THE BASICS OF NEURAL NETS

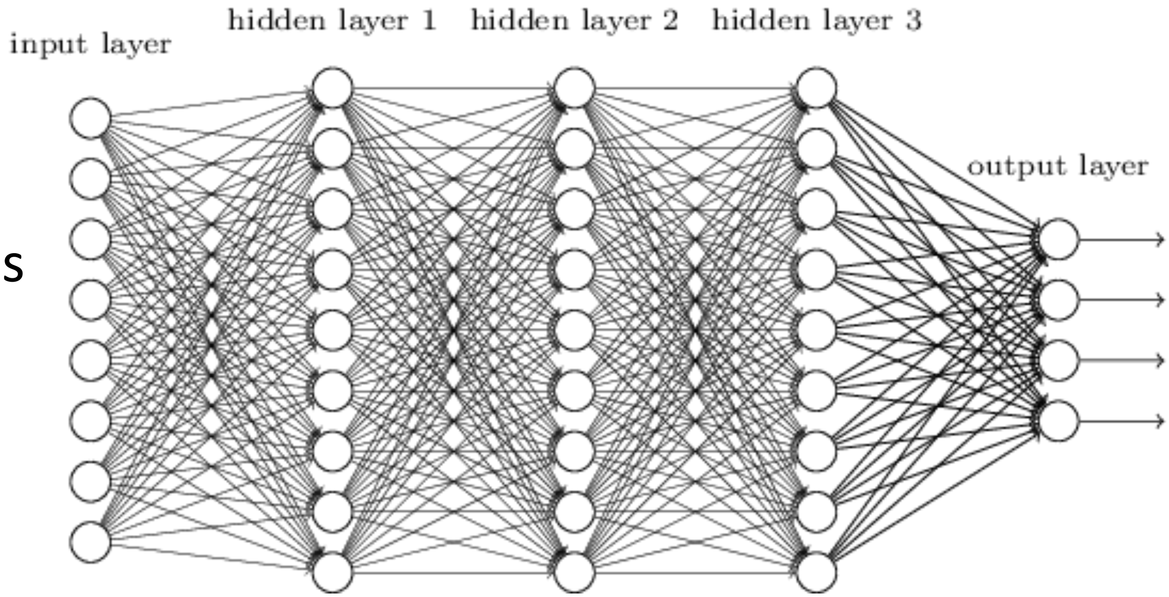
Bryan Pardo

Interactive Audio Lab

Northwestern University

Deep Nets (AKA Neural Nets)

- Machine learners
- made of simple functions
- Organized in layers
- Very popular



Deep Nets (AKA Neural Nets)

- Machine learners made of many simple functions connected by weight parameters
- Can model functions from $\mathbb{R}^d$ to $\mathbb{R}^d$ where d is very large (e.g. 10^5)
- Can learn arbitrary Boolean functions and very complex manifolds
- Often require lots of data (e.g. millions of examples)
- Use gradient descent and are thus susceptible to being stuck in local minima
- Opaque (internal representations are difficult to interpret)

Why I care about deep nets

They are key technology that is enables state of the art performance in….

Image recognition (e.g. Facebook and Google image labeling)

Speech recognition (Cortana)

Machine translation (Google Translate)

Playing video games (Why are we automating this?)

Playing board games (e.g. AlphaGo)

Machine Learning in one slide

1. Pick data **D,** model $\mathbf{M}(\mathbf{w})$ and objective function $\mathbf{J}(\mathbf{D}, \mathbf{w})$
2. Initialize model parameters $\mathbf{w}$ somehow
3. Measure model performance with the objective function $\mathbf{J}(\mathbf{D}, \mathbf{w})$
4. Modify parameters $\mathbf{w}$ somehow, hoping to improve $\mathbf{J}(\mathbf{D}, \mathbf{w})$
5. Repeat 3 and 4 until you stop improving or run out of time

The Perceptron

Rosenblatt, Frank. "The perceptron: A model for information storage and organization in the brain." Psychological review 65.6 (1958): 386.

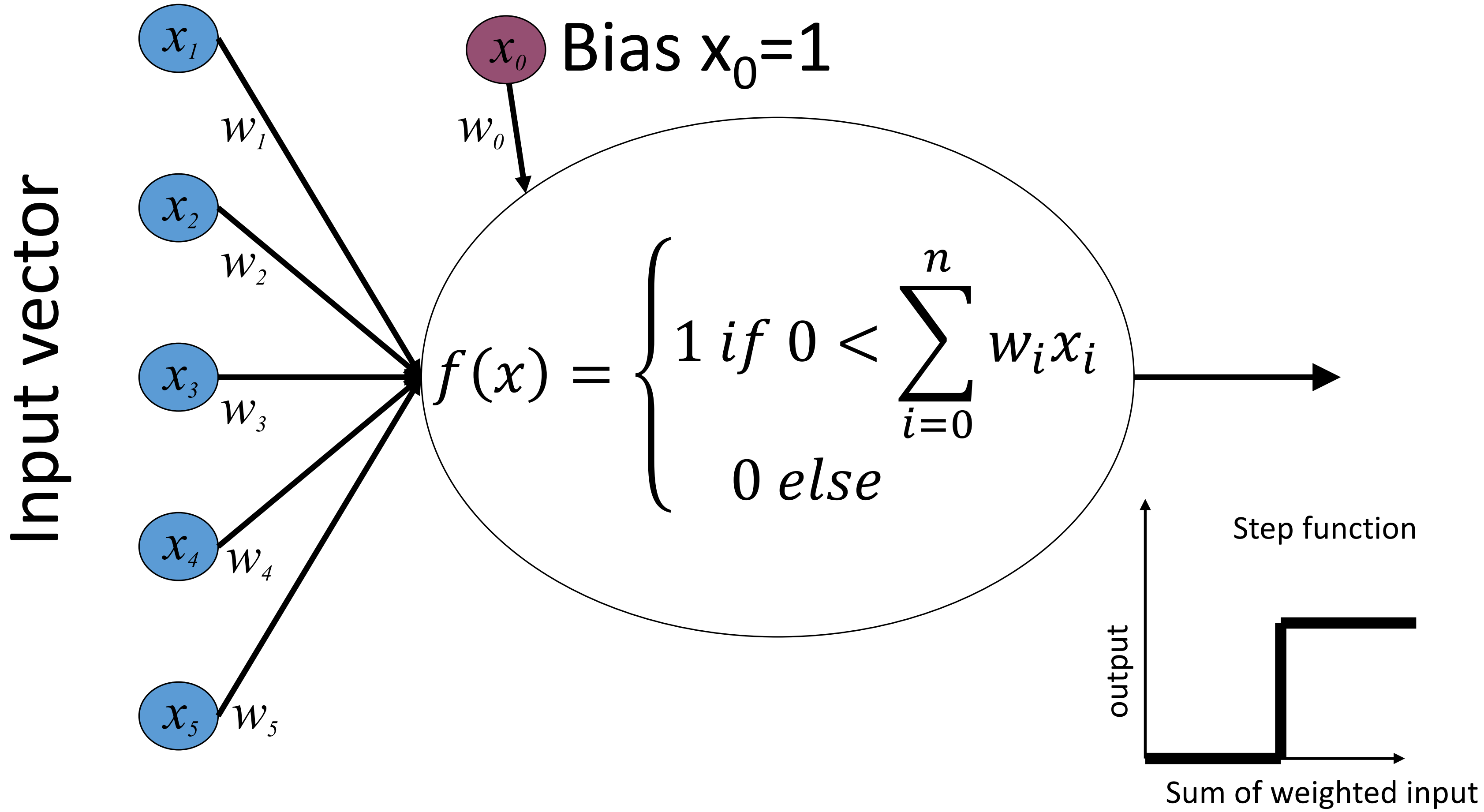
The Perceptron

- Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. Psychological Review, 65(6), 386-408

- The “first wave” in neural networks

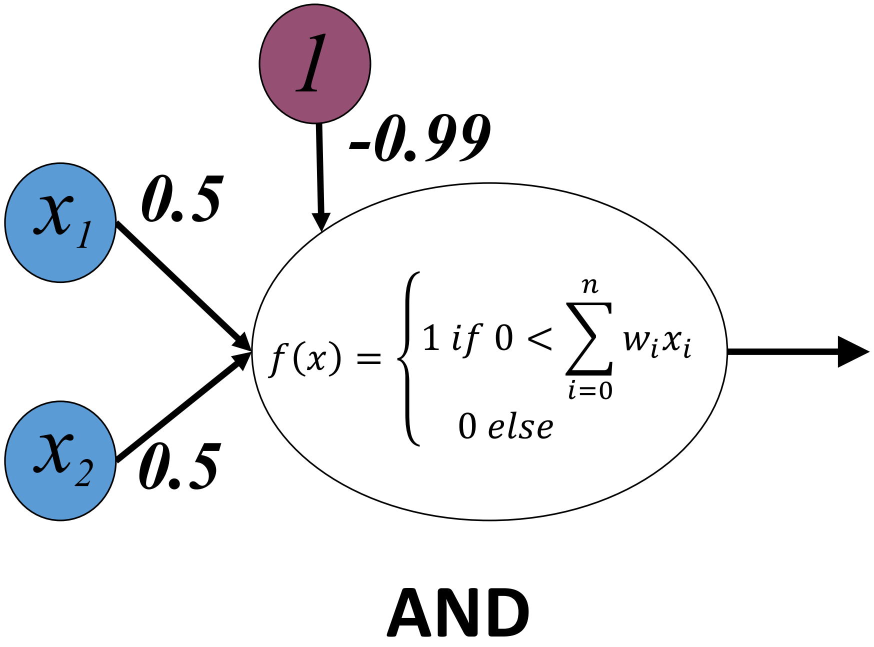
- A linear classifier

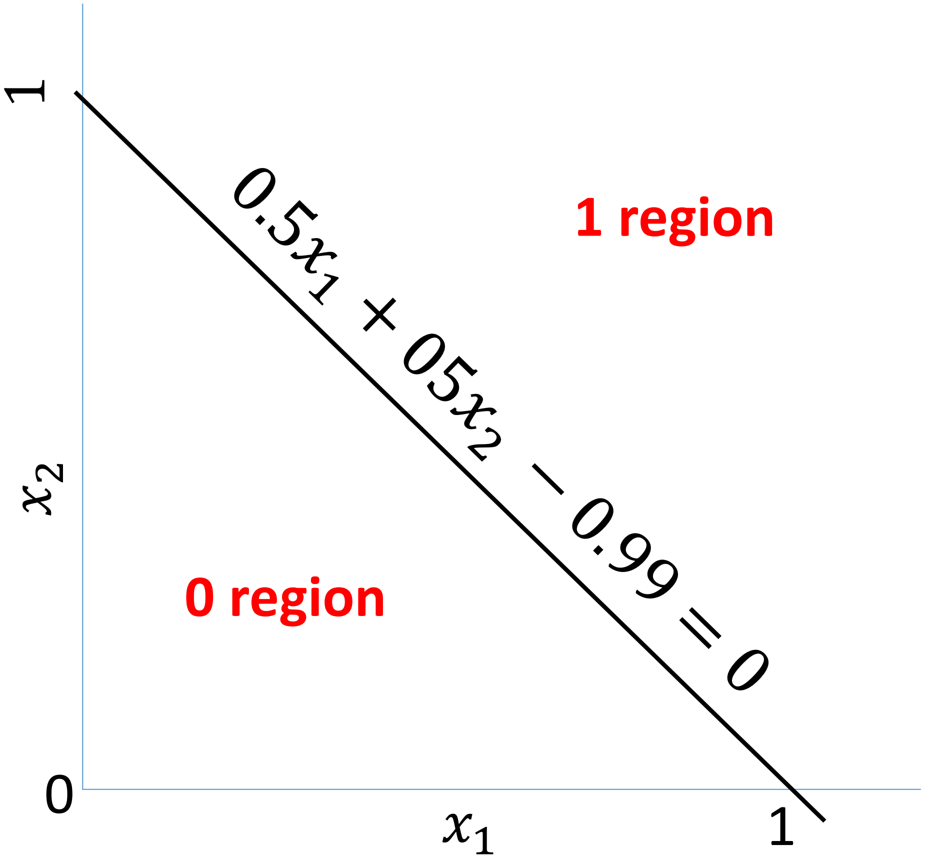
A single perceptron



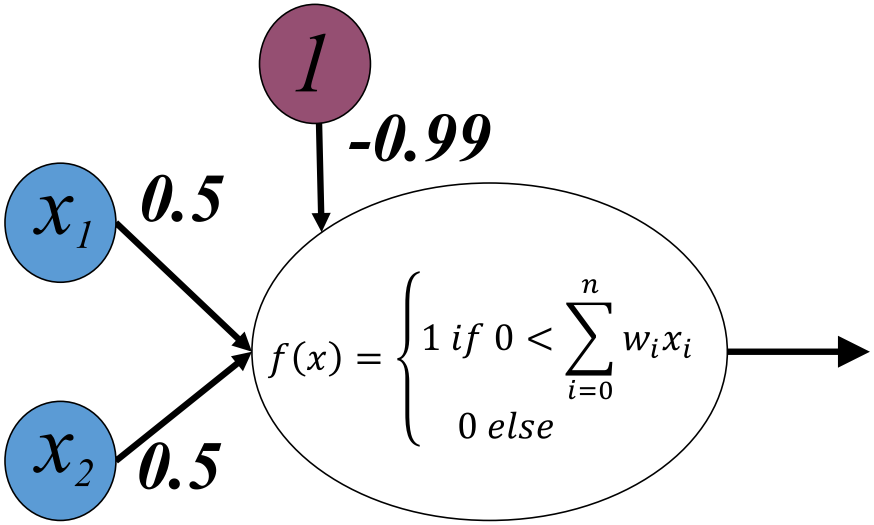
$$f(x) = \begin{cases} 1 \; if \; 0 < \sum_{i=0}^{n} w_i x_i \\ 0 \; else \end{cases}$$

Weights define a hyperplane in the input space


1
-0.99
x_1
0.5
x_2
0.5
$f(x) = \begin{cases} 1 \ if \ 0 < \sum_{i=0}^{n} w_i x_i \\ 0 \ else \end{cases}$
AND

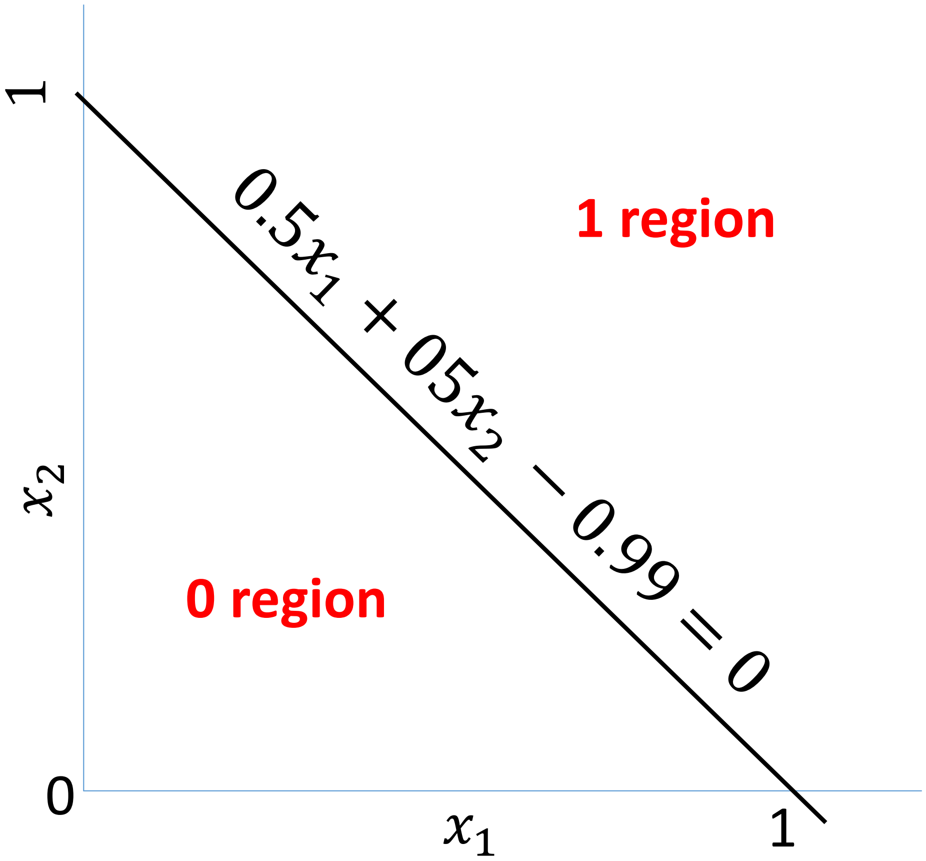


Classifies any (linearly separable) data

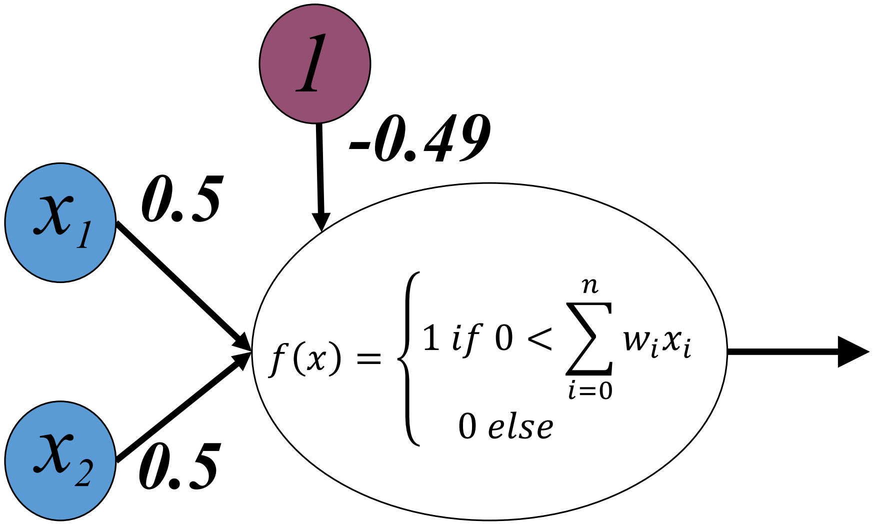


$$f(x) = \begin{cases} 1 \; if \; 0 < \sum_{i=0}^{n} w_i x_i \\ 0 \; else \end{cases}$$

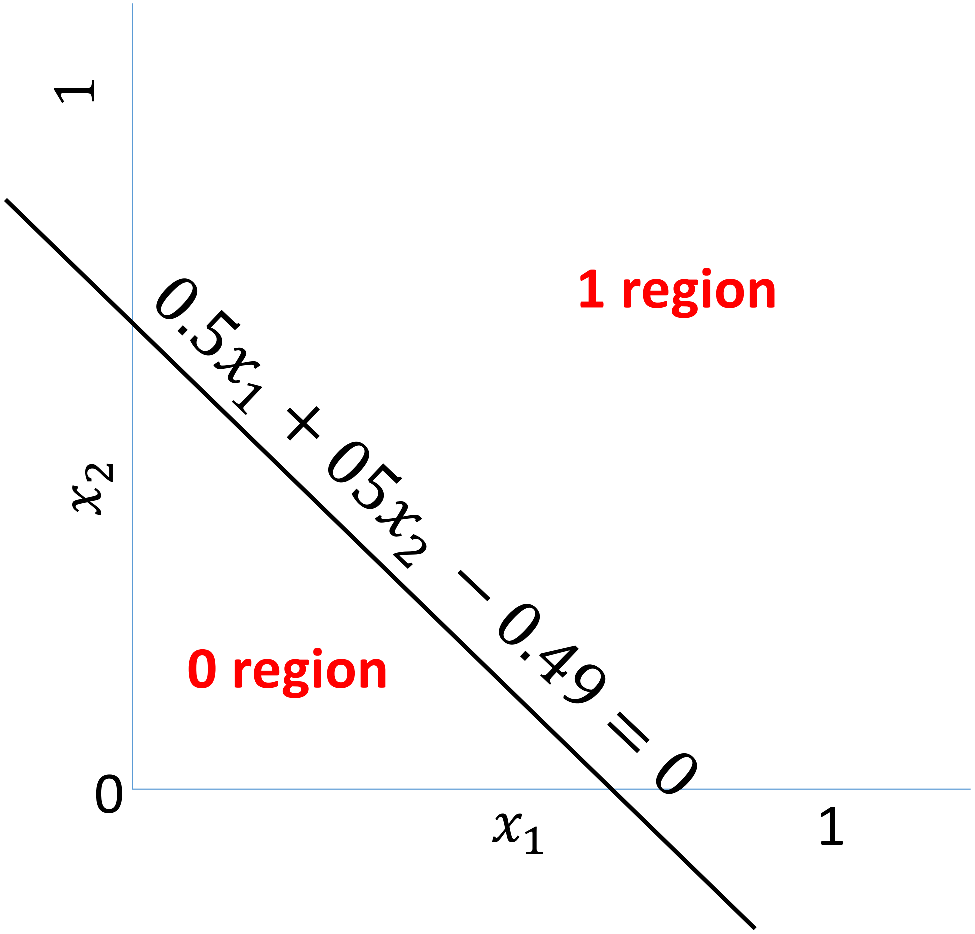
AND



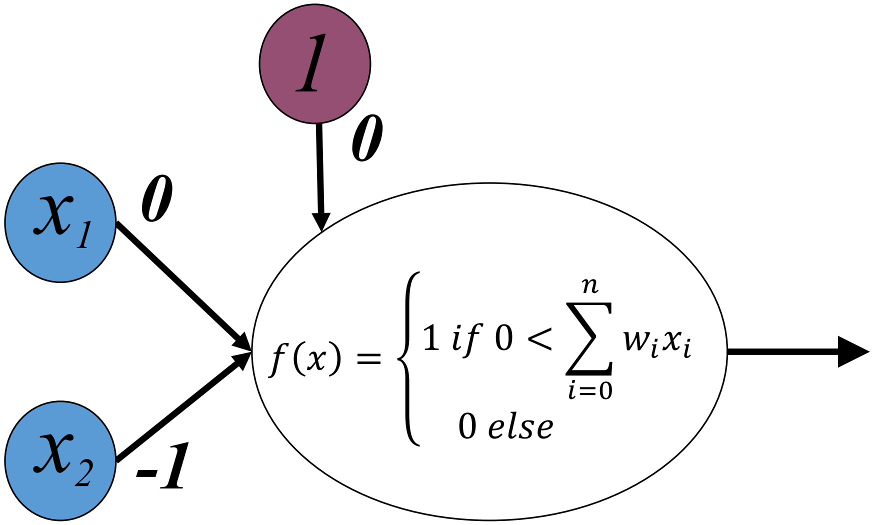
Different logical functions are possible


1
-0.49
x_1
0.5
x_2
0.5
$f(x) = \begin{cases} 1 \; if \; 0 < \sum_{i=0}^{n} w_i x_i \\ 0 \; else \end{cases}$

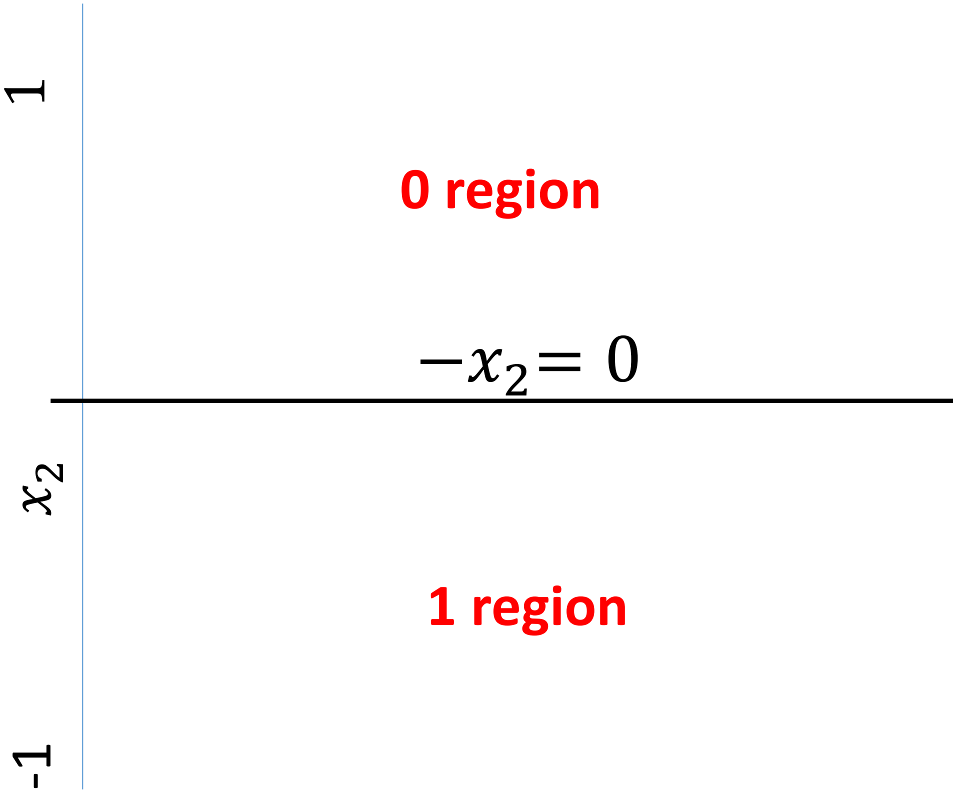
OR



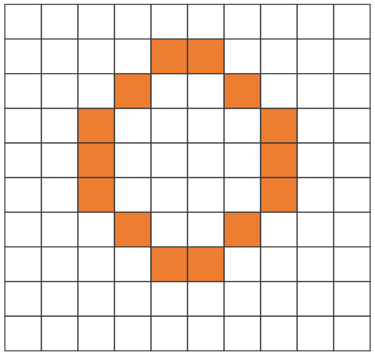
And, Or, Not are easy to define

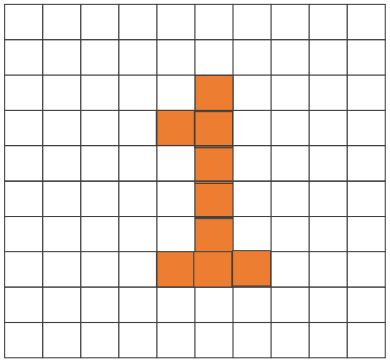

1
0
x_1
0
x_2
-1
$f(x) = \begin{cases} 1 \; if \; 0 < \sum_{i=0}^{n} w_i x_i \\ 0 \; else \end{cases}$

NOT



Classifying image data with a single perceptron



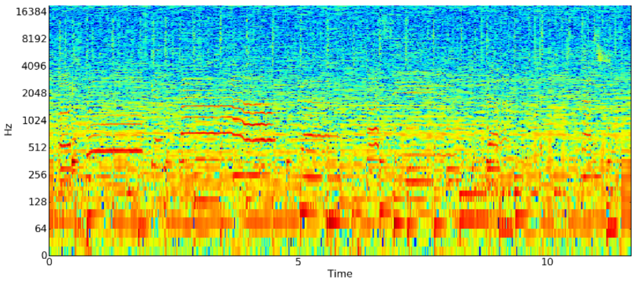


- Each image is an m by n array of pixel values
- Each image is a point in an mn dimensional space
- The right weights make a mn hyperplane to separate the images
- How do we pick those weights?

Classifying song spectrograms

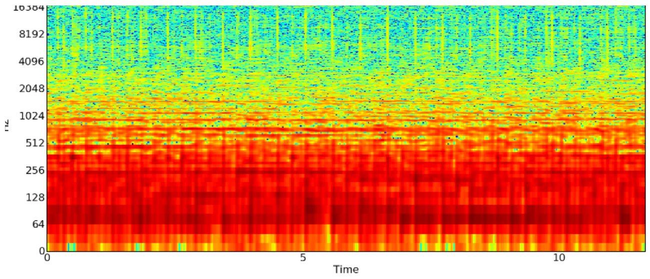
Y = 0 (FLOP SONG)

M = 1024



N = 600

Y = 1 (HIT SONG)



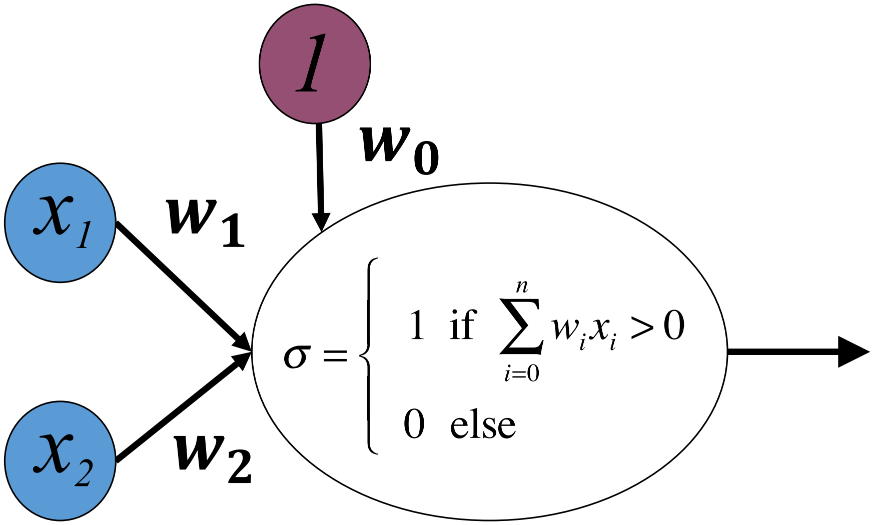
- Each image is an *m* by *n* array of pixel values
- Each image is a point in an mn dimensional space
- The right weights make a mn hyperplane to separate the images
- How do we pick those weights?

Pick model **M**

Right now, it is a single perceptron unit with some small set of weights **w**

Later, we will combine units (thousands of them) and have many weights (millions)

Choosing the types of unit and how they are connected is a challenge.



1
w_0
x_1
w_1
x_2
w_2
$\sigma = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ 0 \text{ else} \end{cases}$

Machine Learning in one slide

1. Pick data **D,** model $\mathbf{M}(\mathbf{w})$ and objective function $\mathbf{J}(\mathbf{D}, \mathbf{w})$

2. Initialize model parameters $\mathbf{w}$ somehow

3. Measure model performance with the objective function $\mathbf{J}(\mathbf{D}, \mathbf{w})$

4. Modify parameters $\mathbf{w}$ somehow, hoping to improve $\mathbf{J}(\mathbf{D}, \mathbf{w})$

5. Repeat 3 and 4 until you stop improving or run out of time

A good objective (loss) function, $J(\mathbf{D}, \mathbf{w})$

Data weights

Required

- $J(\mathbf{D}, \mathbf{w}) \geq 0$
- $J(\mathbf{D}, \mathbf{w})$ decreases as performance improves

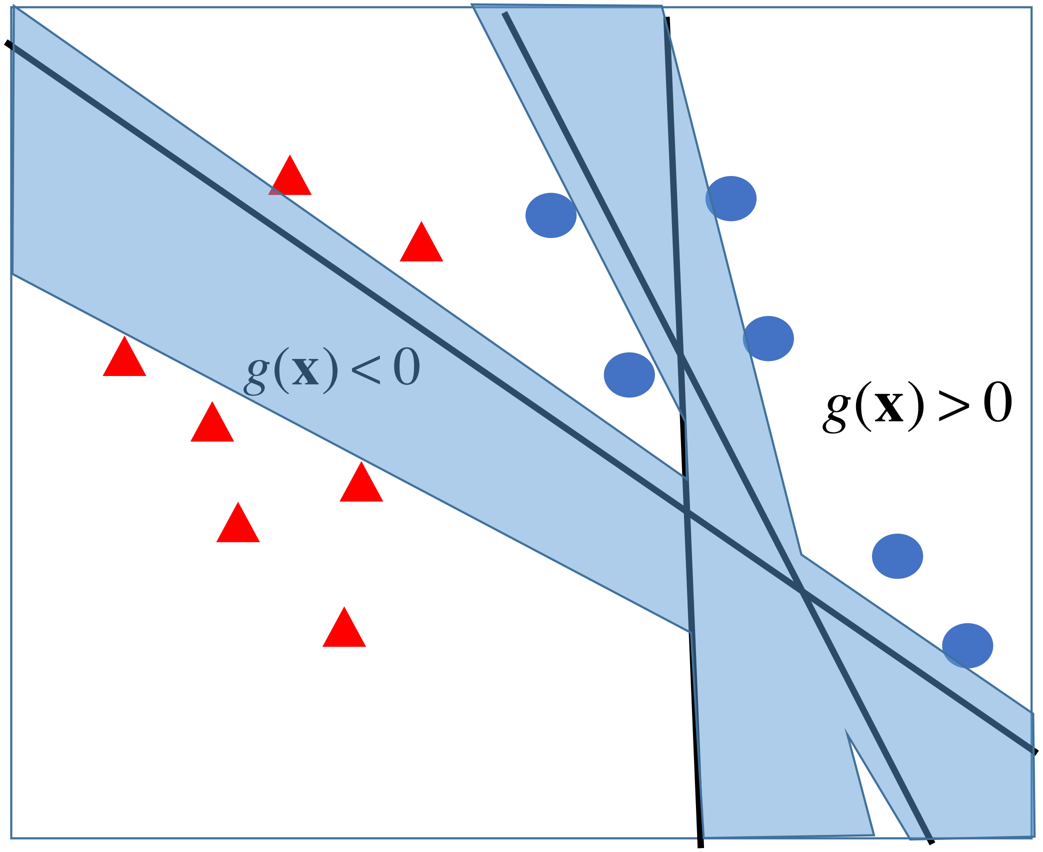
Required for gradient descent

- $J(\mathbf{D}, \mathbf{w})$ is differentiable, with respect to $\mathbf{w}$

Really helpful

- The gradient of J is bounded … $\mathbf{0} < |\nabla J| \ll \infty$

Example objective *J* : sum of squared errors (SSE)



$$g(\mathbf{x}) = w_0 + w_1 x_1 + w_2 x_2 = 0$$

$$h(\mathbf{x}) = \begin{cases} 1 & \text{if } g(\mathbf{x}) > 0 \\ -1 & \text{otherwise} \end{cases}$$

$$SSE = \sum_{i}^{n} (y_i - h(\mathbf{x}_i))^2$$

SSE is same everywhere in the blue

Gradient 0 in the blue region!

Cross Entropy Loss Function

Given: “true” distribution $p = \{p_1, p_2, \dots p_k\}$

estimated distribution $q = \{q_1, q_2, \dots q_k\}$

Define cross entropy between 2 distributions as

$$H(p,q) = -\sum_{i=1}^{k} p_i log q_i$$

Over a set of data points, **D**

$$J(\mathbf{D}) = \frac{\mathbf{1}}{|\mathbf{D}|} \sum_{d \in \mathbf{D}} H_d\,(p,q)$$

Machine Learning in one slide

1. Pick data **D,** model $\mathbf{M}(\mathbf{w})$ and objective function $\mathbf{J}(\mathbf{D}, \mathbf{w})$

2. Initialize model parameters $\mathbf{w}$ somehow

3. Measure model performance with the objective function $\mathbf{J}(\mathbf{D}, \mathbf{w})$

4. Modify parameters $\mathbf{w}$ somehow, hoping to improve $\mathbf{J}(\mathbf{D}, \mathbf{w})$

5. Repeat 3 and 4 until you stop improving or run out of time

Gradient Descent in one slide

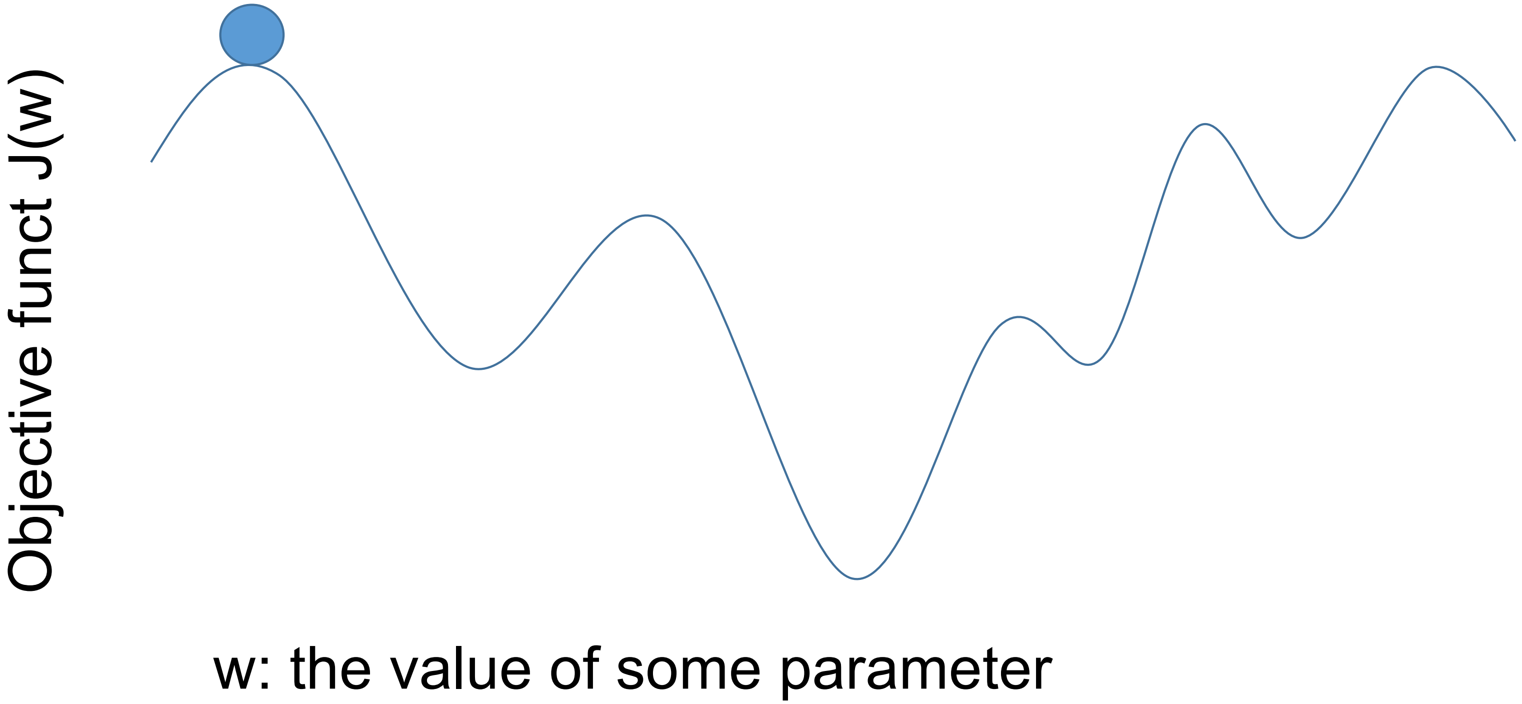
1. Measure how the the objective function changes when we change the $\mathrm{current}$ parameters $\boldsymbol{w}$ slightly (measure the gradient with respect to the weights).

2. Pick the next set of parameters to be close to the current set, but in the direction that most changes the objection function for the better (follow the gradient)

3. Repeat

Gradient Descent: Promises & Caveats

- Much faster than guessing new parameters randomly
- Finds the global optimum only if the objective function is convex



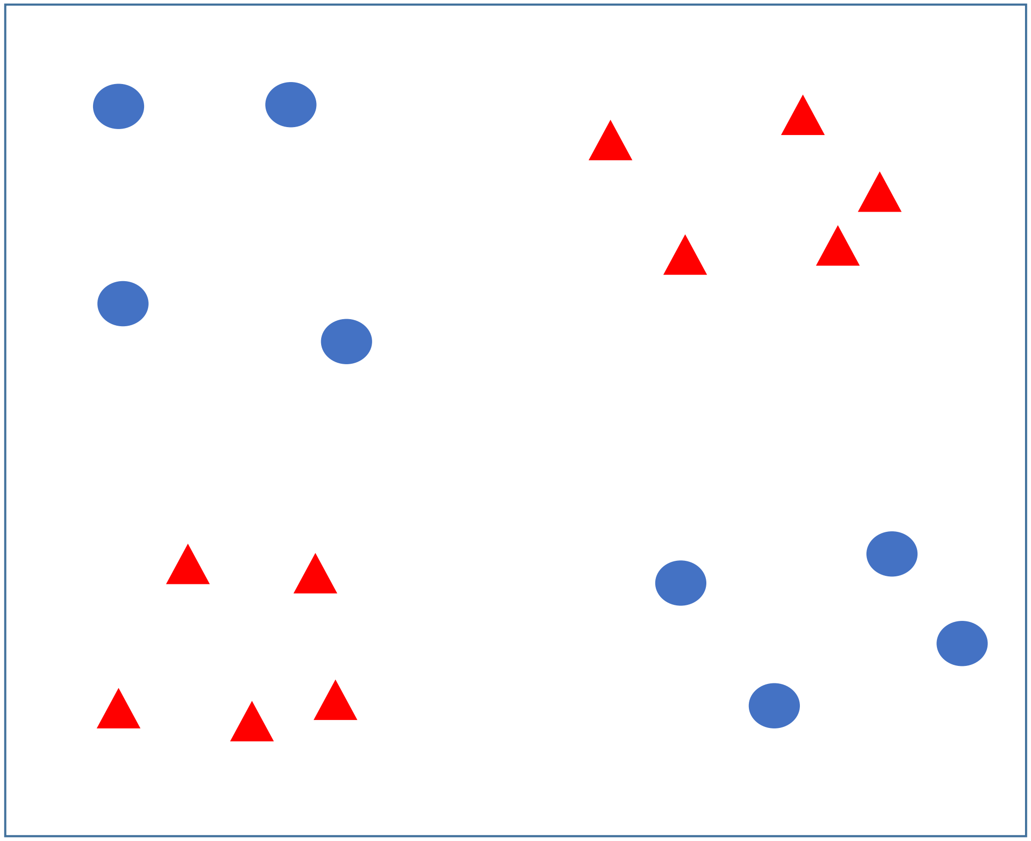
Stochastic, Batch, Mini-Batch Descent

- In **batch gradient descent**, the objective function J is a function of both the parameters and ALL training samples, summing the total error

- In **stochastic gradient descent**, J is a function of the parameters and a different single random training sample at each iteration

- In **mini-batch gradient descent**, random subsets of the data (e.g. 100 examples) are used at each step in the iteration. This is a common approach today.

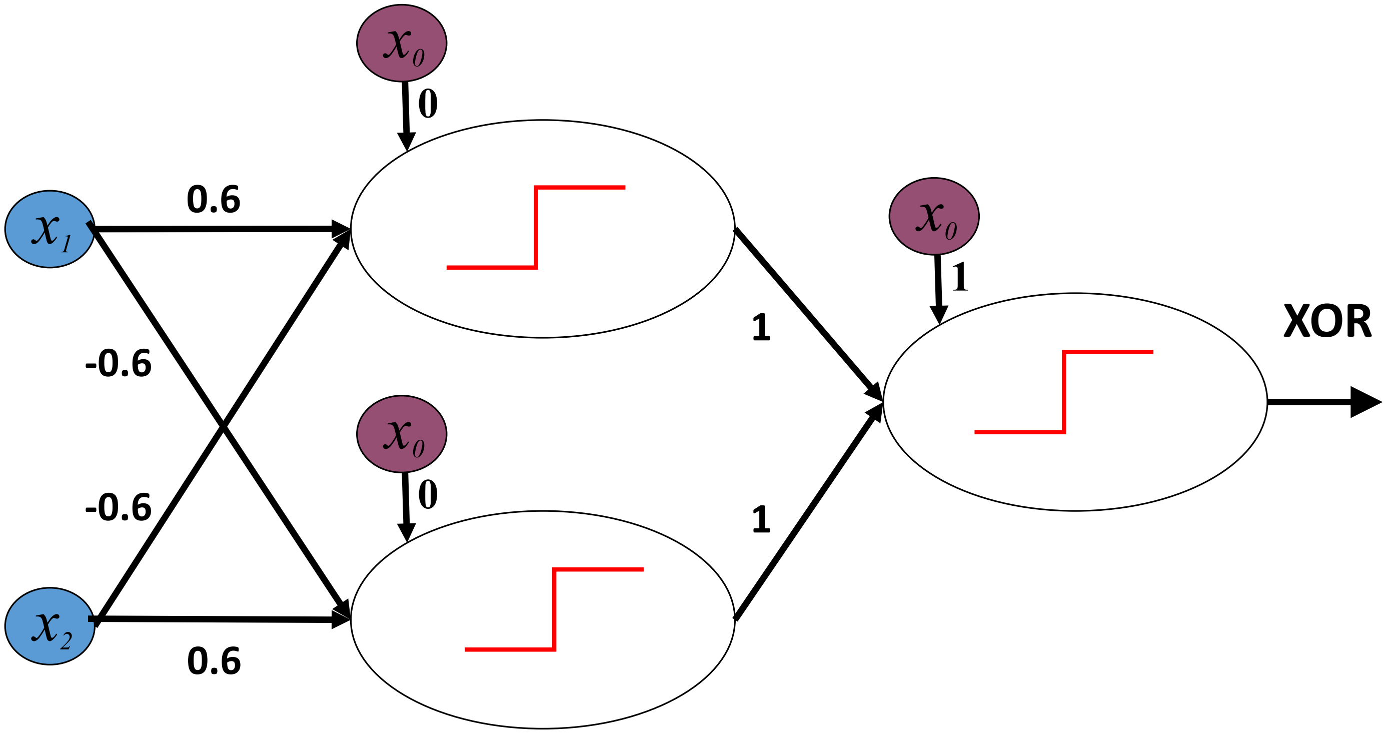
One perceptron: Only linear decisions



This is XOR.

It can’t learn XOR.

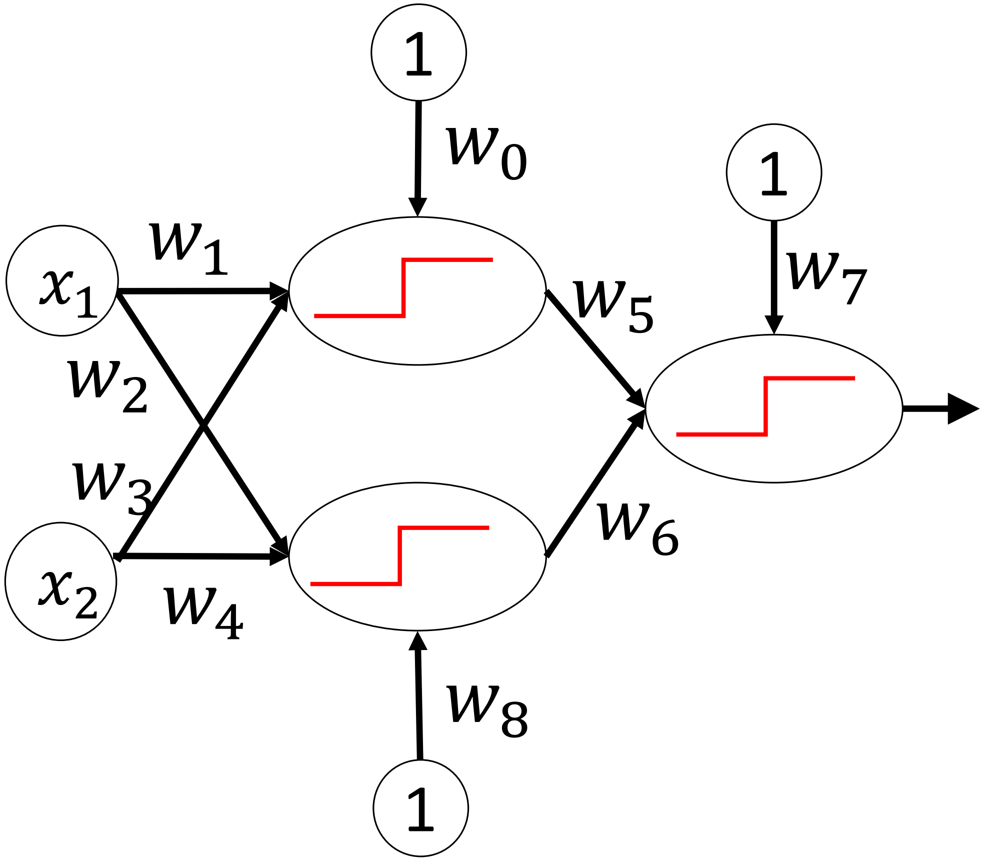
Combining perceptrons can make any Boolean function



…if you can set the weights & connections right

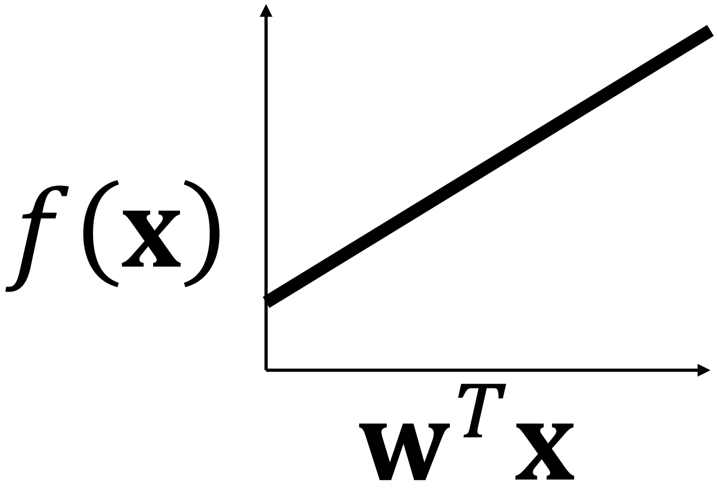
Problem with a step function: Assignment of error

- Stymies multi-layer weight learning
- Limits us to a single layer of units
- Thus, only linear functions
- You can hand-wire XOR perceptrons, but the sytem can’t learn XOR with perceptrons



Linear Units & Delta Rule

Solution: Remove the step function



$$f(\mathbf{x}) = \sum_{i=0}^{n} w_i x_i = \mathbf{w}^T\mathbf{x}$$

Measuring error for linear units

- Output Function

$$f(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$$

- Objective Function:

Correct output

Model output

$$J(\mathbf{w}) = \frac{1}{2} \sum_{<\mathbf{x},y> \in D} (y - \mathbf{w}^T\mathbf{x})^{\mathbf{2}}$$

Dataset

Gradient Decsent Rule

- To figure out how to change each weight to best reduce error we take the derivative with respect to that weight

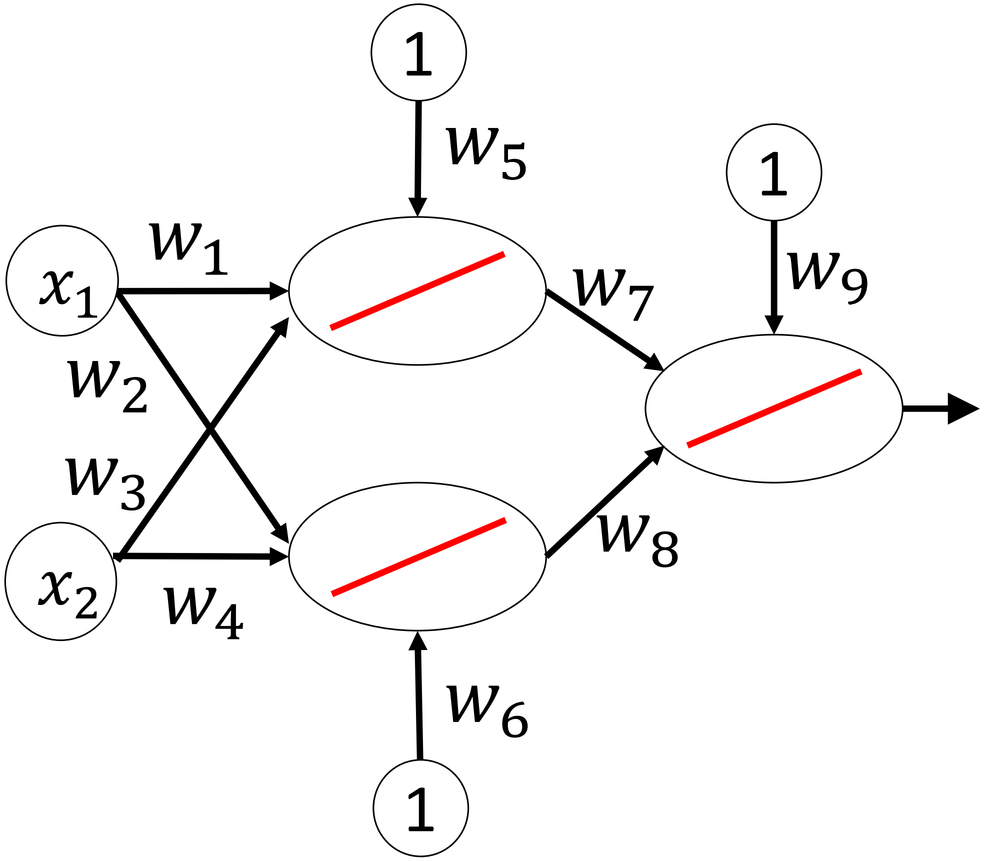
$$\frac{\partial J}{\partial w_i} \equiv \frac{1}{2} \frac{\partial}{\partial w_i} \sum_{<\mathbf{x},y> \in D} (y - \mathbf{w}^T \mathbf{x})^{\mathbf{2}}$$

- …and then do some math to get this update rule

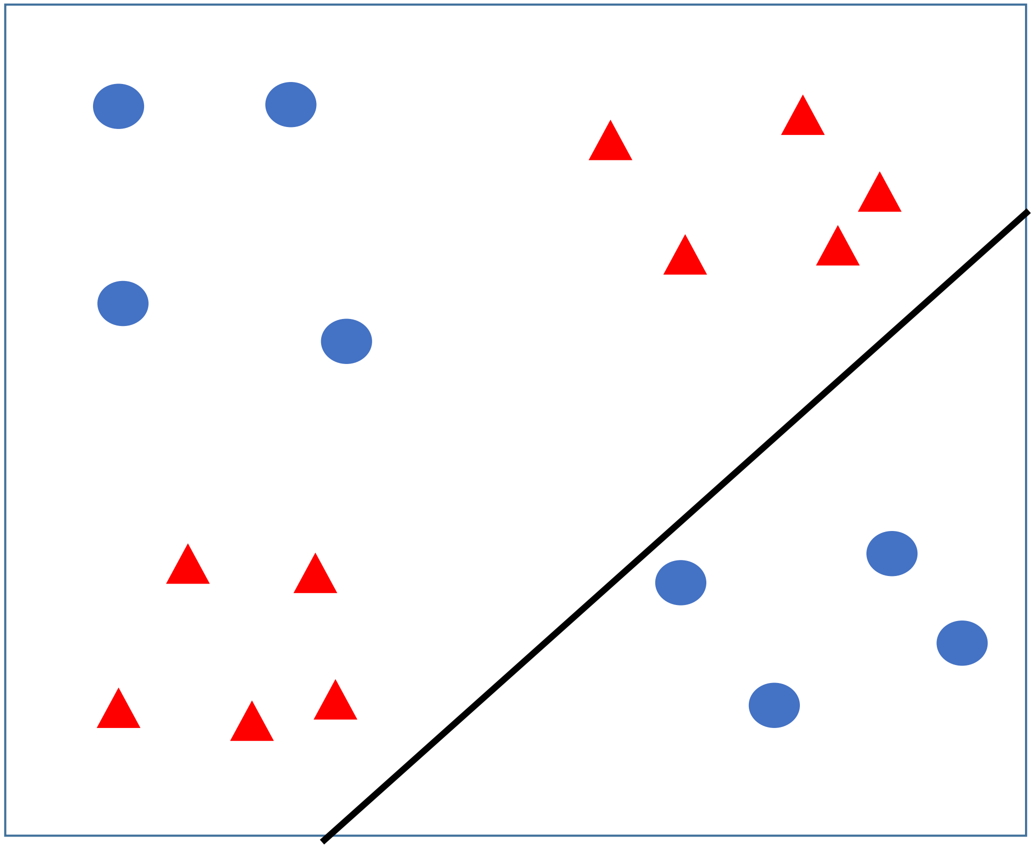
$$w_i \leftarrow w_i + \eta \textstyle\sum_{<x,y> \in D} (y - w^T x)\, x_i$$

Better & worse than a perceptron

- All changes in input result in changed output
- This gives us a gradient everywhere
- We can learn multiple layers of weights.
- Combining linear functions only gives you linear functions
- you can't represent XOR



Many linear units: Only linear decisions



This is XOR.

A multilayer perceptron with linear units CANNOT learn XOR

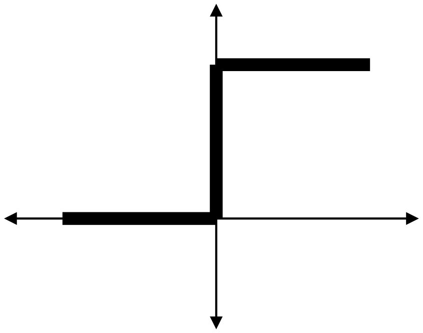
The Sigmoid Unit

Rumelhart, David E., James L. McClelland, and PDP Research Group. Parallel distributed processing. Vol. 1. Cambridge, MA, USA:: MIT press, 1987.

Sigmoid (aka Logistic) function: best of both

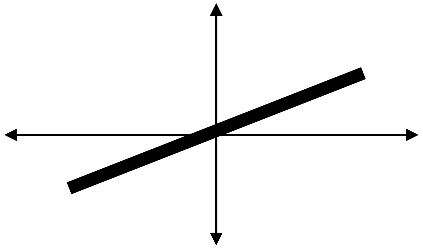
- Perceptron

$$f(x) = \begin{cases} 1\ if\ 0 < \sum_{i=0}^{n} w_i x_i \\ 0\ else \end{cases}$$



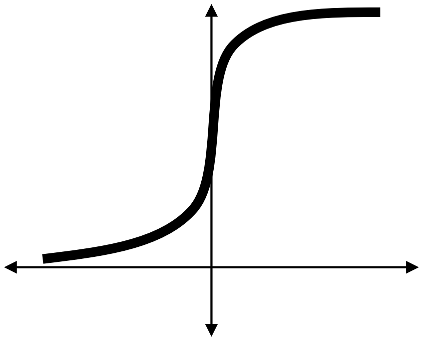
- Linear

$$f(x) = \mathbf{w}^T \mathbf{x} = \sum_{i=0}^{n} w_i x_i$$



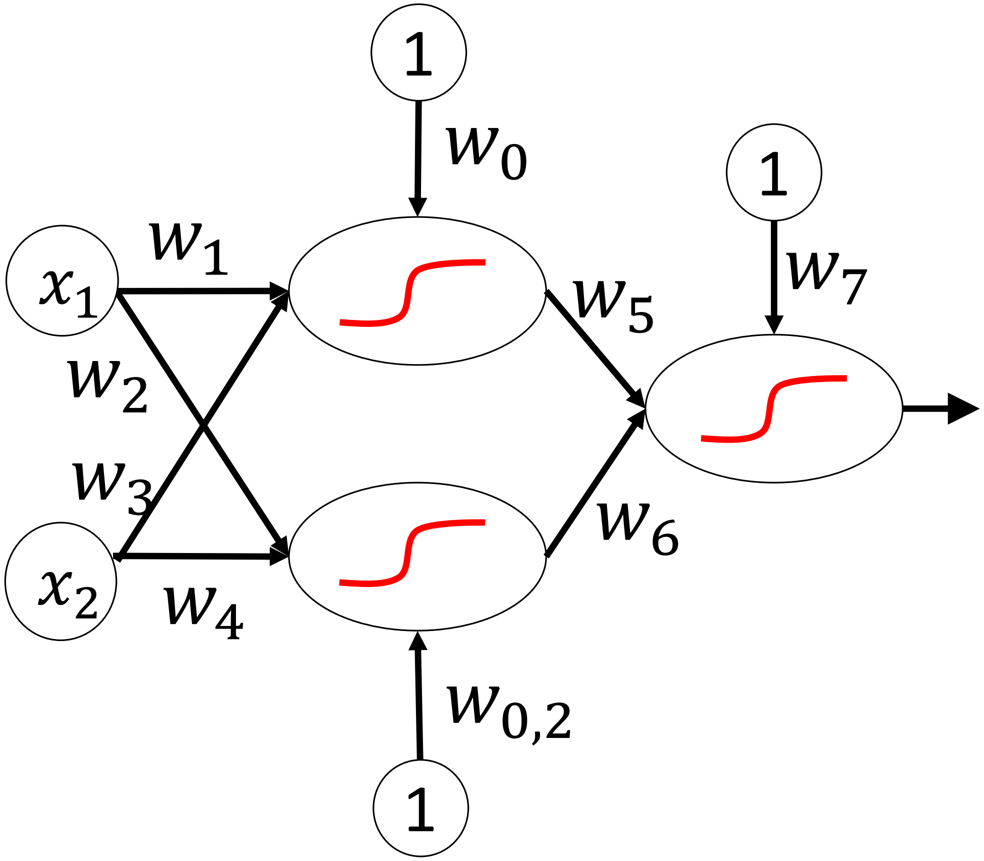
- Sigmoid

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



A network of sigmoid units

- Small changes in input result in output
- This gives us a gradient everywhere
- We can learn multiple layers of weights.
- Combining layers gives non-linear functions



Sigmoid changes (almost) everything

Easy to differentiate

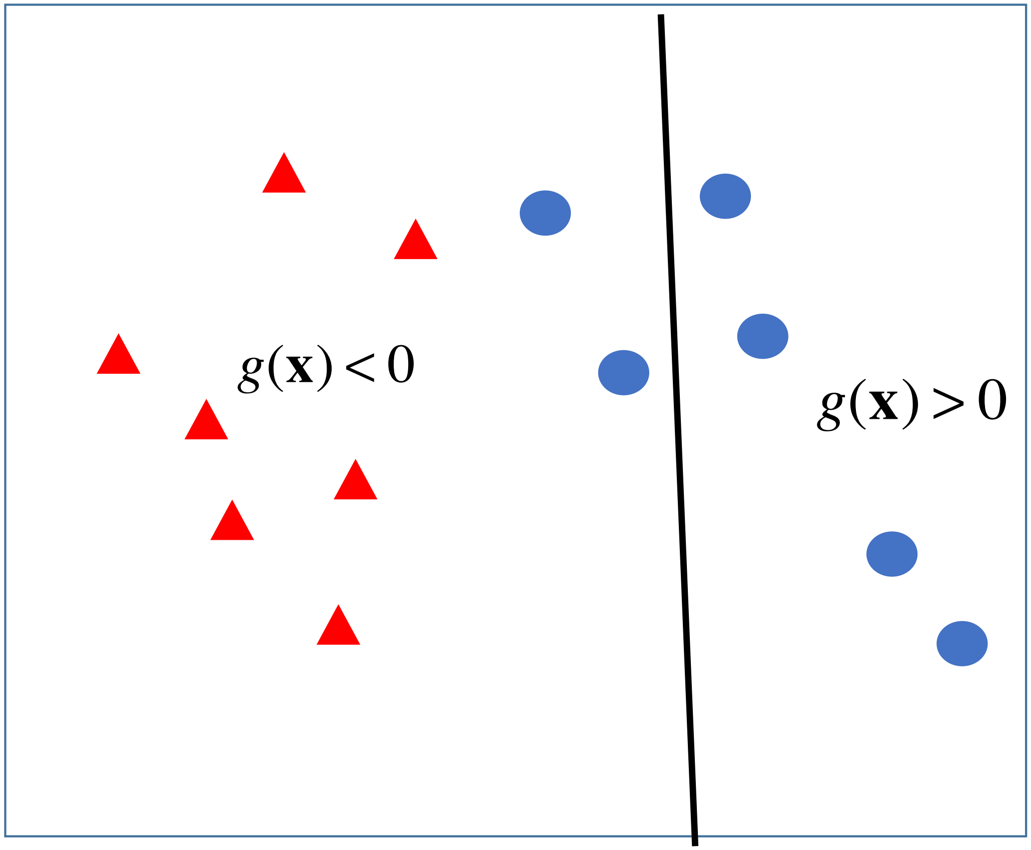
$\sigma'\left(\mathbf{w}^{\boldsymbol{T}}\mathbf{x}\right) = \sigma\left(\mathbf{w}^{\boldsymbol{T}}\mathbf{x}\right)$(1- $\sigma(\mathbf{w}^{\boldsymbol{T}}\mathbf{x})$)

Gradient everywhere

This allows backpropagation of the gradient through multiple layers

Nonlinearity allows arbitrary nonlinear functions to be built by using multiple layers.

Example objective *J* : sum of squared errors

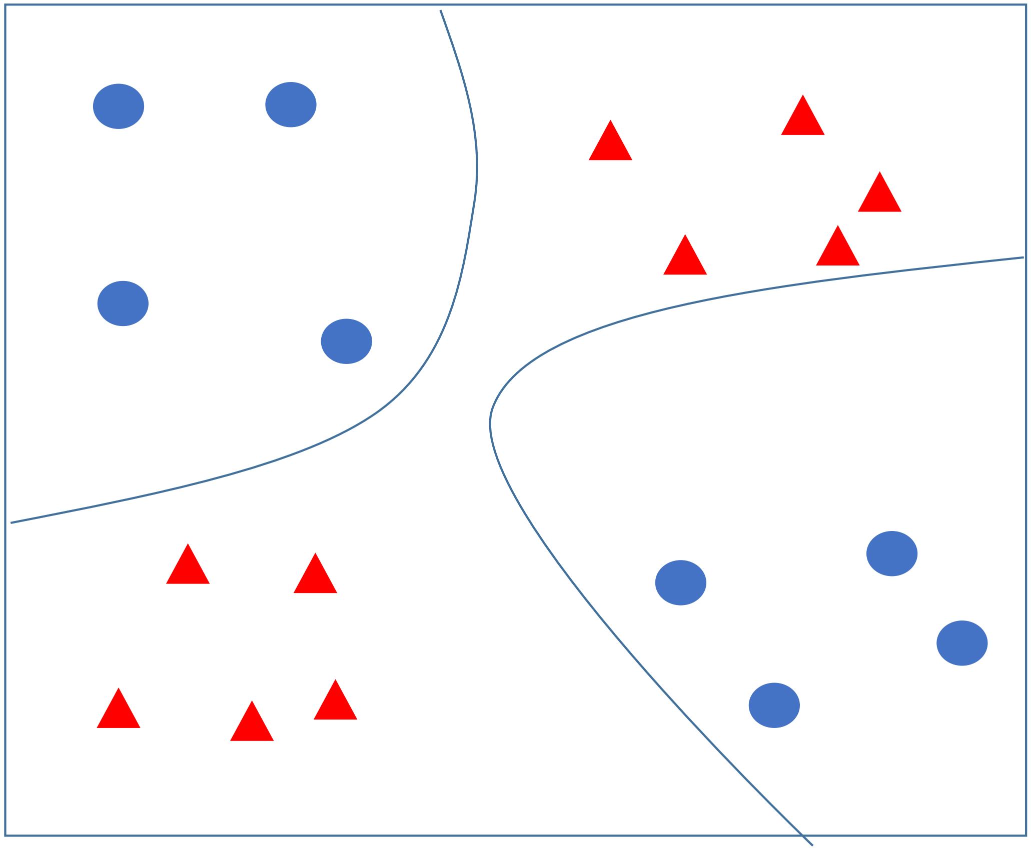


$$h(x) = f(x) = \frac{1}{1 + e^{-(\mathbf{w}^T \mathbf{x})}}$$

$$SSE = \sum_{i}^{n} (y_i - h(\mathbf{x}_i))^2$$

Gradient non-zero everywhere!

Multilayer Perceptron with sigmoid units



This is XOR.

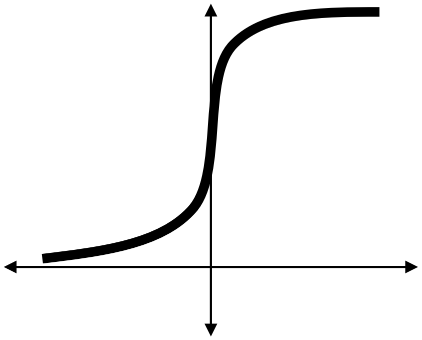
A multilayer perceptron with sigmoid units CAN learn XOR…or any other arbitrary Boolean function.

The promise of many layers

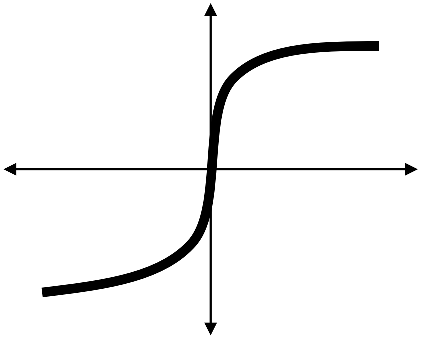
- Each layer learns an abstraction of its input representation (we hope)
-
- As we go up the layers, representations become increasingly abstract
- The hope is that the intermediate abstractions facilitate learning functions that require non-local connections in the input space (recognizing rotated & translated digits in images, for example)
- Modern neural networks are up to 100 layers deep

TanH: A shifted sigmoid

- Sigmoid $$f(x) = \frac{1}{1 + e^{-(\mathbf{w}^T\mathbf{x})}}$$

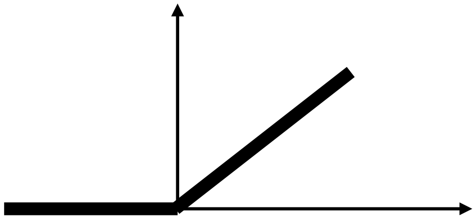


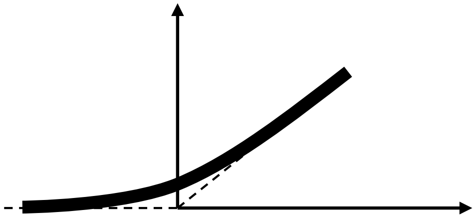
- TanH $$f(x) = \frac{2}{1 + e^{-2(\mathbf{w}^T\mathbf{x})}} - 1$$



Rectified Linear Unit (ReLU) & Soft Plus :

- ReLU $f(x) = \max(0, \mathbf{w}^T\mathbf{x})$





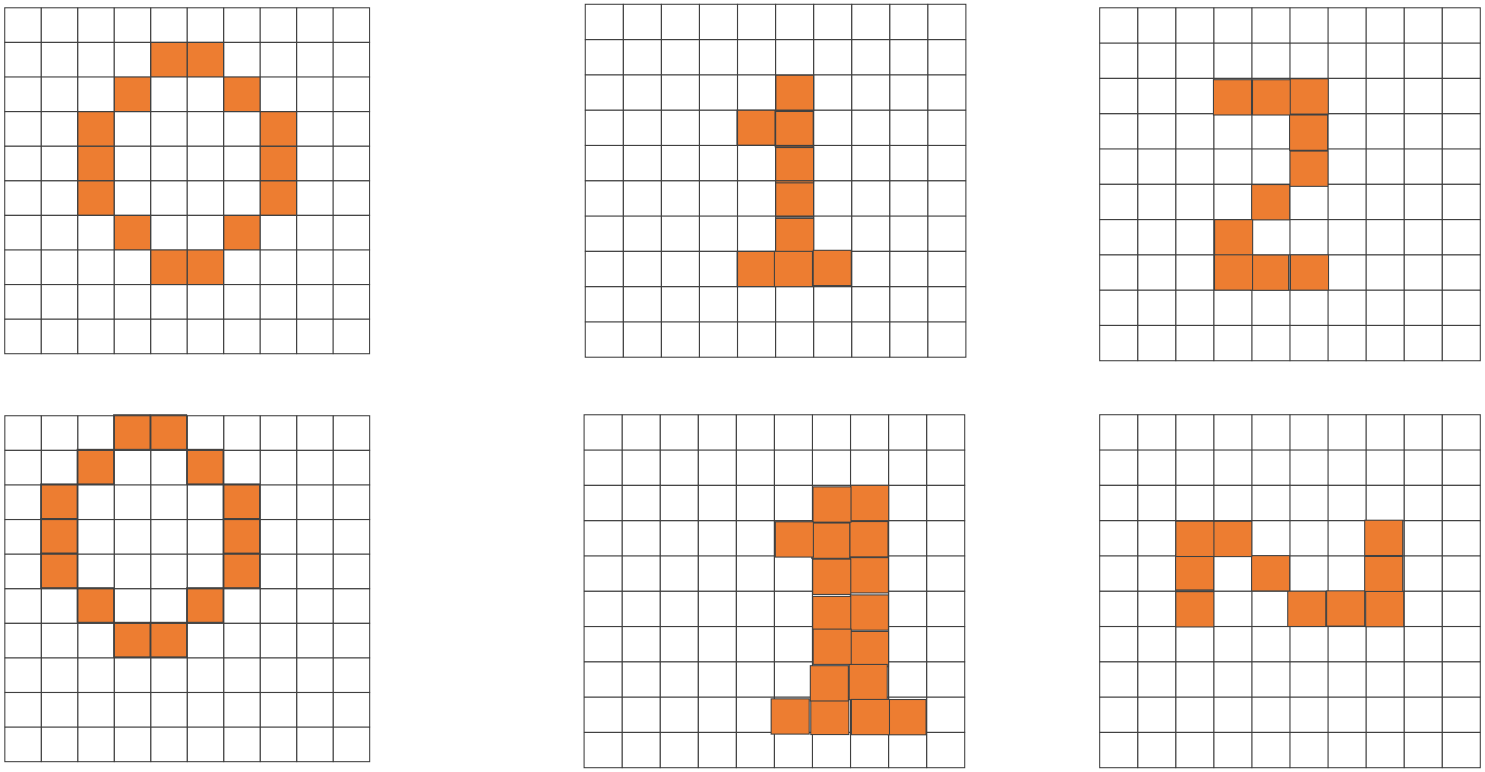
- Soft Plus $f(x) = \ln(1 + e^{\mathbf{w}^T\mathbf{x}})$

- Both can be combined in layers to make non-linear functions

Design choices

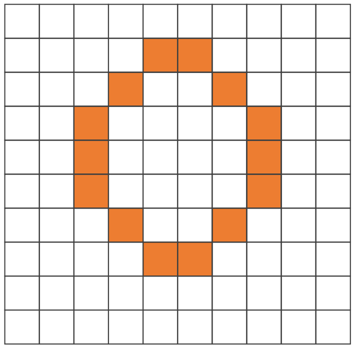
- Define the function you want to learn
- Determine an encoding for the data
- Pick a network architecture
  - Number of layers (between 3 and 100)
  - Activation functions function (tanh,ReLU, linear)
  - Select how units connect within and between layers
- Pick a gradient descent algorithm
- Pick regularization approach (e.g. dropout)

Classifying images of digits



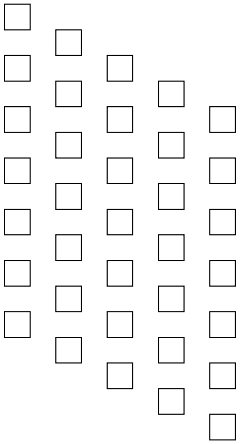
One possibility

INPUT LAYER



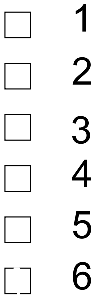
One input
per pixel

HIDDEN LAYER



One hidden node per
potential shifted image
(ReLU)

OUTPUT LAYER



One output node
per category
(Sigmoid)

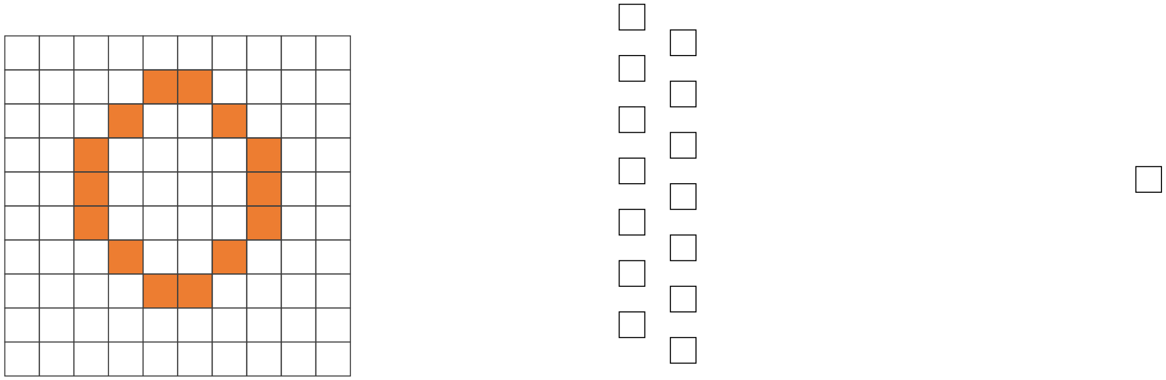
Each node is connected to EVERY node in the prior layer
(it is just too many lines to draw)

Another possibility

INPUT LAYER

HIDDEN LAYER

OUTPUT LAYER



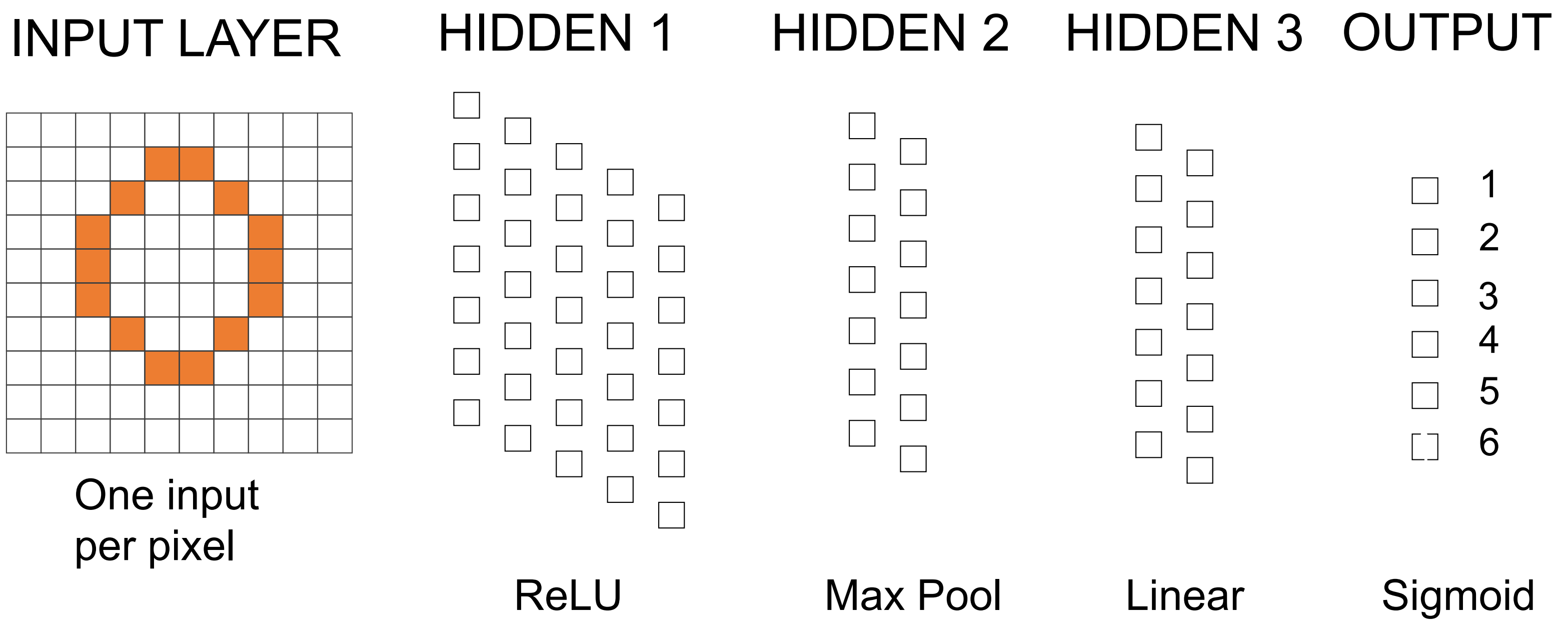
One input
per pixel

small number of nodes:
1 per important feature
(TanH)

A single linear node with
(scaled) output

Each node is connected to EVERY node in the prior layer
(it is just too many lines to draw)

Another possibility



INPUT LAYER

HIDDEN 1

HIDDEN 2

HIDDEN 3

OUTPUT

One input
per pixel

ReLU

Max Pool

Linear

Sigmoid

HUGE DESIGN SPACE!

Convolutional networks

LeCun, Yann, and Yoshua Bengio. "Convolutional networks for images, speech, and time series." *The handbook of brain theory and neural networks*3361.10 (1995): 1995.

How big is that image?

500

1000

How many weights in a fully connected net?

Input Layer

M = 500

N = 1000

$(500000 \cdot 100)$ Unique weights

Hidden Layer

100 Nodes

$(100 \cdot 50)$ Unique weights

Hidden Layer

50 Nodes

$(50 \cdot 2)$ Unique Weights

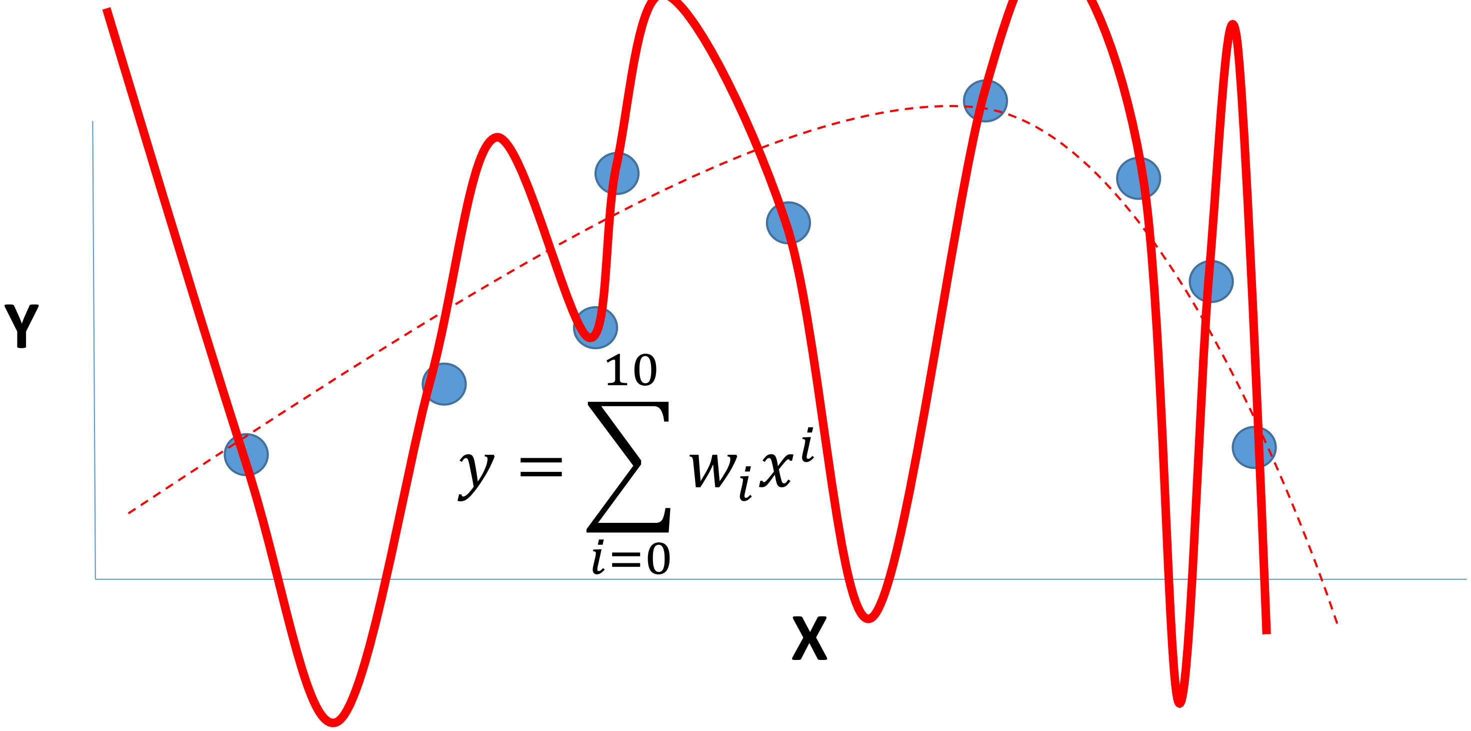
Output Layer

2 Nodes

$$50{,}000{,}000 + 5{,}000 + 100 = 50{,}005{,}100 \text{ weights}$$

Fitting & Hypothesis space

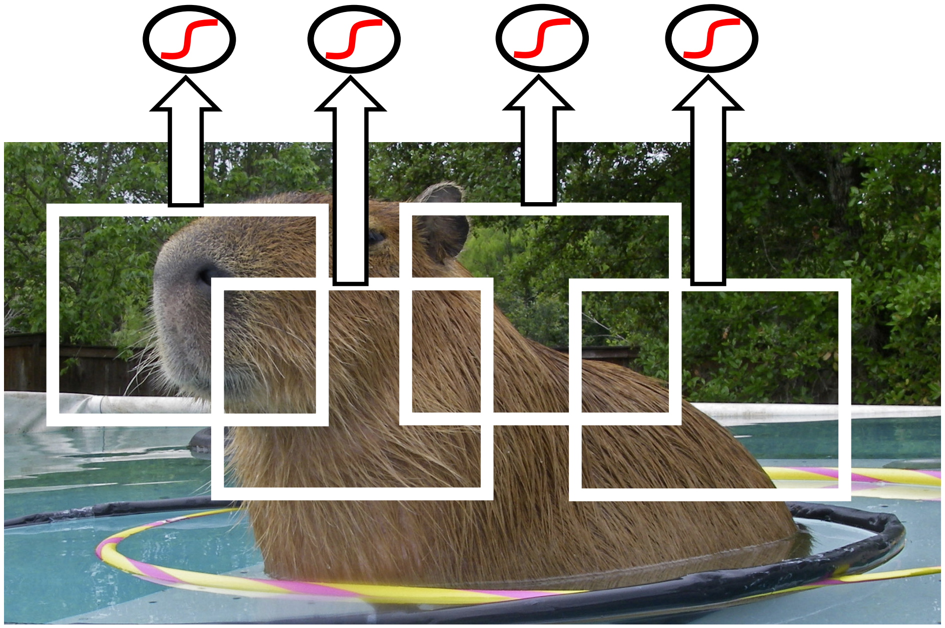
If a model’s hypothesis space is too big, it can learn a crazy, overly specific function (overfitting)



$$y = \sum_{i=0}^{10} w_i x^i$$

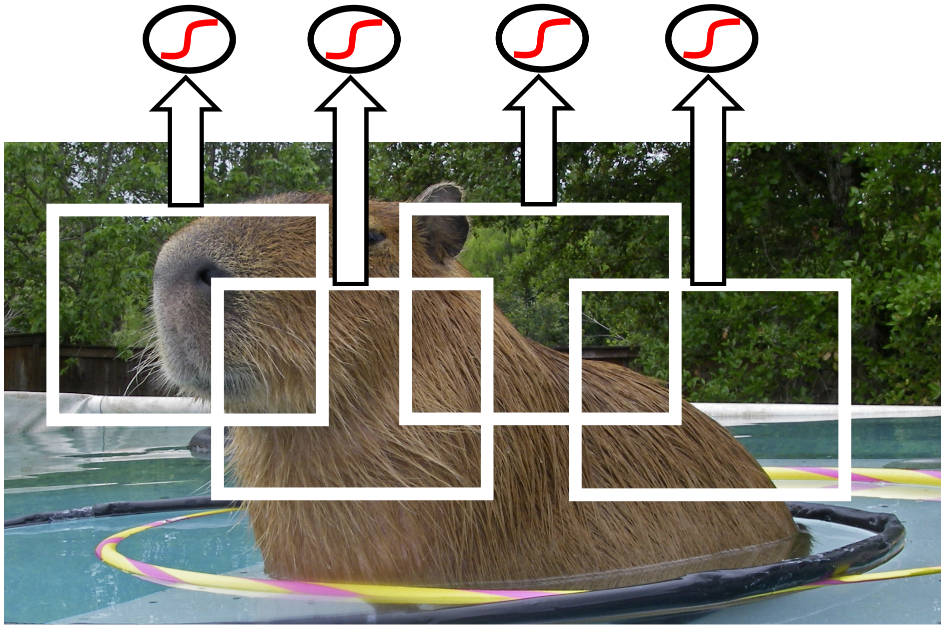
Small Fixed Windows (filter size/receptive field)

- If important features fall within a bounded size region, we can bound the receptive field of each unit to that size.
- This greatly reduces the number of weights.

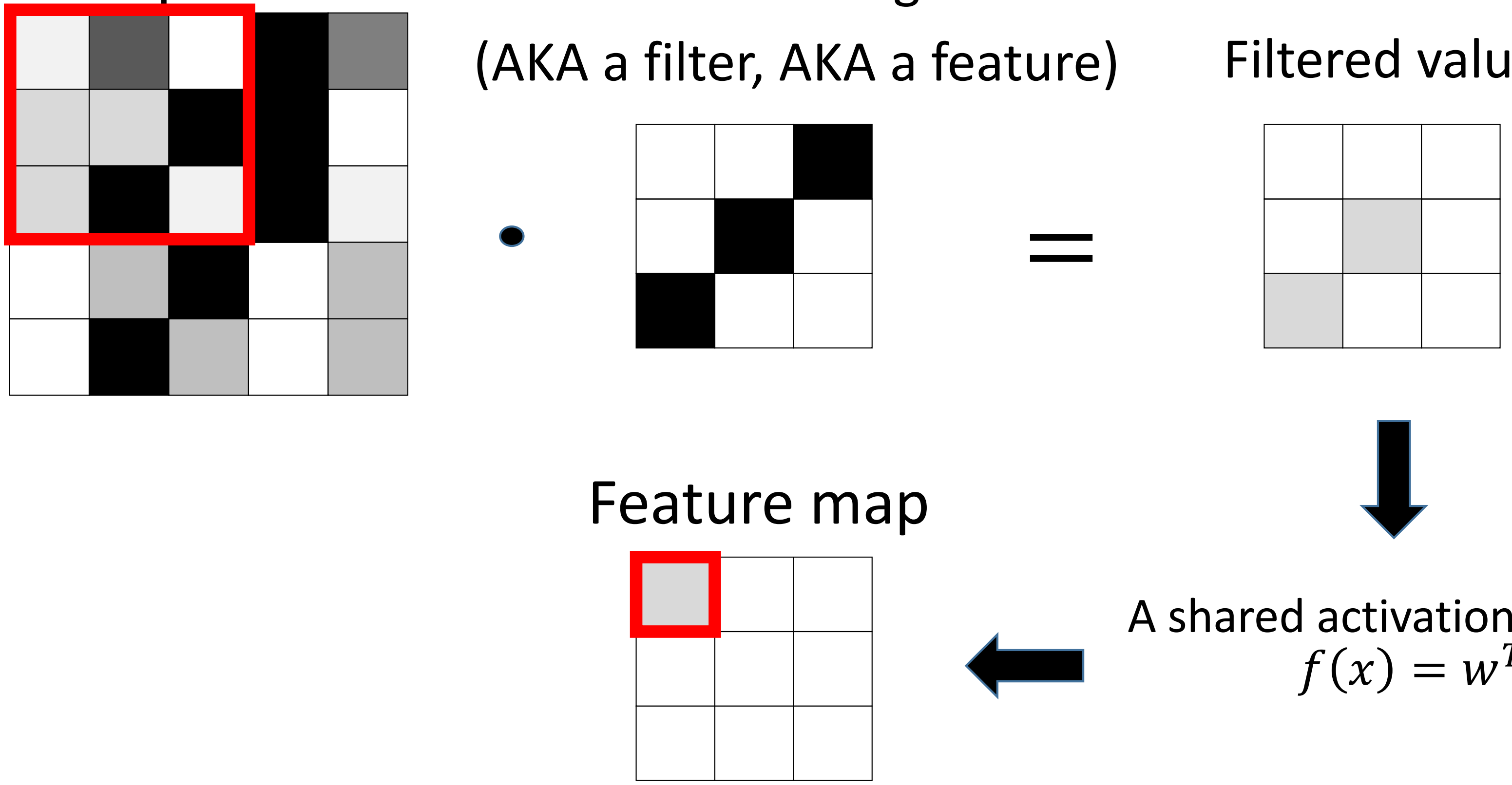


Shared weights

- If a feature is good to find in one region, it may be good to find in other regions.
- Units look for the same feature if they share weights.
- A set of units that share weights is a feature map.



Building that feature map



Input X

The shared weights W

(AKA a filter, AKA a feature)

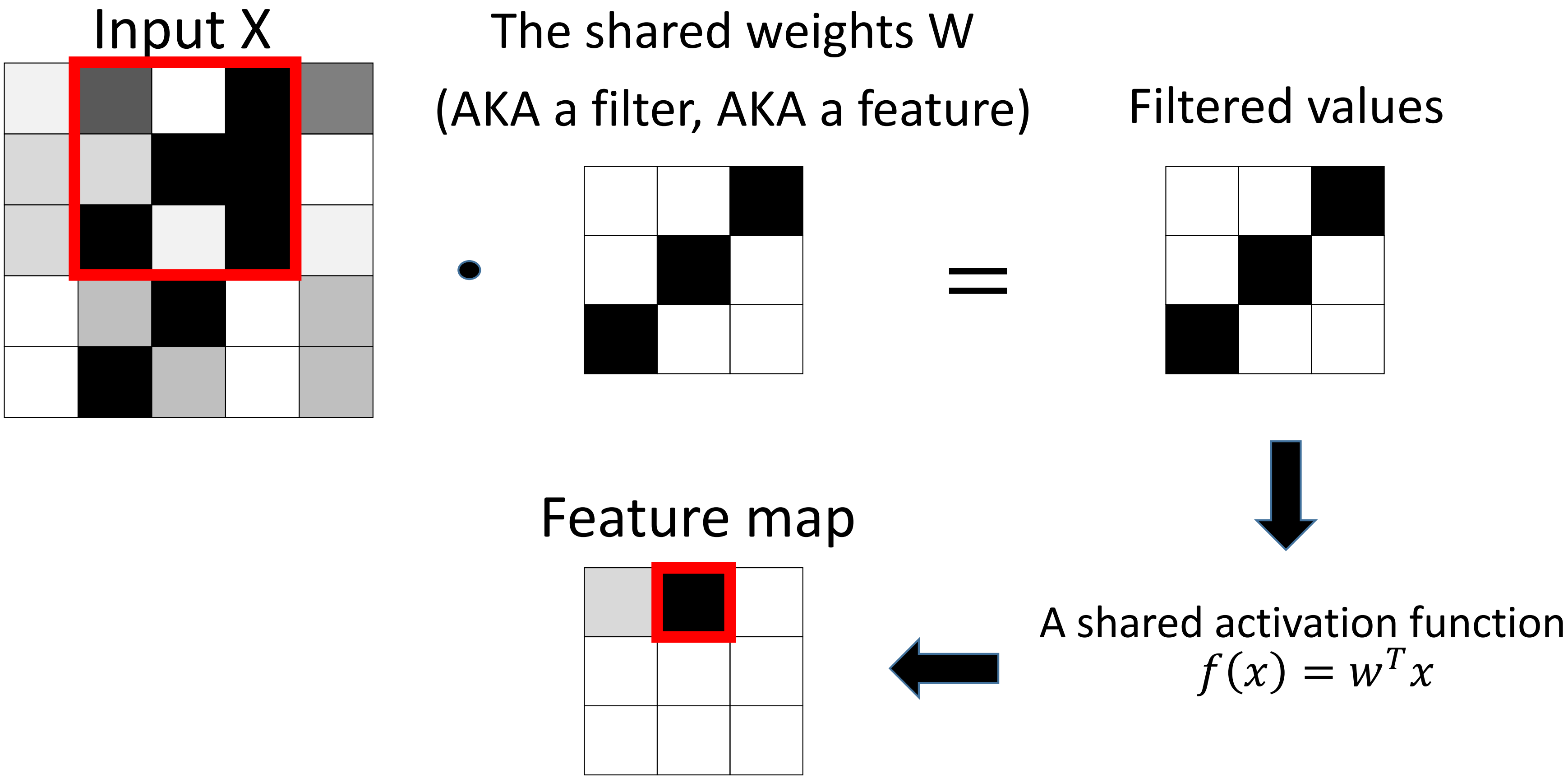
Filtered values

Feature map

A shared activation function

$f(x) = w^T x$

Building that feature map



Input X

The shared weights W

(AKA a filter, AKA a feature)

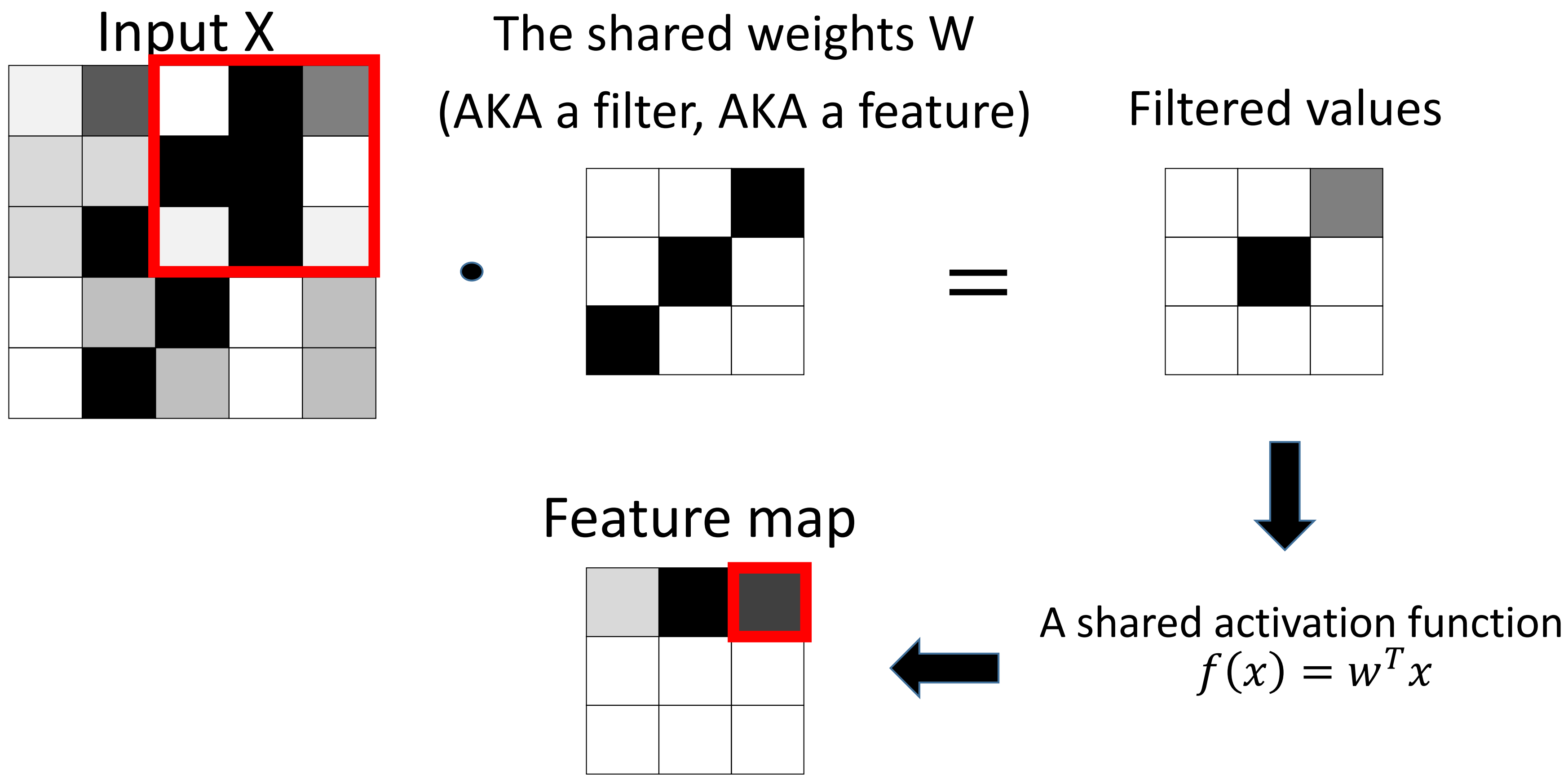
Filtered values

Feature map

A shared activation function

$f(x) = w^T x$

Building that feature map



Input X

The shared weights W

(AKA a filter, AKA a feature)

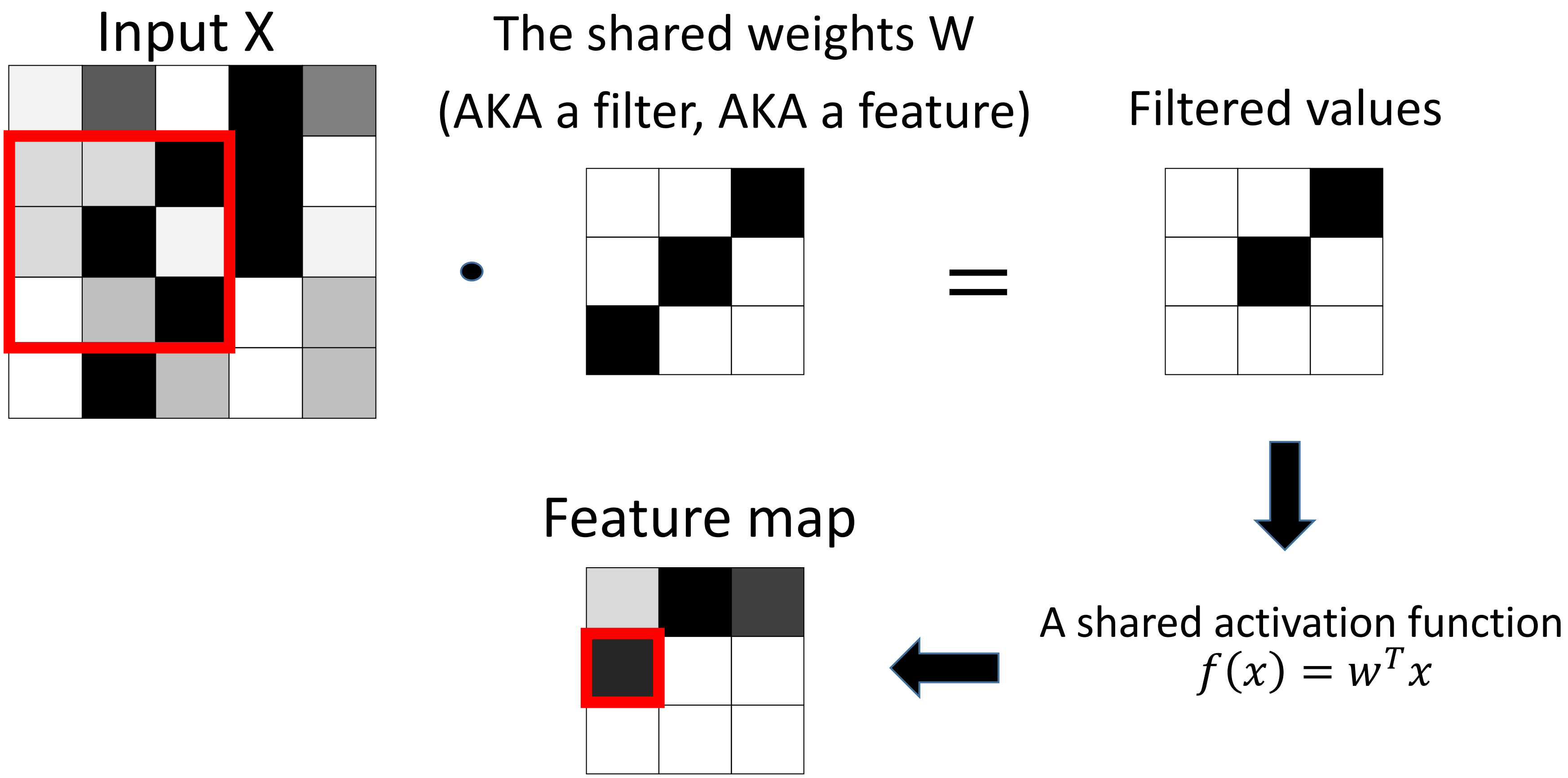
Filtered values

Feature map

A shared activation function

$f(x) = w^T x$

Building that feature map



Input X

The shared weights W

(AKA a filter, AKA a feature)

Filtered values

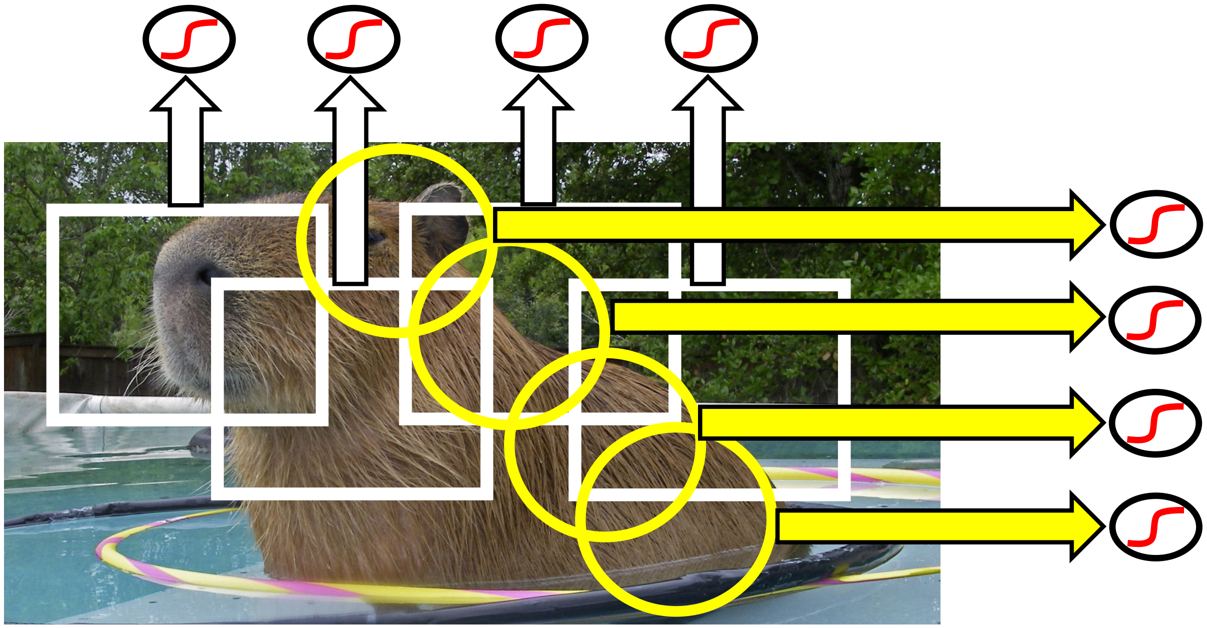
Feature map

A shared activation function

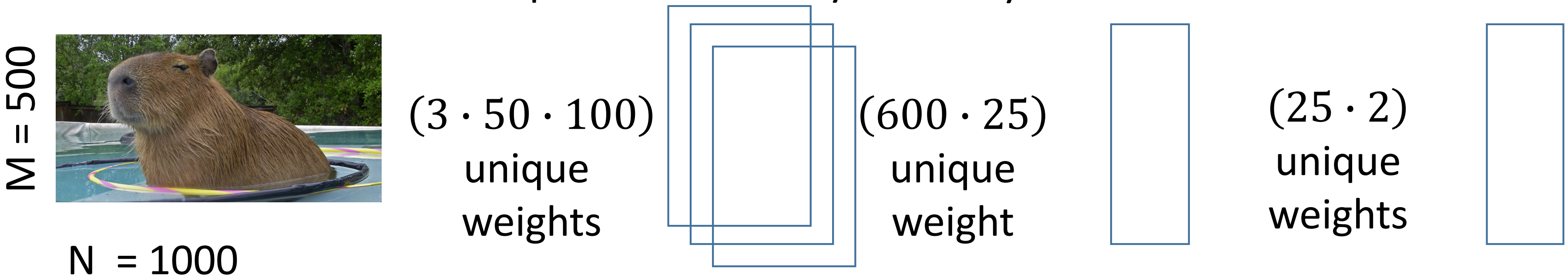
$f(x) = w^T x$

Multiple Feature Maps

- To look for multiple features, use multiple feature maps.
- Each map will specialize on one thing.
- Even with many feature maps, you still have far fewer weights



How many weights in a convolutional net?



Input Layer

M = 500

N = 1000

3 feature maps of 200 nodes. Receptive fields: 50 by 100

$(3 \cdot 50 \cdot 100)$ unique weights

Fully connected layer of 25 nodes

$(600 \cdot 25)$ unique weight

2 fully connected nodes

$(25 \cdot 2)$ unique weights

$15{,}000 + 15{,}000 + 50 = 30{,}050$ unique weights

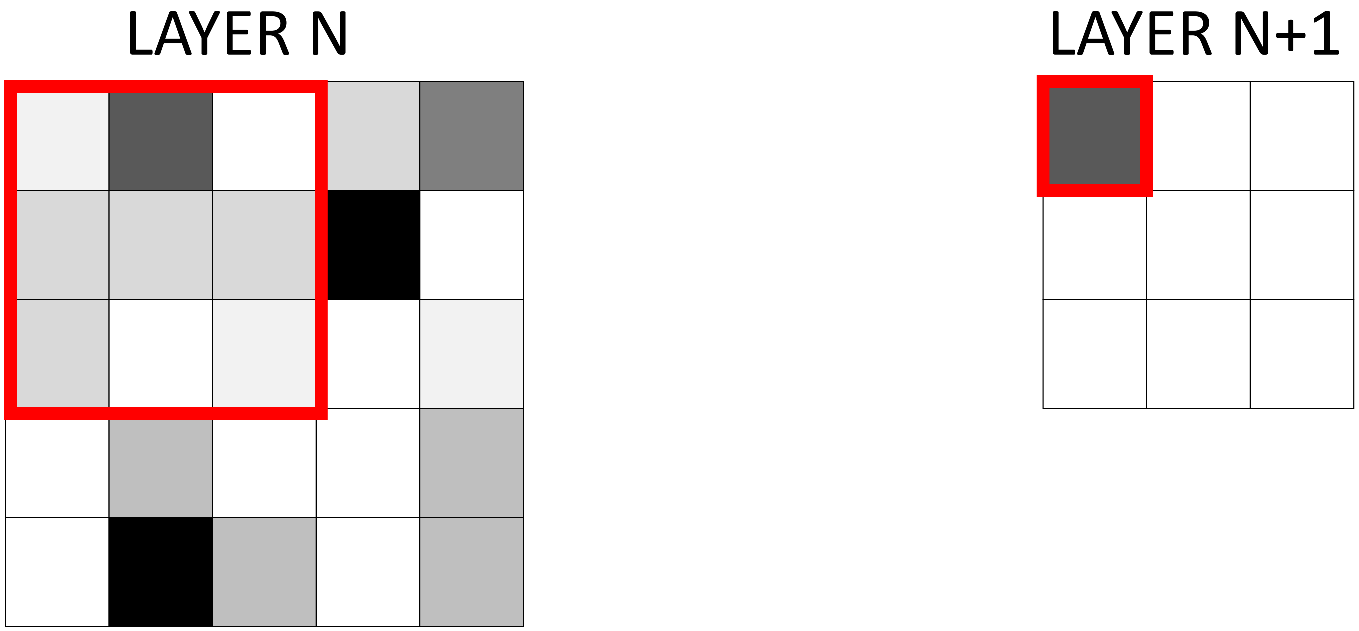
Compare that to the $50{,}005{,}100$ weights in the other network

Is that enough reduction?

- That picture of the adorable Capybara was 500,000 pixels.
- The 2017 iPhone X takes 12 megapixel images. That’s 24 times as big.
- Making the network on the previous slide 24 times bigger would have us at over 600,000 weights.
- Can we do some kind of down sampling on our data?

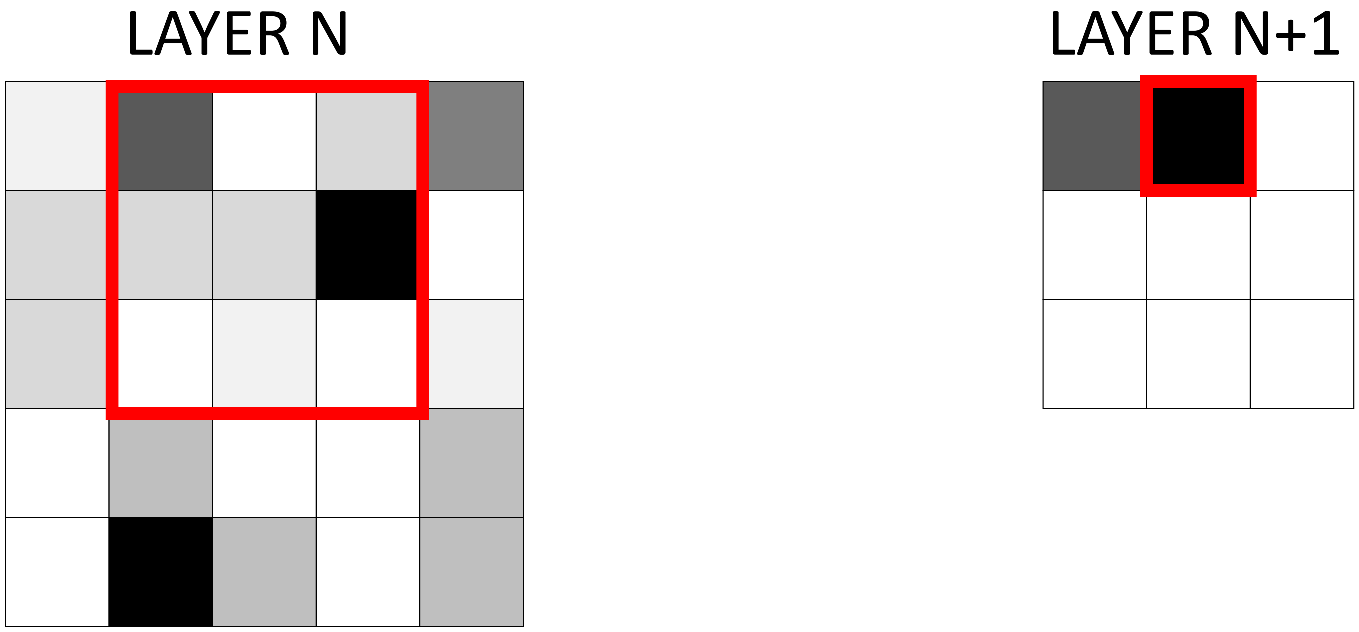
Max Pool Layer: A kind of downsampling

- Max Pool $f(x) = \max(x_1, x_2, \dots x_n)$



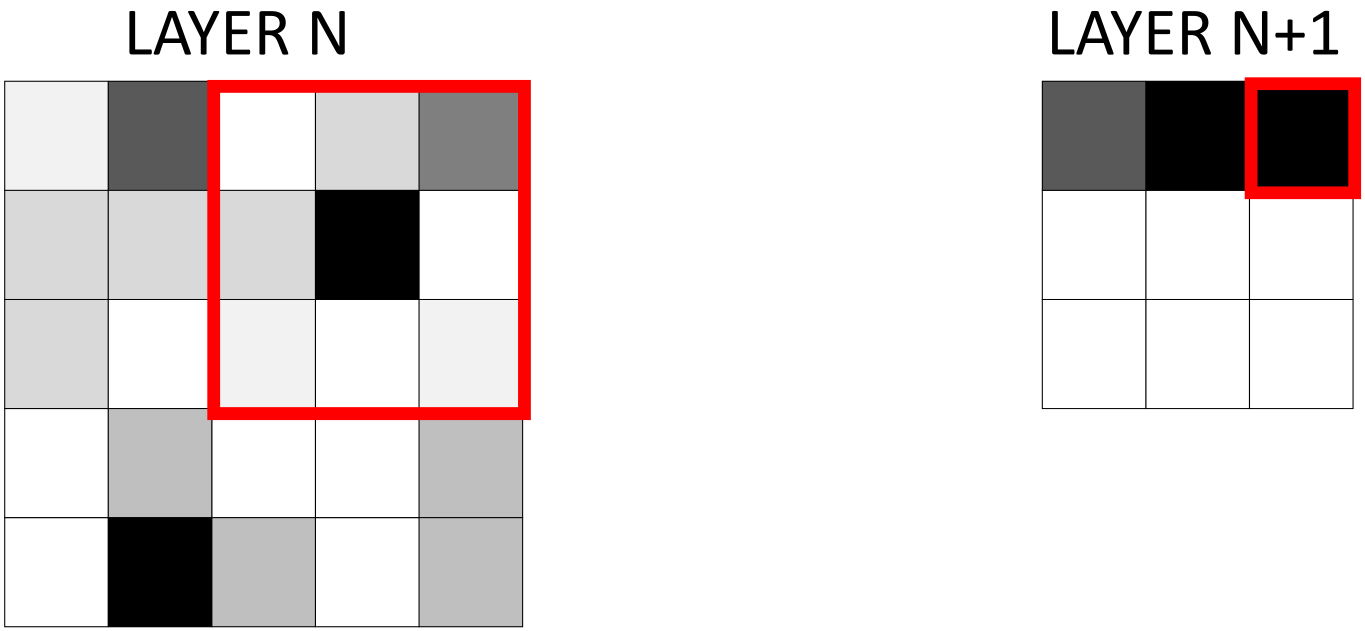
Max Pool Layer: A kind of downsampling

- Max Pool $f(x) = \max(x_1, x_2, \ldots x_n)$



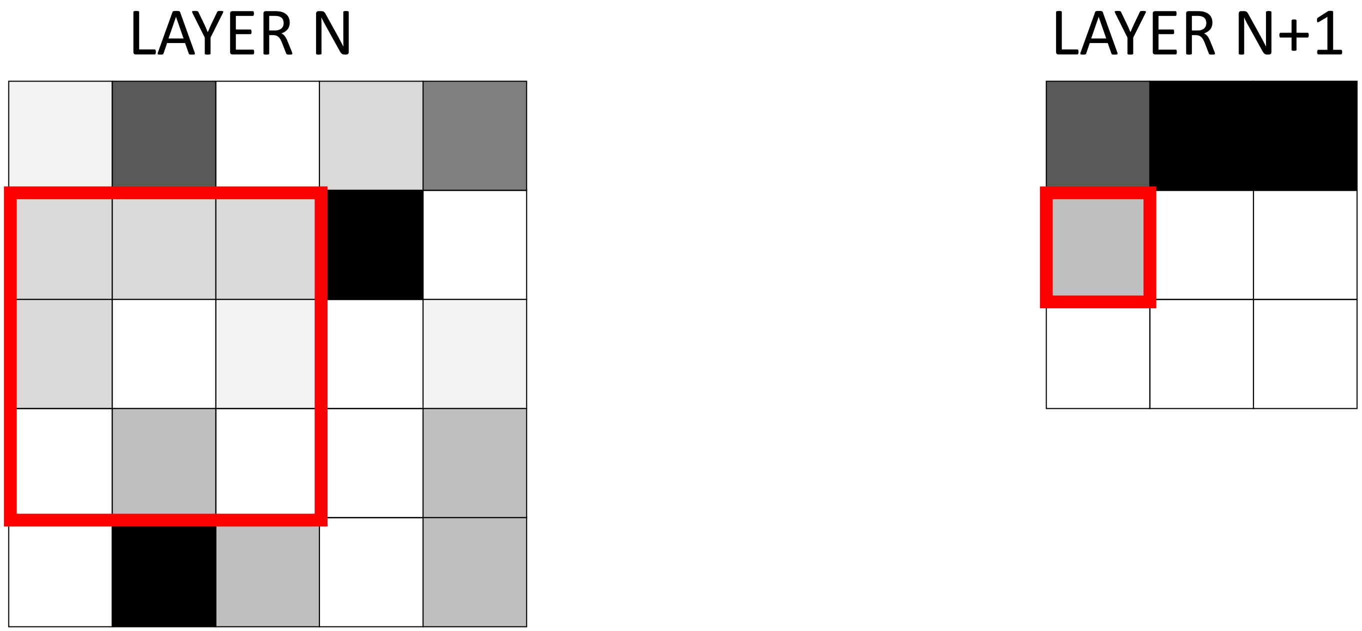
Max Pool Layer: A kind of downsampling

- Max Pool $f(x) = \max(x_1, x_2, \ldots x_n)$



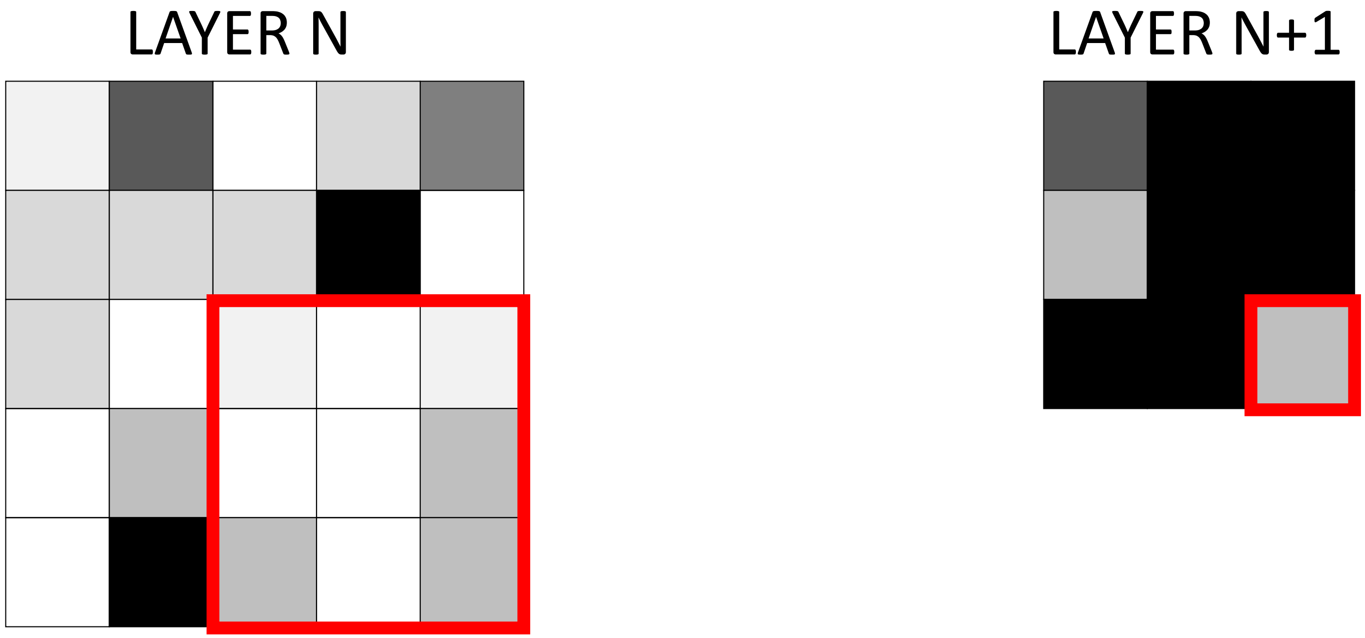
Max Pool Layer: A kind of downsampling

- Max Pool $f(x) = \max(x_1, x_2, \ldots x_n)$



Max Pool Layer: A kind of downsampling

- Max Pool $f(x) = \max(x_1, x_2, \dots x_n)$



So…what is a convolutional net?

- A network with one or more layers that are feature maps

- A layer with feature maps is called a “convolutional layer”

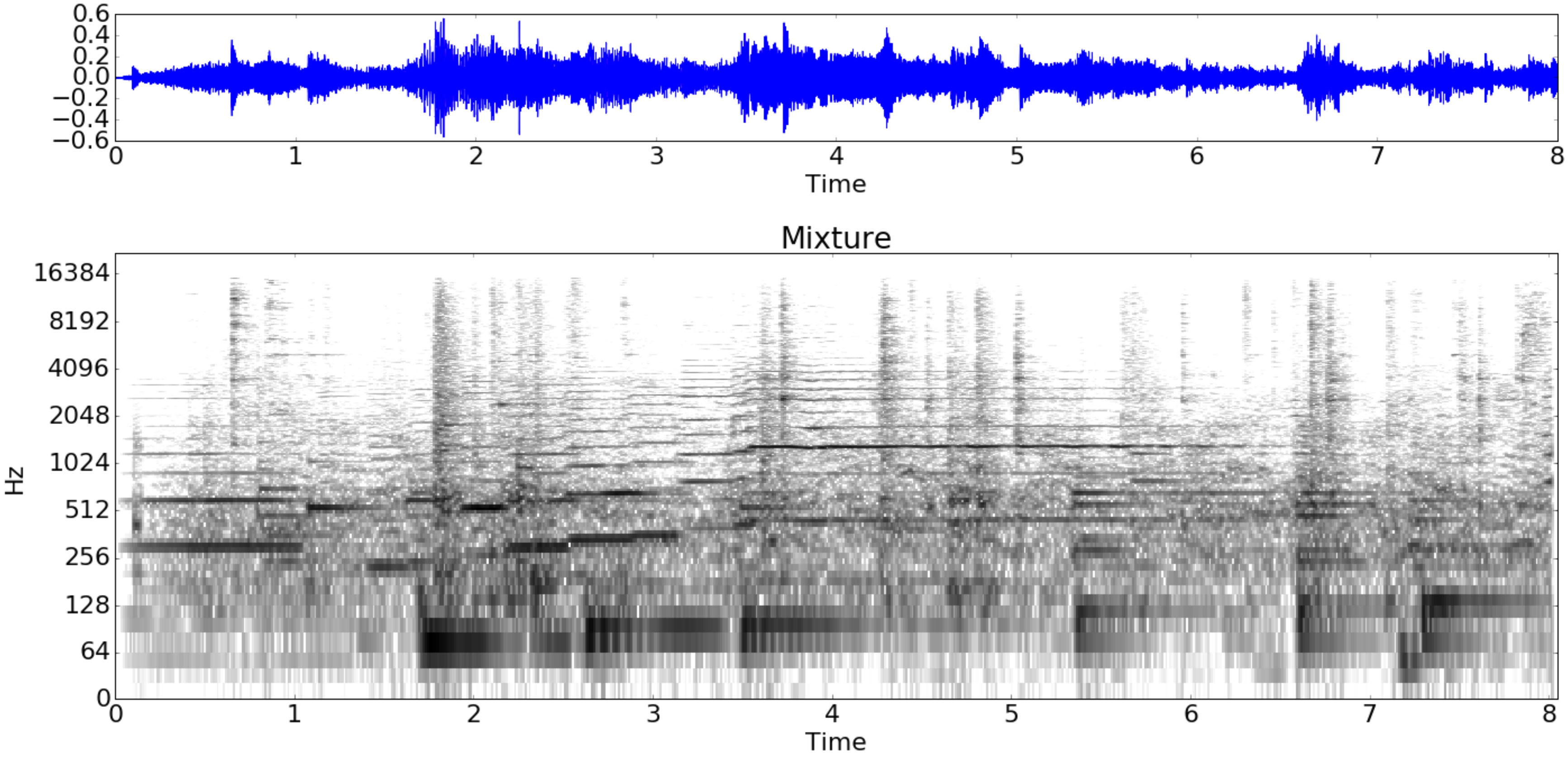
- Often, convolutional layers are alternated with pooling layers.

- Since these nets have many fewer connections
  - They train faster
  - They need fewer training examples

RECURRENT NETS

Werbos, Paul J. "Backpropagation through time: what it does and how to do it." *Proceedings of the IEEE* 78.10 (1990): 1550-1560.

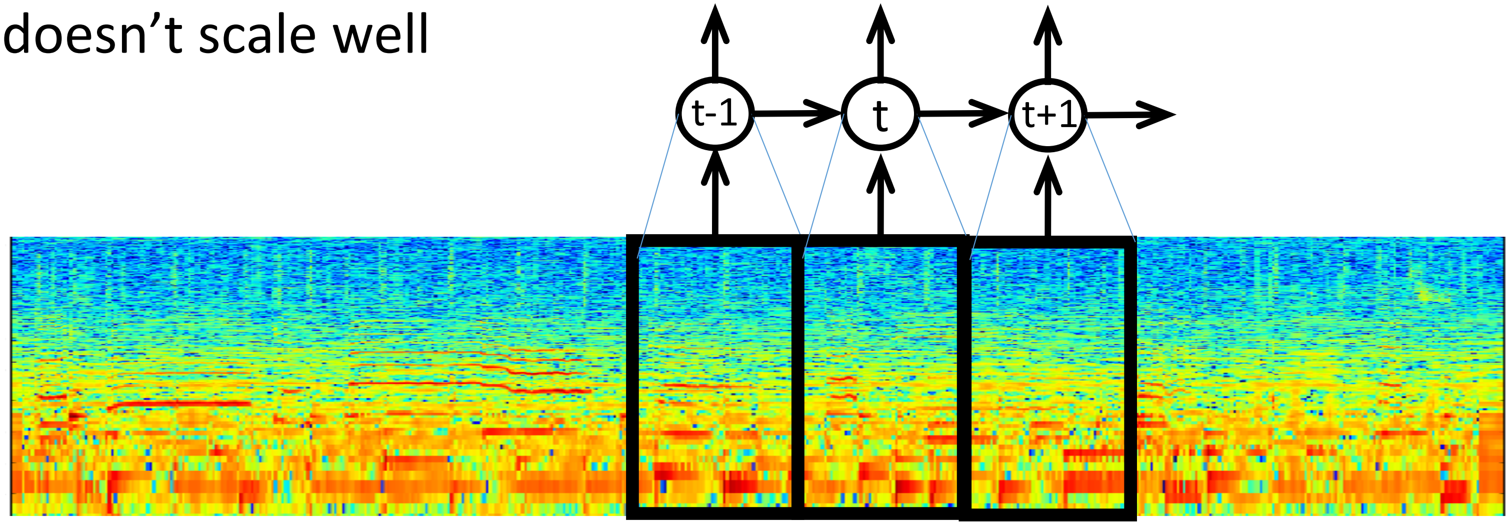
A complex auditory scene



Music + dishes + people talking

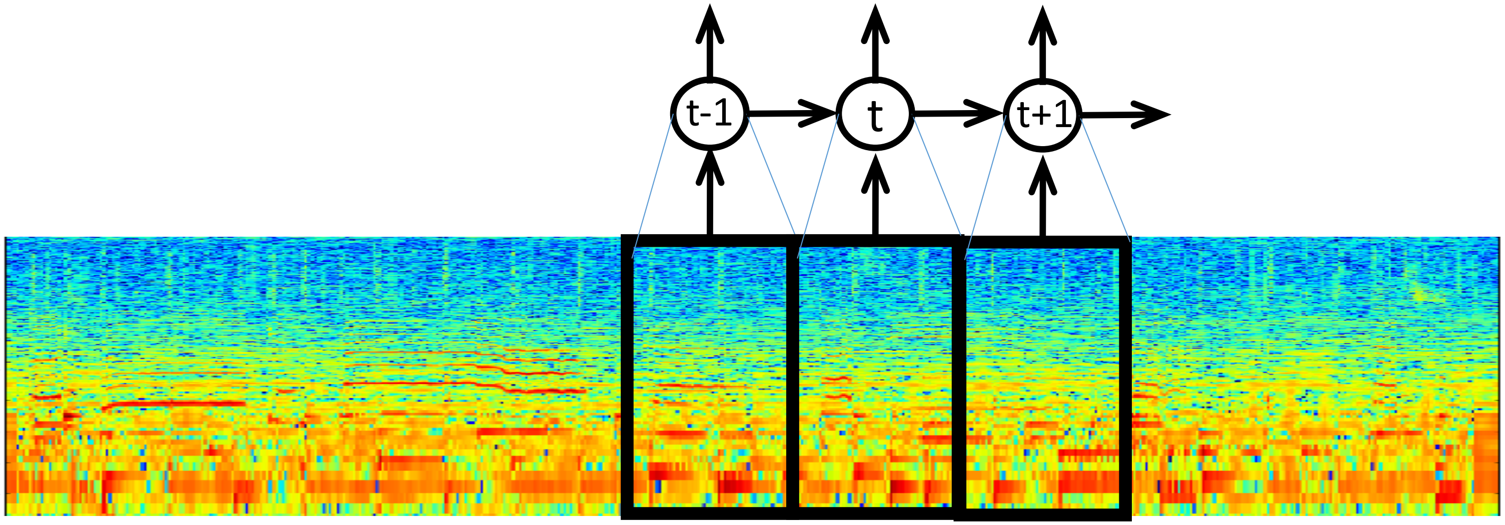
Dealing with time

- With a ”standard” feed-forward architecture, you process data from within a window, ignoring everything outside the window.
- To get influence from the processing of earlier time steps, add nodes and connections
- This doesn’t scale well



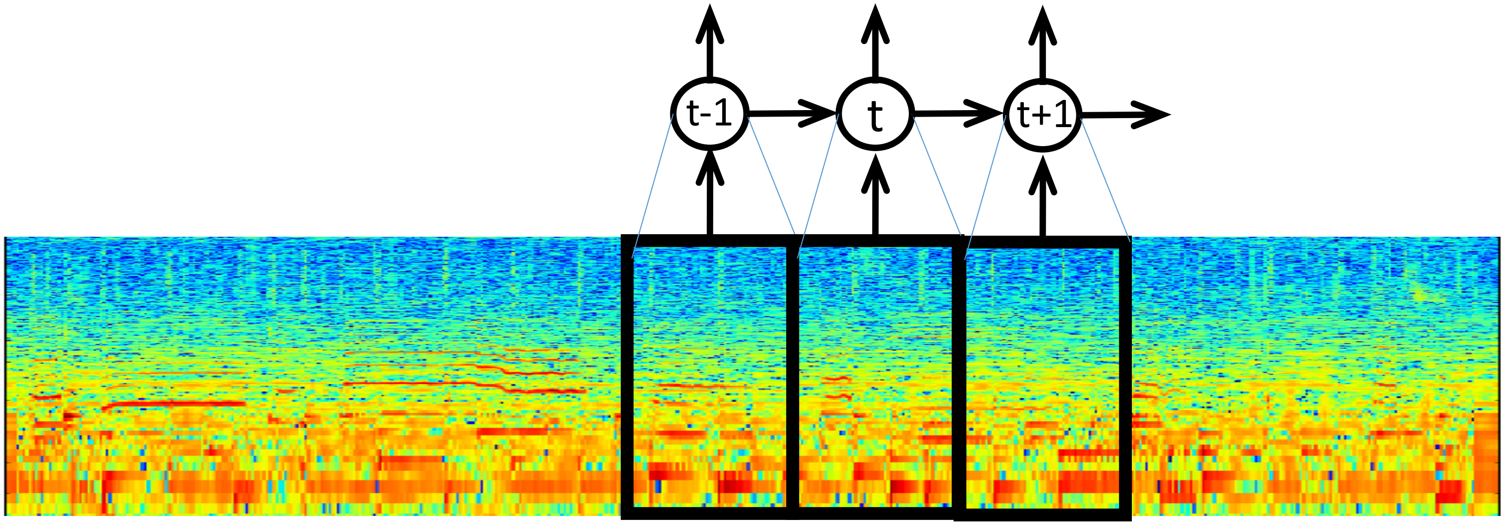
Weight sharing

- If all the windows share the same input weights (like in a feature map), then we only have the same number of weights as if we had a single window.
- This is a recurrent net.



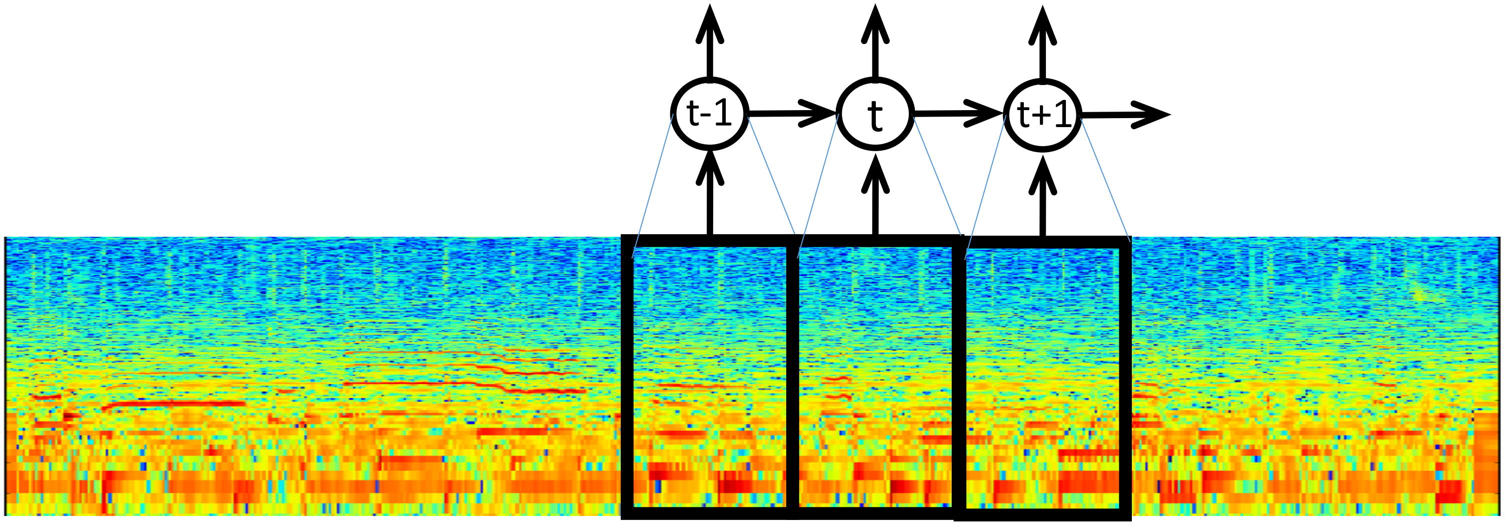
Exponentially decaying influence

- If your network needs to connect information from a distant timestep, the influence of the earlier one tends to get lost
- This problem was solved by the LSTM



Exponentially decaying influence

- If your network needs to connect information from a distant timestep, the influence of the earlier one tends to get lost
- This problem was solved by the LSTM

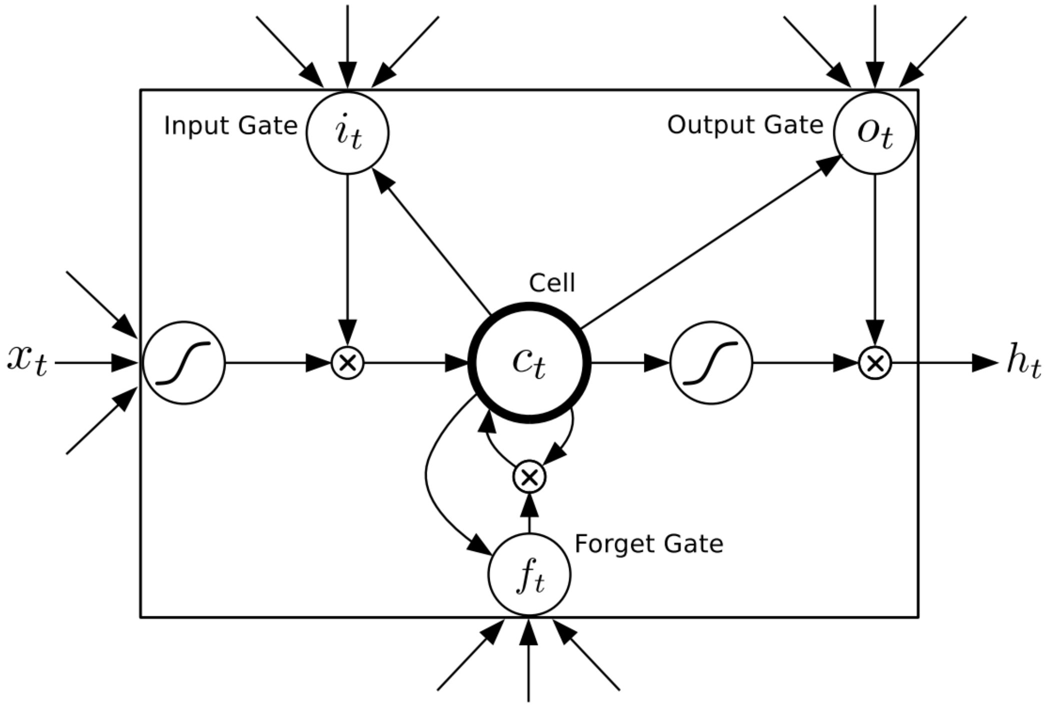


Long-Short Term Memories

Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term memory." *Neural computation* 9.8 (1997): 1735-1780.

Long Short Term Memory Units (LSTMs)

- Added a way of storing data over many time steps without decay
- Let networks to handle problems with long term dependencies
- Are too complicated to explain right now.



A single LSTM memory unit

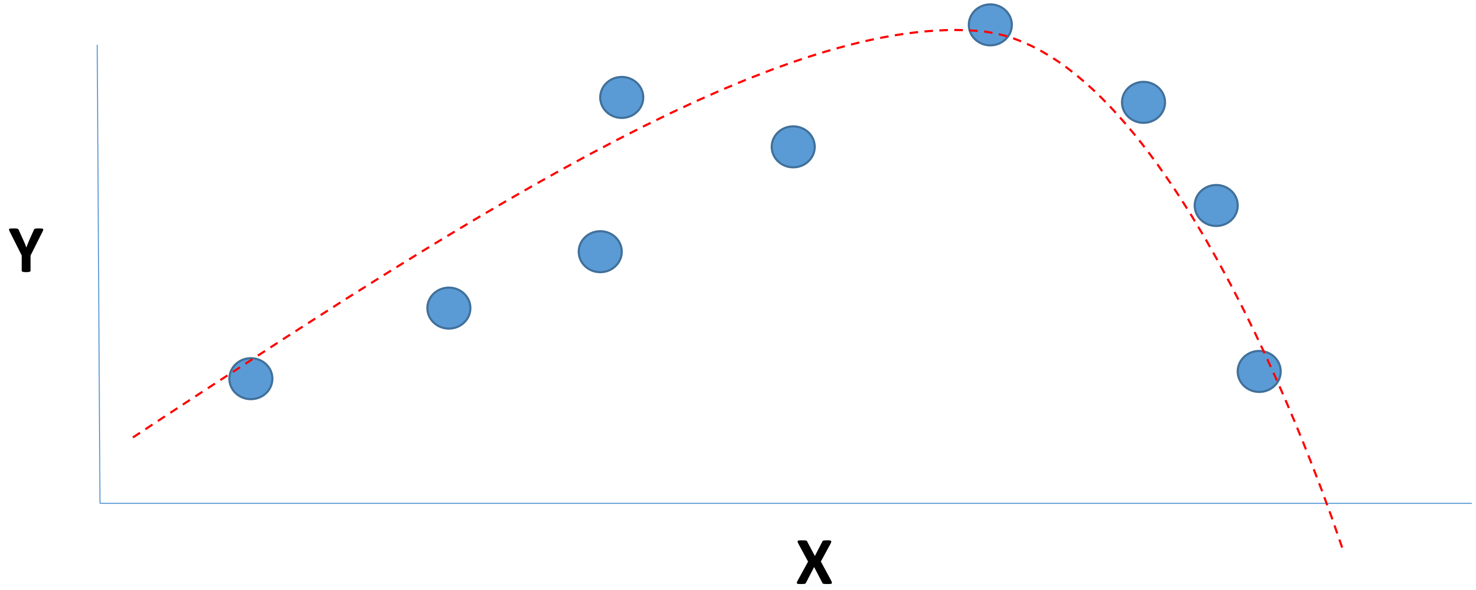
Data collection & augmentation

Pick data **D**

The data defines a function to learn: $f(x) = y$

Typically, this is from $\mathbb{R}^d$ to $\mathbb{R}^d$.

This is called **regression**.

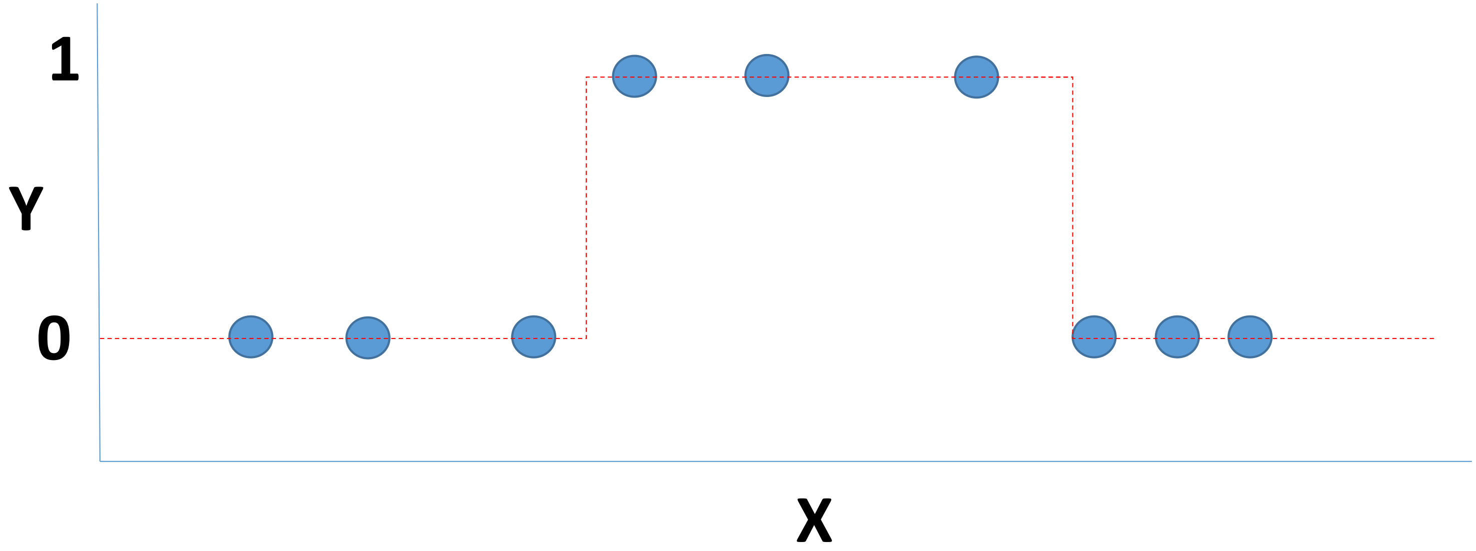


Pick data **D**

The data defines a function to learn: $f(x) = y$

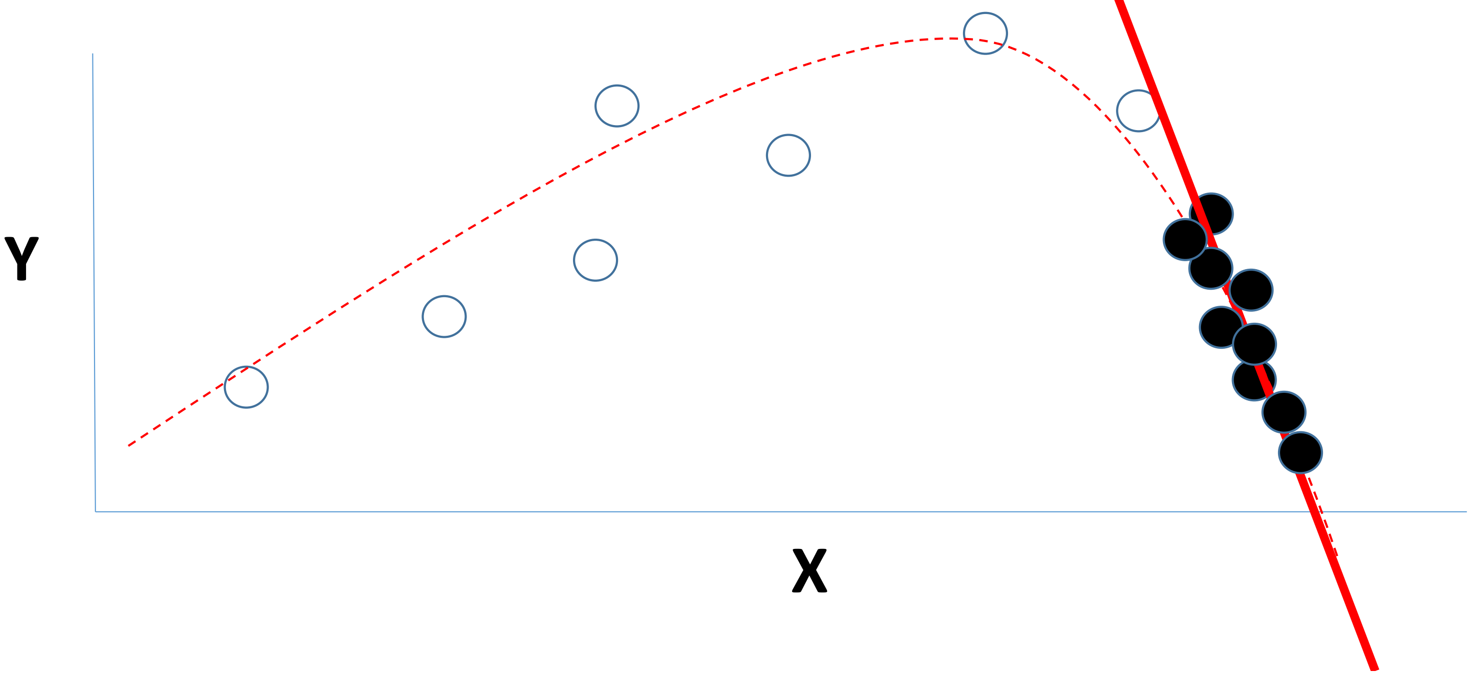
This can also be from $\mathbb{R}^d$ to a finite set of labels, e.g. {0,1}.

This is **classification**.



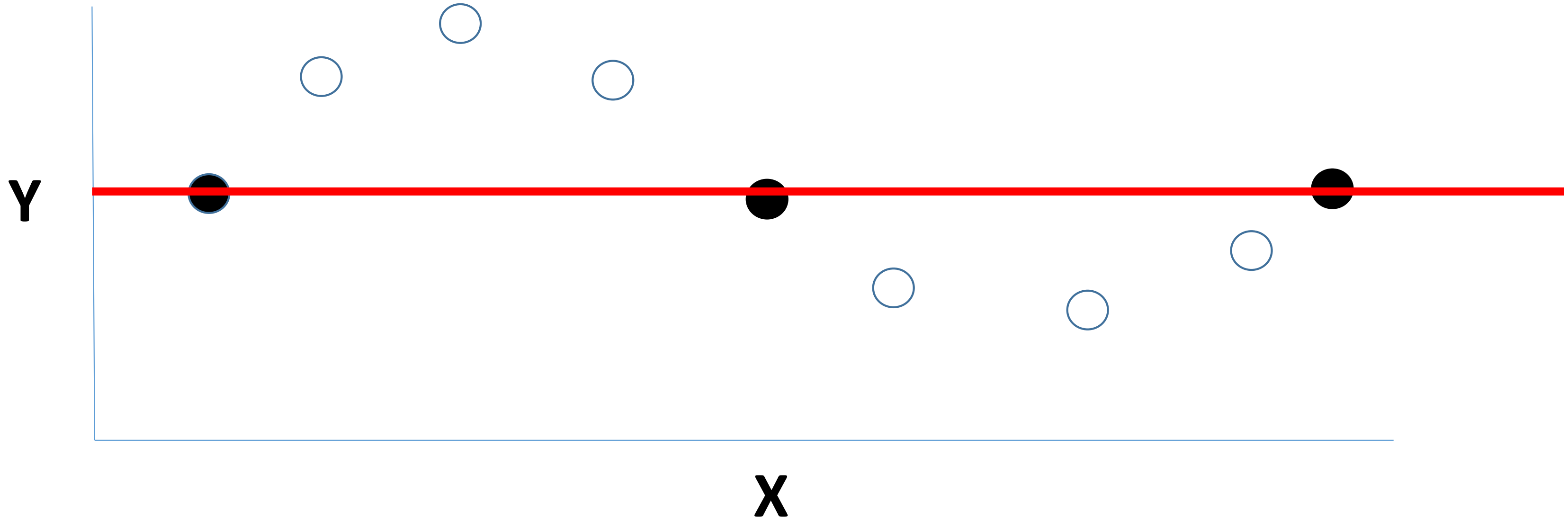
Pick data D: Is there enough?

- Good coverage of the range of possible values?
- Just because you got lots of data, doesn’t mean it covers the space.



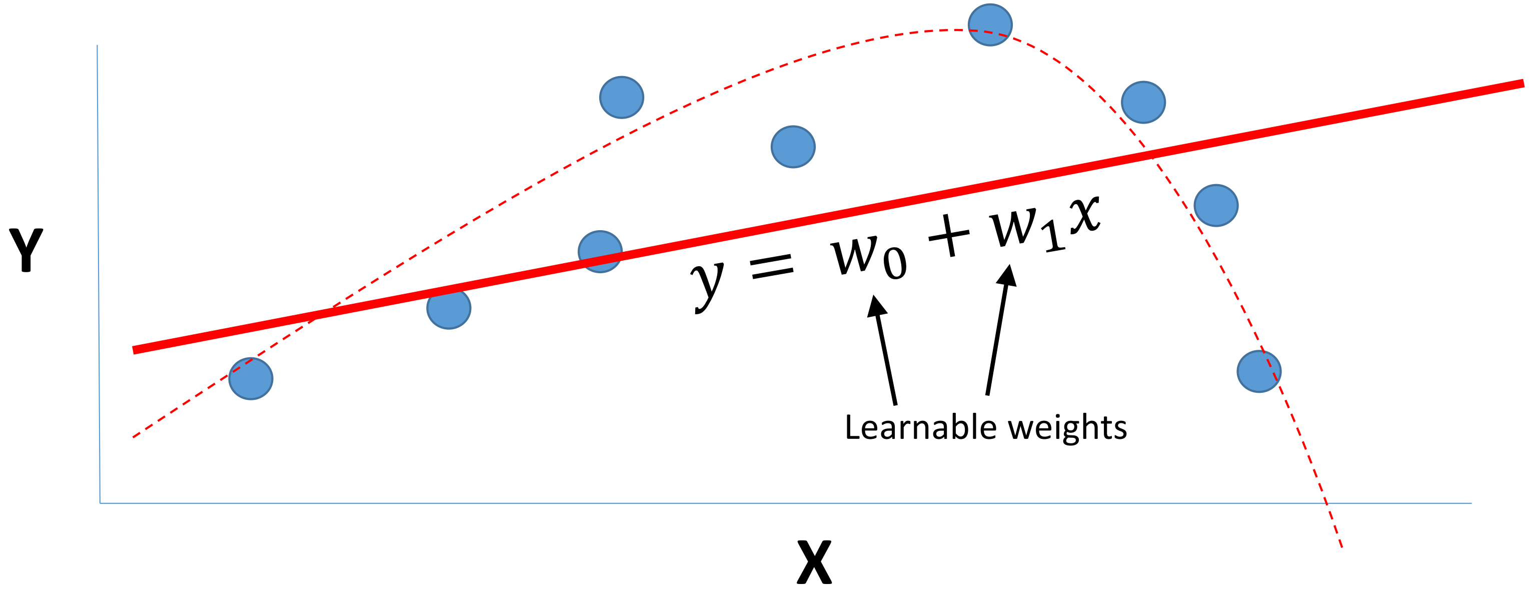
Pick data D: Is there enough?

- Enough density in the space?
- Just because you cover the range, doesn't mean you captured the function.



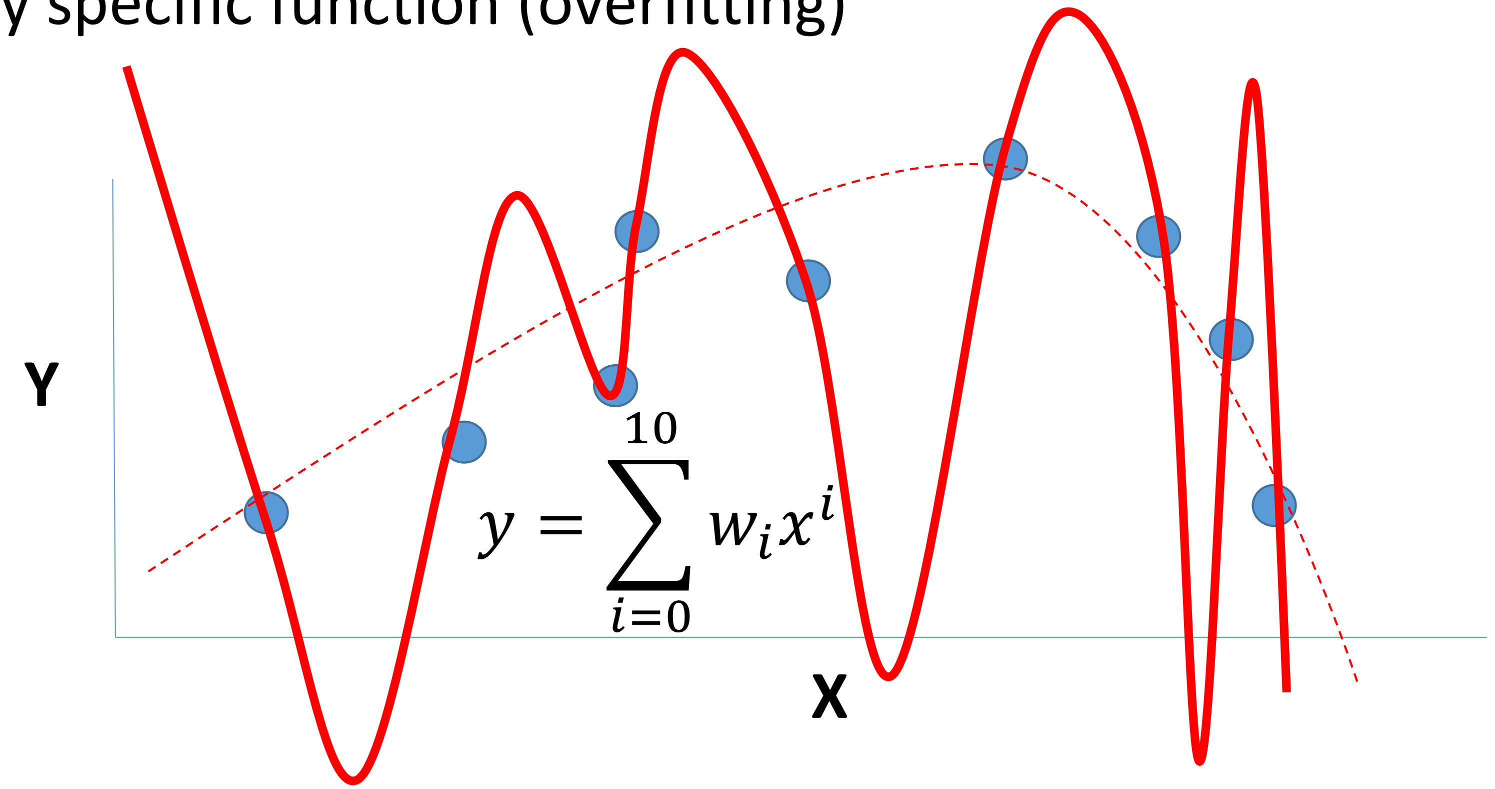
Fitting & Hypothesis space.

If a model's hypothesis space is too small, the true function is probably not in its vocabulary (underfitting)



Fitting & Hypothesis space

If a model's hypothesis space is too big, it can learn a crazy, overly specific function (overfitting)



$$y = \sum_{i=0}^{10} w_i x^i$$

Dimensions and data

- The more dimensions your data has, the more data you need to cover the space
- The more dimensions, the more parameters your model needs (at least 1 per dimension)
- The more parameters, the more data you need to prevent overfitting
- Conclusion: You probably don’t have enough data. You probably overfit somehow.

Data Augmentation

- Make perturbed copies of your data that vary in ways that should not change the value nature of the output function.

- This can help prevent spurious correlations between data and output.

- Example: Distinguishing clarinet sounds from flute sounds
  - Vary the pitch of each note by + or – 1%, 2%, 3%, 4%....
  - Add background noise of different kinds and at different dB
  - Time-stretch each note a bit
  - Delay or advance the onset of the note

- This can turn 1000 data points into 100,000.

Reduce your patch size, if you can

- Use a small patch of the spectrogram as input (e.g. 100 by 100 patch of the spectrogram)
- Reduces the number of model parameters needed
- Increases the number of training examples

1000 Spectrograms * 600 patches* 100augmentations = 60 million

Conclusions

- Deep nets are powerful, but not all-powerful
- The design space is very large
- Success requires
  - Lots of patience
  - Much architecture hacking
  - Lots of data curation

To get started

Read *Deep Learning* by Goodfellow, Bengio and Courville

www.deeplearningbook.org

Get the Tensorfow open source library for ML

www.tensorflow.org

Make it easier with TFLearn, a wrapper for Tensorflow

tflearn.org

Stuff we won’t have time for starts HERE

Hebbian Learning

Donald Hebb

- A cognitive psychologist active mid 20th century
- Influential book: The Organization of Behavior (1949)
- Hebb’s postulate

"Let us assume that the persistence or repetition of a reverberatory activity (or "trace") tends to induce lasting cellular changes that add to its stability.… When an axon of cell *A* is near enough to excite a cell *B* and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that *A*'s efficiency, as one of the cells firing *B*, is increased.

Pithy version of Hebb’s postulate

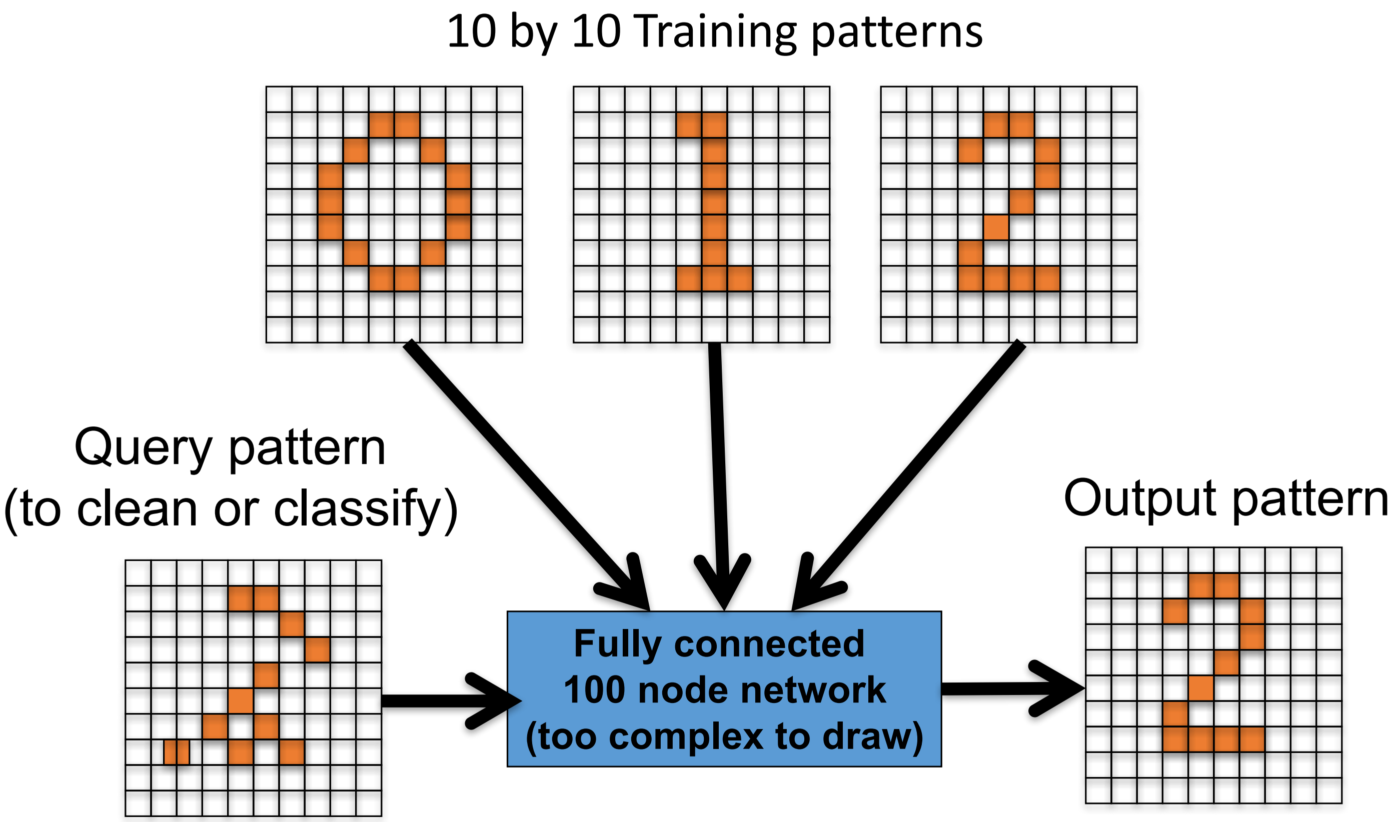
Cells that fire together, wire together.

Hopfield networks

Hopfield nets are

- “Hebbian” in their learning approach
- Old technology. Nobody uses this. These are proof of concept things from the 80s and earlier.
- Fast to train, slower to use
- Weights are symmetric
- All nodes are input & output nodes
- Use binary (1, -1) inputs and output
- Used as
  - Associative memory (image cleanup)
  - Classifier

Using a Hopfield Net



10 by 10 Training patterns

Query pattern
(to clean or classify)

Fully connected
100 node network
(too complex to draw)

Output pattern

Training a Hopfield Net

- Assign connection weights as follows

$$w_{ij} = \begin{cases} \sum_{c=1}^{C} x_{c,i} x_{c,j} & \text{if } i \neq j \\ 0 & \text{if } i = j \end{cases}$$

c index number for C many class exemplars

i, j index numbers for nodes

$w_{i,j}$ connection weight from node i to node j

$x_{c,i} \in \{+1, -1\}$ element i of the exemplar for class c

Using a Hopfield Net

Force output to match an unknown input pattern

$s_i(0) = x_i \qquad \forall i$ Here, $\mathrm{s}_i(t)$ is the state of node i at time t and x_i is the value of the input pattern at node i

Iterate the following function until convergence

$$s_i(t+1) = \begin{cases} 1 & \text{if } 0 \leq \sum_{j=1}^{N} w_{ij} s_j(t) \\ -1 & else \end{cases}$$

Note: this means you have to pick an order for updating nodes.
People often update all the nodes in random order

Using a Hopfield Net

Once it has converged…

FOR INPUT CLEANUP: You’re done. Look at the final state of the network.

FOR CLASSIFICATION: Compare the final state of the network to each of your input examples. Classify it as the one it matches best.

Input Training Examples

Why isn’t 5 in the set of examples?

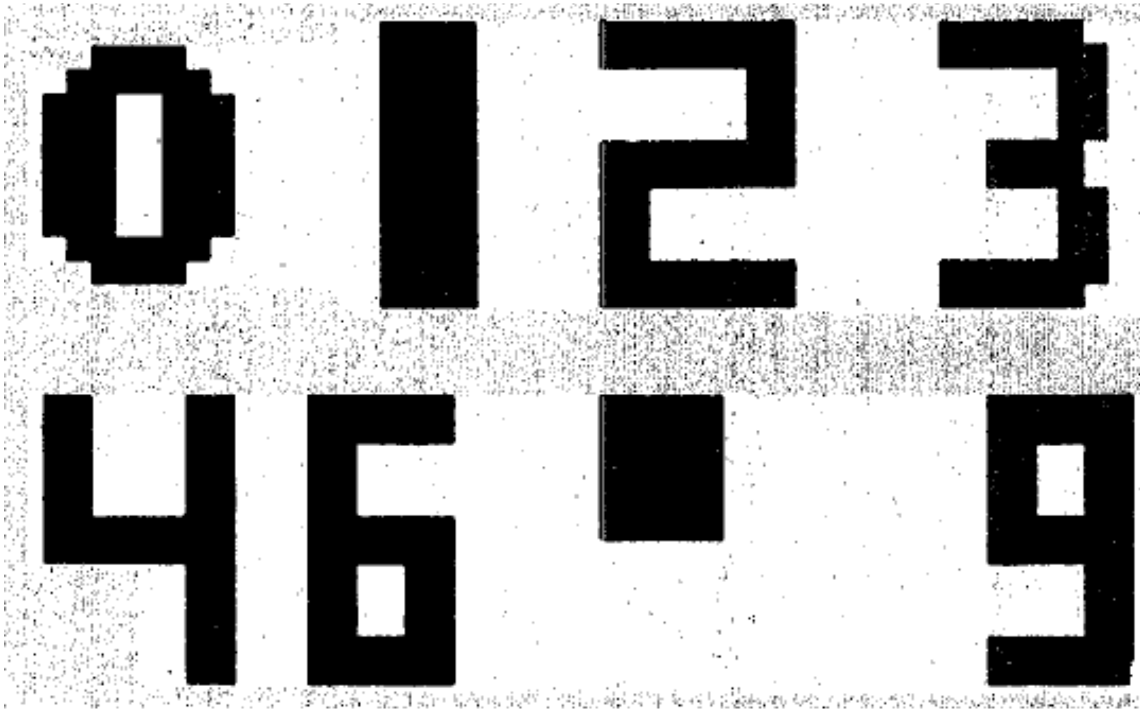


Image from:R. Lippman, An Introduction to Computing with Neural Nets, IEEE ASSP Magazine, April 1987

Output of network over 8 iterations

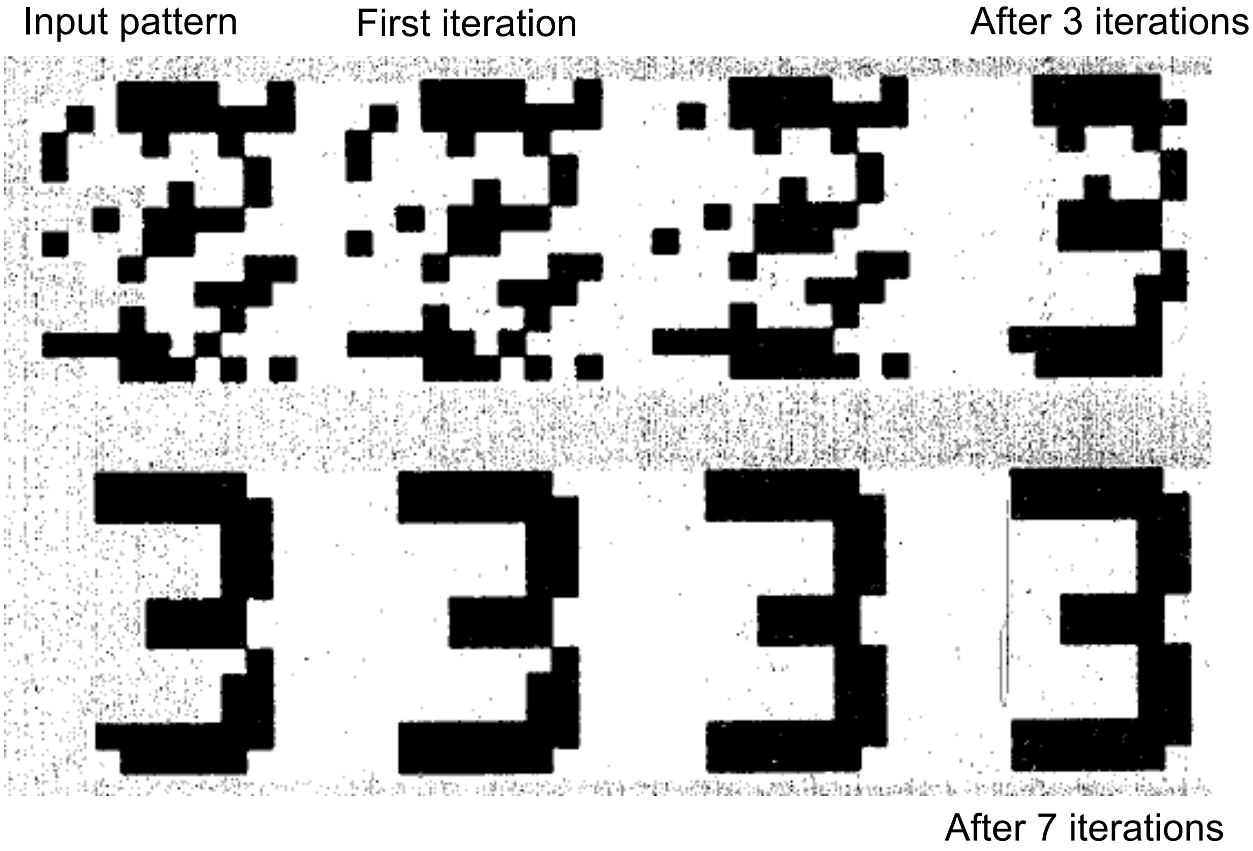
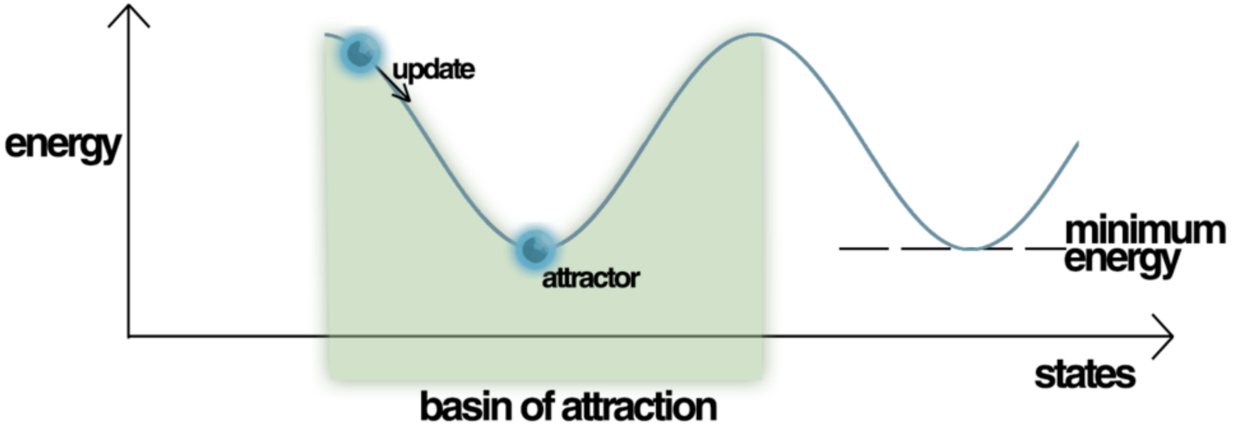


Image from:R. Lippman, An Introduction to Computing with Neural Nets, IEEE ASSP Magazine, April 1987

Characterizing “Energy”



- As $s_i(t)$ is updated, the state of the system converges on an “attractor”, where $s_i(t+1) = s_i(t)$
- Convergence is measured with this “Energy” function:

$$E(t) = -\frac{1}{2}\sum_{i,j} w_{ij} s_i(t) s_j(t)$$

Note: people often add a “bias” term to this function. I’m assuming we’ve added an extra “always on” node to make our “bias”

Image from: http://en.wikipedia.org/wiki/Hopfield_network#mediaviewer/File:Energy_landscape.png

Limits of Hopfield Networks

- Input patterns become confused if they overlap
- The number of patterns it can store is about 0.15 times the number of nodes
- Retrieval can be slow, if there are a lot of nodes (it can take thousands of updates to converge)

Restricted boltzman machine (RBM)

About RBNs

- Related to Hopfield nets
- Used extensively in Deep Belief Networks
- You can’t understand DBNs without understanding these

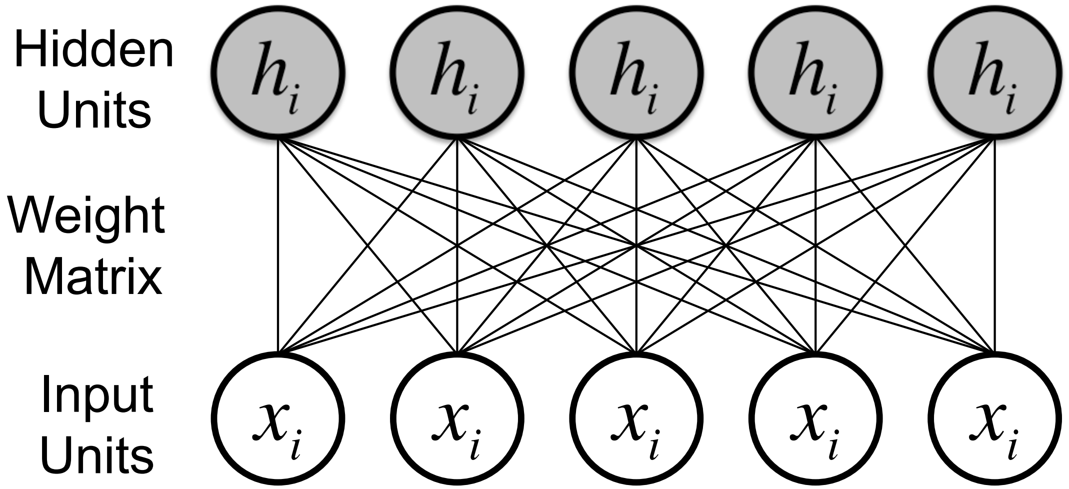
About RBNs

- Related to Hopfield nets

- Used extensively in Deep Belief Networks

- You can’t understand DBNs without understanding these

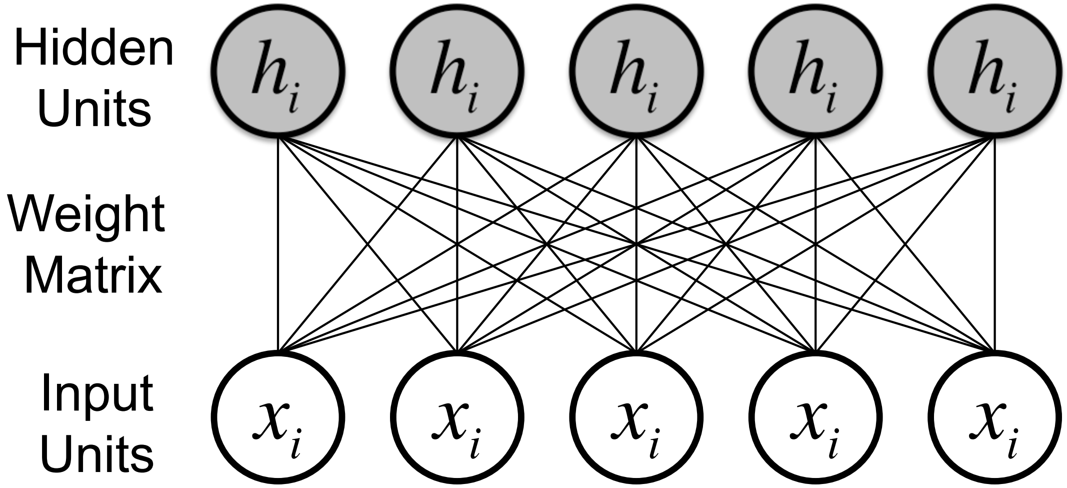
Standard RBM Architecture



2 layers (hidden & input) of Boolean nodes

Nodes only connected to the other layer

Standard RBM Architecture



Hidden Units

Weight Matrix

Input Units

- Setting the hidden nodes to a vector of values updates the visible nodes…and vice versa

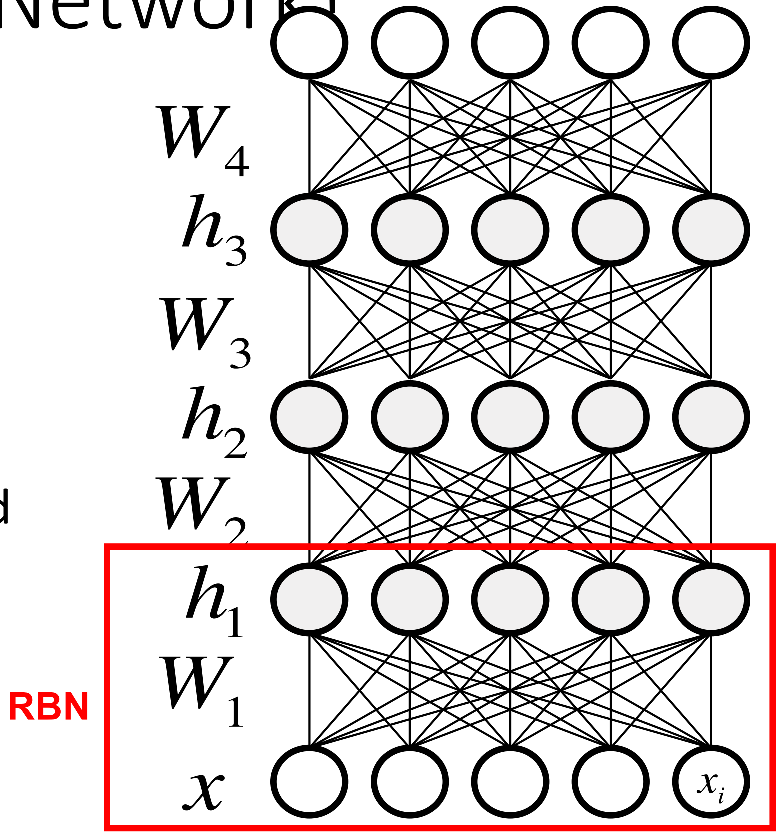
Contrastive Divergence Training

1. Pick a training example.
2. Set the input nodes to the values given by the example.
3. See what activations this gives the hidden nodes.
4. Set the hidden nodes at the values from step 3.
5. Set the input node values, given the hidden nodes
6. Compare the input node values from step 5 to the the input node values from step 2
7. Update the connection weights to decrease the difference found in step 6.
8. If that difference falls below some epsilon, quit. Else, go to step 1.

Deep BELIEF Network (DBN)

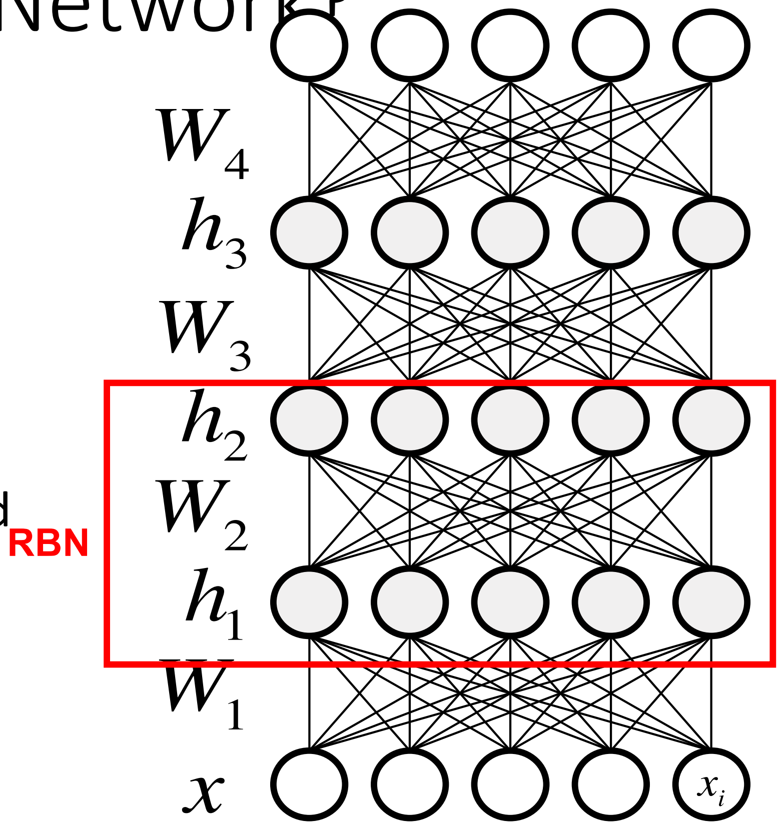
What is a Deep Belief Network?

- A stack of RBNS
- Trained bottom to top with Contrastive Divergence
- Trained AGAIN with supervised training (similar to backprop in MLPs)



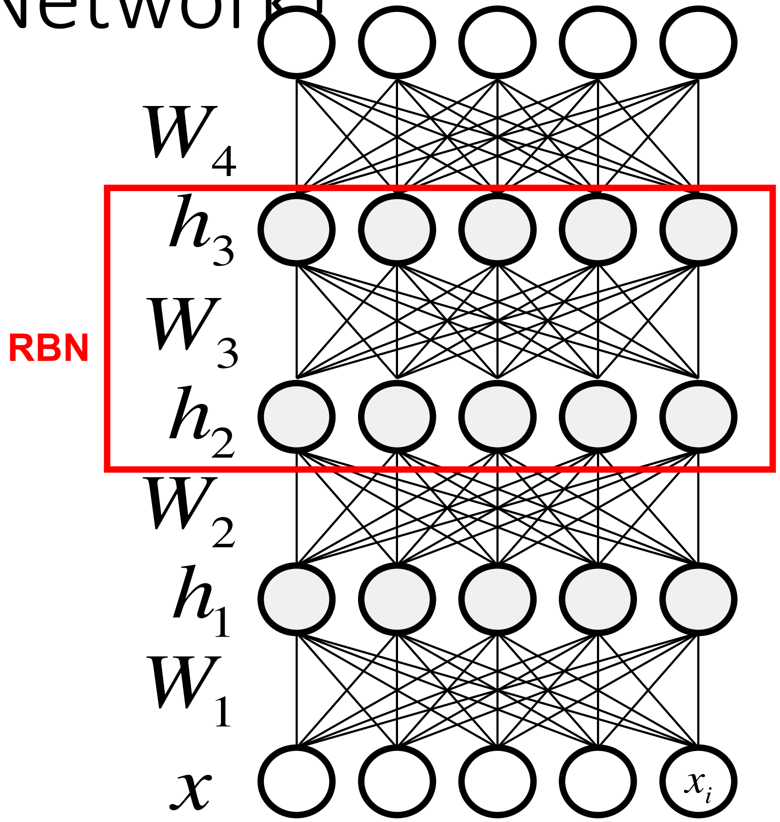
What is a Deep Belief Network?

- A stack of RBNS
- Trained bottom to top with Contrastive Divergence
- Trained AGAIN with supervised training (similar to backprop in MLPs)



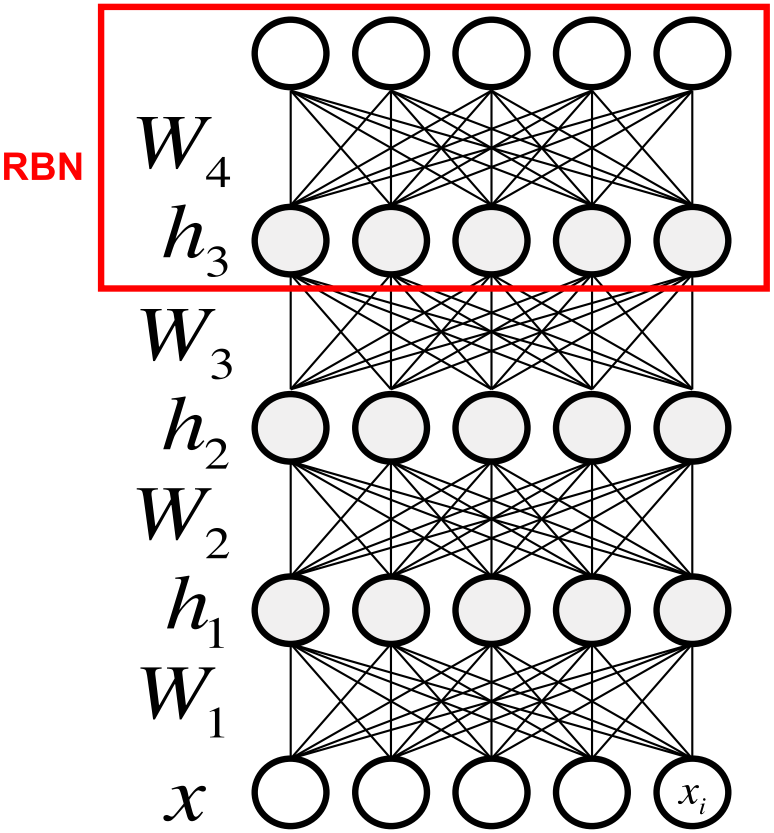
What is a Deep Belief Network?

- A stack of RBNS
- Trained bottom to top with Contrastive Divergence
- Trained AGAIN with supervised training (similar to backprop in MLPs)



What is a Deep Belief Network?

- A stack of RBNS
- Trained bottom to top with Contrastive Divergence
- Trained AGAIN with supervised training (similar to backprop in MLPs)



Why are DBNs important?

- They are state-of-the-art systems for doing certain recognition tasks
    - Handwritten digits
    - Phonemes

- They are very “hot” right now

- They have a good marketing campaign “Deep learning” vs “shallow learning”

How does “deep” help?

- It may be possible to much more naturally encode problems like the parity problem with deep representations than with shallow ones

Why not use standard MLP training?

- Fading signal from backprop
- The more complex the network, the more likely there are local minima
- Memorization issues
- Training set size and time to learn

Benefits

- Allows relatively deep networks (e.g. 10 layers), compared to regular perceptrons
- In the mid 2000's this approach made networks better than anything else out there for some problems (e.g. digit recognition, phoneme recognition)
- Today, they've been superseded by other approaches