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 a 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 best encodes the variation among known face images.
- The face space is defined by the "eigenfaces", which are the eigenvectors of the set of faces.
Eigenfaces (1)

- **Calculation of Eigenfaces**
  1. Calculate **average face**: \( \bar{v} \).
  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.
  3. The **eigenvectors of covariance matrix** \( C \) (M by M) give the eigenfaces.
     - M is usually big, so this process would be time consuming.

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

What to do?

\[
C = AA^T
\]
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 \(N\)).

If \(\lambda_i\) are the eigenvectors of \(L\) then \(A \lambda_i\) are the eigenvectors for \(C\).

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

\[ L = A^T A \]
**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 $u$:

$$u = \sum_i a_i \phi_i$$

Since the eigenvectors are orthogonal:

$$a_i = u^T \phi_i$$
Fisherfaces

- Fisherfaces is developed by the statistical discrimination theory.
- Once the weight matrix is obtained, the computation is similar to eigenfaces.
- An unknown 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 eigenfaces and fisherfaces require 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 $(x_1...x_n, \text{output})$
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, \( y = w_0x_0 + w_1x_1 + ... + w_nx_n \)
  - A threshold function, i.e. 1 if \( y > 0 \), -1 if \( y \leq 0 \)
$\sum_{i=0}^{n} w_i x_i$

$y = \sum_{i=0}^{n} w_i x_i$

Threshhold

1 if $y > 0$
-1 otherwise
Linear Separator

This...

But not this (XOR)
Boolean Functions

Thus all boolean functions can be represented by layers of perceptrons!
The Perceptron Training Rule is given by:

\[ w_i := w_i \pm \eta (t - o)x_i \]

- \( w_i \): The weight of input \( i \)
- \( \eta \): The 'learning rate' between 0 and 1
- \( t \): The target output
- \( o \): The actual output
- \( x_i \): The \( i \)th input

This rule updates the weights based on the difference between the target output and the actual output, scaled by the learning rate.
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**: 
\[ E(\mathbf{x}) = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \]

\[ w_i = w_i - \frac{\partial E}{\partial w_i} \]

\[ w_i = \sum_{d \in D} (t_d - o_d x_{id}) \]
Gradient Descent Algorithm

GRADIENT-DESCENT(training_examples, )

Each training example is a pair of the form (x, t) where x is the vector of input values, and t is the target output value, is learning rate (0< <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 x to the unit and compute the output o

--------For each linear unit weight $w_i$, Do

\[ w_i = w_i t - o x_i \]

----For each linear unit $w_i$, Do

\[ w_i = w_i w_i \]
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
  - $\eta$ 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: \( o = \frac{1}{1 + e^{-y}} \)

\( \frac{\partial y}{\partial y} = y(1 - y) \)
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
- $\text{Downstream}(j) =$ the set of units whose immediate inputs include the output of unit $j$
Backpropagation Rule

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

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

For output units:

$$w_{ij} = t_j - o_j o_j 1 - o_j x_{ij}$$

For internal units:

$$w_{ij} = j x_{ij}$$

$$= o_j 1 - o_j \sum_{k \in \text{Downstream of } j} w_{jk}$$
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, \( n_{in}, n_{out}, n_{hidden} \))

Create a feed-forward network with \( n_{in} \) 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 \( e_k \)

\[
e_k = o_k (1 - o_k) (t_k - o_k)
\]

--- For each hidden unit \( h \), calculate its error term \( e_h \)

\[
e_h = o_h (1 - o_h) \sum_{k \in \text{outputs}} w_{hk} d_k
\]

--- Update each network weight \( w_{ij} \)

\[
w_{ij} = w_{ij} (1 - x_{ij}) x_{ij}
\]
Momentum

- Add the a fraction $0 \leq \alpha < 1$ 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
  - $30 \times 32 = 960$ 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