs & computed quantities
  - Edges: which nodes are computed directly as function of which other nodes



Univariate Chain Rule

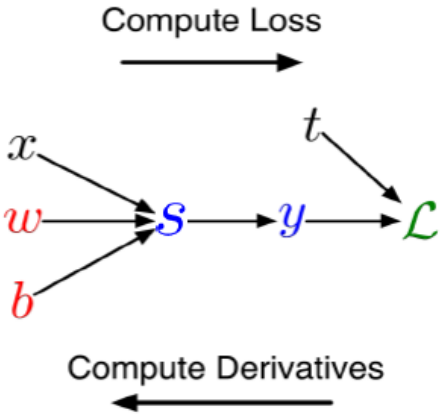
- A shorthand notation
  - Use $\delta_y := d\mathcal{L}/dy$, called the error signal
  - Note that the error signals are values computed by the program

Computing the loss:

$$s = wx + b$$
$$y = \sigma(s)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

Computing the derivatives:

$$\delta_y = y - t$$
$$\delta_s = \delta_y \sigma'(s)$$
$$\delta_w = \delta_s x$$
$$\delta_b = \delta_s$$



General Backpropagation

- Given a computation graph

Let $v_1, \ldots, v_N$ be a topological ordering of the computation graph (i.e. parents come before children.)

v_N denotes the variable we're trying to compute derivatives of (e.g. loss)

forward pass

For $i = 1, \ldots, N$

Compute v_i as a function of $\text{Pa}(v_i)$

backward pass

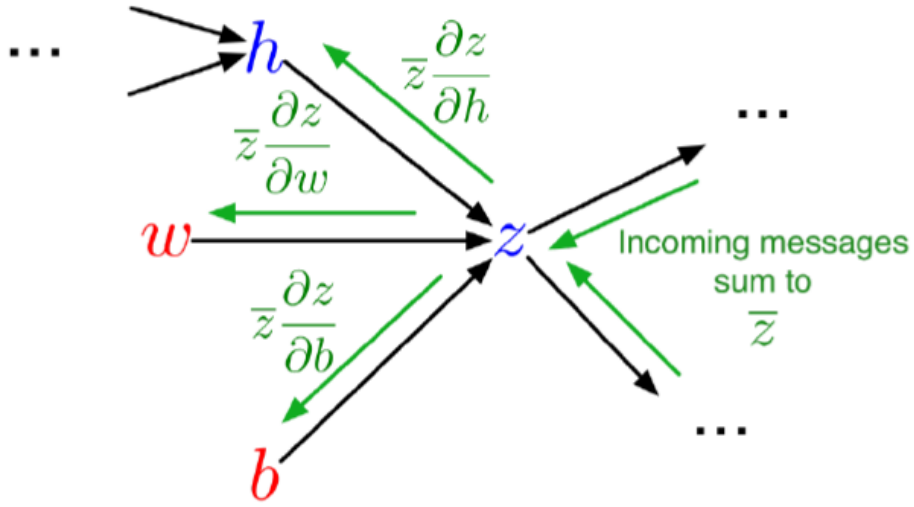
$\delta_{v_N} = 1$

For $i = N - 1, \ldots, 1$

$\delta_{v_i} = \sum_{j \in \text{Ch}(v_i)} \delta_{v_j} \frac{\partial v_j}{\partial v_i}$

General Backpropagation

- Backprop as message passing:



  - Each node receives a set of messages from its children, which are aggregated into its error signal, then it passes messages to its parents
  - Modularity: each node only has to know how to compute derivatives w.r.t. its arguments – local computation in the graph

Backpropagation

- Backprop is used to train the majority of neural nets
  - Even generative network learning, or advanced optimization algorithms (second-order) use backprop to compute the update of weights
- However, backprop seems biologically implausible
  - No evidence for biological signals analogous to error derivatives
  - All the existing biologically plausible alternatives learn much more slowly on computers.
  - So how on earth does the brain learn???

Gradient Descent Procedure

1. **Specify the neural network architecture:**
   a. Input layer size → how many input features we feed into network
   b. Output layer size, output unit type → depends on output shape, task at hand
   c. Hidden units/layer, number of hidden layers*

2. **Initialize weights to small random numbers.**

3. **Repeat following until convergence:**
   a. Forward pass
   b. Backpropagation
      i. Compute share of error at the output unit
      ii. Propagate share of error for hidden units
   c. Gradient Descent
      i. Update each network weight based on gradient

Forms of Gradient Descent

Stochastic Gradient Descent: compute error using a single sample at a time, update weights, repeat

Batch Gradient Descent: compute error on all examples, update weights based on error, repeat

Mini-batch Gradient Descent: randomly select a subset from the training data, calculate error on subset, update weights, repeat

** Most Commonly Done in Practice

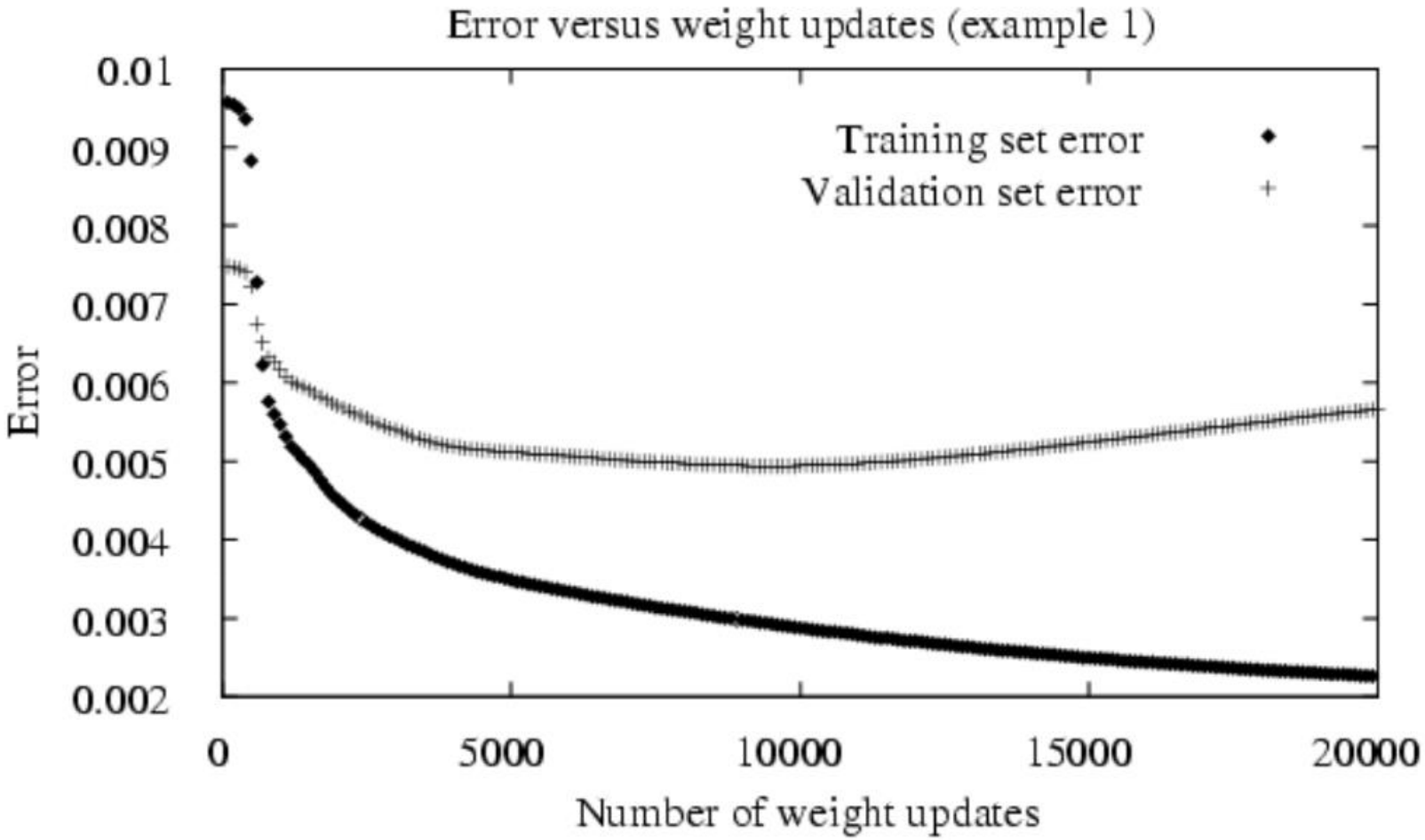
More on Backpropagation

- Gradient descent over entire *network* weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
  - In practice, often works well (can run multiple times)
- Often include weight *momentum* α

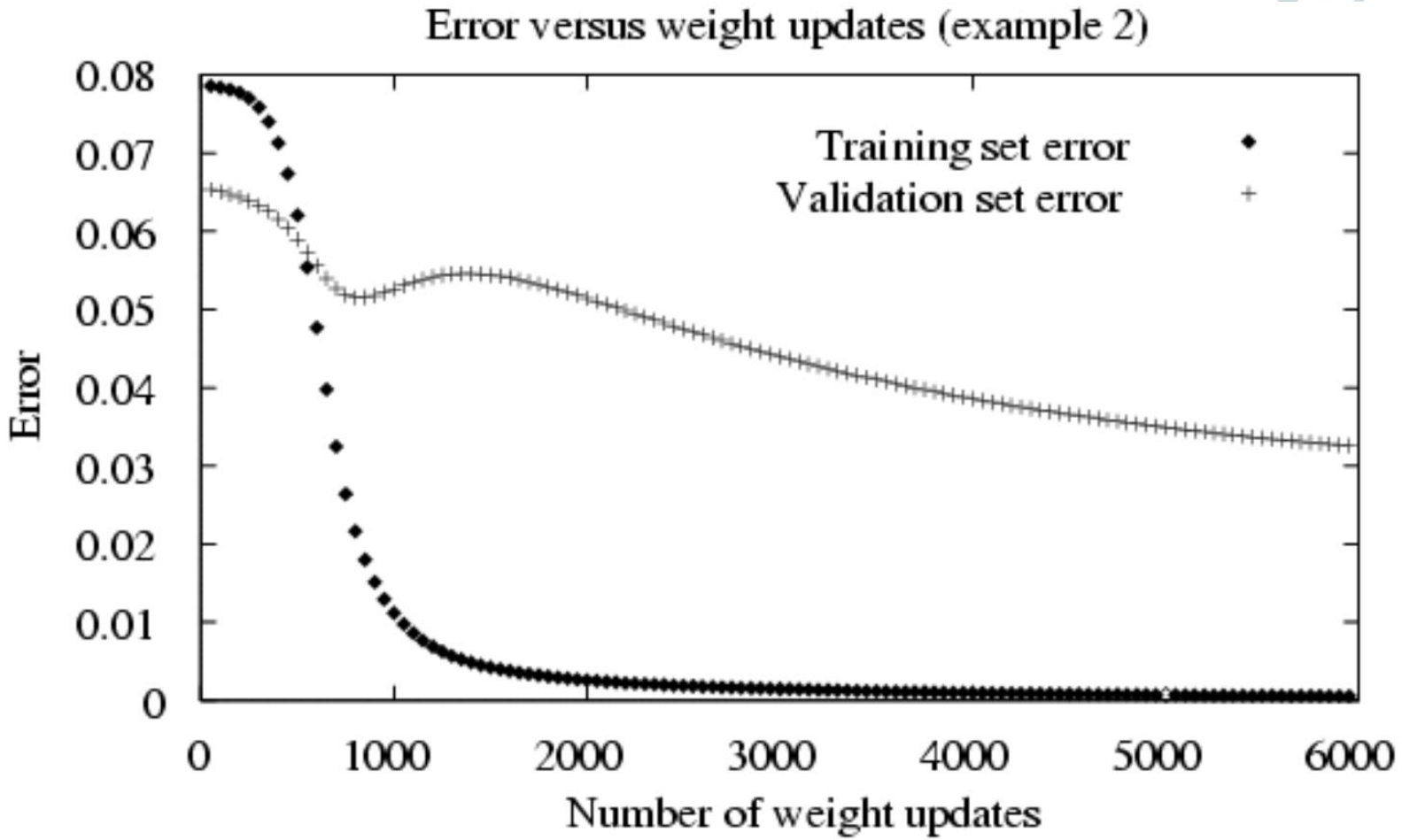
$$\Delta w_{i,j}(n) = \eta \delta_j x_{i,j} + \alpha \Delta w_{i,j}(n-1)$$

- Minimizes error over *training* examples
  - Will it generalize well to subsequent examples?
- Training can take thousands of iterations $\rightarrow$ slow!
- Using network after training is very fast

Overfitting in NN



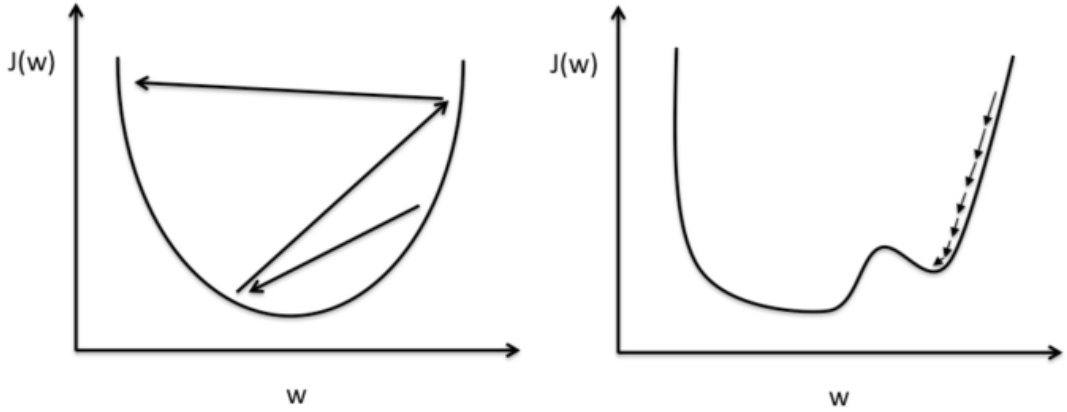
Underfitting in NN



Picking α - Adaptive Learning Rates

- Optimization methods that are used in practice compute more optimal learning rates for better performance
- Typically, we use larger learning rates at the beginning (to take bigger steps towards a local minima) and start to slow down as we get closer to the desired value (once again, to not oscillate too much)

E.g. ADAM, RMSProp, Adagrad, etc.
More Information on Stochastic Gradient Descent and Different Optimization Methods



Multi-Layer Neural Network

- Limitations of Single-Layer NN
- Generalization Ability
- Inference and Learning
- **Practice**

Use Numpy to Implement A Neural Network

- A fully-connected two-layer ReLU network: one hidden layer, no bias.
- Input: x and output: y and use L2 Loss

$$h=W_1X$$
$$a=max(0,h)$$
$$y=W_2a$$

- Three key components:

1. forward pass
2. loss computation
3. backward pass

- We need to do everything by ourselves.

Use Numpy to Implement A Neural Network

Step 0 -- Initialization:

samples: 64

features: 1000

hidden units: 100

output units: 10

```
N, D_in, H, D_out = 64, 1000, 100, 10

# random data generation
x = np.random.randn(N, D_in)
y = np.random.randn(N, D_out)

w1 = np.random.randn(D_in, H)
w2 = np.random.randn(H, D_out)

learning_rate = 1e-6
```

Use Numpy to Implement A Neural Network

Step 1 -- Forward pass:

$h=W_1X$

$a=max(0,h)$

$y=W_2a$

```
# Forward pass
h = x.dot(w1) # N * H
h_relu = np.maximum(h, 0) # N * H
y_pred = h_relu.dot(w2) # N * D_out
```

Use Numpy to Implement A Neural Network

Step 2 – Compute loss:

Squared loss

```
# compute loss
loss = np.square(y_pred - y).sum()
print(it, loss)
```

Use Numpy to Implement A Neural Network

Step 3 – Backpropagation:

Compute gradient

Update weights W_1 and W_2

```
# Backward pass
# compute the gradient
grad_y_pred = 2.0 * (y_pred - y)
grad_w2 = h_relu.T.dot(grad_y_pred)
grad_h_relu = grad_y_pred.dot(w2.T)
grad_h = grad_h_relu.copy()
grad_h[h<0] = 0
grad_w1 = x.T.dot(grad_h)

# update weights of w1 and w2
w1 -= learning_rate * grad_w1
w2 -= learning_rate * grad_w2
```

PyTorch is a scientific computing library based on Python:

- Similar to NumPy, but it can use GPU
- Can be used to build, train and test deep network models in a flexible manner
- Tensor is similar to ndarray in NumPy, and the main difference is that Tensor can use GPU for faster computation.

Can we use PyTorch to re-implement it?

- A fully-connected two-layer ReLU network: one hidden layer, no bias.
- Input: x and output: y and use L2 Loss

$$h=W_1X$$
$$a=max(0,h)$$
$$y=W_2a$$

- Three key components:

1. forward pass
2. loss computation
3. backward pass

- We need to do everything by ourselves.

```
N, D_in, H, D_out = 64, 1000, 100, 10

# random data generation
x = np.random.randn(N, D_in)
y = np.random.randn(N, D_out)

w1 = np.random.randn(D_in, H)
w2 = np.random.randn(H, D_out)

learning_rate = 1e-6
for it in range(500):
    # Forward pass
    h = x.dot(w1) # N * H
    h_relu = np.maximum(h, 0) # N * H
    y_pred = h_relu.dot(w2) # N * D_out

    # compute loss
    loss = np.square(y_pred - y).sum()
    print(it, loss)

    # Backward pass
    # compute the gradient
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.T.dot(grad_y_pred)
    grad_h_relu = grad_y_pred.dot(w2.T)
    grad_h = grad_h_relu.copy()
    grad_h[h<0] = 0
    grad_w1 = x.T.dot(grad_h)

    # update weights of w1 and w2
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

```python
N, D_in, H, D_out = 64, 1000, 100, 10

# random data generation
x = np.random.randn(N, D_in)
y = np.random.randn(N, D_out)

w1 = np.random.randn(D_in, H)
w2 = np.random.randn(H, D_out)

learning_rate = 1e-6
for it in range(500):
    # Forward pass
    h = x.dot(w1) # N * H
    h_relu = np.maximum(h, 0) # N * H
    y_pred = h_relu.dot(w2) # N * D_out

    # compute loss
    loss = np.square(y_pred - y).sum()
    print(it, loss)

    # Backward pass
    # compute the gradient
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.T.dot(grad_y_pred)
    grad_h_relu = grad_y_pred.dot(w2.T)
    grad_h = grad_h_relu.copy()
    grad_h[h<0] = 0
    grad_w1 = x.T.dot(grad_h)

    # update weights of w1 and w2
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

Numpy

```python
N, D_in, H, D_out = 64, 1000, 100, 10

# random data generation
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

w1 = torch.randn(D_in, H)
w2 = torch.randn(H, D_out)

learning_rate = 1e-6
for it in range(500):
    # Forward pass
    h = x.mm(w1) # N * H
    h_relu = h.clamp(min=0) # N * H
    y_pred = h_relu.mm(w2) # N * D_out

    # compute loss
    loss = (y_pred - y).pow(2).sum().item()
    print(it, loss)

    # Backward pass
    # compute the gradient
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h<0] = 0
    grad_w1 = x.t().mm(grad_h)

    # update weights of w1 and w2
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

PyTorch

Further Readings

Deep Learning Chapter 6 - Ian GoodFellow, Yoshua Bengio, and Aaron Courville

Article of Backpropagation and Gradient Descent

How the backpropagation algorithm works - Michael Nielsen

A Practical Guide to ReLU

Empirical Evaluation of Rectified Activations in Convolution Network (Xu et al. 2015)

Visualizing Neural Networks from a Manifold Perspective (Olah 2014)