Review on Face Recognition

- Initialization :
  - Acquire the training set and calculate eigenfaces (using PCA projections) which define eigenspace.
- When a new face is encountered, calculate its weight.
- Determine if the image is face.
- If yes, classify the weight pattern as known or unknown.
- (Learning) If the same unknown face is seen several times incorporate it into known faces.

Review on Eigen Faces

- Face Images are projected into a feature space ("Face Space") that <u>best encodes the variation</u> among known face images.
- The face space is defined by the "eigenfaces", which are the <u>**eigenvectors**</u> of the set of faces.

Eigenfaces (1)

- Calculation of Eigenfaces

  (1) Calculate **average face** : *v*.

  (2) Collect **difference between training images** and average face in matrix A (M by N), where M is the number of pixels and N is the number of images.

$$A = [u_1^1 - v, \dots, u_n^1 - v, \dots, u_1^p - v, \dots, u_n^p - v]$$

  (3) The **eigenvectors of covariance matrix C** (M by M) give the eigenfaces.

  – M is usually big, so this process would be time consuming.

What to do?

$$C = AA^T$$

Eigenfaces (2)

- Calculation of Eigenvectors of C

  If the number of data points is smaller than the dimension (N<M), then there will be only N-1 meaningful eigenvectors.

  Instead of directly calculating the eigenvectors of C, we can **calculate the eigenvalues and the corresponding eigenvectors of a much smaller matrix L (N by N).**

$$\boxed{L = A^T A}$$

  if λ_i are the eigenvectors of L then A λ_i are the eigenvectors for C.

  – The eigenvectors are in the descent order of the corresponding eigenvalues.

Eigenfaces (3)

- Representation of Face Images using Eigenfaces
- The training face images and new face images can be represented as linear combination of the eigenfaces.
- When we have a face image u :

$$\boxed{u = \sum_i a_i \phi_i}$$

Since the eigenvectors are orthogonal :

$$\boxed{a_i = u^T \phi_i}$$

Fisherfaces

- Fisherfaces is developed by the statistical discrimination theory.
- Once the weight matrix is obtained, the computation is similar to eigenfaces.
- An unkown face image is subject to a dimension reduction (projection) and decision is made based on nearest neighbor rule in the projected subspace.
- In summary, both eignenfaces and fisherfaces requires the projection of an image into a subspace and classification is done in the projected space. The difference is on how to obtain a meaningful projection matrix.

Artificial Neural Networks

- What can they do?
- How do they work?
- What might we use them for it for face recognition?
- Why are they so cool?

History

- late-1800's - Neural Networks appear as an analogy to biological systems
- 1960's and 70's – Simple neural networks appear
  - Fall out of favor because the perceptron is not effective by itself, and there were no good algorithms for multilayer nets
- 1986 – Backpropagation algorithm appears
  - Neural Networks have a resurgence in popularity

Applications

- Handwriting recognition
- Recognizing spoken words
- Face recognition
  - You will get a chance to play with this later!

Basic Idea

- Modeled on biological systems
  - This association has become much looser
- Learn to classify objects
  - Can do more than this
- Learn from given training data of the form (x1...xn, output)

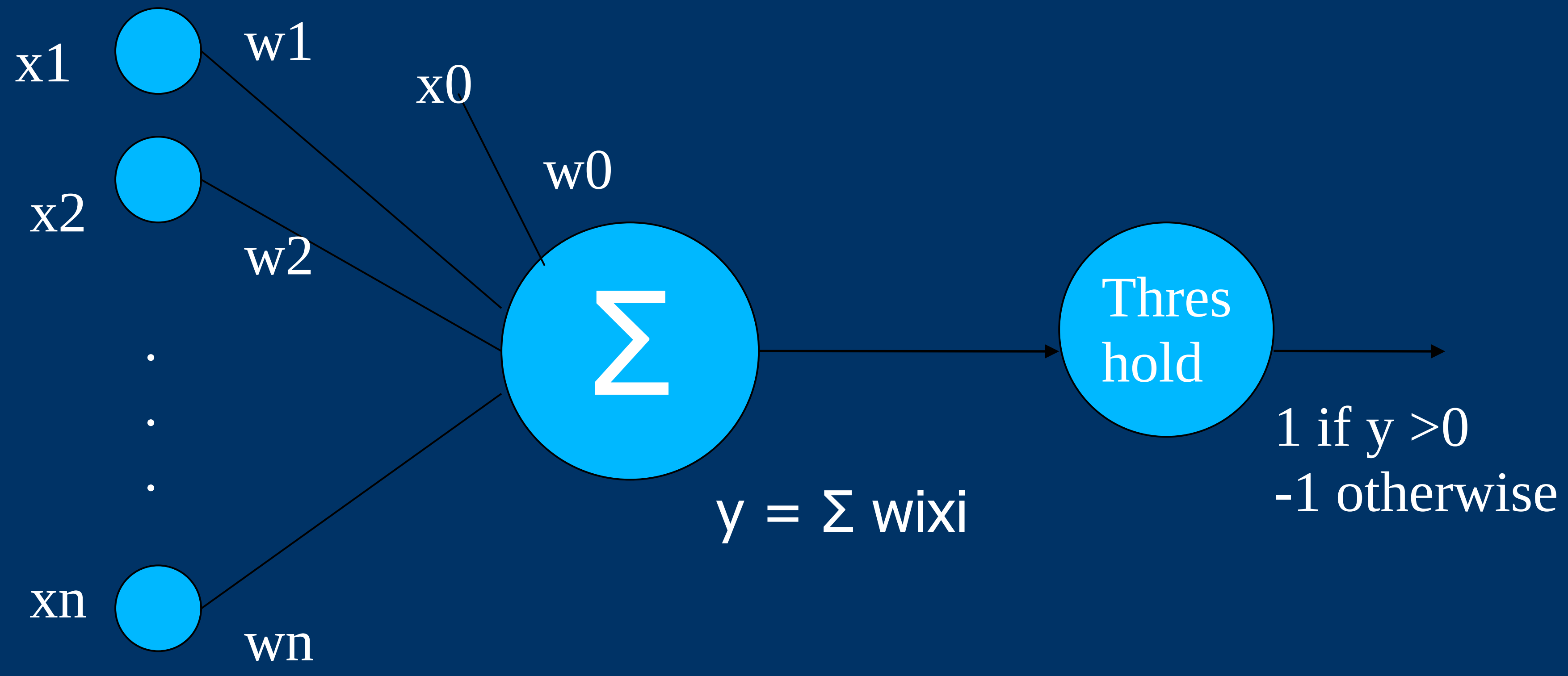
Properties

- Inputs are flexible
  - any real values
  - Highly correlated or independent
- Target function may be discrete-valued, real-valued, or vectors of discrete or real values
  - Outputs are real numbers between 0 and 1
- Resistant to errors in the training data
- Long training time
- Fast evaluation
- The function produced can be difficult for humans to interpret

Perceptrons

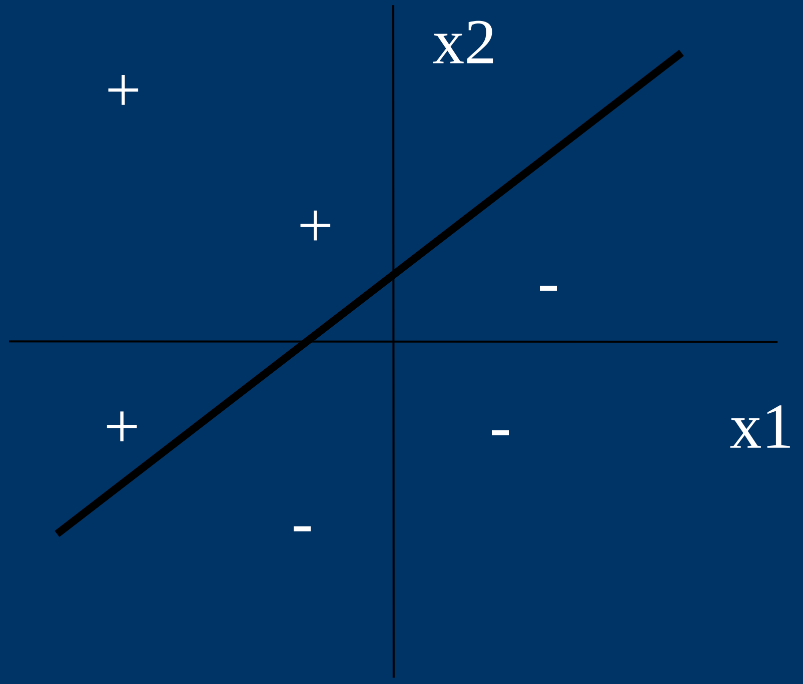
- Basic unit in a neural network
- Linear separator
- Parts
  - N inputs, x1 ... xn
  - Weights for each input, w1 ... wn
  - A bias input x0 (constant) and associated weight w0
  - Weighted sum of inputs, y = w0x0 + w1x1 + ... + wnxn
  - A threshold function, i.e 1 if y > 0, -1 if y <= 0

Diagram

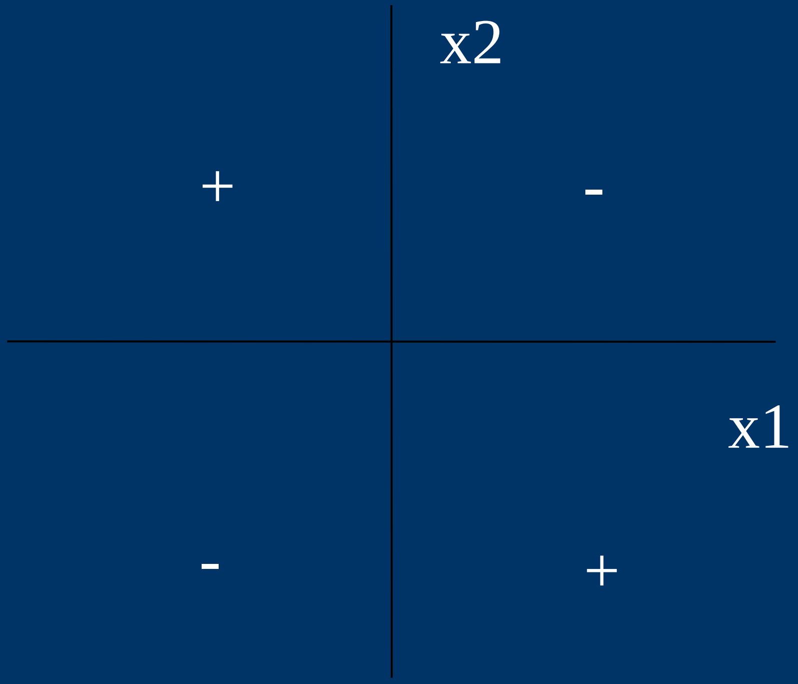


Linear Separator

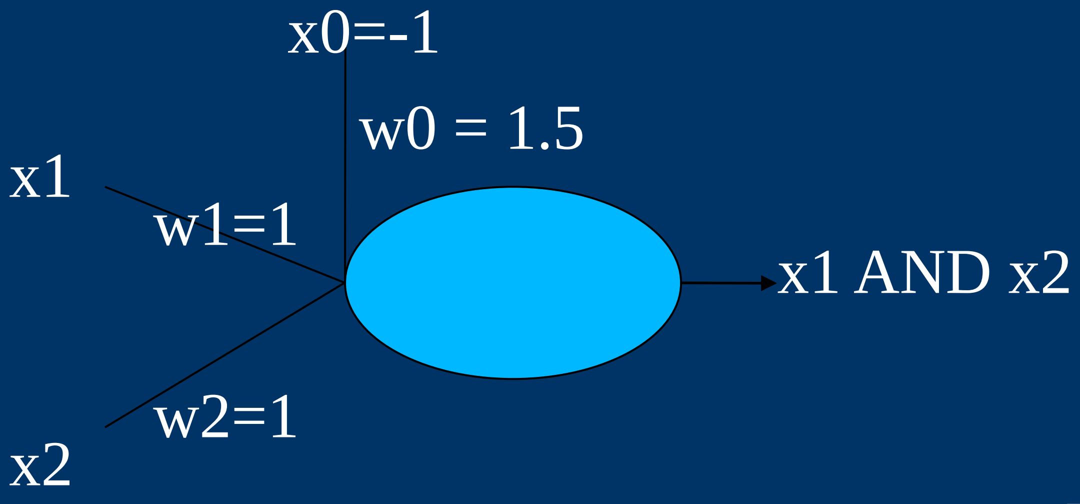
This...

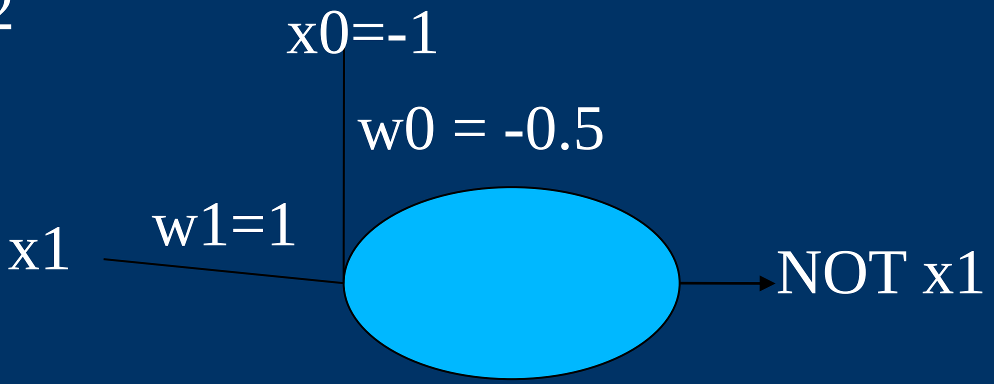


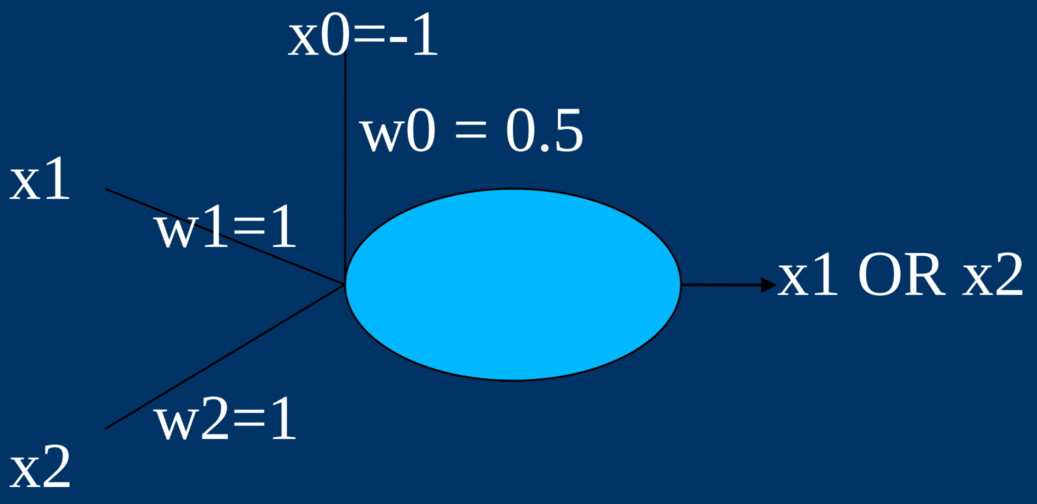
But not this (XOR)



Boolean Functions







Thus all boolean functions can be represented by layers of perceptrons!

Perceptron Training Rule

$$w_i = w_i + \Delta w_i$$

$$\Delta w_i = \eta\left(t - o\right)x_i$$

w_i: The weight of input i

η: The 'learning rate' between 0 and 1

t: The target output

o: The actual output

x_i: The ith input

Gradient Descent

- Perceptron training rule may not converge if points are not linearly separable
- Gradient descent will try to fix this by changing the weights by the total error for all training points, rather than the individual
  - If the data is not linearly separable, then it will converge to the best fit

Gradient Descent

$$Error\ \ function: E\,\square x\square = \frac{1}{2}\sum_{d\in D}\square t_d - o_d\square^2$$

$$w_i = w_i \square\ \square w_i$$

$$\square\, w_i = -\,\square\frac{\partial E}{\partial w_i}$$

$$\square\, w_i = \square\sum_{d\in D}\square t_d - o_d\square x_{id}$$

Gradient Descent Algorithm

```
GRADIENT-DESCENT(training_examples, $\eta$)
Each training example is a pair of the form $(\vec{x}, t)$ where x is the
vector of input values, and t is the target output value,
$\eta$ is learning rate (0< $\eta$<1)

Initialize each $w_i$ to some small random value
Until the termination condition is met, Do
----For each (vec x, t) in training_examples, Do
--------Input the instance $\vec{x}$ to the unit and compute the output o
--------For each linear unit weight $w_i$, Do
                    $\Delta w_i = \Delta w_i + \eta(t - o)x_i$
----For each linear unit wi, Do
                    $w_i = w_i + \Delta w_i$
```

Gradient Descent Issues

- Converging to a local minimum can be very slow
  - The while loop may have to run many times
- May converge to a local minima
- Stochastic Gradient Descent
  - Update the weights after each training example rather than all at once
  - Takes less memory
  - Can sometimes avoid local minima
  - η must decrease with time in order for it to converge

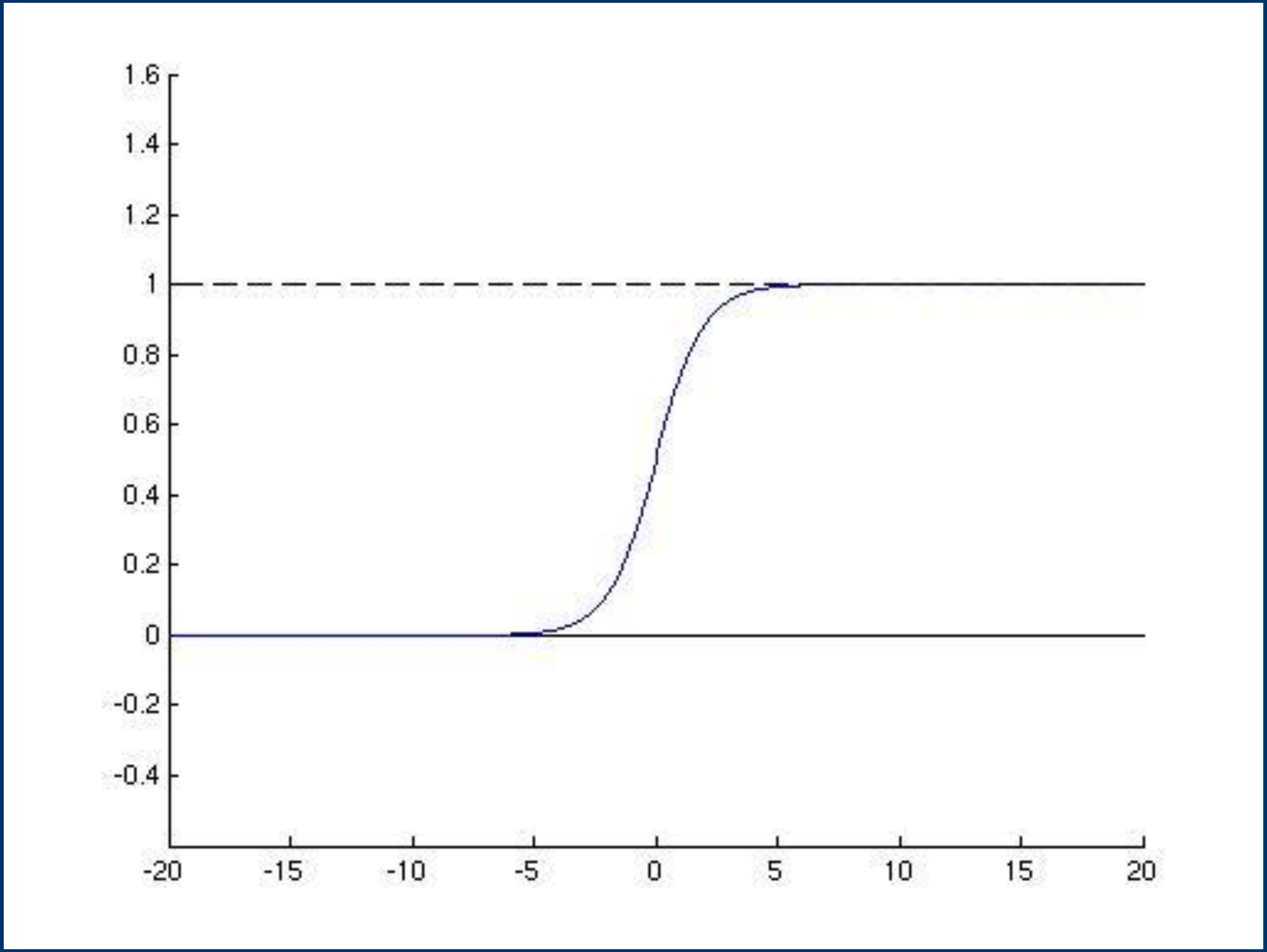
Multi-layer Neural Networks

- Single perceptron can only learn linearly separable functions
- Would like to make networks of perceptrons, but how do we determine the error of the output for an internal node?
- Solution: Backpropogation Algorithm

Differentiable Threshold Unit

- We need a differentiable threshold unit in order to continue
- Our old threshold function (1 if y > 0, 0 otherwise) is **not differentiable**
- **One solution is the sigmoid unit**

Graph of Sigmoid Function



Sigmoid Function

$$Output: o = \sigma(w \cdot x)$$

$$\sigma(y) = \frac{1}{1 + e^{-y}}$$

$$\frac{\partial \sigma(y)}{\partial y} = \sigma(y)(1 - \sigma(y))$$

Variable Definitions

- x_{ij} = the input from to unit j from unit i
- w_{ij} = the weight associated with the input to unit j from unit i
- o_j = the output computed by unit j
- t_j = the target output for unit j
- outputs = the set of units in the final layer of the network
- Downstream(j) = the set of units whose immediate inputs include the output of unit j

Backpropagation Rule

$$E_d(\vec{w}) = \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

$$\Delta w_{ij} = -\eta \frac{\partial E_d}{\partial w_{ij}}$$

For output units:

$$\Delta w_{ij} = \eta (t_j - o_j) o_j (1 - o_j) x_{ij}$$

For internal units:

$$\Delta w_{ij} = \eta \delta_j x_{ij}$$

$$\delta_j = o_j (1 - o_j) \sum_{k \in Downstream(j)} \delta_k w_{jk}$$

Backpropagation Algorithm

- For simplicity, the following algorithm is for a two-layer neural network, with one output layer and one hidden layer
  - Thus, Downstream(j) = outputs for any internal node j
  - Note: Any boolean function can be represented by a two-layer neural network!

```
BACKPROPAGATION(training_examples, $\eta, n_{\text{in}}, n_{out}, n_{hidden}$)

Create a feed-forward network with $n_{\text{in}}$ inputs, $n_{hidden}$ units in the hidden layer,
and $n_{out}$ output units
Initialize all the network weights to small random numbers
(e.g. between -.05 and .05
Until the termination condition is met, Do
--- *Propogate the input forward through the network :*
---Input the instance $\vec{x}$ to the network and compute the output $o_u$ for every
---unit u in the network
--- *Propogate the errors backward through the network*
---For each network output unit k, calculate its error term $\delta_k$
        $$\delta_k = o_k (1 - o_k)(t_k - o_k)$$
---For each hidden unit h, calculate its error term $\delta_h$
        $$\delta_h = o_h (1 - o_h) \sum_{k \in outputs} w_{hk} d_k$$
---Update each network weight $w_{ij}$
        $$w_{ij} = w_{ij} + \eta \delta_j x_{ij}$$
```

Momentum

- Add the a fraction $0 <= \alpha < 1$ of the previous update for a weight to the current update
- May allow the learner to avoid local minimums
- May speed up convergence to global minimum

When to Stop Learning

- Learn until error on the training set is below some threshold
  - Bad idea! Can result in overfitting
    - If you match the training examples too well, your performance on the real problems may suffer
- Learn trying to get the best result on some validation data
  - Data from your training set that is not trained on, but instead used to check the function
  - Stop when the performance seems to be decreasing on this, while saving the best network seen so far.
  - There may be local minimums, so watch out!

Representational Capabilities

- Boolean functions – Every boolean function can be represented exactly by some network with two layers of units
  - Size may be exponential on the number of inputs
- Continuous functions – Can be approximated to arbitrary accuracy with two layers of units
- Arbitrary functions – Any function can be approximated to arbitrary accuracy with three layers of units

Example: Face Recognition

- From *Machine Learning by Tom M. Mitchell*
- *Input: 30 by 32 pictures of people with the following properties:*
  - *Wearing eyeglasses or not*
  - *Facial expression: happy, sad, angry, neutral*
  - *Direction in which they are looking: left, right, up, straight ahead*
- *Output: Determine which category it fits into for one of these properties (we will talk about direction)*

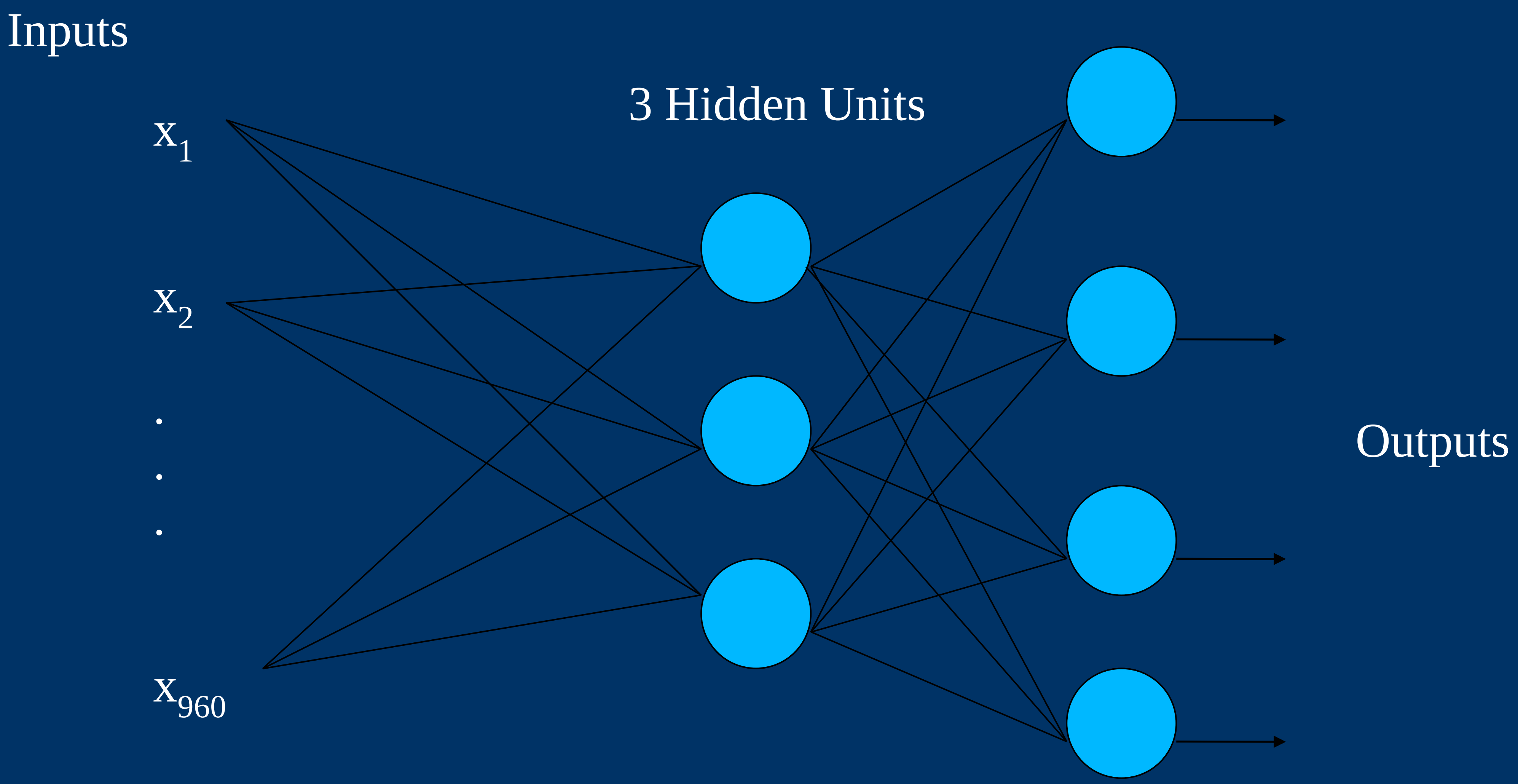
Input Encoding

- Each pixel is an input
  - 30*32 = 960 inputs
- The value of the pixel (0 – 255) is linearly mapped onto the range of reals between 0 and 1

Output Encoding

- Could use a single output node with the classifications assigned to 4 values (e.g. 0.2, 0.4, 0.6, and 0.8)
- Instead, use 4 output nodes (one for each value)
  - 1-of-N output encoding
  - Provides more degrees of freedom to the network
- Use values of 0.1 and 0.9 instead of 0 and 1
  - The sigmoid function can never reach 0 or 1!
- Example: (0.9, 0.1, 0.1, 0.1) = left, (0.1, 0.9, 0.1, 0.1) = right, etc.

Network structure



Other Parameters

- training rate: $\eta = 0.3$
- momentum: $\alpha = 0.3$
- Used full gradient descent (as opposed to stochastic)
- Weights in the output units were initialized to small random variables, but input weights were initialized to 0
  - Yields better visualizations
- Result: 90% accuracy on test set!

Try it yourself!

- Get the code from http://www.cs.cmu.edu/~tom/mlbook.html
  - Go to the Software and Data page, then follow the “Neural network learning to recognize faces” link
  - Follow the documentation
- You can also copy the code and data from my ACM account (provide you have one too), although you will want a fresh copy of facetrain.c and imagenet.c from the website
  - /afs/acm.uiuc.edu/user/jcander1/Public/NeuralNetwork