AI Programming

CS662-2008F-20

Neural Networks

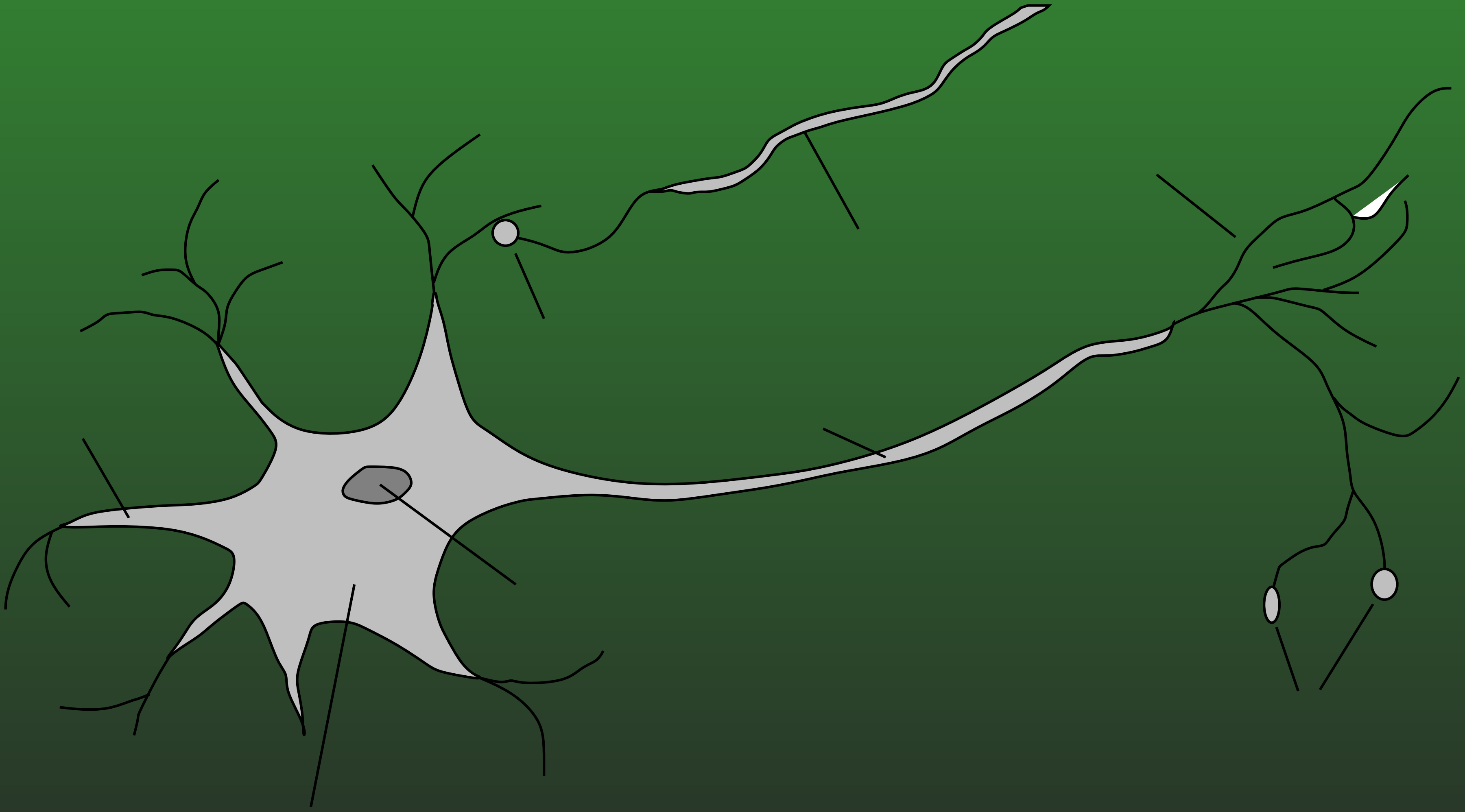
David Galles

Department of Computer Science
University of San Francisco

Symbolic AI

- Most of this class has been focused on *Symbolic AI*
    - Focus or symbols and relationships between them
    - Search, logic, decision trees, etc.
- Assumption: Key requirement for intelligent behavior is the manipulation of symbols
- Neural networks are a little different: *subsymbolic* behavior

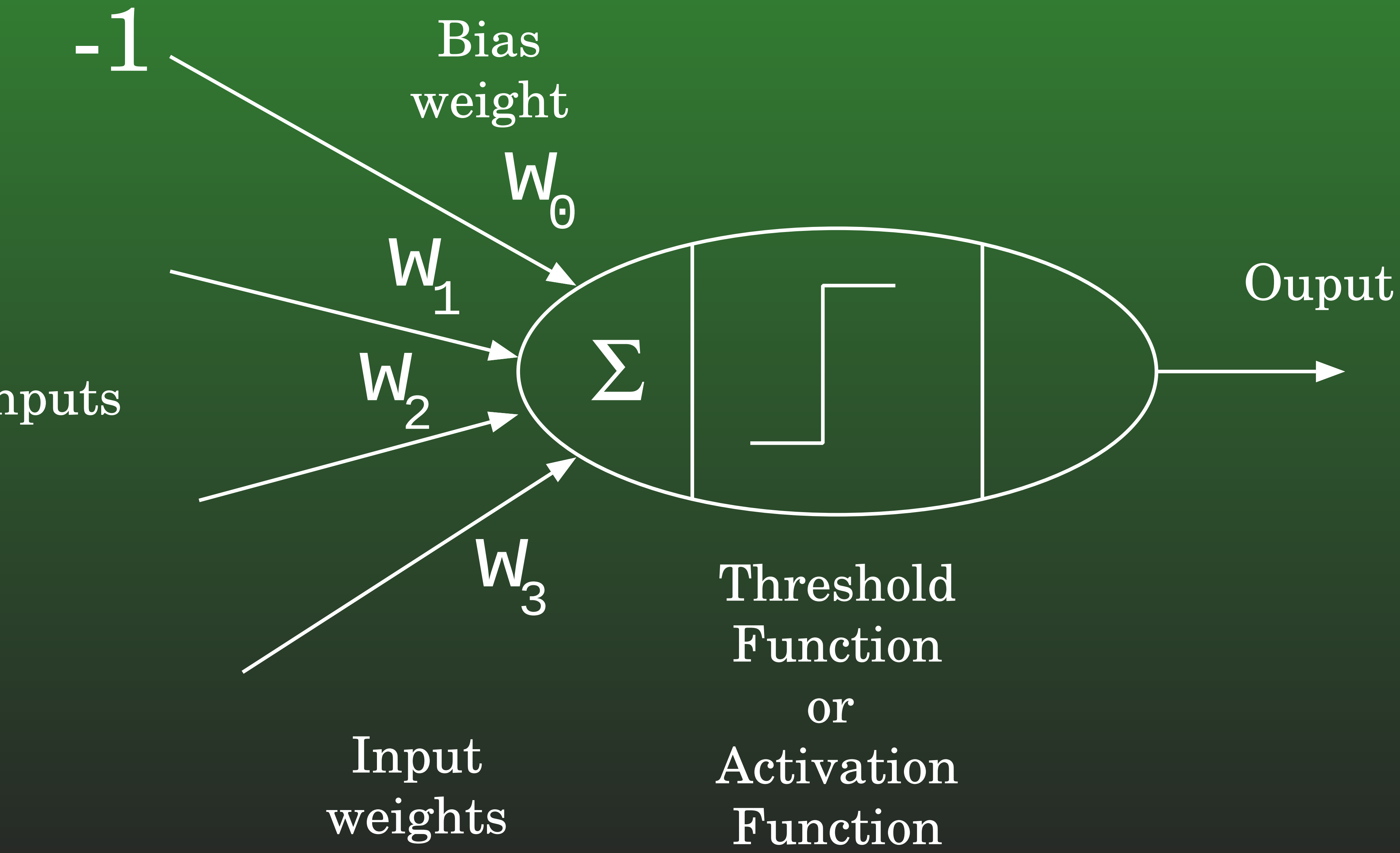
20-1: Biological Neurons



20-2: Biological Neurons

- Biological neurons transmit (more-or-less) electrical signals
- Each Nerve cell is connected to the “outputs” of several other neurons
- When there is a sufficient total input from all inputs, the cell “fires”, and sends ouputs to other cells
- Extreme oversimplification, simple version is the model for Artificial Neural Networks

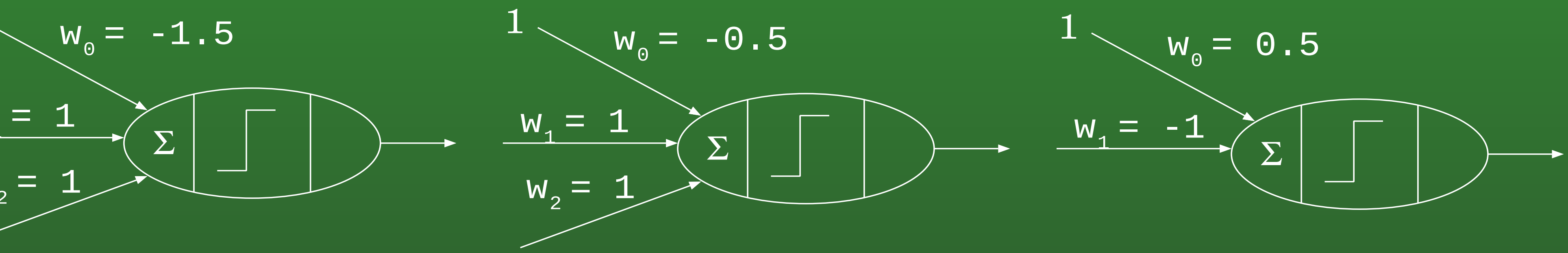
20-3: Artificial Neural Networks



20-4: Activation Function

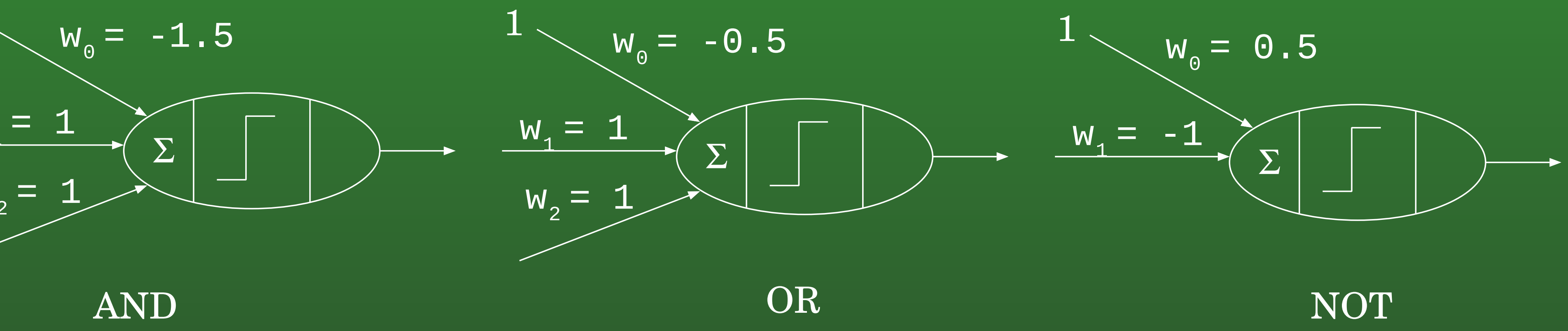
- Neurons are mostly binary
  - Fire, or don’t fire
  - Not “fire at 37%”
- Model this with an activation function
  - Step Function
  - Sigmoid function: $f(x) = \frac{1}{1+e^{-x}}$
- Talk about why the sigmoid function can be better than the step function when we do training

20-5: Single Neuron NNs



- A single Neuron can compute a simple function
- What functions do each of these neurons compute?

20-6: Single Neuron NNs

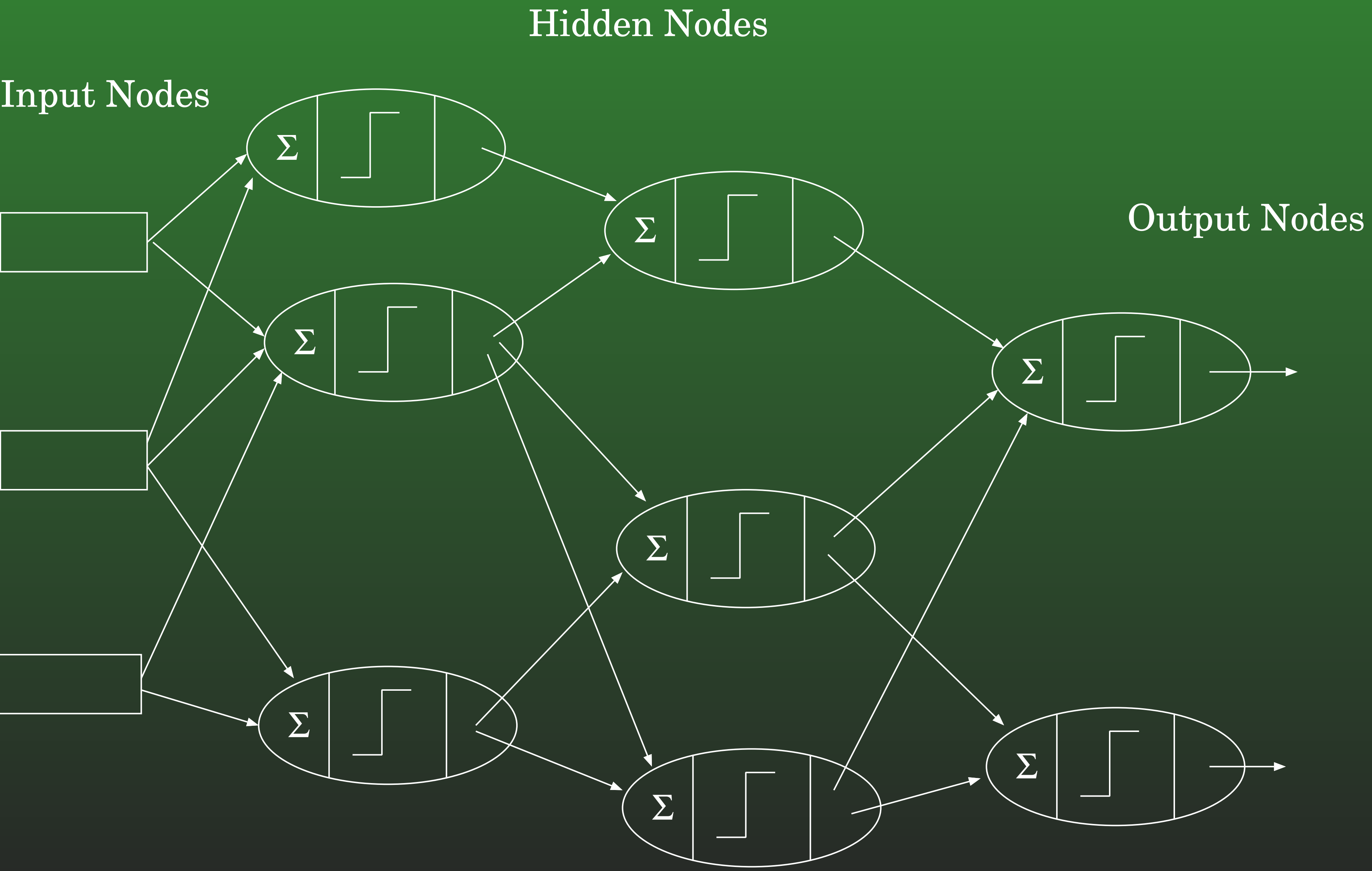


- A single Neuron can compute a simple function
- What functions do each of these neurons compute?

20-7: Neural Networks

- Of course, things get more fun when we connect individual neurons together into a network
    - Ouputs of some neurons feed into the inputs of other neurons
- Add special “input nodes”, used for input

20-8: Neural Networks



Neural Networks

- Feed Forward Networks
  - No cycles, signals flow in one direction
- Recurrent Networks
  - Cycles in signal propagation
  - Much more complicated: Need to deal with time, learning is much harder
- We will focus on Feed Forward networks

Function Approximators

- Feed Forward Neural Networks are *Nonlinear Function Approximators*
- Output of the network is a function of its inputs
- Activation function is non-linear, allows for representation of non-linear functions
- Adjust weights, change function
- Neural Networks are used to efficiently approximate complex functions

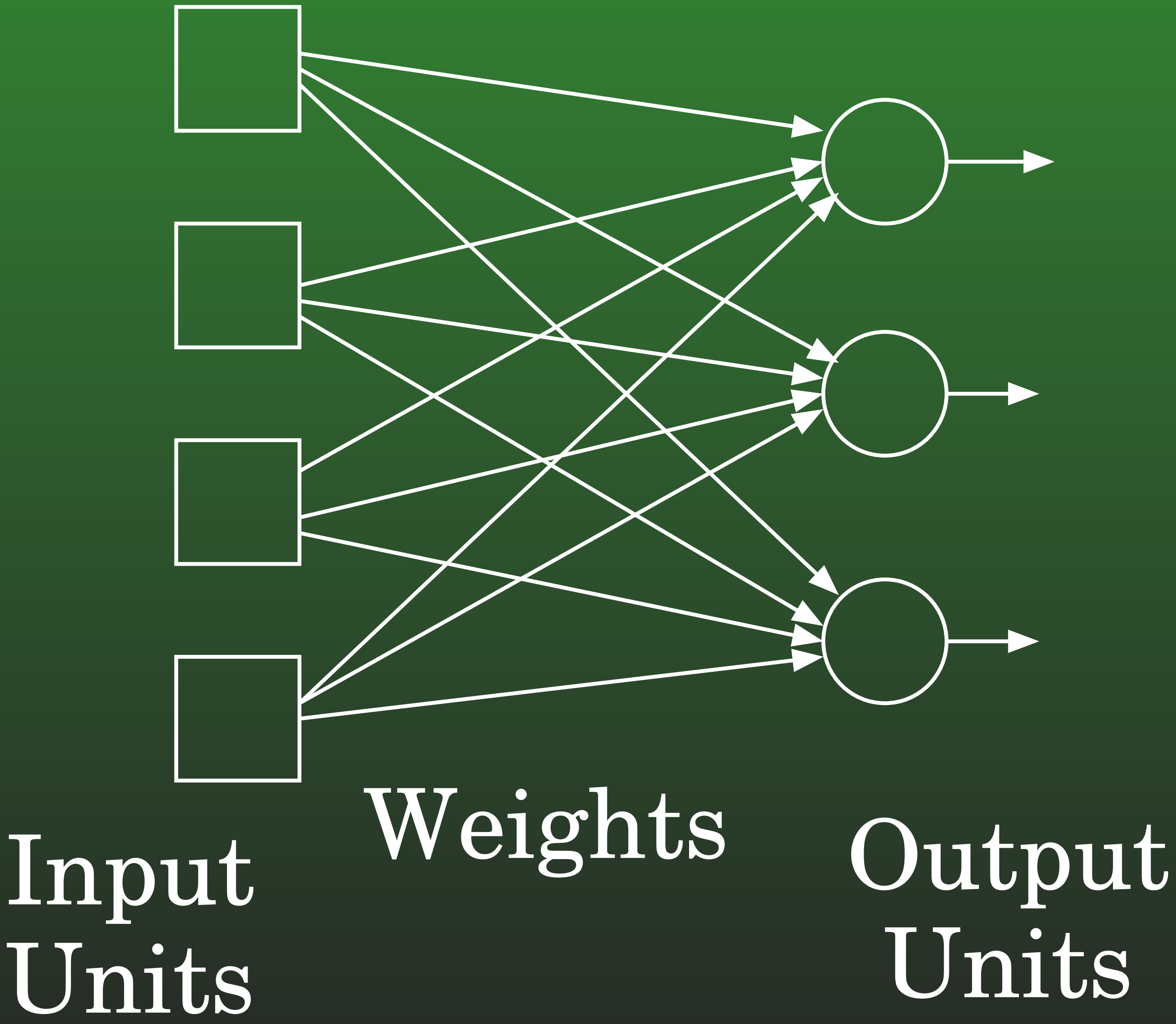
20-11: Classification

- Common use for Neural Networks is classification
    - We've already seen classification with decision trees and naive bayes
    - Map inputs into one or more outputs
    - Output range is split into discrete "classes" (like "spam" and "not spam"
- Useful for learning tasks where "what to look for" is unknown
    - Face recognition
    - Handwriting recognition

20-12: Perceptrons

- Feed Forward
- Single-layer network
- Each input is directly connected to one or more outputs

20-13: Perceptrons



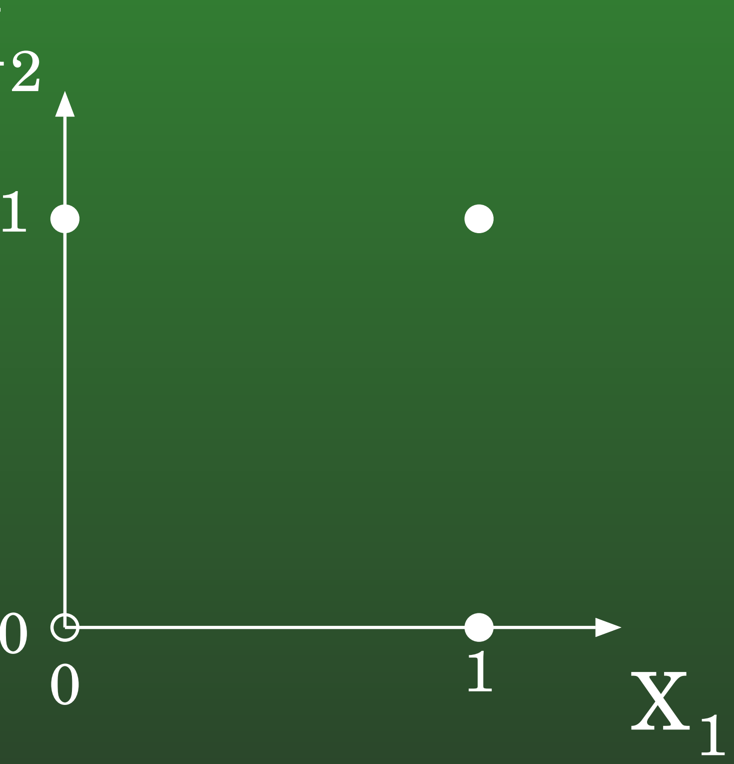
Perceptrons

- For each perceptron:
  - Threshold firing function is used (not sigmoid)
  - Output function o:
    - $o(x_1, \ldots x_n) = 1$ if $w_o + w_1x_1 + w_2x_2 + \ldots + w_nx_n > 0$
    - $o(x_1, \ldots x_n) = 0$ if $w_o + w_1x_1 + w_2x_2 + \ldots + w_nx_n \leq 0$

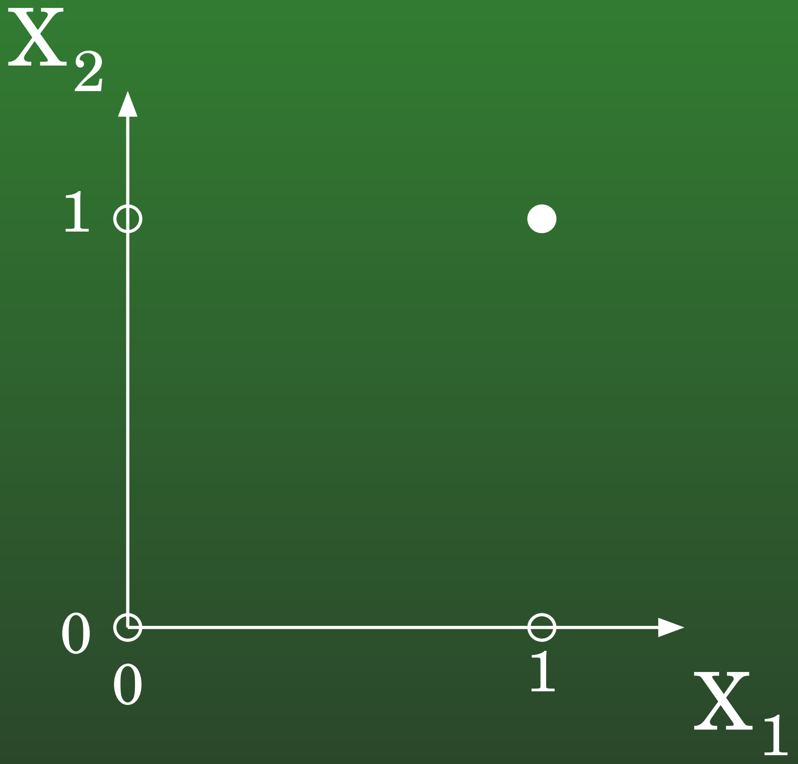
20-15: Perceptrons

- Since each perceptron is independent of others, we can examine each in isolation
- Output of a single perceptron:
  - $\sum_{j=1}^{n} W_j x_j > 0$
  - (or, $W \cdot x > 0$)
- Perceptrons can represent any linearly separable function
- Perceptrons can *only* represent linearly separable functions

20-16: Linearly Seperable

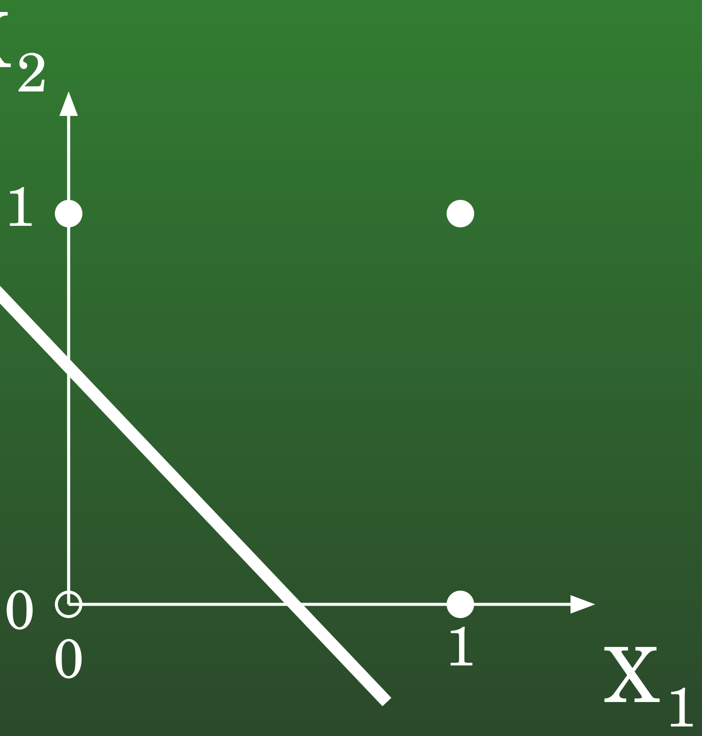


Function: X1 or X2

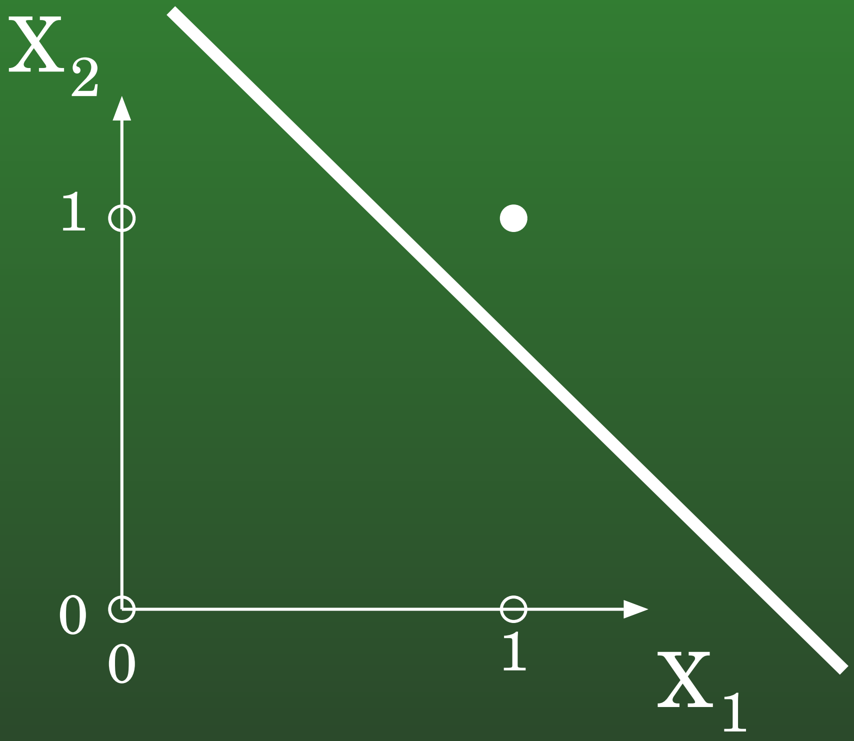


Function: X1 and X2

20-17: Linearly Seperable

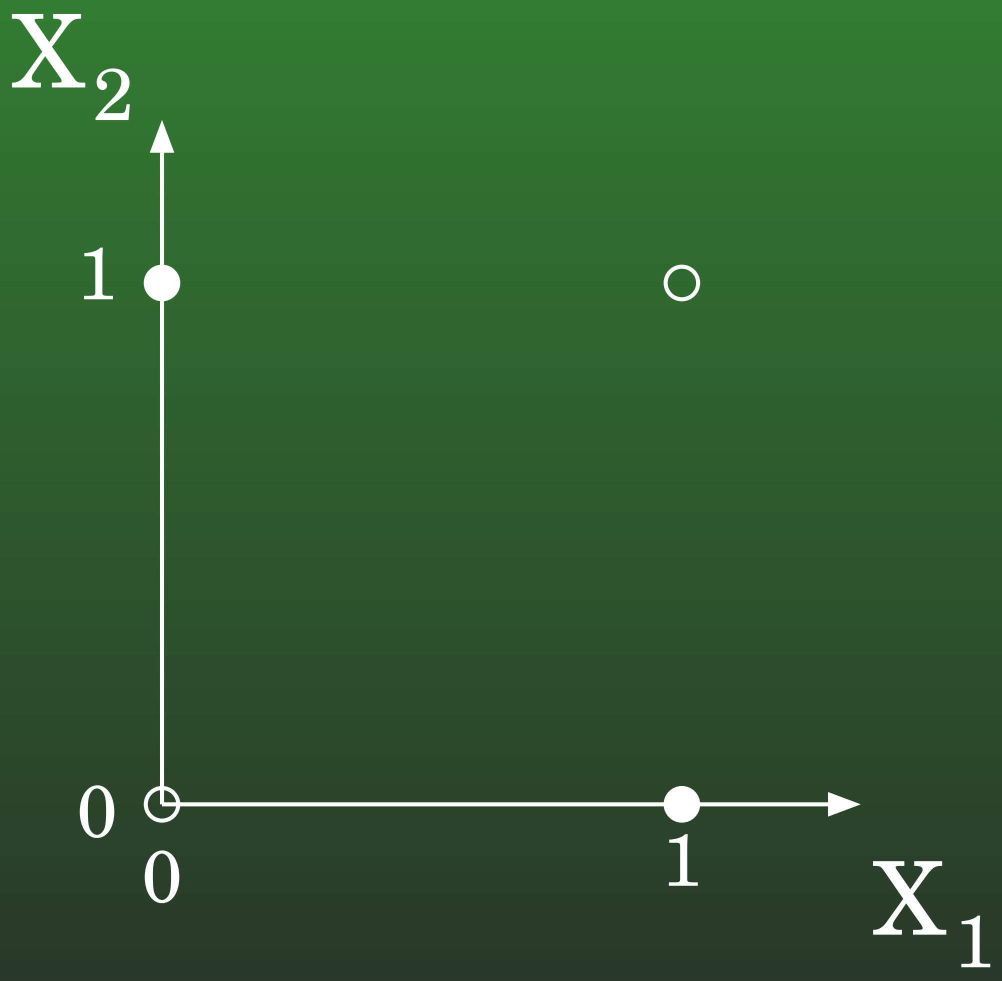


Function: X1 or X2



Function: X1 and X2

Linearly Seperable



Function: X1 xor X2

20-19: Perceptron Learning

```
inptuts : in1, in2, ..., inj
weights : w1, w2, ... wn
training examples: t1 = (tin1, to1),
                   t2 = (tin2, to2), ...

do
  for t in training examples
    inputs = tin
    o = compute output with current weights
    E = to - o
    for eacxh wi in weight
      wi = wi + alpha * tin[i] * E
while notConverged
```

20-20: Perceptron Learning

- If the output signal is too high, weights need to be reduced
  - “Turn down” weights that contributed to output
  - Weights with zero input are not affected
- If output is too low, weights need to be increased
  - “Turn up” weights that contribute to output
  - Zero-input weights not affected
- Doing a hill-climbing search through weight space

20-21: **Perceptron Example**

- Learn the majority function with 3 inputs
  - (plus bias input)
- out = 1 if $\sum_j w_j in_j > 0$, 0 otherwise
- $\alpha = 0.2$
- Initially, all weights 0

Perceptron Example

bias	inputs			expected out
1	1	0	0	0
1	0	1	1	1
1	0	1	0	0
1	1	1	1	1
1	0	0	1	0
1	1	0	1	1
1	1	1	0	1
1	0	0	0	0

Perceptron Example

bias	inputs			expected out	w0	w1	w2	w3	actual out	new weights			
1	1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1									
1	0	1	0	0									
1	1	1	1	1									
1	0	0	1	0									
1	1	0	1	1									
1	1	1	0	1									
1	0	0	0	0									

Perceptron Example

bias	inputs			expected out	w0	w1	w2	w3	actual out	new weights			
1	1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	0	0	0	0	0	0.2	0	0.2	0.2
1	0	1	0	0									
1	1	1	1	1									
1	0	0	1	0									
1	1	0	1	1									
1	1	1	0	1									
1	0	0	0	0									

Perceptron Example

bias	inputs			expected out	w0	w1	w2	w3	actual out	new weights			
1	1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	0	0	0	0	0	0.2	0	0.2	0.2
1	0	1	0	0	0.2	0	0.2	0.2	1	0	0	0	0.2
1	1	1	1	1									
1	0	0	1	0									
1	1	0	1	1									
1	1	1	0	1									
1	0	0	0	0									

Perceptron Example

bias	inputs			expected out	w0	w1	w2	w3	actual out	new weights			
1	1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	0	0	0	0	0	0.2	0	0.2	0.2
1	0	1	0	0	0.2	0	0.2	0.2	1	0	0	0	0.2
1	1	1	1	1	0	0	0	0.2	1	0	0	0	0.2
1	0	0	1	0									
1	1	0	1	1									
1	1	1	0	1									
1	0	0	0	0									

Perceptron Example

bias	inputs			expected out	w0	w1	w2	w3	actual out	new weights			
1	1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	0	0	0	0	0	0.2	0	0.2	0.2
1	0	1	0	0	0.2	0	0.2	0.2	1	0	0	0	0.2
1	1	1	1	1	0	0	0	0.2	1	0	0	0	0.2
1	0	0	1	0	0	0	0	0.2	1	-0.2	0	0	0
1	1	0	1	1									
1	1	1	0	1									
1	0	0	0	0									

Perceptron Example

bias	inputs			expected out	w0	w1	w2	w3	actual out	new weights			
1	1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	0	0	0	0	0	0.2	0	0.2	0.2
1	0	1	0	0	0.2	0	0.2	0.2	1	0	0	0	0.2
1	1	1	1	1	0	0	0	0.2	1	0	0	0	0.2
1	0	0	1	0	0	0	0	0.2	1	-0.2	0	0	0
1	1	0	1	1	-0.2	0	0	0	0	0	0.2	0	0.2
1	1	1	0	1									
1	0	0	0	0									

20-29: Perceptron Example

bias	inputs			expected out	w0	w1	w2	w3	actual out	new weights			
1	1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	0	0	0	0	0	0.2	0	0.2	0.2
1	0	1	0	0	0.2	0	0.2	0.2	1	0	0	0	0.2
1	1	1	1	1	0	0	0	0.2	1	0	0	0	0.2
1	0	0	1	0	0	0	0	0.2	1	-0.2	0	0	0
1	1	0	1	1	-0.2	0	0	0	0	0	0.2	0	0.2
1	1	1	0	1	0	0.2	0	0.2	1	0	0.2	0	0.2
1	0	0	0	0									

20-30: Perceptron Example

bias	inputs			expected out	w0	w1	w2	w3	actual out	new weights			
1	1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	0	0	0	0	0	0.2	0	0.2	0.2
1	0	1	0	0	0.2	0	0.2	0.2	1	0	0	0	0.2
1	1	1	1	1	0	0	0	0.2	1	0	0	0	0.2
1	0	0	1	0	0	0	0	0.2	1	-0.2	0	0	0
1	1	0	1	1	-0.2	0	0	0	0	0	0.2	0	0.2
1	1	1	0	1	0	0.2	0	0.2	1	0	0.2	0	0.2
1	0	0	0	0	0	0.2	0	0.2	0	0	0.2	0	0.2

Still hasn’t converged, need more iterations

20-31: Perceptron Example

After 3 more iterations (of all weights):

bias	inputs			expected out	w0	w1	w2	w3	actual out
1	1	0	0	0	-0.4	0.2	0.4	0.4	0
1	0	1	1	1	-0.4	0.2	0.4	0.4	1
1	0	1	0	0	-0.4	0.2	0.4	0.4	0
1	1	1	1	1	-0.4	0.2	0.4	0.4	1
1	0	0	1	0	-0.4	0.2	0.4	0.4	0
1	1	0	1	1	-0.4	0.2	0.4	0.4	1
1	1	1	0	1	-0.4	0.2	0.4	0.4	1
1	0	0	0	0	-0.4	0.2	0.4	0.4	0

20-32: Gradient Descent & Delta Rule

- What if we can’t learn the function exactly?
  - Function is not linearly seperable
- Want to do “as well as possible”
- Minimize the sum of the squared error
- $E = \sum(t_d - o_d)^2$ for d in the training set

Gradient Descent & Delta Rule

- Searching through a space of weights
- Much like local search, define an error E as a function of the weights
- Find values of weights to minimize E
- Follow the gradient – largest negative change in E
    - Alas, E is discontinuous, hard to differentiate
    - Instead of using actual output, use *unthresholded* output

20-34: Gradient Descent & Delta Rule

- Garient descent: follow the steepest slope down the error surface
- Consider the derivative of E with respect to each weight
- $E = \sum(t_d - o_d)^2$ for d in the training set

$$\frac{dE}{dw_i} = \sum 2(t_d - o_d)\frac{d(t_d - o_d)}{dw_i}$$

First, we will simplify for looking at a single training data point (then we can sum over all of them, since the derivative of a sum is the sum of the derivatives)

20-35: Gradient Descent & Delta Rule

- For a single training example:

$$
\begin{aligned}
\frac{dE}{dw_i} &= \frac{d(t_d - o_d)^2}{dw_i} \\
&= 2(t_d - o_d)\frac{d(t_d - o_d)}{dw_i}
\end{aligned}
$$

since $\frac{d(f(x)^2)}{dx} = 2f(x)\frac{d(f(x))}{dx}$

Gradient Descent & Delta Rule

- For a single training example:

$$\begin{aligned}\frac{dE}{dw_i} &= 2(t_d - o_d)\frac{d(t_d - o_d)}{dw_i} \\ &= 2(d_d - o_d)\frac{d(-x_{id}w_i)}{dw_i}\end{aligned}$$

  - t_d doesn't involve w_i, so $\frac{d(t_t)}{dw_i} = 0$
  - $o_d = w_1x_{1d} + w_2x_{2d} + w_3x_{3d} + \ldots$, the only term that involves w_i is w_ix_{id}

20-37: Gradient Descent & Delta Rule

- For a single training example:

$$
\begin{aligned}
\frac{dE}{dw_i} &= 2(d_d - o_d)\frac{d(-x_{id}w_i)}{dw_i} \\
&= 2(t_d - o_d)(-x_{id})
\end{aligned}
$$

  - Since $\frac{d(cx)}{dx} = c$

Gradient Descent & Delta Rule

- Garient descent: follow the steepest slope down the error surface
- Consider the derivative of E with respect to each weight
- $E = \sum(t_d - o_d)^2$ for d in the training set

$$\begin{aligned} \frac{dE}{dw_i} &= \sum 2(t_d - o_d)\frac{d(t_d - o_d)}{dw_i} \\ &= \sum 2(t_d - o_d)(-x_{id}) \end{aligned}$$

- Want to go *down* the gradiant,
  $\Delta w_i = \alpha \sum_{d \in D}(t_d - o_d)x_{id}$

20-39: Gradient Descent & Delta Rule

- Garient descent: follow the steepest slope down the error surface
- Consider the derivative of E with respect to each weight
- After derivation, updating rule (called the Delta Rule) is:

$$\Delta w_i = \alpha \sum_{d \in D} (t_d - o_d) x_{id}$$

  - D is the training set, α is the training rate, t_d is the expected output, and o_d is the actual output, $x_i d$ is the input along weight w_i.

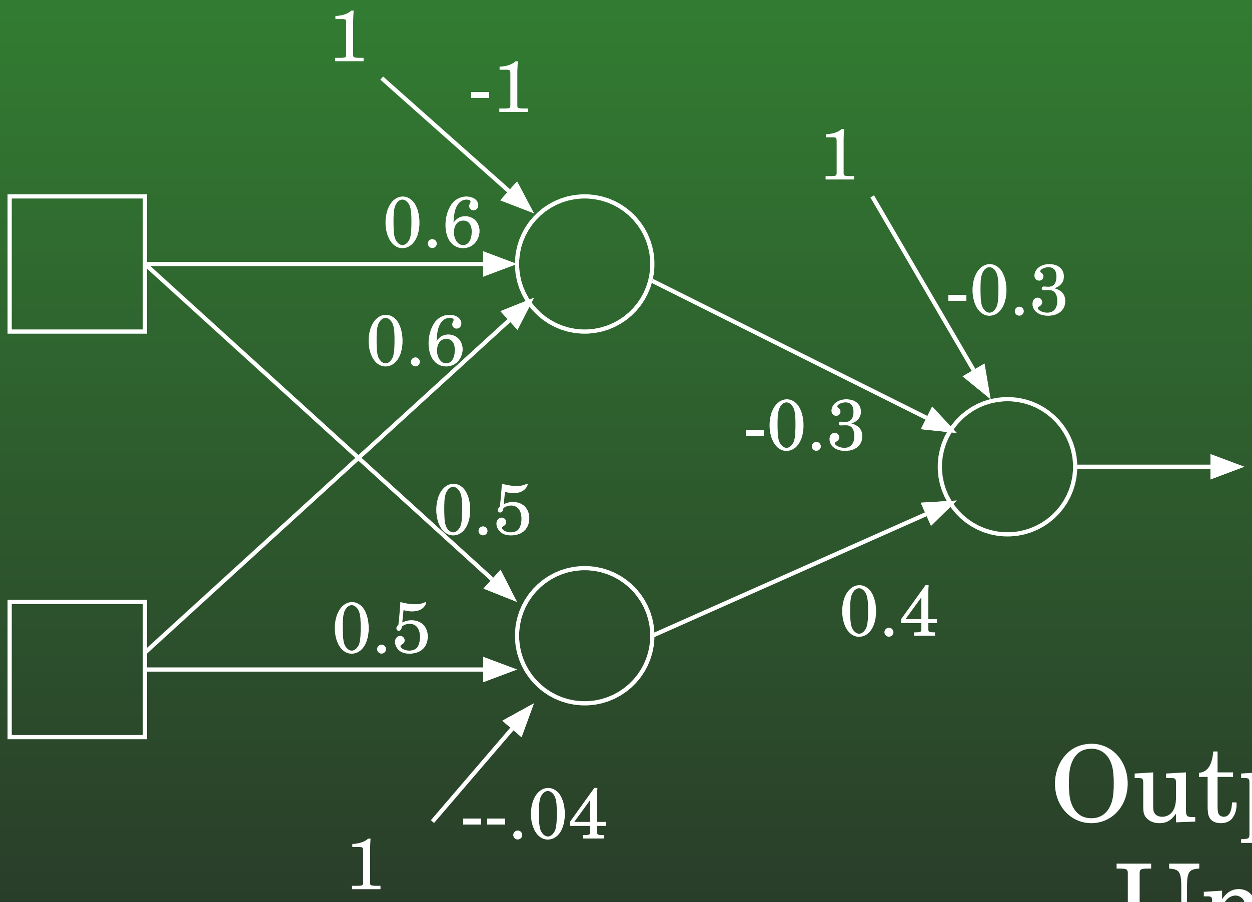
Incremental Learning

- Often not practical to compute global weight change for entire training set
- Instead, update weights incrementally
  - Observe one piece of data, then update
- Update rule: $w_i = \alpha(t - o)x_i$
  - Like perceptron learning rule – except uses unthresholded output
- Smaller training rate α typically used
- No theoretical guarantees of convergence

Multilayer Networks

- While perceptrons have the advantage of a simple learning algorithm, their computational limitations are a problem.
- What if we add another “hidden” layer?
- Computational power increases
  - With one hidden layer, can represent any continuous function
  - With two hidden layers, can represent any function
- Example: Create a multi-layer network that computes XOR

XOR



Input Units

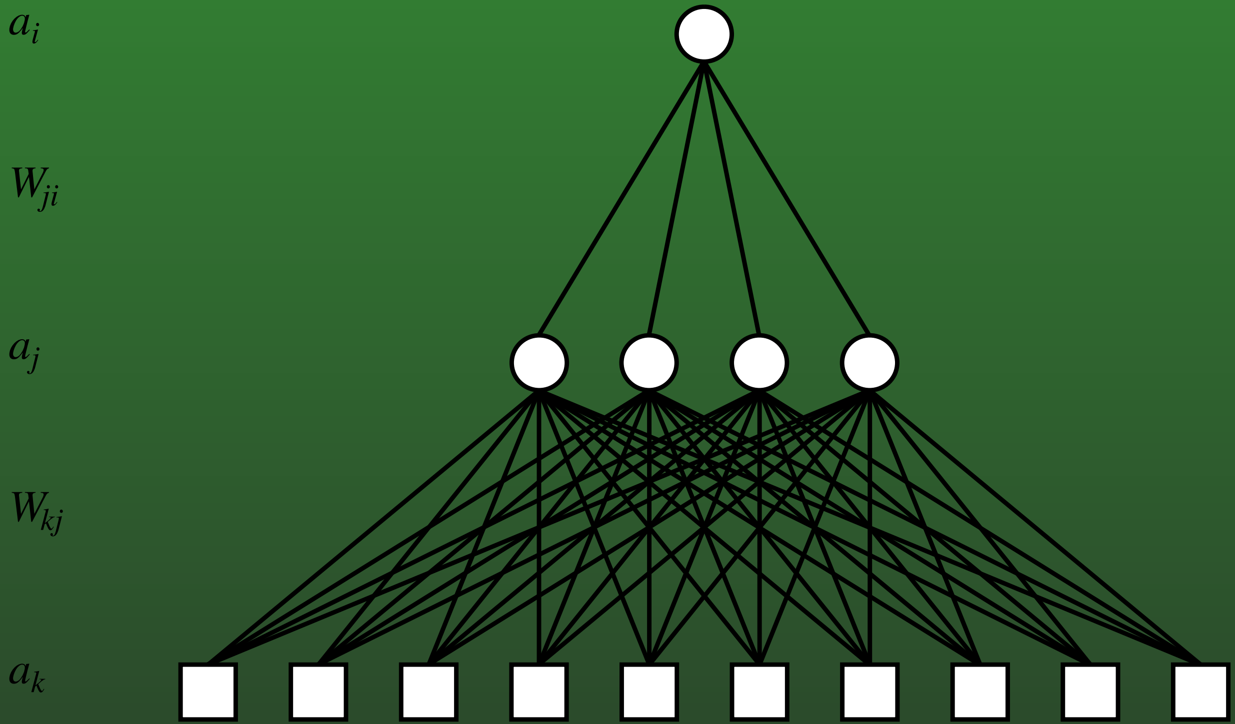
Hidden Units

Output Unit

Multilayer Networks

- While perceptrons have the advantage of a simple learning algorithm, their computational limitations are a problem.
- What if we add another “hidden” layer?
- Computational power increases
  - With one hidden layer, can represent any continuous function
  - With two hidden layers, can represent any function
- Problem: How to find the correct weights for hidden nodes?

Multilayer Network Example



More on computing error

- Backpropagation is an extension of the perceptron learning algorithm to deal with multiple layers of nodes.
- Our goal is to minimize the error function.
- To do this, we want to change the weights so as to reduce the error.
- We define error as a function of the weights like so:
  - $E(\mathbf{w}) = expected - actual$
  - $E(\mathbf{w}) = expected - g(input)$
- So, to determine how to change the weights, we compute the derivative of error with respect to the weights.
  - This tells us the slope of the error curve at that

Backpropagation

- Nodes use sigmoid activation function, rather than the step function
- Sigmoid function works in much the same way, but is differentiable.
  - $g(input_i) = \frac{1}{1+e^{-input_i}}$.
  - $g'(input_i) = g(input_i)(1 - g(input_i))$ (good news here -calculating the derivative only requires knowing the output!)

More on computing error

- Recall that our goal is to minimize the error function.
- To do this, we want to change the weights so as to reduce the error.
- We define error as a function of the weights like so:
    - $E(\mathbf{w}) = expected - actual$
    - $E(\mathbf{w}) = expected - g(input)$
- So, to determine how to change the weights, we compute the derivative of error with respect to the weights.
    - This tells us the slope of the error curve at that point.

More on computing error

- $E(\mathbf{w}) = expected - g(input)$
- $E(\mathbf{w}) = expected - g(\mathbf{w} * \mathbf{i})$
- $\frac{dE}{dW} = 0 - g'(input)\mathbf{i}$
- $\delta_w = \frac{dE}{dW} = -g(input) * (1 - g(input)) * \mathbf{i}$

20-49: Updating hidden weights

- Each weight is updated by $\alpha * \Delta_i$
- $W_{j,i} = W_{j,i} + \alpha * a_j * \Delta_i$

Backpropagation

- Updating input-hidden weights:
- Idea: each hidden node is responsible for a fraction of the error in δ_i.
- Divide δ_i according to the strength of the connection between the hidden and output node.
- For each hidden node j
- $\delta_j = g(input)(1 - g(input)) \sum_{i \in outputs} W_{j,i} \delta_i$
- Update rule for input-hidden weights:
- $W_{k,j} = W_{k,j} + \alpha * input_k * \delta_j$

20-51: Backpropagation Algorithm

- The whole algorithm can be summed up as:

```
While not done:
    for d in training set
        Apply inputs of d, propagate forward.
        for node $i$ in output layer
            $\delta_i = output * (1 - output) * (t_{exp} - output)$
        for each hidden node $j$
            $\delta_j = output * (1 - output) * \sum W_{j,i}\delta_i$
        Adjust each weight
            $W_{j,i} = W_{j,i} + \alpha * \delta_i * input_j$
```

Theory vs Practice

- In the definition of backpropagation, a single update for all weights is computed for all data points at once.
  - Find the update that minimizes total sum of squared error.
- Guaranteed to converge in this case.
- Problem: This is often computationally space-intensive.
  - Requires creating a matrix with one row for each data point and inverting it.
- In practice, updates are done incrementally instead.

Stopping conditions

- Unfortunately, incremental updating is not *guaranteed* to converge.
- Also, convergence can take a long time.
- When to stop training?
  - Fixed number of iterations
  - Total error below a set threshold
  - Convergence - no change in weights

Backpropagation

- Also works for multiple hidden layers
- Backpropagation is only guaranteed to converge to a local minimum
  - May not find the absolute best set of weights
- Low initial weights can help with this
  - Makes the network act more linearly - fewer minima
- Can also use random restart - train multiple times with different initial weights.

Momentum

- Since backpropagation is a hillclimbing algorithm, it is susceptible to getting stuck in plateaus
  - Areas where local weight changes don’t produce an improvement in the error function.
- A common extension to backpropagation is the addition of a momentum term.
  - Carries the algorithm through minima and plateaus.
- Idea: remember the “direction” you were going in, and by default keep going that way.
- Mathematically, this means using the second derivative.

Momentum

- Implementing momentum typically means remembering what update was done in the previous iteration.
- Our update rule becomes:
- $\Delta w_{ji}(n) = \alpha \Delta_j x_{ji} + \beta \mathbf{\Delta w_{ji}(n-1)}$
- To consider the effect, imagine that our new delta is zero (we haven’t made any improvement)
- Momentum will keep the weights “moving” in the same direction.
- Also gradually increases step size in areas where gradient is unchanging.
  - This speeds up convergence, helps escape plateaus and local minima.

Design issues

- One difficulty with neural nets is determining how to *encode* your problem
  - Inputs must be 1 and 0, or else real-valued numbers.
  - Same for outputs
- Symbolic variables can be given binary encodings
- More complex concepts may require care to represent correctly.

Design issues

- Like some of the other algorithms we’ve studied, neural nets have a number of paramters that must be tuned to get good performance.
    - Number of layers
    - Number of hidden units
    - Learning rate
    - Initial weights
    - Momentum term
    - Training regimen
- These may require trial and error to determine

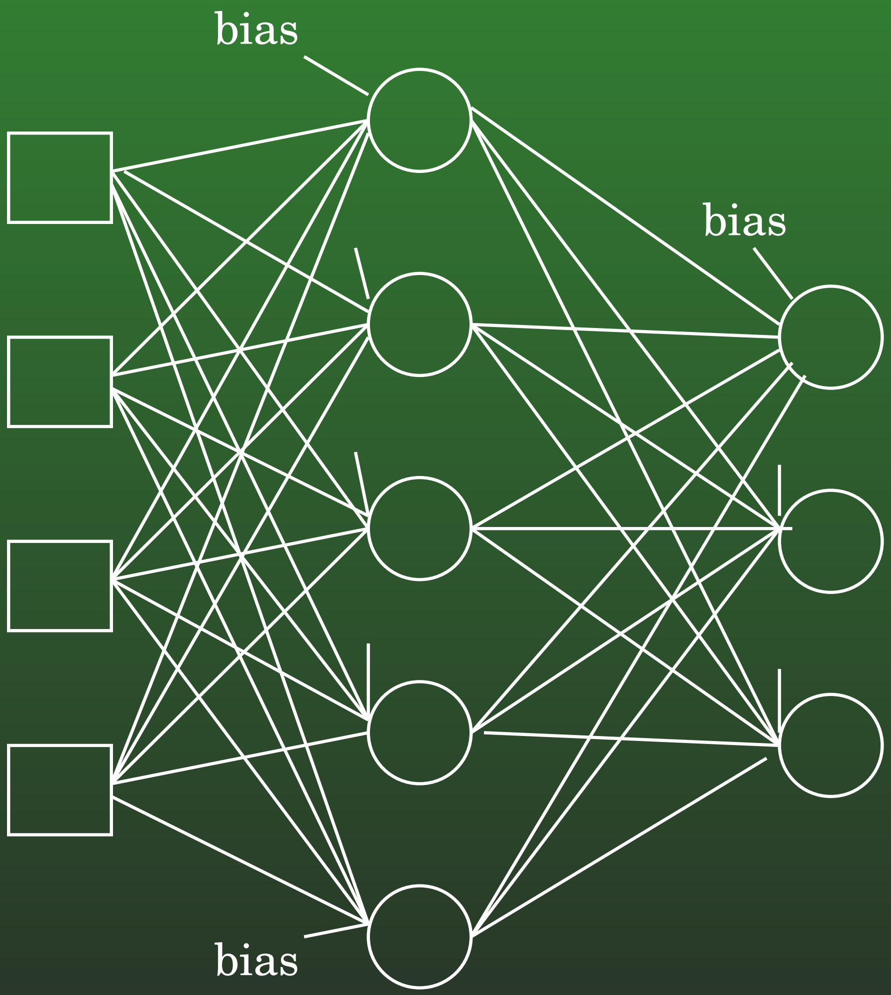
Design issues

- The more hidden nodes you have, the more complex function you can approximate
- Is this always a good thing? That is, are more hidden nodes better?

20-60: Overfitting

- Overfitting
  - Conisder a network with i input nodes, o output nodes, and k hidden nodes
  - Training set has k examples
  - Could end up learning a lookup table

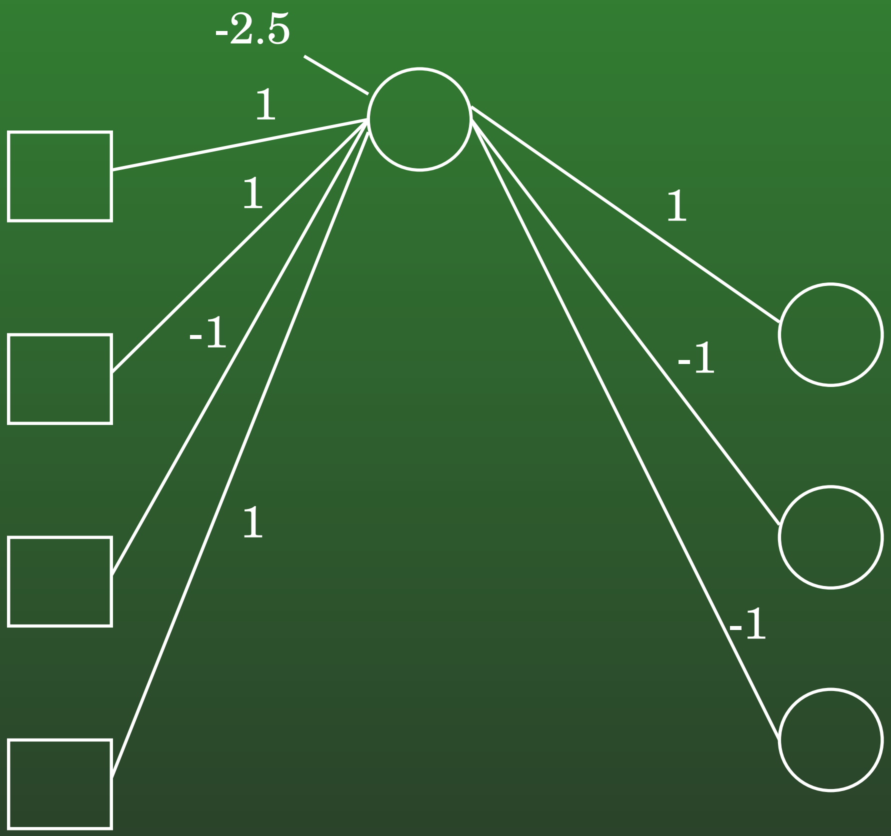
Overfitting



Training Data

1101	100
0110	010
1111	001
0011	100
0000	010

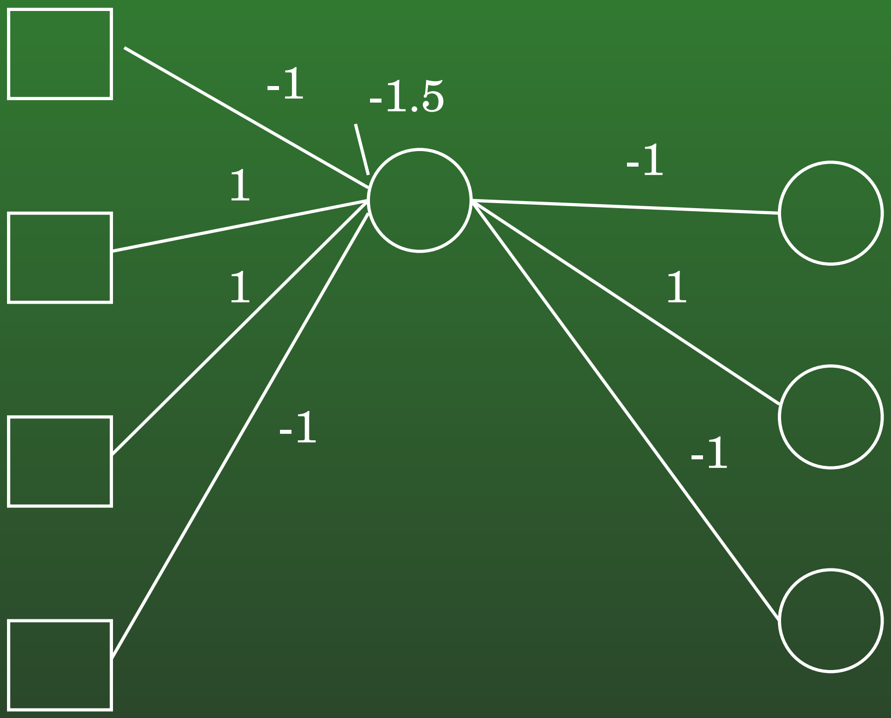
20-62: Overfitting



Training Data

1101 100
0110 010
1111 001
0011 100
0000 010

20-63: Overfitting



Training Data

1101 100

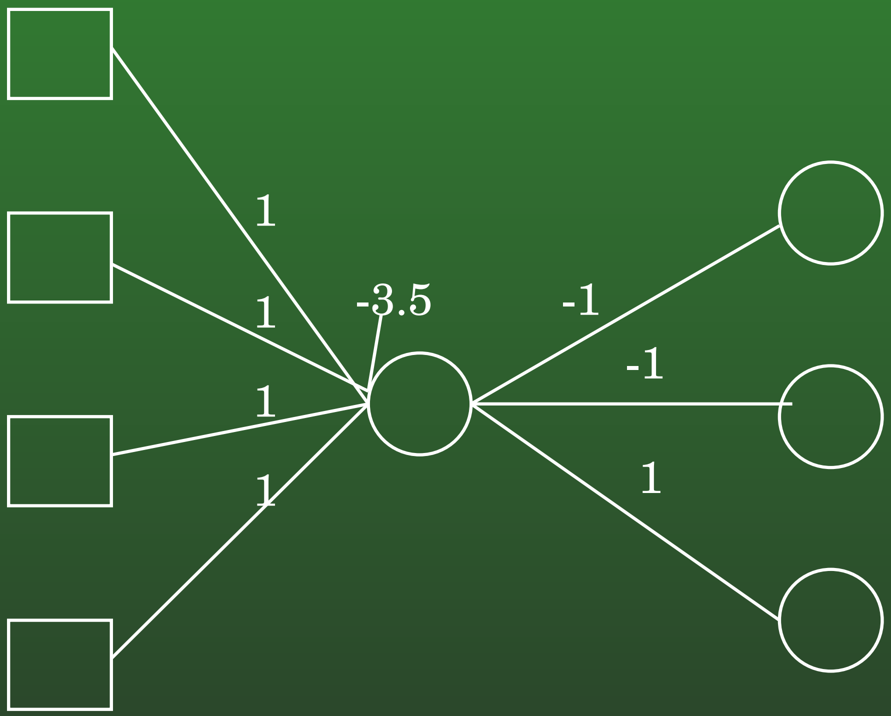
$\boxed{0110\ \ 010}$

1111 001

0011 100

0000 010

20-64: Overfitting



Training Data

1101 100

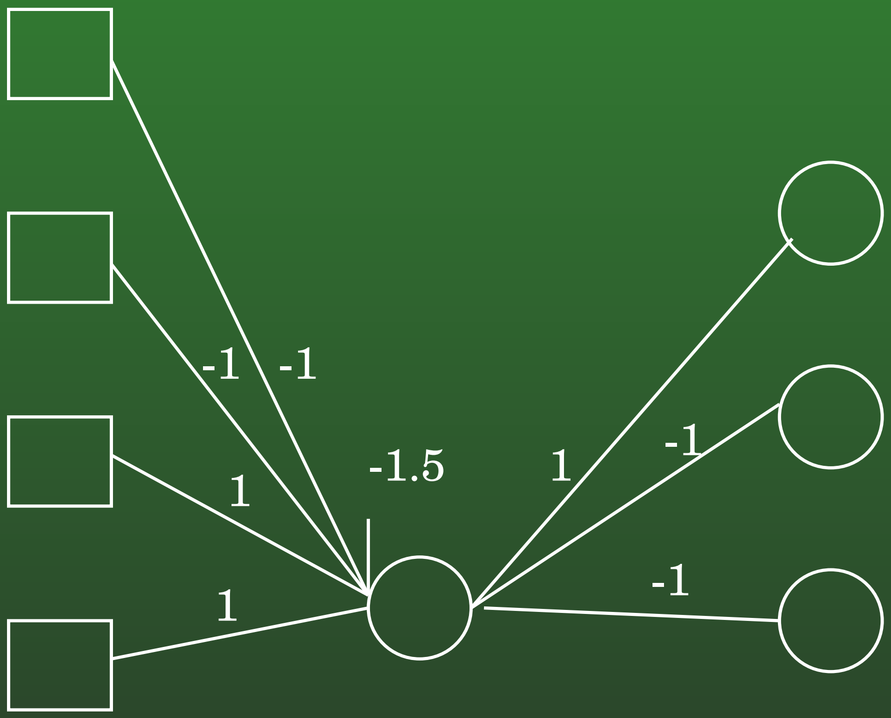
0110 010

1111 001

0011 100

0000 010

20-65: Overfitting



Training Data

1101 100

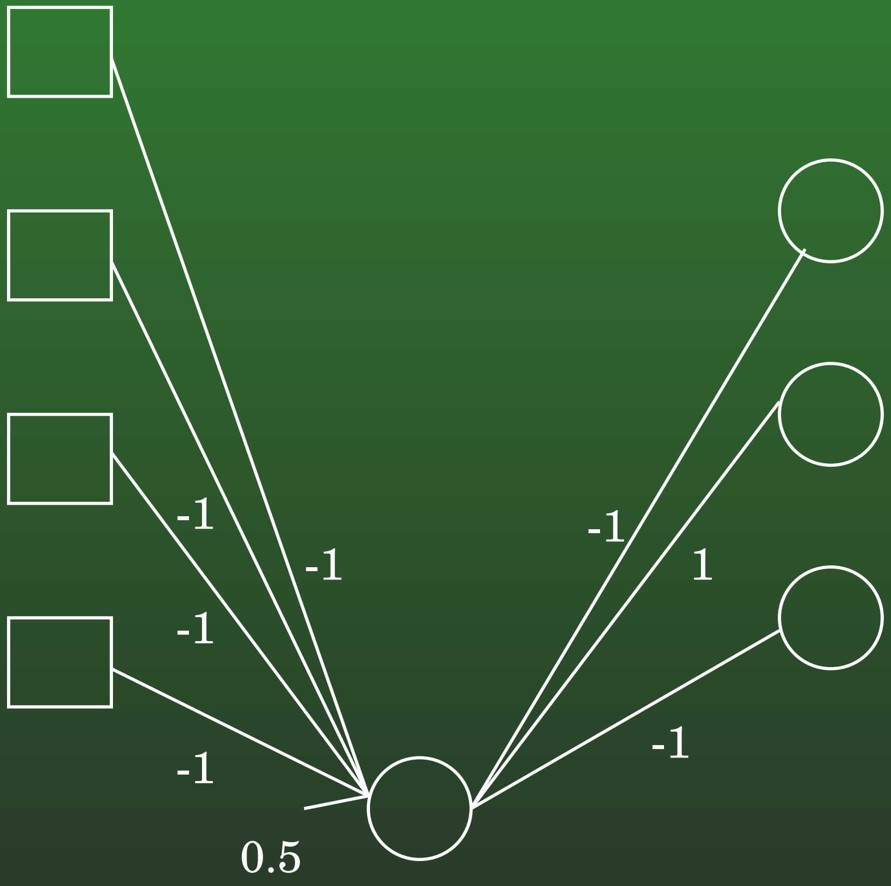
0110 010

1111 001

$\boxed{0011\ \ 100}$

0000 010

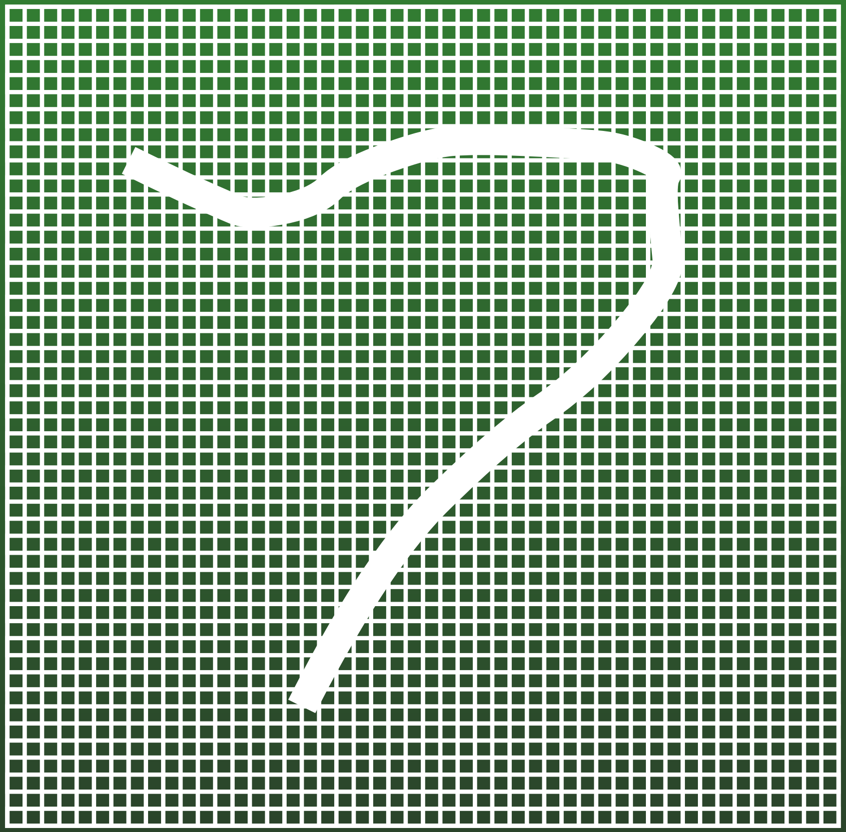
Overfitting



Training Data
1101 100
0110 010
1111 001
0011 100
0000 010

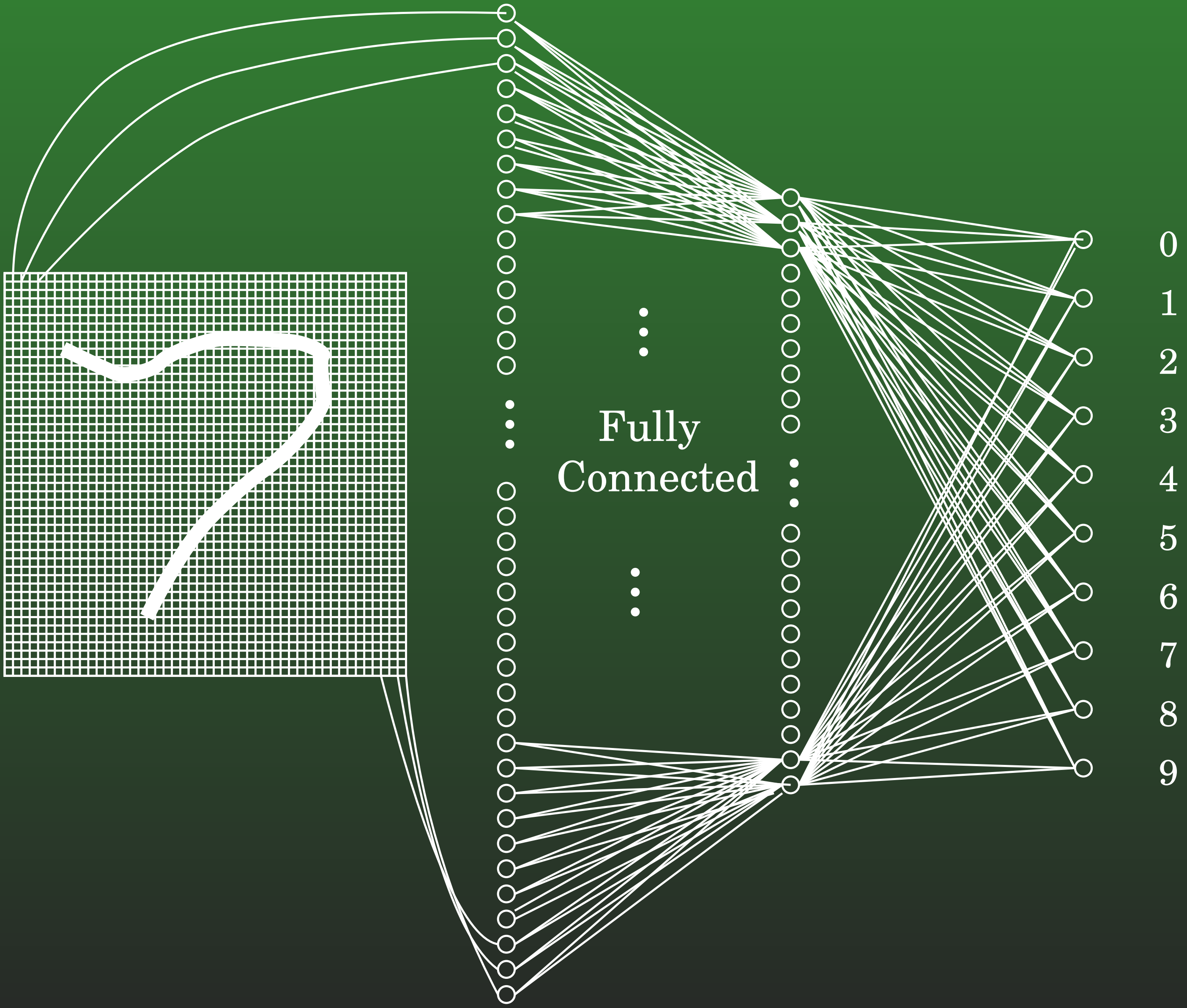
Number Recognition

20-68: Number Recognition



Each pixel is an input unit

Number Recognition

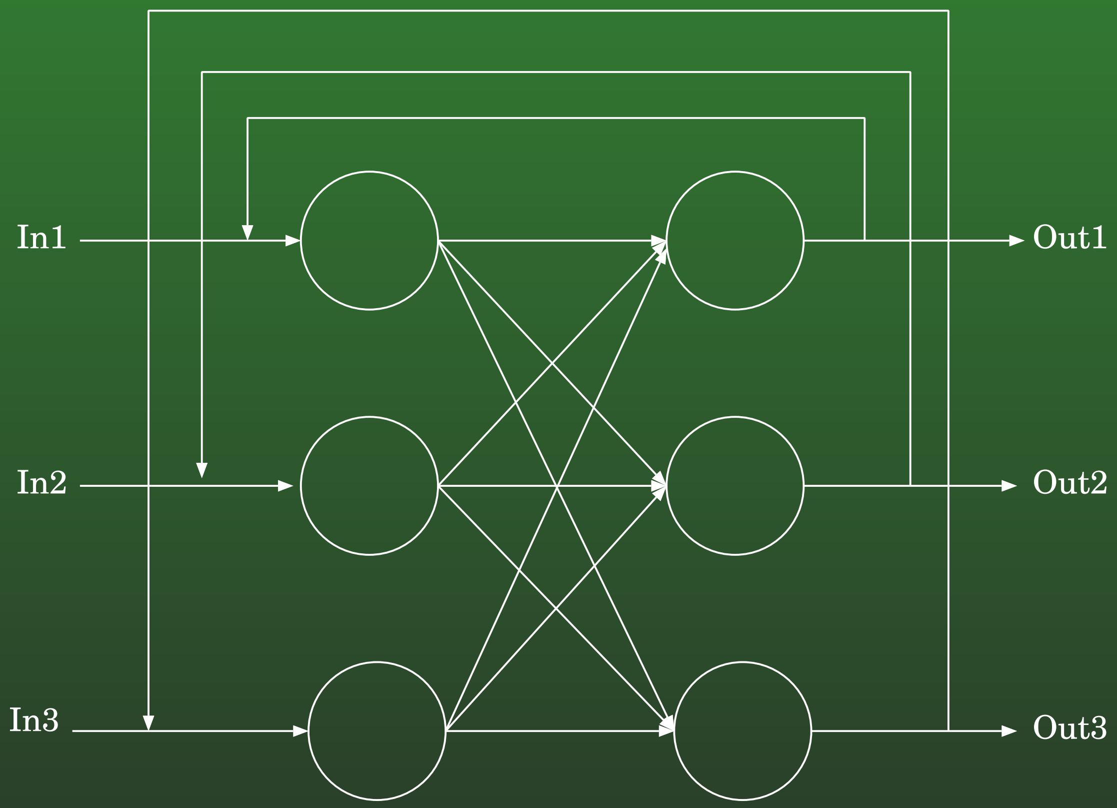


Recurrent NNs

- So far, we’ve talked only about feedforward networks.
  - Signals propagate in one direction
  - Output is immediately available
  - Well-understood training algorithms
- There has also been a great deal of work done on recurrent neural networks.
  - At least some of the outputs are connected back to the inputs.

Recurrent NNs

- This is a single-layer recurrent neural network



Hopfield networks

- A Hopfield network has no special input or output nodes.
- Every node receives an input and produces an output
- Every node connected to every other node.
- Typically, threshold functions are used.
- Network does not immediately produce an output.
  - Instead, it oscillates
- Under some easy-to-achieve conditions, the network will eventually stabilize.
- Weights are found using simulated annealing.

20-73: Hopfield networks

- Hopfield networks can be used to build an *associative memory*
- A portion of a pattern is presented to the network, and the net “recalls” the entire pattern.
- Useful for letter recognition
- Also for optimization problems
- Often used to model brain activity

Neural nets - summary

- Key idea: simple computational units are connected together using weights.
- Globally complex behavior emerges from their interaction.
- No direct symbol manipulation
- Straightforward training methods
- Useful when a machine that approximates a function is needed
  - No need to understand the learned hypothesis