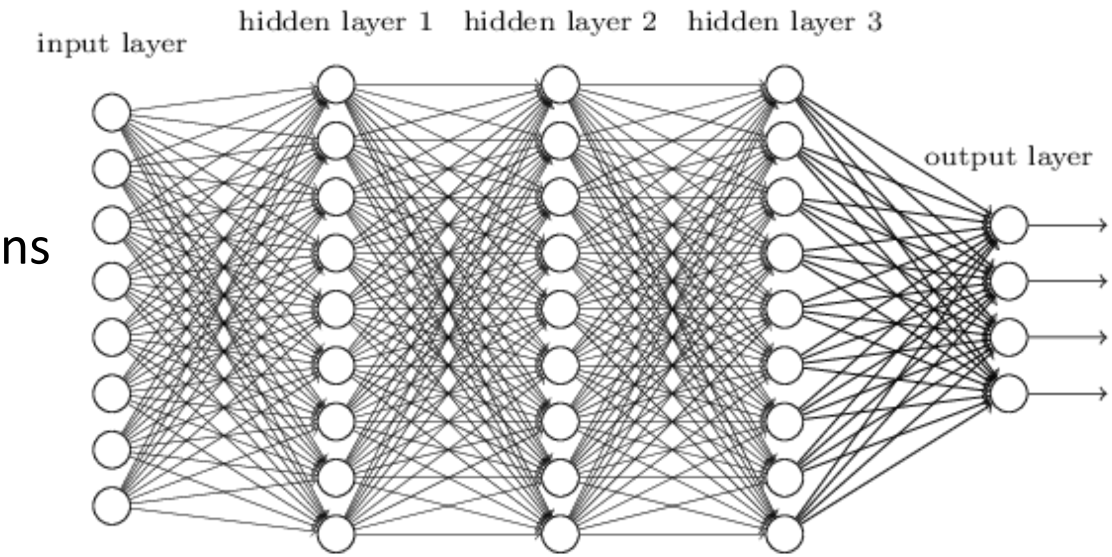
DEEP LEARNING AND AUDIO

Bryan Pardo
Interactive Audio Lab
Northwestern University

Deep Nets (AKA Neural Nets)

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



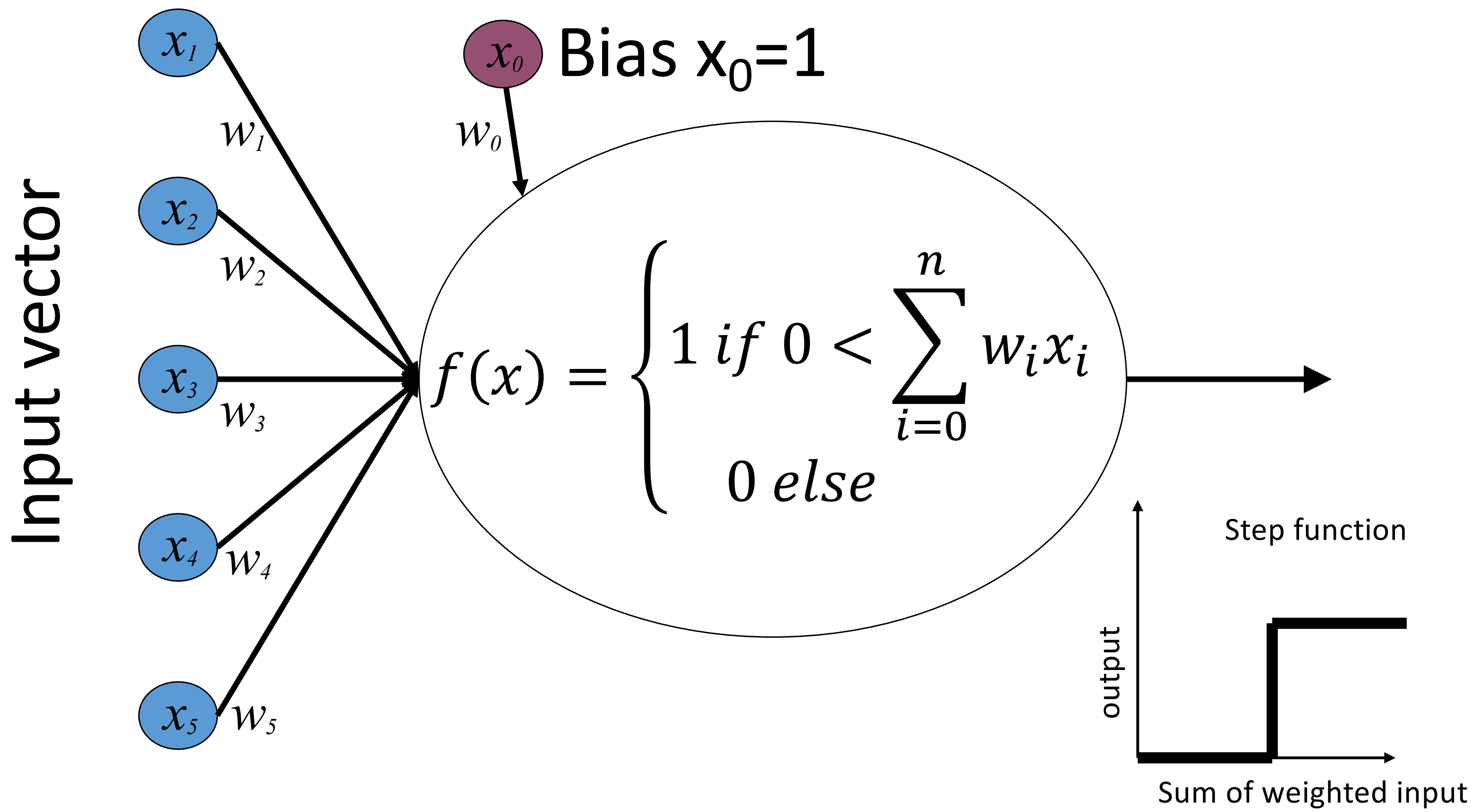
Machine Learning in one slide

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

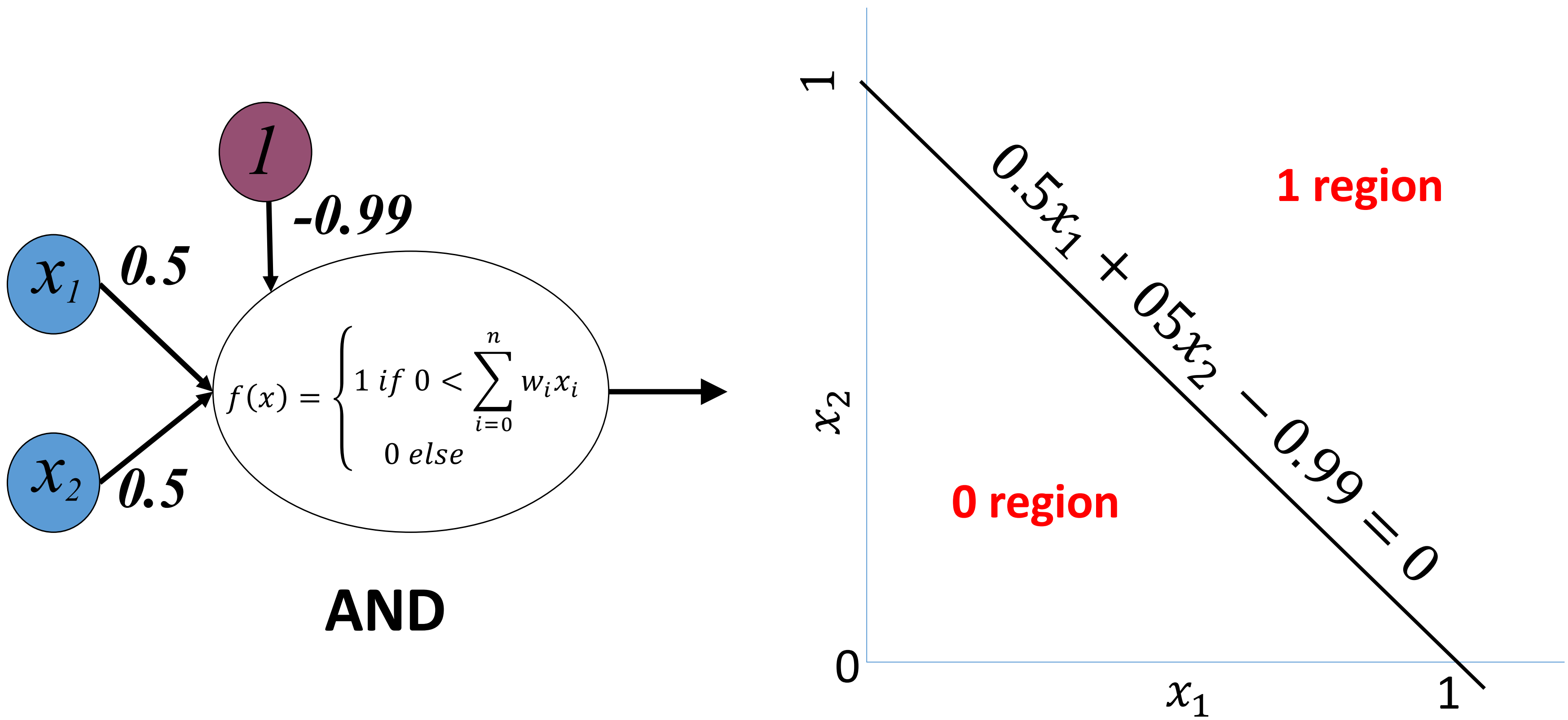
The Perceptron

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

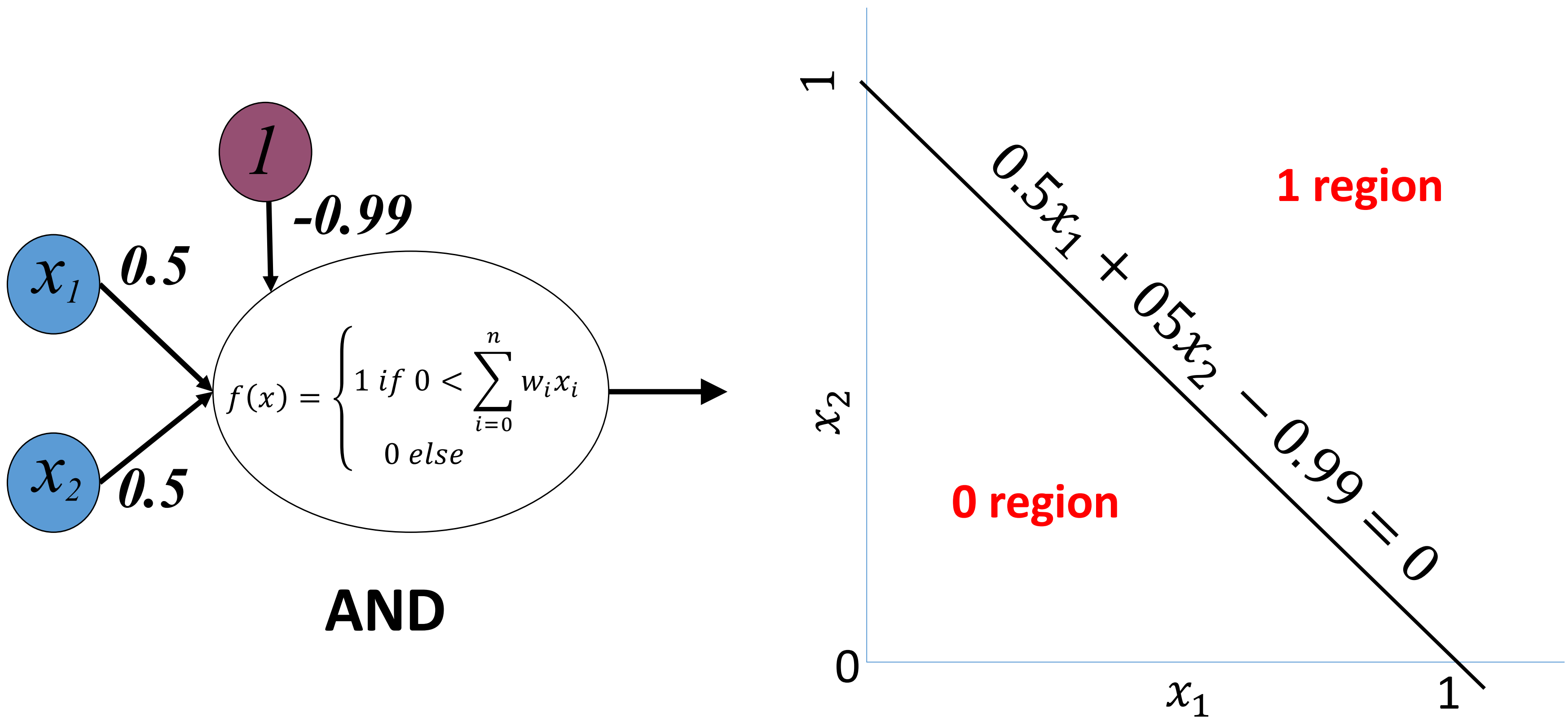
A single perceptron



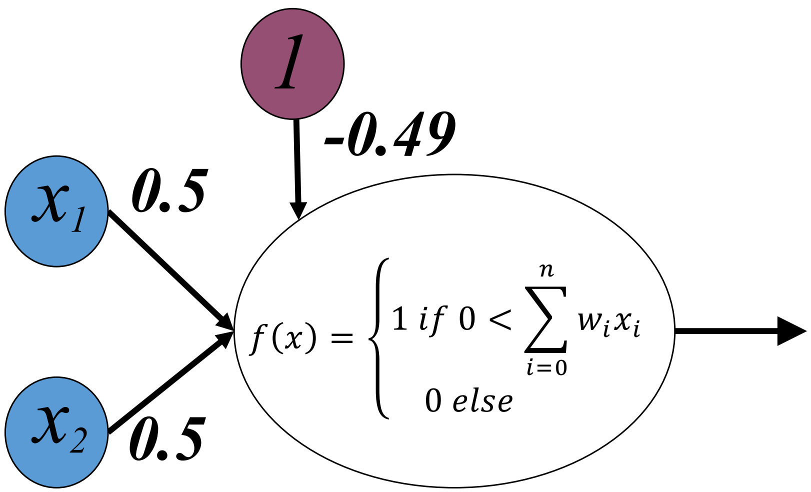
Weights define a hyperplane in the input space



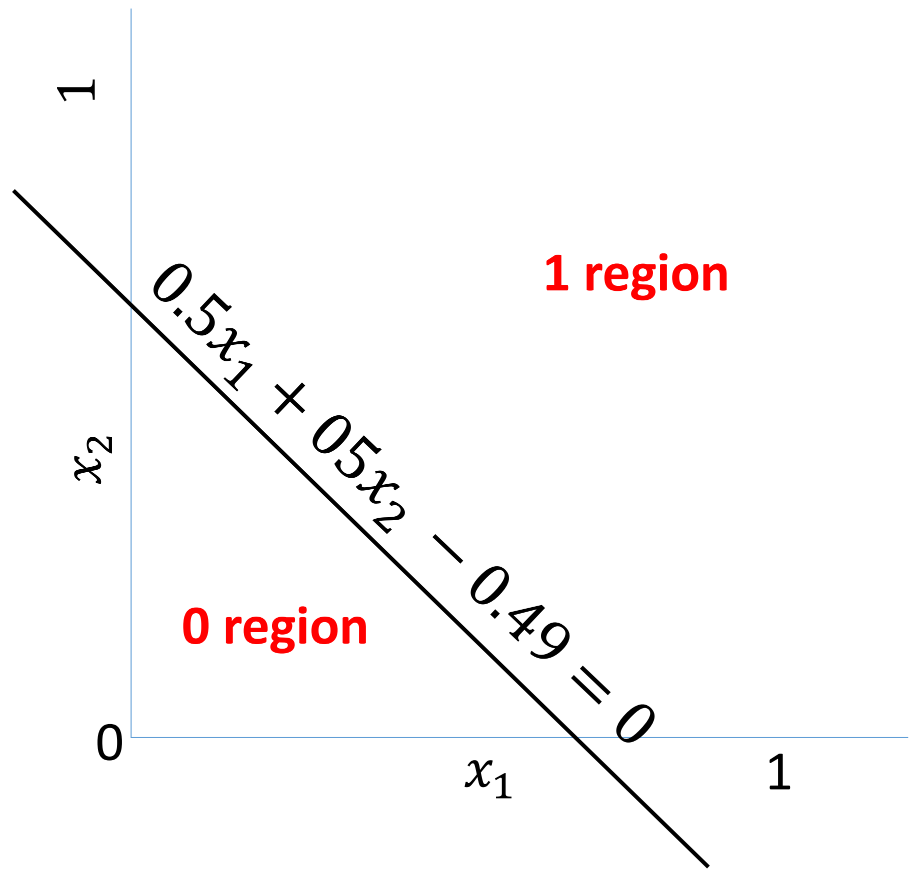
Classifies any (linearly separable) data



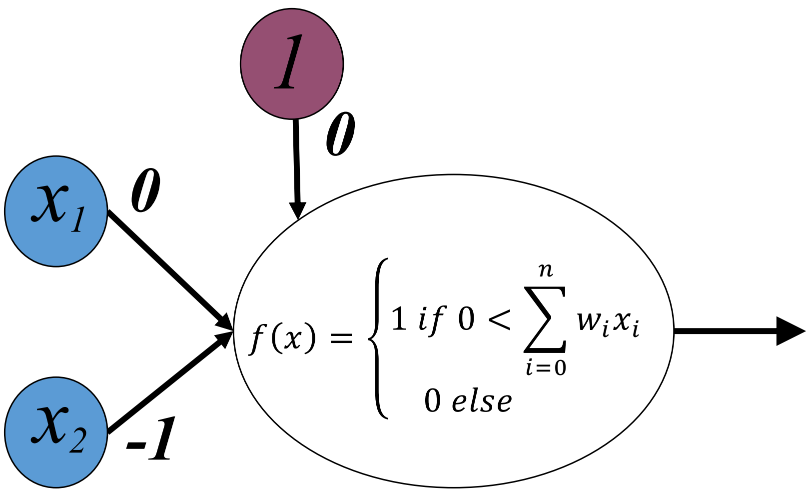
Different logical functions are possible



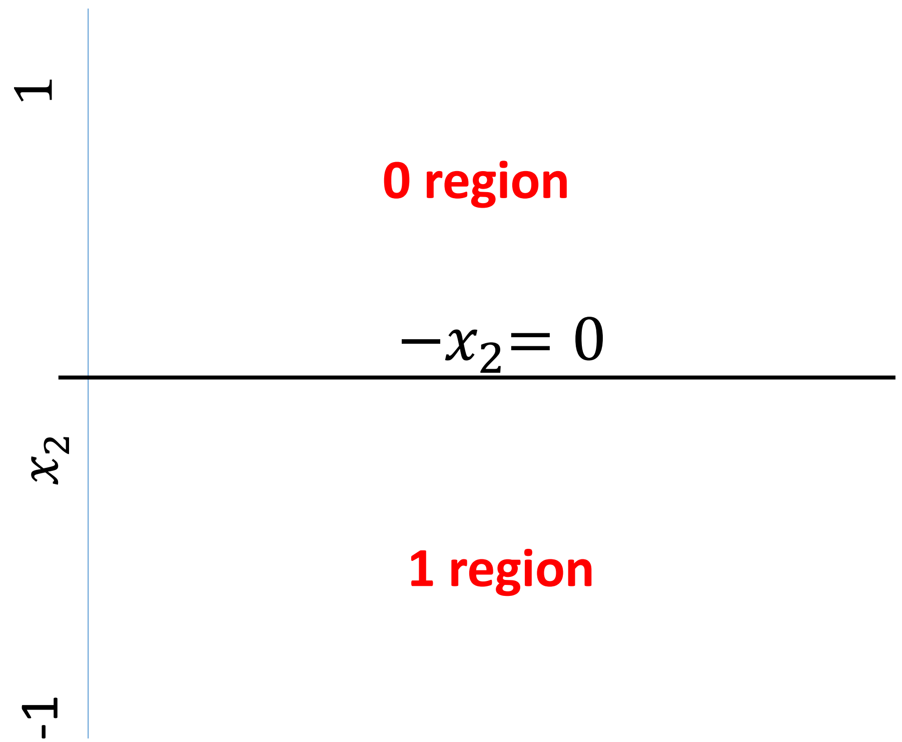
OR



And, Or, Not are easy to define



NOT



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

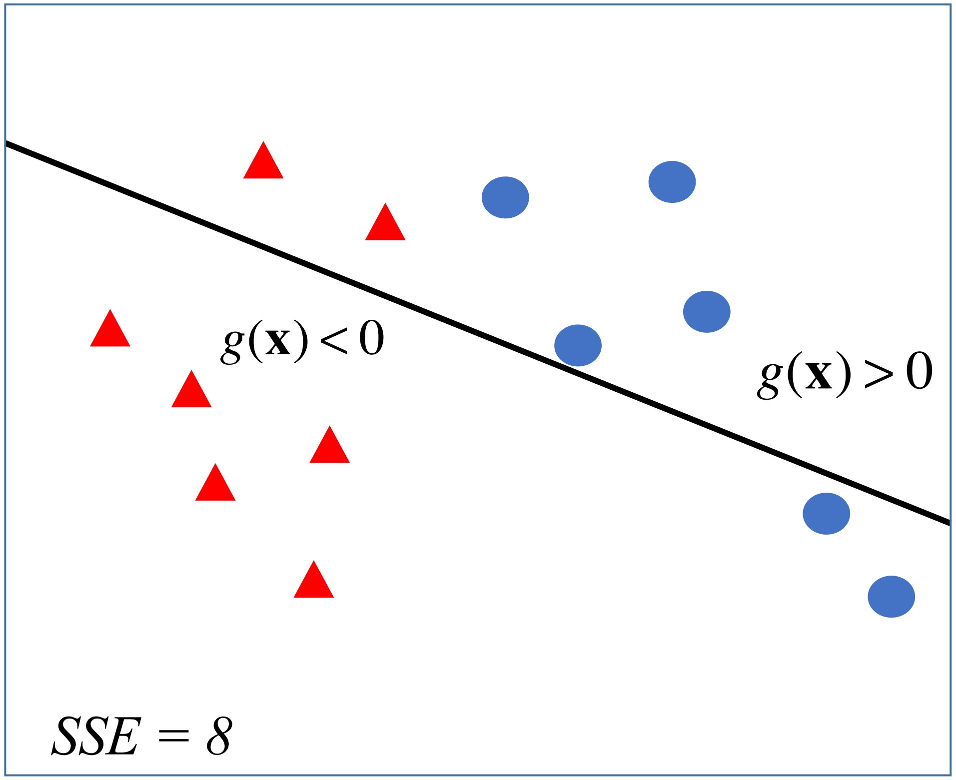
Required for gradient descent

- $J(\mathbf{D}, \mathbf{w})$ 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)

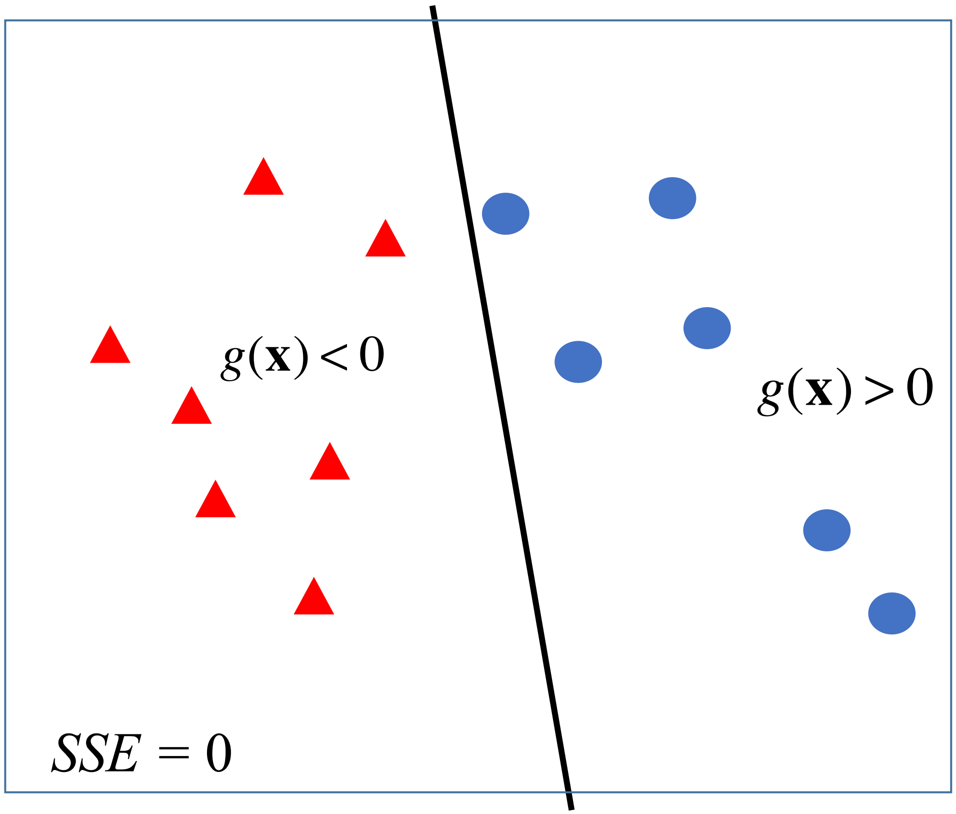


$$g(\mathbf{x}) = w_0 + w_1 x_1 + w_2 x_2 = 0$$

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

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

Example objective J : sum of squared errors (SSE)

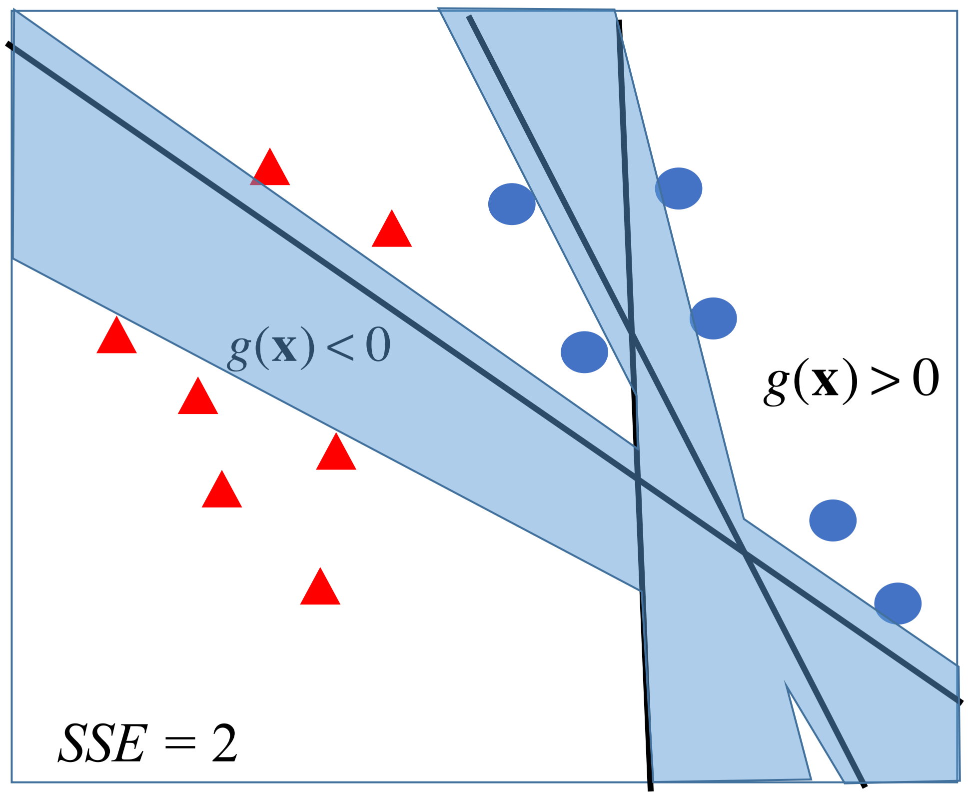


$$g(\mathbf{x}) = w_0 + w_1 x_1 + w_2 x_2 = 0$$

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

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

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



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

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

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

Gradient 0 in the blue region!

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

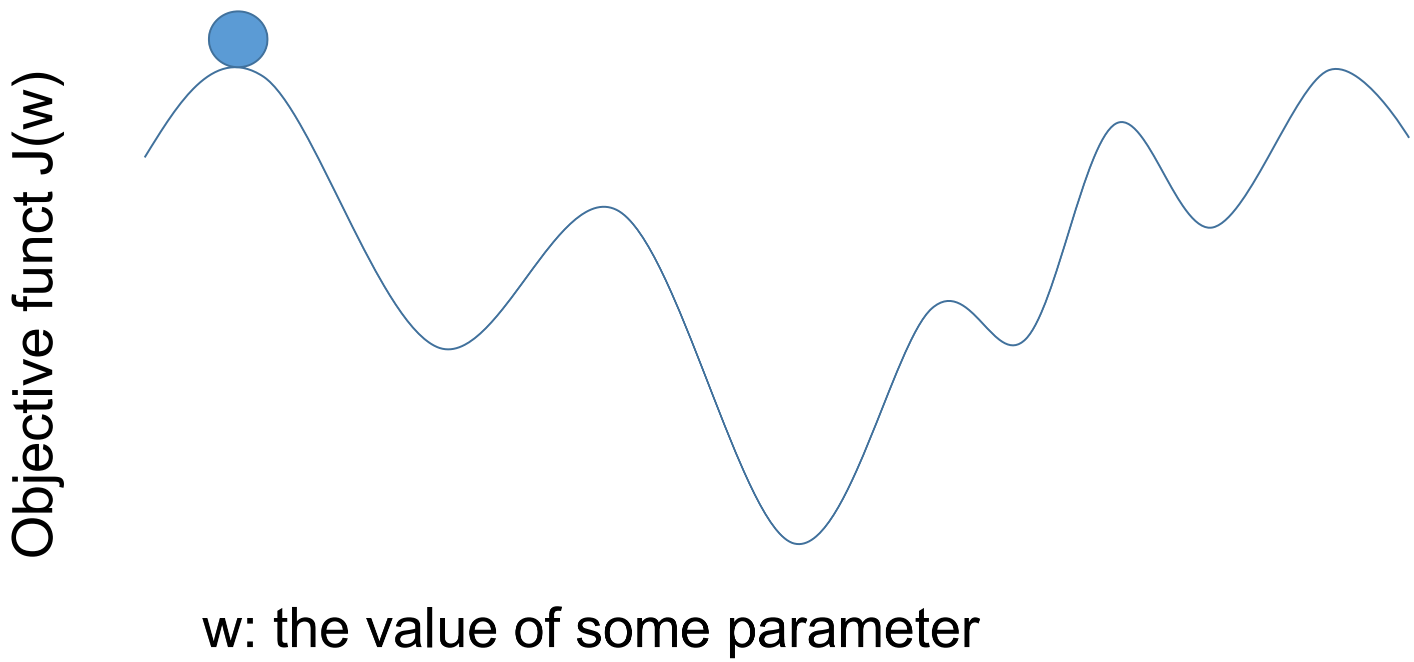
1. Measure how the the objective function changes when we change the $\mathrm{current}$ parameters $\boldsymbol{w}$ slightly (measure the gradient with respect to the weights).

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

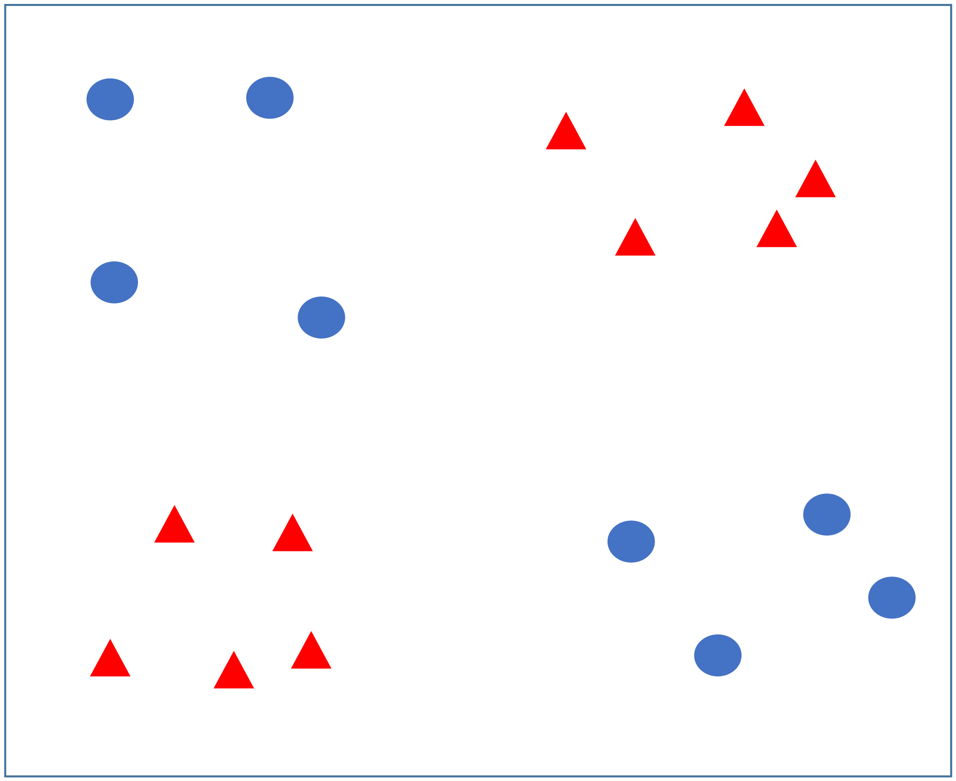
- Much faster than guessing new parameters randomly
- Finds the global optimum only if the objective function is convex



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.

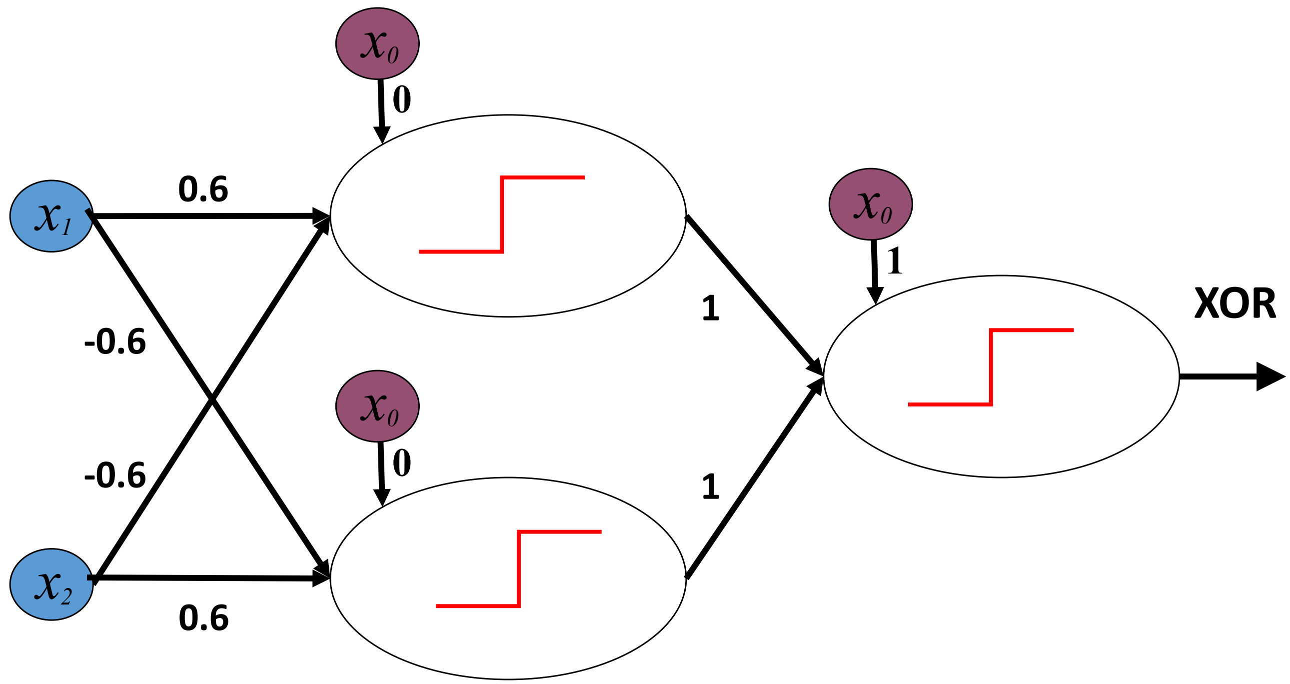
One perceptron: Only linear decisions



This is XOR.

It can't learn XOR.

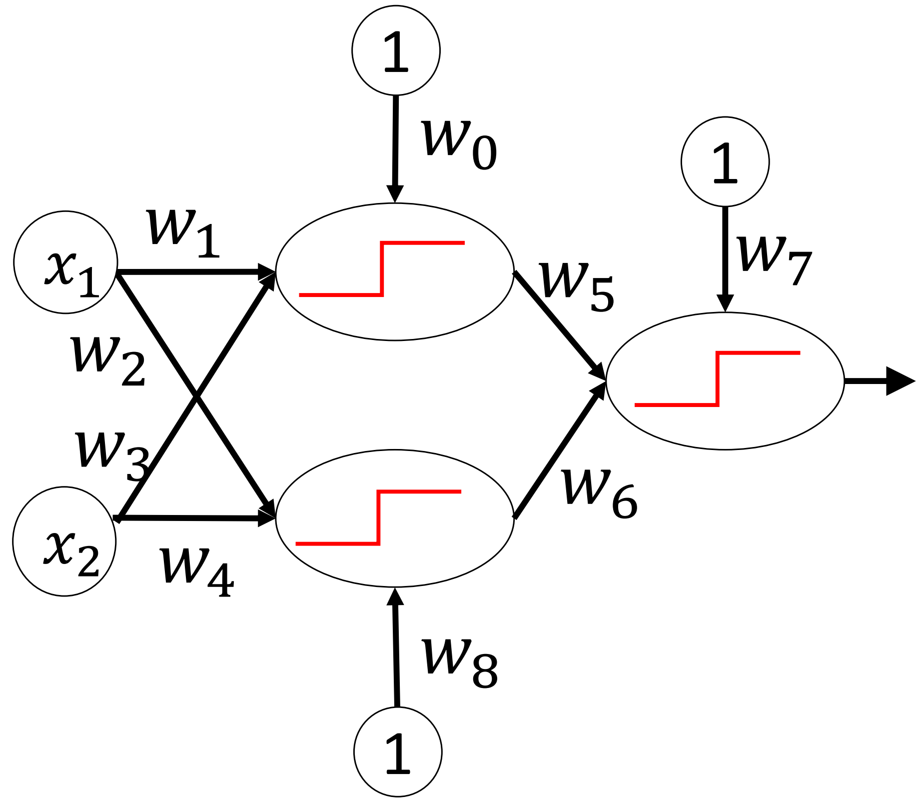
Combining perceptrons can make any Boolean function



…if you can set the weights & connections

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 you can't learn XOR with perceptrons



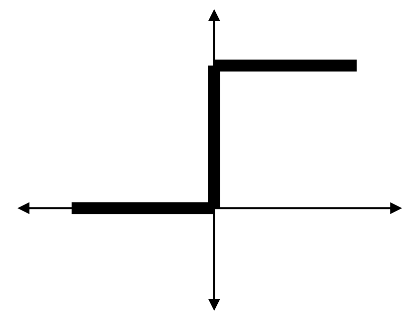
The Sigmoid Unit

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

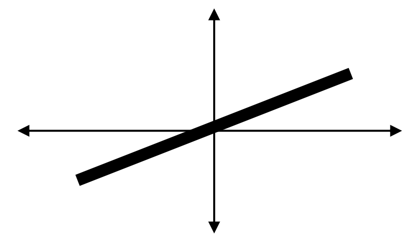
- Perceptron

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



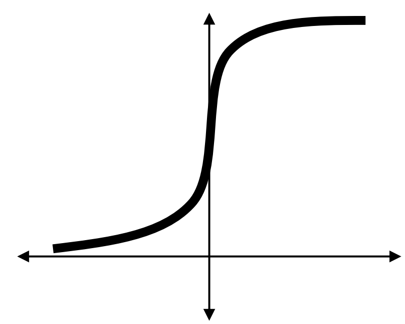
- Linear

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



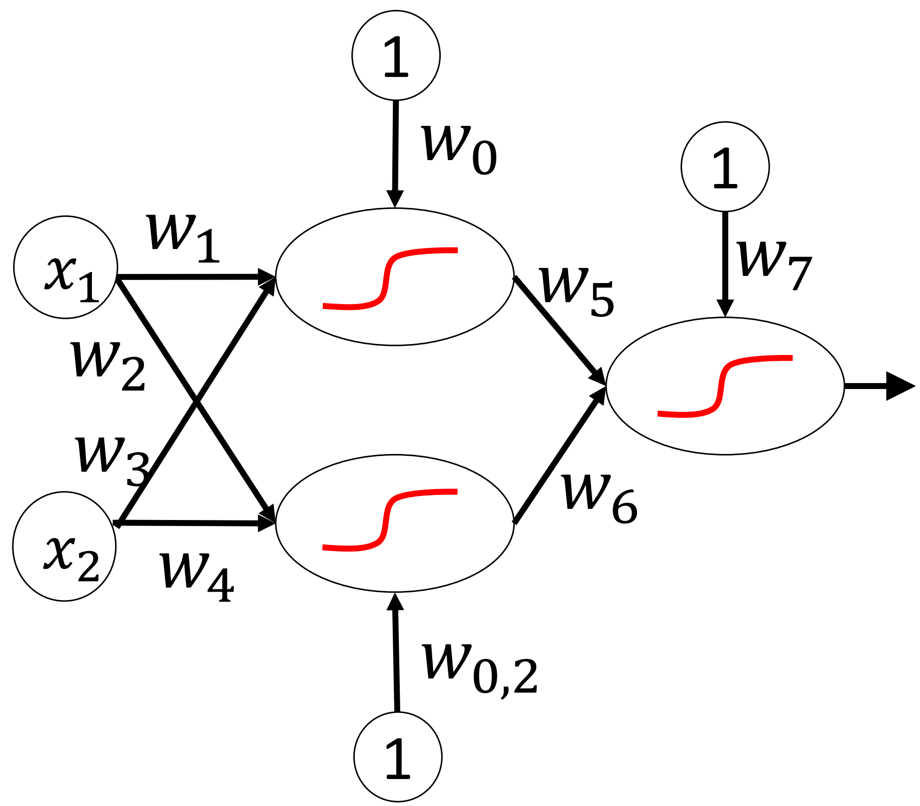
- Sigmoid

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

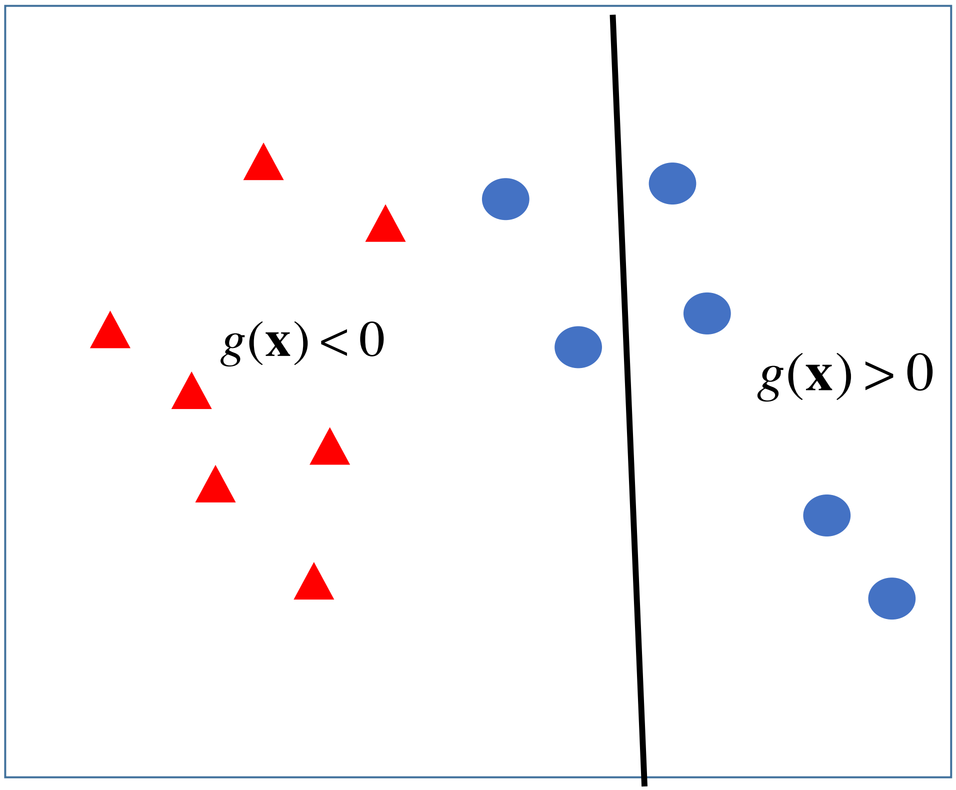


A network of sigmoid units

- 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



Example objective *J* : sum of squared errors

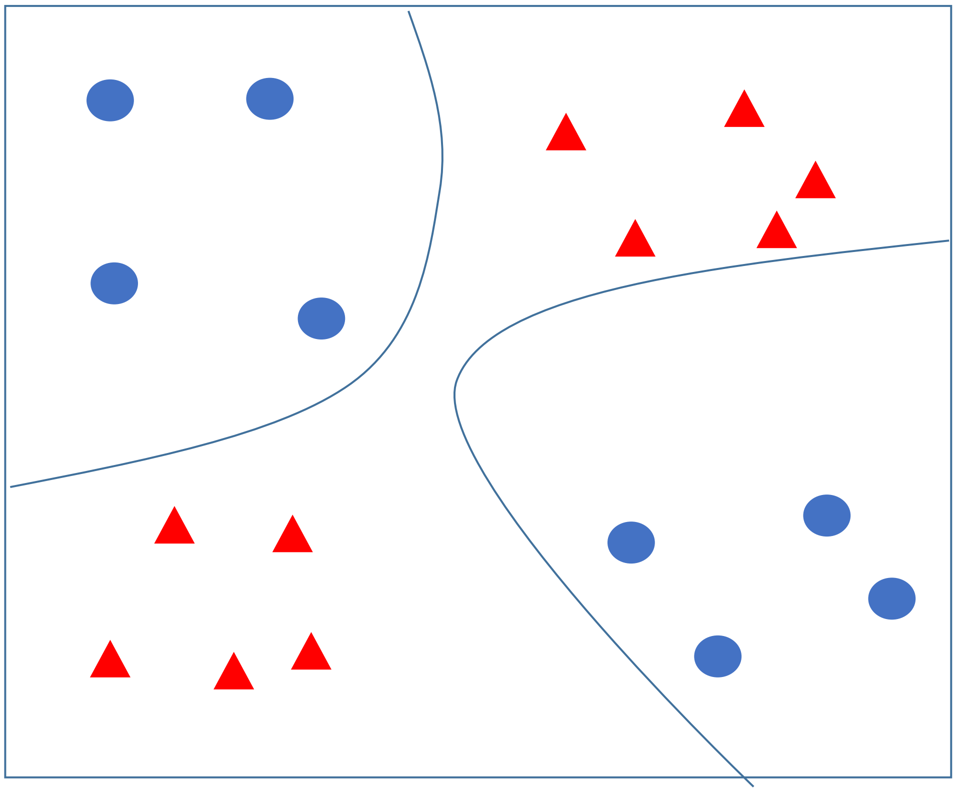


$$h(x) = f(x) = \frac{1}{1 + e^{-(\mathbf{w}^T \mathbf{x})}}$$

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

Gradient non-zero everywhere!

Multilayer Perceptron with sigmoid units



This is XOR.

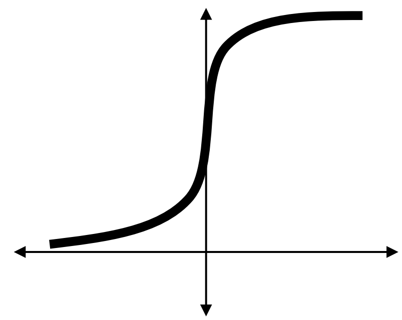
A multilayer perceptron with sigmoid units CAN learn XOR…or any other arbitrary Boolean function.

The promise of many layers

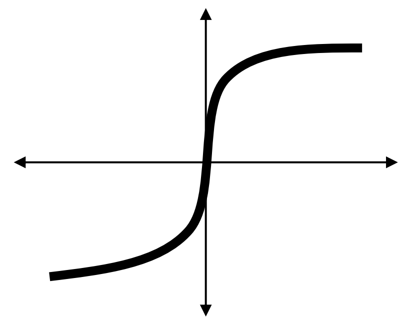
- Each layer learns an abstraction of its input representation (we hope)
-
- As we go up the layers, representations become increasingly abstract
- The hope is that the intermediate abstractions facilitate learning functions that require non-local connections in the input space (recognizing rotated &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

TanH: A shifted sigmoid

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

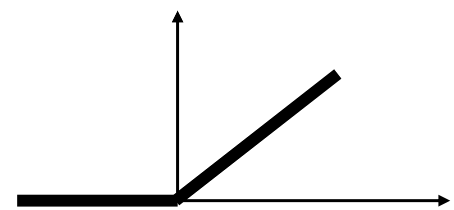


- TanH $$f(x) = \frac{2}{1 + e^{-2(\mathbf{w}^T\mathbf{x})}} - 1$$

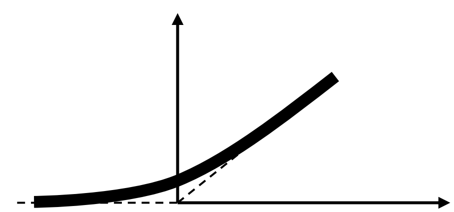


Rectified Linear Unit (ReLU) & Soft Plus :

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



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

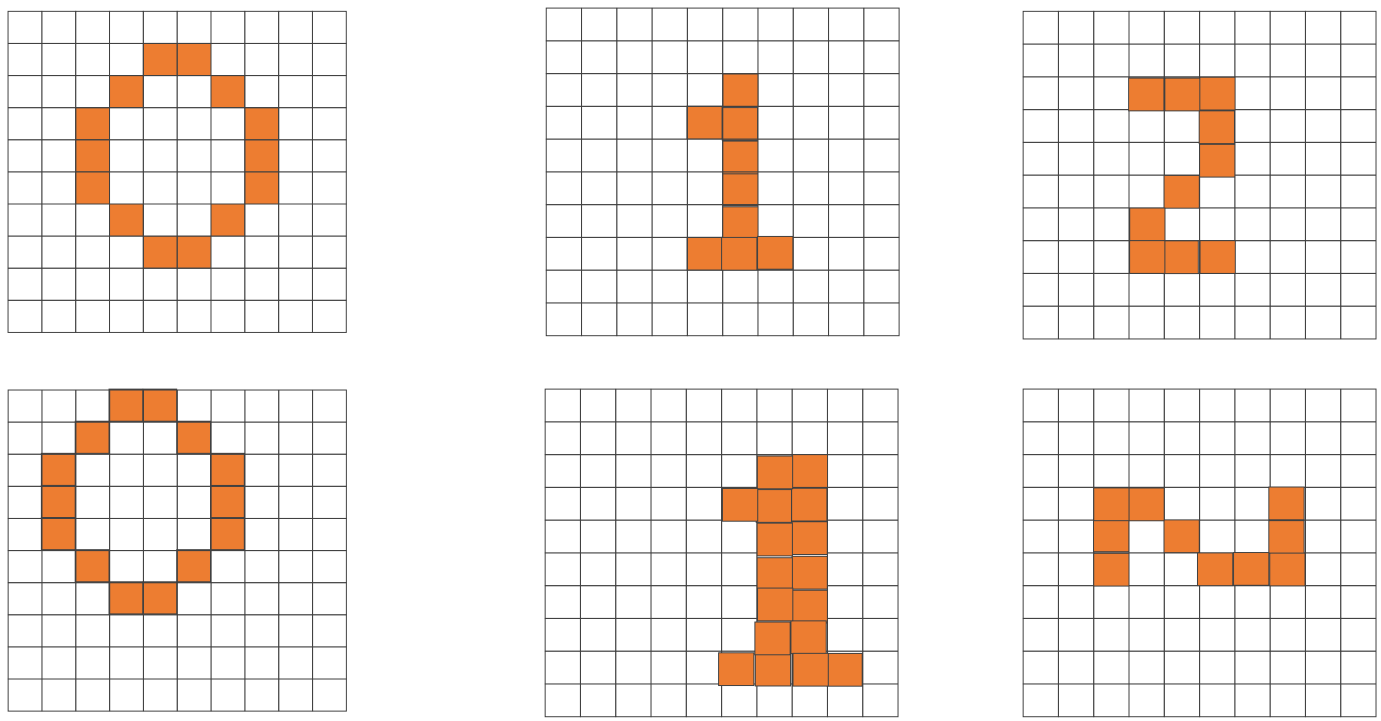


- Both can be combined in layers to make non-linear functions

Design choices

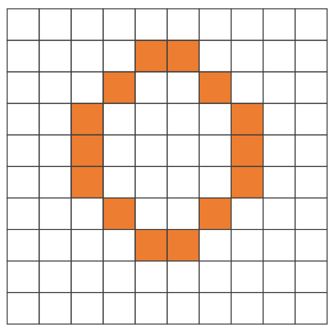
- Define the function you want to learn
- Determine an encoding for the data
- Pick a network architecture
  - Number of layers (between 3 and 100)
  - Activation functions function (tanh,ReLU, linear)
  - Select how units connect within and between layers
- Pick a gradient descent algorithm
- Pick regularization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

Classifying images of digits



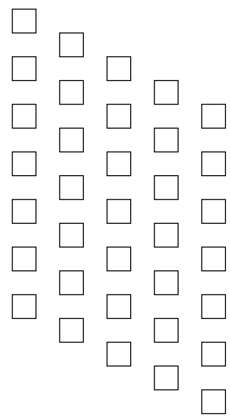
One possibility

INPUT LAYER



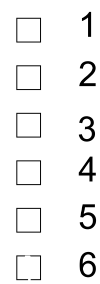
One input
per pixel

HIDDEN LAYER



One hidden node per
potential shifted image
(ReLU)

OUTPUT LAYER

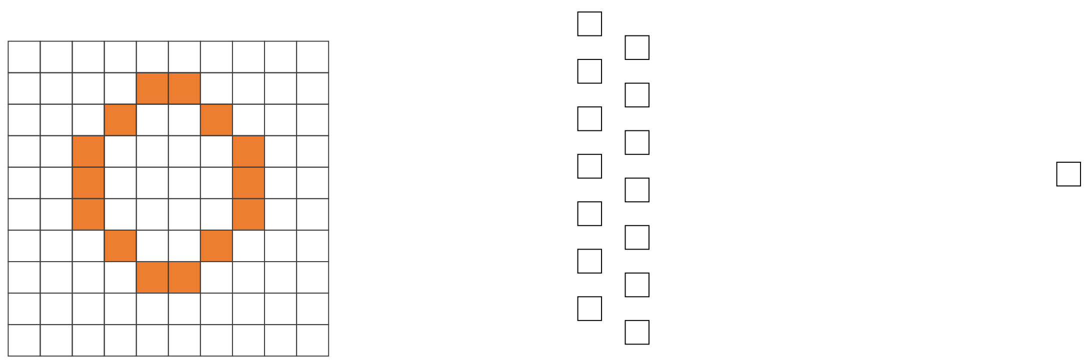


One output node
per category
(Sigmoid)

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



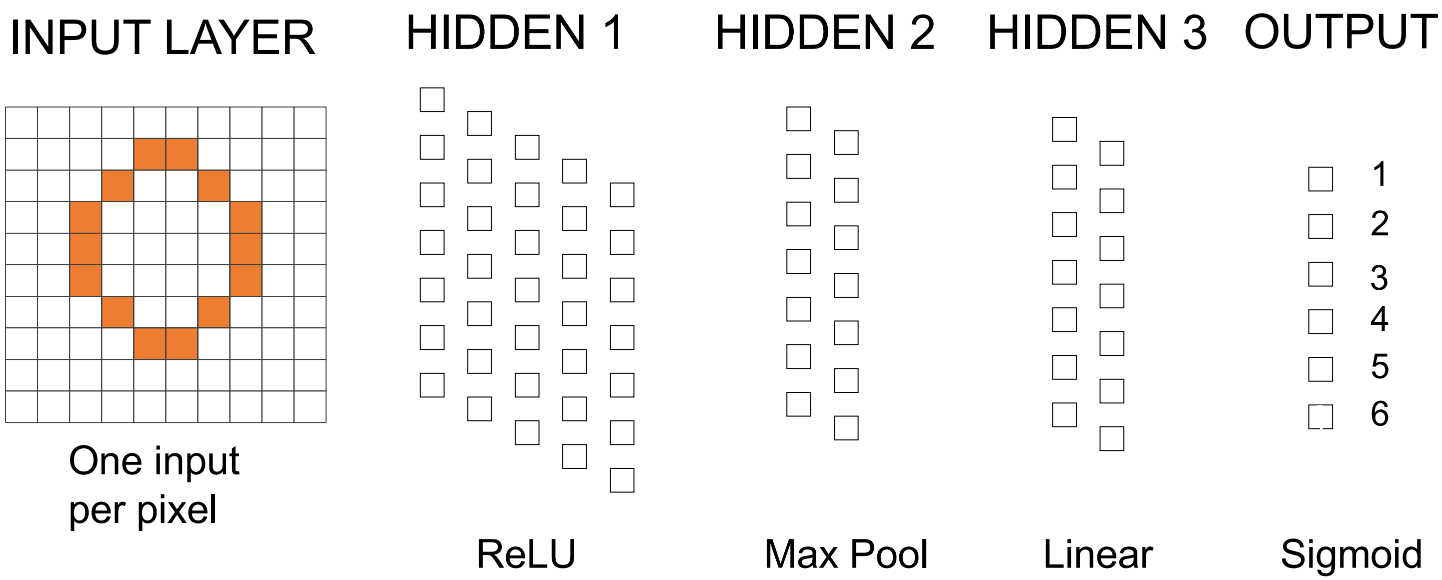
One input per pixel

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



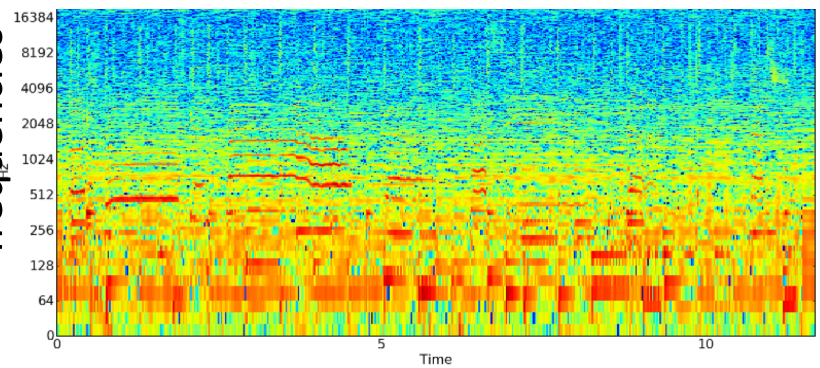
HUGE DESIGN SPACE!

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 magnitude spectrogram?

1024 frequencies



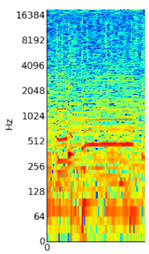
600 time steps

$= 614{,}400$ inputs

30 seconds of audio
20 frames per second
22.5kHz
FFT padded to 2048
only frequencies below Nyquist

How many weights in a fully connected net?

M = 1024



N = 60

Input Layer		Hidden Layer		Hidden Layer		Output Layer
61440 Nodes	$(61440 \cdot 1000)$ connections	1000 Nodes	$(1000 \cdot 100)$ connections	100 Nodes	$(100 \cdot 2)$ connections	2 Nodes

$$61{,}444{,}000 + 100{,}000 + 200 = 61{,}544{,}200 \text{ weights}$$

To reduce the number of connections
Use a 3 second window, instead of 30

Small Fixed Windows (filter size/receptive field)

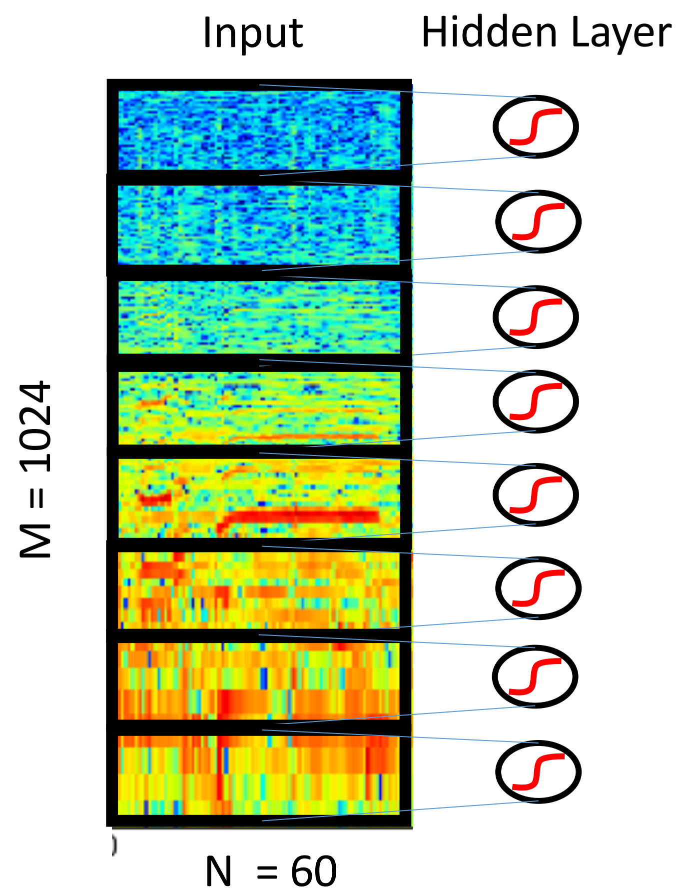
- If important relationships in the input fall within a bounded region.
- Then we can bound the receptive field of each node to a fixed region size

Fully connected between layers

61440 inputs * 8 nodes = 491,520 weights

Receptive field on 1/8 of the input

7680 inputs * 8 nodes = 61440 weights



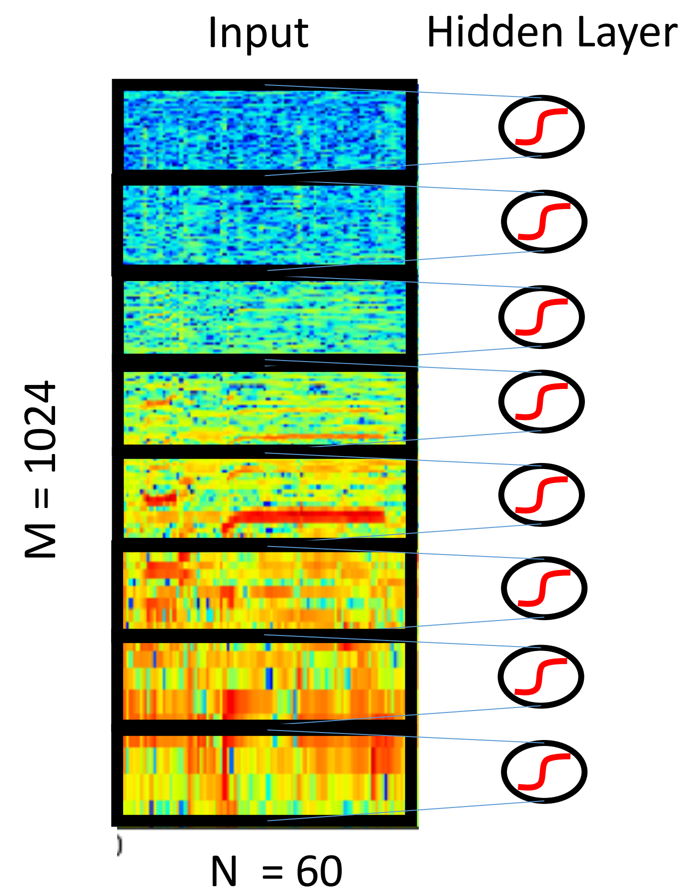
A feature map (AKA a "channel")

- If a feature is good to find in one region, they may be good to find in other regions.
- Units looking at different sub-regions of the input will look for the same feature if they share weights.
- A set of nodes that share connection weights is a feature map

491520: fully connected

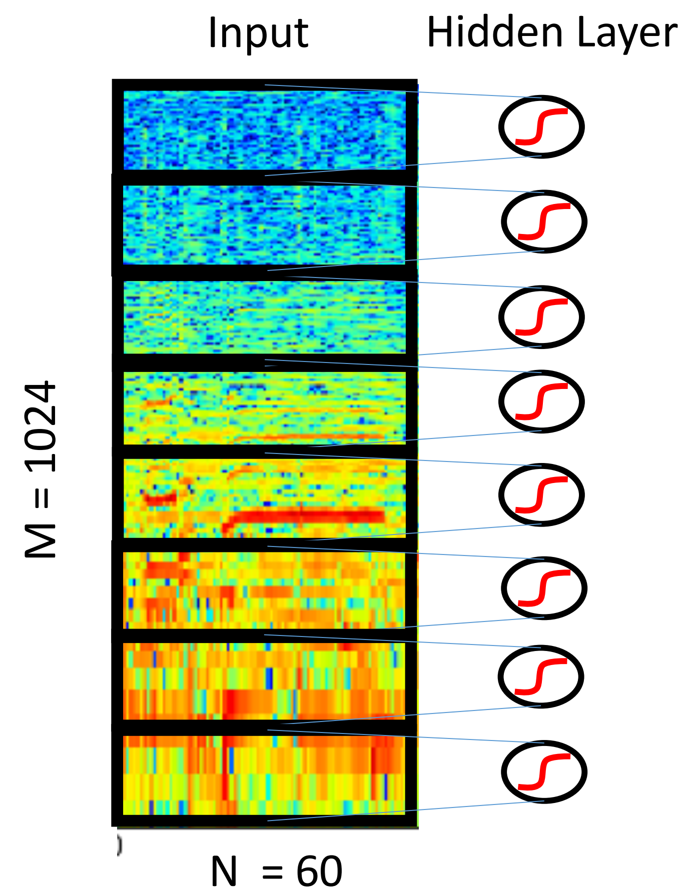
61440: limited receptive field

7680: limited field + shared weights



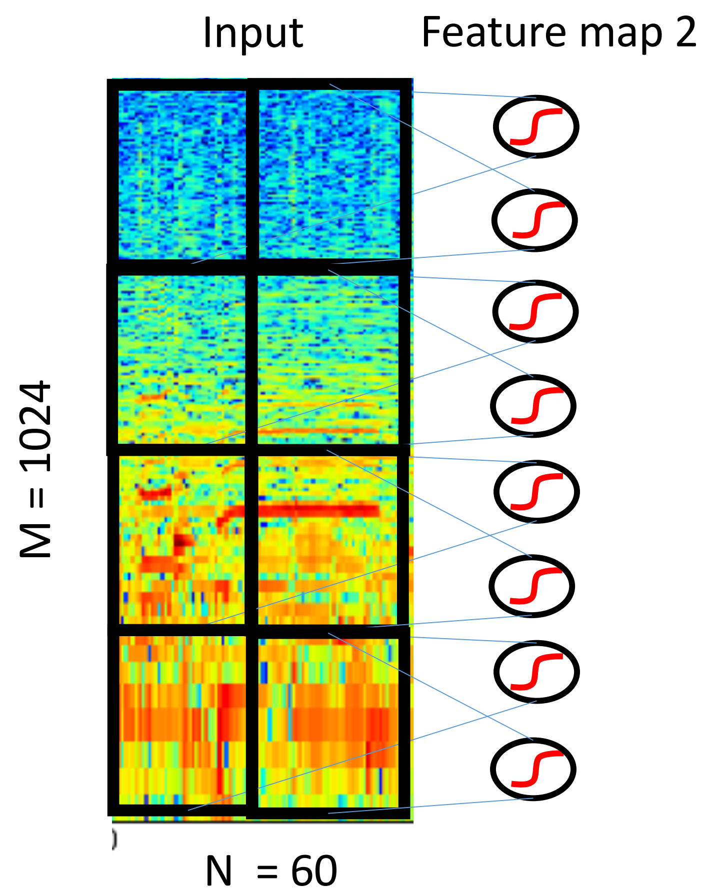
Multiple feature maps

- 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



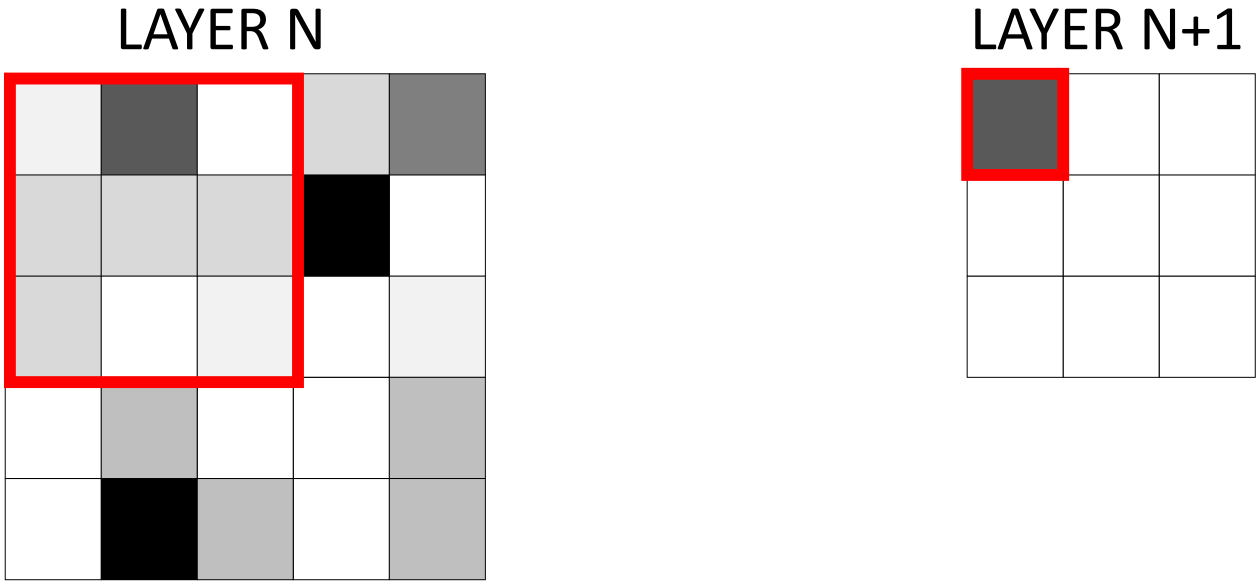
Multiple feature maps

- 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



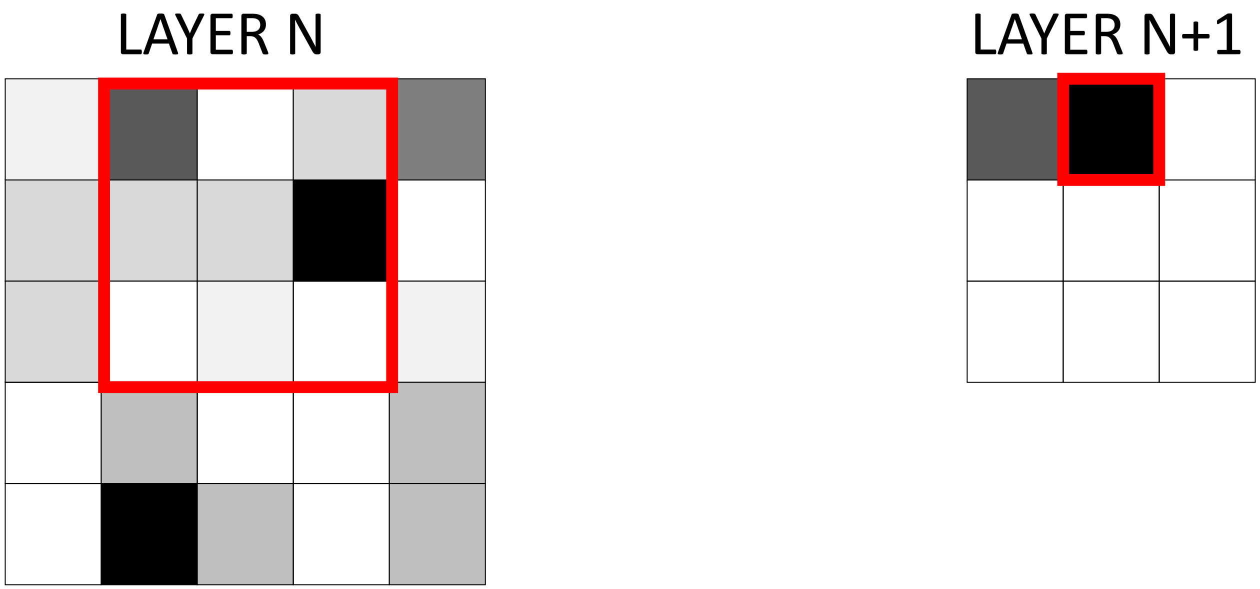
Max Pool Layer: A kind of downsampling

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



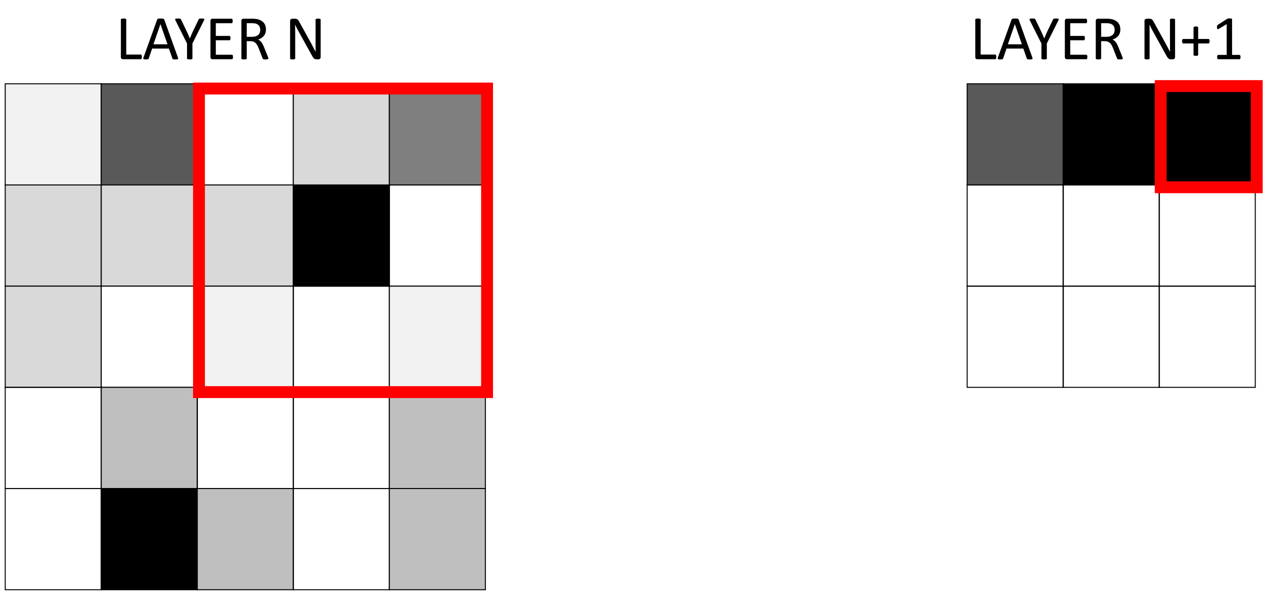
Max Pool Layer: A kind of downsampling

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



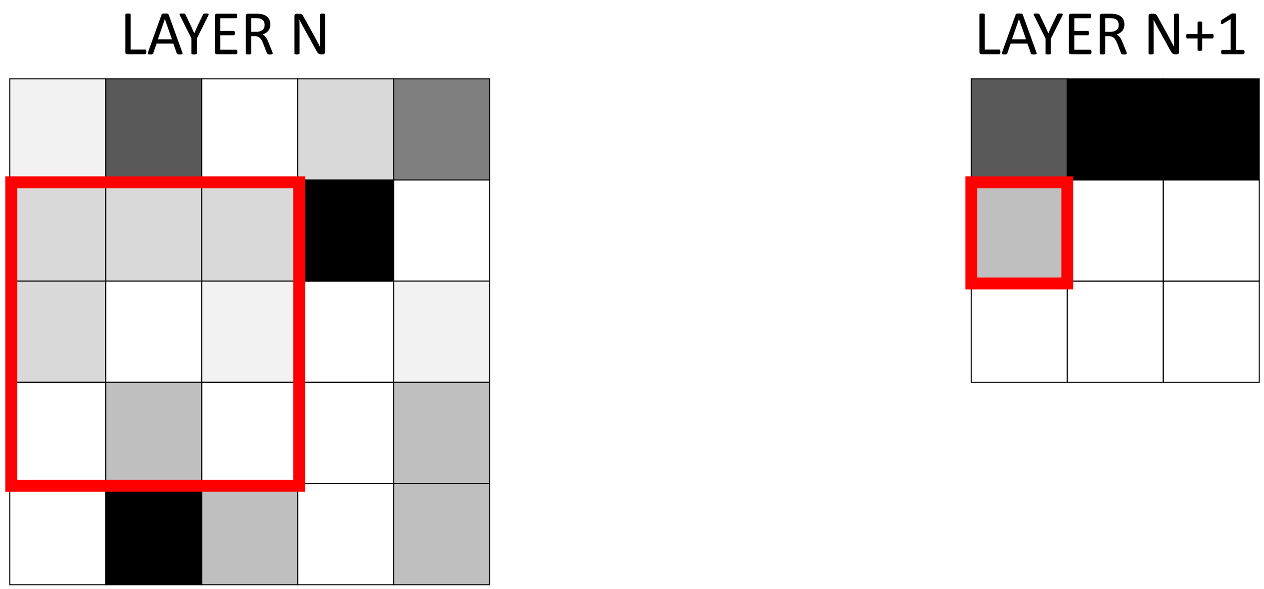
Max Pool Layer: A kind of downsampling

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



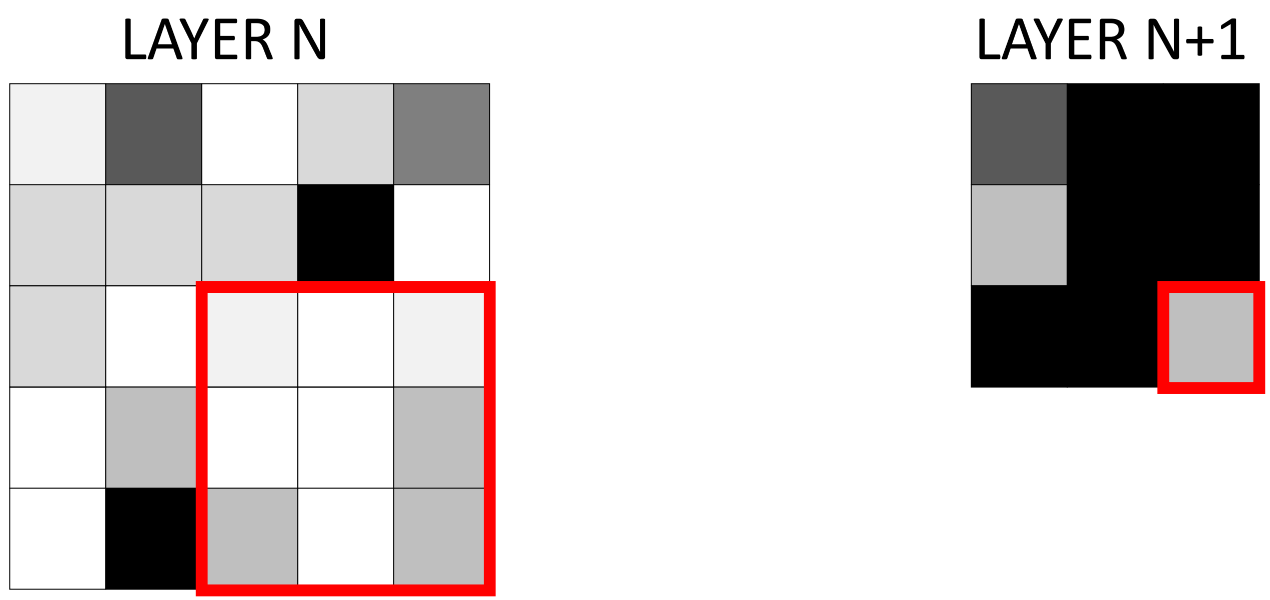
Max Pool Layer: A kind of downsampling

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



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?

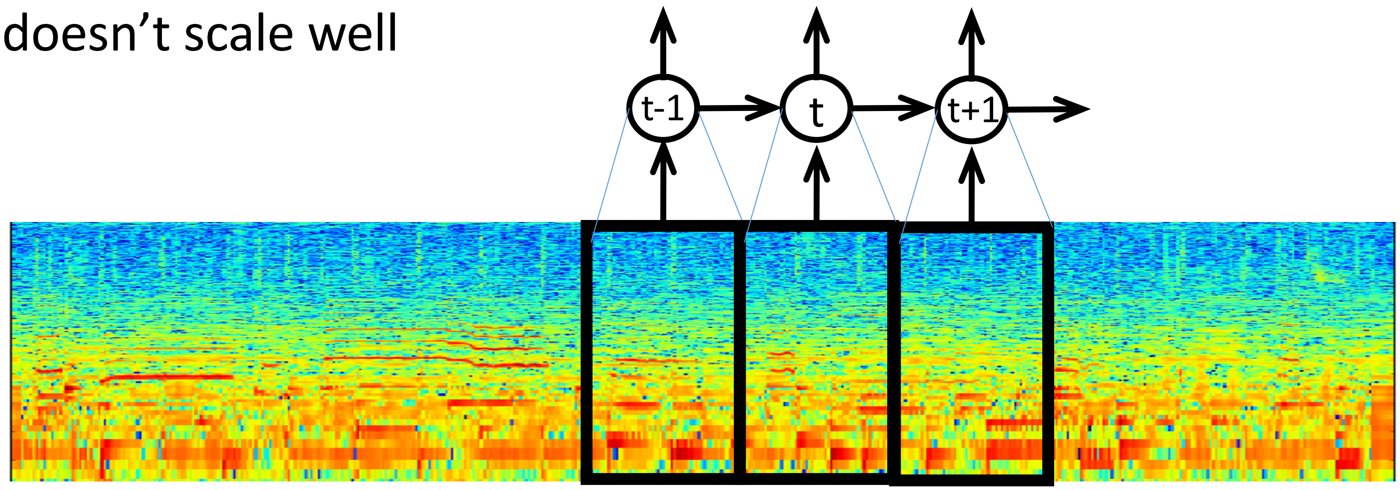
- A network with one or more layers that are feature maps
- A layer with feature maps is called a “convolutional layer”
- Often, convolutional layers are alternated with pooling layers.
- Since these nets have many fewer connections
  - They train faster
  - They need fewer training examples

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.

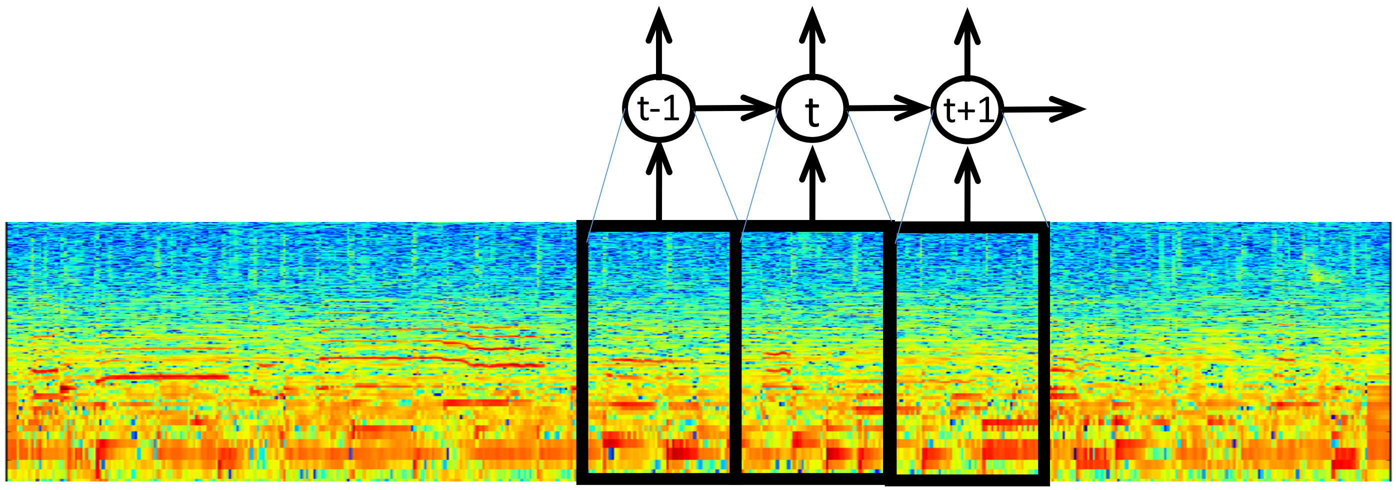
Dealing with time

- 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



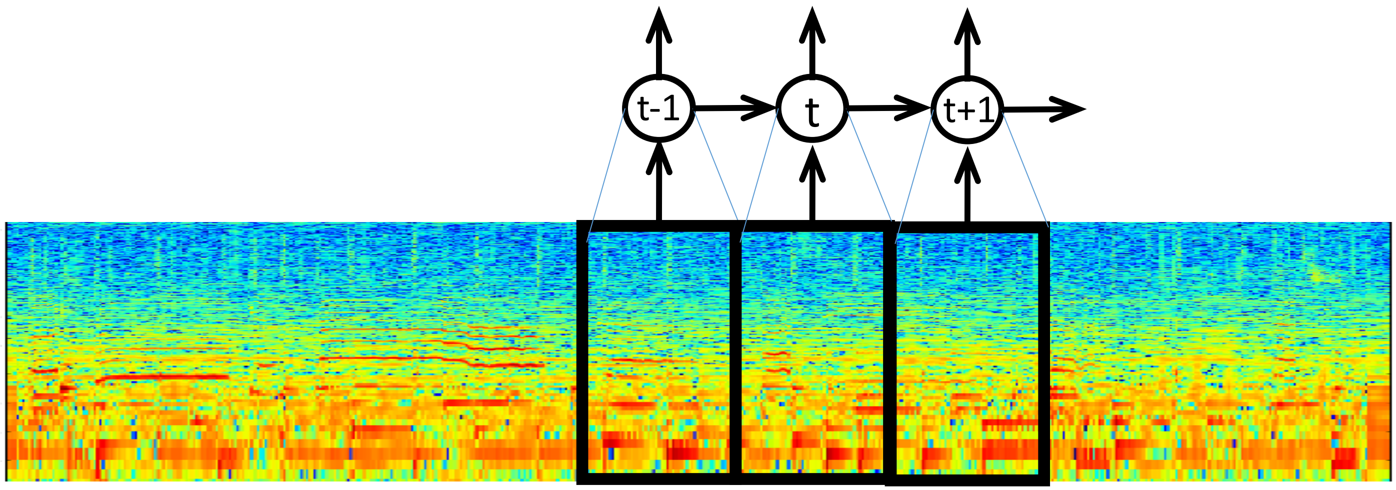
Weight sharing

- 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.



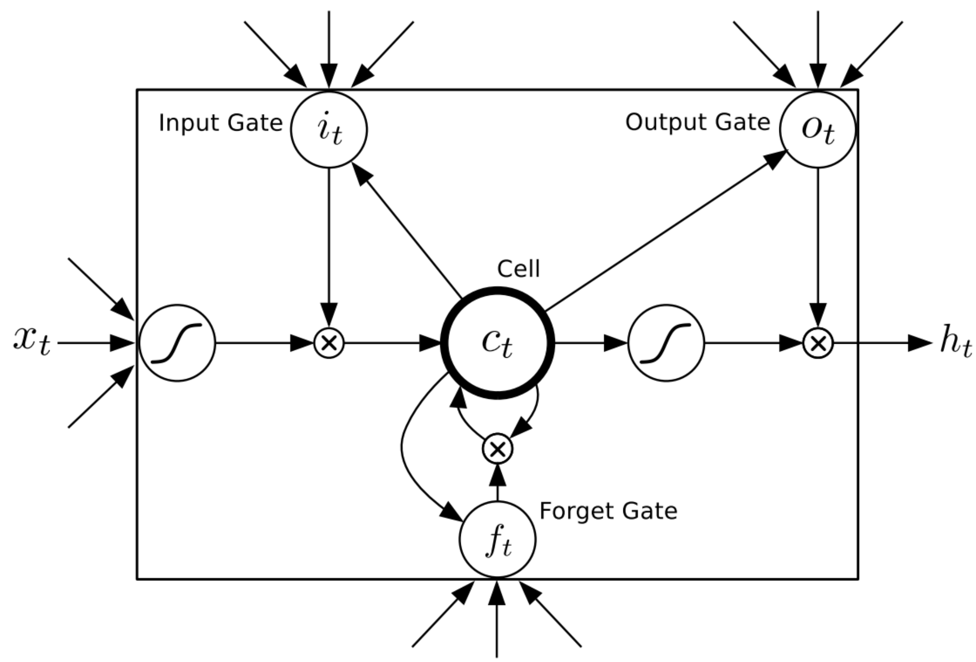
Exponentially decaying influence

- 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



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