Introduction

Consider humans:
- Total number of neurons $\approx 10^{10}$
- Neuron switching time $\approx 10^{-3}$ second (vs. $10^{-10}$)
- Connections per neuron $\approx 10^3$–$10^5$
- Scene recognition time $\approx 0.1$ second
- 100 inference steps doesn’t seem like enough
  $\Rightarrow$ much parallel computation

Properties of artificial neural nets (ANNs):
- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
- Emphasis on tuning weights automatically

Strong differences between ANNs for ML and ANNs for biological modeling

When to Consider ANNs

- Input is high-dimensional discrete- or real-valued (e.g., raw sensor input)
- Output is discrete- or real-valued
- Output is a vector of values
- Possibly noisy data
- Form of target function is unknown
- Human readability of result is unimportant
- Long training times acceptable
**Perceptron Training Rule**

\[ w_{j+1}^i = w_j^i + \Delta w_j^i, \text{ where } \Delta w_j^i = \eta (y^i - y^i) x_j^i \]

- \( r^i \) is label of training instance \( i \)
- \( y^i \) is perceptron output on training instance \( i \)
- \( \eta \) is small constant (e.g., 0.1) called learning rate

I.e., if \((y^i - y^i) > 0\) then increase \( w_{j+1}^i \) w.r.t. \( x_j^i \), else decrease

Can prove rule will converge if training data is linearly separable and \( \eta \) sufficiently small

**Where Does the Training Rule Come From?**

- Consider simpler linear unit, where output

\[ y^i = w_0^i + w_1^i x_1^i + \ldots + w_n^i x_n^i \]

(i.e., no threshold)

- For each example, want to compromise between correctness and conservativeness
  1. Correctiveness: Tendency to improve on \( x^i \) (reduce error)
  2. Conservativeness: Tendency to keep \( w_j^{i+1} \) close to \( w_j^i \) (minimize distance)

- Use cost function that measures both:

\[
U(w) = \text{dist} (w^{i+1}, w^i) + \eta \text{ error } (y^i, w_j^{i+1} \cdot x^i)
\]

**Gradient Descent**

\[
E(w) = \frac{1}{2} \sum_{i=1}^{n} (y^i - y^i)^2
\]

\[
\Delta w_j = -\eta \frac{\partial E}{\partial w_j}, \forall i
\]

\[
w_{j+1} = w_j + \Delta w_j
\]

Approximate with

\[
0 = 2 \left( w_{j+1}^i - w_j^i \right) - 2 \eta \left( r^i - \sum_{j=1}^{n} w_j^{i+1} x_j^i \right) x_j^i
\]

which yields

\[
w_j^{i+1} = w_j^i + \eta (r^i - y^i) x_j^i
\]
**Implementation Approaches**

- Can use rules on previous slides on an example-by-example basis, sometimes called *incremental, stochastic, or on-line GD*
  - Has a tendency to “jump around” more in searching, which helps avoid getting trapped in local minima
- Alternatively, can use *standard or batch GD*, in which the classifier is evaluated over all training examples, summing the error, and then updates are made
  - I.e., sum up $\Delta w_i$ for all examples, but don’t update $w_i$ until summation complete
  - This is an inherent averaging process and tends to give better estimate of the gradient

**Handling Nonlinearly Separable Problems**

**The XOR Problem**

Represent with *intersection* of two linear separators

$$g_1(x) = 1 \cdot x_1 + 1 \cdot x_2 - 1/2$$

$$g_2(x) = 1 \cdot x_1 + 1 \cdot x_2 - 3/2$$

$$\text{pos} = \{ x \in \mathbb{R}^2 : g_1(x) > 0 \ \text{AND} \ g_2(x) < 0 \}$$

$$\text{neg} = \{ x \in \mathbb{R}^2 : g_1(x), g_2(x) < 0 \ \text{OR} \ g_1(x), g_2(x) > 0 \}$$

In other words, we remapped all vectors $x$ to $z$ such that the classes are linearly separable in the new vector space

This is a two-layer perceptron or two-layer feedforward neural network

Each neuron outputs 1 if its weighted sum exceeds its threshold, 0 otherwise

**The Sigmoid Unit**

$$\sigma(\text{net})$$ is the *logistic function*

$$\frac{1}{1 + e^{-\text{net}}}$$

*Squashes* $\text{net}$ into $[0, 1]$ range

Nice property:

$$\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$$

Continuous, differentiable approximation to threshold
Sigmoid Unit

Gradent Descent

Again, use squared error for correctness:

$$E(w') = \frac{1}{2} (y' - y')^2$$

(folding 1/2 of correctness into error func)

Thus

$$\frac{\partial E}{\partial w'_j} = \frac{\partial}{\partial w'_j} \frac{1}{2} (y' - y')^2$$

$$= \frac{1}{2} 2 (y' - y') \frac{\partial}{\partial w'_j} (y' - y') = (y' - y') \left( \frac{\partial y'}{\partial w'_j} \right)$$

SVMs

Problems

Separable

Nonlinearly

Introduction

Stephen Scott

Outline

Introduction

Outline

The Perceptron

Nonlinearly

Separable

Problems

Backprop Alg

Networks

Multilayer Networks

Sigmoid Unit

Remarks

Overfitting

Training Multilayer Networks

Gradient Descent

Output Units

Adjust weight $w'_j$ according to $E'$ as before

For output units, this is easy since contribution of $w'_j$ to $E'$ when $j$ is an output unit is the same as for single neuron case\(^1\), i.e.,

$$\frac{\partial E'}{\partial w'_j} = -(y' - y_j) y_j (1 - y_j) x'_j = -\delta'_j x'_j$$

where $\delta'_j = -\frac{\partial y}{\partial w'_j}$ is error term of unit $j$

\(^1\)This is because all other outputs are constants w.r.t. $w'_j$

Multilayer Networks

Input layer

$\mathbf{x}_i = \text{input from } i \text{ to } j$

$w_{ji} = \text{wt from } i \text{ to } j$

Hidden layer

$\mathbf{x}_n = \text{input from } i \text{ to } j$

$w_{ni} = \text{wt from } i \text{ to } j$

Output layer

$\mathbf{y}_j = \text{output layer}$

Use sigmoid units since continuous and differentiable

$$E' = E(w') = \frac{1}{2} \sum_{k}\left( y'_k - y'_k \right)^2$$

Training Multilayer Networks

Hidden Units

- How can we compute the error term for hidden layers when there is no target output $y'$ for these layers?
- Instead propagate back error values from output layer toward input layers, scaling with the weights
- Scailing with the weights characterizes how much of the error term each hidden unit is “responsible for”

Training Multilayer Networks

Hidden Units (cont'd)

The impact that $w'_j$ has on $E'$ is only through $net'_j$ and units immediately “downstream” of $j$:

$$\frac{\partial E'}{\partial w'_j} = \frac{\partial E'}{\partial net'_j} \frac{\partial net'_j}{\partial w'_j} = \frac{x'_j}{k \in \text{down}(j)} \frac{\partial E'}{\partial net'_k} \frac{\partial net'_k}{\partial w'_j}$$

$$= x'_j \sum_{k \in \text{down}(j)} -\delta'_k \frac{\partial net'_k}{\partial w'_j} = x'_j \sum_{k \in \text{down}(j)} -\delta'_k \frac{\partial net'_k}{\partial y'_j} \frac{\partial y'_j}{\partial w'_j}$$

$$= x'_j \sum_{k \in \text{down}(j)} -\delta'_k w_{kj} y'_j = x'_j \sum_{k \in \text{down}(j)} -\delta'_k w_{kj} y'_j (1 - y'_j)$$

Works for arbitrary number of hidden layers
Backpropagation Algorithm

Initialize all weights to small random numbers

Until termination condition satisfied do
- For each training example \((i, x_i)\) do
  - Input \(x_i\) to the network and compute the outputs \(y_i\)
  - For each output unit \(k\)
    - \(\delta_k^i \leftarrow y_k^i (1 - y_k^i) (y_k^i - y_k^\text{target})\)
  - For each hidden unit \(h\)
    - \(\delta_h^i \leftarrow y_h^i (1 - y_h^i) \sum_j w_{hj} \delta_j^i\)
- Update each network weight \(w_{ij}\)
  - \(w_{ij}^t \leftarrow w_{ij}^t + \eta \delta_j^i y_i\)

Remarks

- When to stop training? When weights don’t change much, error rate sufficiently low, etc. (be aware of overfitting: use validation set)
- Cannot ensure convergence to global minimum due to myriad local minima, but tends to work well in practice (can re-run with new random weights)
- Generally training very slow (thousands of iterations), use is very fast
- Setting \(\eta\): Small values slow convergence, large values might overshoot minimum, can adapt it over time

Remarks (cont’d)

- Alternative error function: cross entropy
  \[
  E = \sum_{k=\text{outputs}} (y_k^i \ln y_k^i + (1 - y_k^i) \ln (1 - y_k^i))
  \]
- “Blows up” if \(y_k \approx 1\) and \(1 - y_k \approx 0\) or vice-versa (vs. squared error, which is always in \([0, 1]\))
- Regularization: penalize large weights to make space more linear and reduce risk of overfitting:
  \[
  E' = \frac{1}{2} \sum_{k=\text{outputs}} (y_k^i - y_k^\text{target})^2 + \gamma \sum_{i,j} (w_{ij})^2
  \]

Backpropagation Algorithm Example

\[
\begin{align*}
\text{target} &= y \\
\text{trial 1:} & \ a = 1, \ b = 0, \ y = 1 \\
\text{trial 2:} & \ a = 0, \ b = 1, \ y = 0 \\
\end{align*}
\]

\[
\begin{align*}
\text{Input:} & \quad x = \begin{bmatrix} a \ c \ b \ w_{x0} \end{bmatrix} \\
\text{Output:} & \quad y = \begin{bmatrix} y_c \ end{bmatrix} \\
\text{Output for trial 1:} & \quad \begin{bmatrix} 0.0028376 \\ -0.004005 \\ 0.1146431 \\ 0.5500447 \\ 0.1008513 \\ 0.1189104 \\ 0.1189104 \\ 0.5500447 \\ 0.1008513 \end{bmatrix} \\
\text{Output for trial 2:} & \quad \begin{bmatrix} 0.0935679 \\ 0.0964548 \\ 0.0935679 \\ 0.0964548 \\ 0.1008513 \\ 0.0996498 \\ 0.1008513 \\ 0.0996498 \\ 0.1008513 \end{bmatrix} \\
\end{align*}
\]

Overfitting

Danger of stopping too soon!

Representational power:
- Any boolean function can be represented with 2 layers
- Any bounded, continuous function can be represented with arbitrarily small error with 2 layers
- Any function can be represented with arbitrarily small error with 3 layers

Number of required units may be large
May not be able to find the right weights
Hypothesis Space

- Hyp. space \( \mathcal{H} \) is set of all weight vectors (continuous vs. discrete of decision trees)
- Search via Backprop: Possible because error function and output functions are continuous & differentiable
- Inductive bias: (Roughly) smooth interpolation between data points

Support Vector Machines

Introduction

Similar to ANNs, polynomial classifiers, and RBF networks in that it remaps inputs and then finds a hyperplane

- Main difference is how it works

Features of SVMs:

- Maximization of margin
- Duality
- Use of kernels
- Use of problem convexity to find classifier (often without local minima)

Support Vector Machines

Margins

- A hyperplane's margin \( \gamma \) is the shortest distance from it to any training vector
- Intuition: larger margin \( \Rightarrow \) higher confidence in classifier's ability to generalize
  - Guaranteed generalization error bound in terms of \( 1/\gamma^2 \) (under appropriate assumptions)
- Definition assumes linear separability (more general definitions exist that do not)

Support Vector Machines

The Perceptron Algorithm Revisited

The Perceptron Algorithm Revisited (partial example)

\[ w_0 \leftarrow 0, \ b_0 \leftarrow 0, \ m \leftarrow 0, \ r' \in \{-1, +1\} \forall t \]

While mistakes are made on training set

- For \( t = 1 \) to \( N \) (= # training vectors)
  - If \( r' (w_m \cdot x + b_m) \leq 0 \)
    - \( w_{m+1} \leftarrow w_m + \eta r' x' \)
    - \( b_{m+1} \leftarrow b_m + \eta r' \)
    - \( m \leftarrow m + 1 \)

Final predictor: \( h(x) = \text{sgn} \ (w_m \cdot x + b_m) \)

Support Vector Machines

The Perceptron Algorithm Revisited (partial example)

At this point, \( w = (0.2, -0.2) \), \( b = -0.2 \), \( \alpha = (3, 0, 0, 5, 0, 0) \)

Can compute

\[ w_1 = \eta (\alpha_1 r_1 x_1^1 + \alpha_4 r_4 x_4^1) = 0.1(3(1)4 + 5(-1)2) = 0.2 \]
\[ w_2 = \eta (\alpha_1 r_1 x_1^1 + \alpha_4 r_4 x_4^1) = 0.1(3(1)1 + 5(-1)1) = -0.2 \]

i.e., \( w = \eta \sum_{t=1}^{N} \alpha_t r_t x' \)
Another way of representing predictor:

\[ \mathbf{h}(\mathbf{x}) = \text{sgn} \left( \mathbf{w} \cdot \mathbf{x} + b \right) = \text{sgn} \left( \eta \sum_{i=1}^{N} \alpha_i' \mathbf{x}_i \cdot \mathbf{x} + b \right) \]

\[ = \text{sgn} \left( \eta \sum_{i=1}^{N} \alpha_i' \mathbf{x}_i \cdot \mathbf{x} + b \right) \]

\[ (\alpha_i = \# \text{ prediction mistakes on } \mathbf{x}_i) \]

### XOR Revisited

**XOR Revisited**

Remap to new space:

\[ \phi(x_1, x_2) = [x_1^1, x_1^2, \sqrt{2} x_1 x_2, \sqrt{2} x_1, \sqrt{2} x_2, 1] \]

### XOR Revisited (cont’d)

- Can easily compute the dot product \( \phi(x) \cdot \phi(z) \) (where \( x = [x_1, x_2] \)) without first computing \( \phi \):

\[ K(x, z) = (\mathbf{x} \cdot \mathbf{z} + 1)^2 = (x_1 z_1 + x_2 z_2 + 1)^2 \]

\[ = (x_1 z_1)^2 + (x_2 z_2)^2 + 2 x_1 z_1 x_2 z_2 + 2 x_1 z_1 + 2 x_2 z_2 + 1 \]

\[ = \begin{bmatrix} x_1^2, x_2^2, \sqrt{2} x_1 x_2, \sqrt{2} x_1, \sqrt{2} x_2, 1 \end{bmatrix} \cdot \begin{bmatrix} x_1^2, x_2^2, \sqrt{2} x_1 x_2, \sqrt{2} x_1, \sqrt{2} x_2, 1 \end{bmatrix} \]

\[ \phi(x) \cdot \phi(z) \]

- i.e., since we use dot products in new Perceptron algorithm, we can implicitly work in the remapped \( y \) space via \( K \)

### Kernels

- A **kernel** is a function \( K \) such that \( \forall \mathbf{x}, \mathbf{z}, K(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x}) \cdot \phi(\mathbf{z}) \)

- E.g., previous slide (quadratic kernel)

- In general, for degree-\( q \) polynomial kernel, computing \( (\mathbf{x} \cdot \mathbf{z} + 1)^q \) takes \( \frac{q(q+1)}{2} \) multiplications + 1 exponentiation for \( \mathbf{x}, \mathbf{z} \in \mathbb{R}^d \)

- In contrast, need over \( \frac{(q+1)q}{2} \) multiplications if compute \( \phi \) first
Kernels (cont’d)

- Typically start with kernel and take the feature mapping that it yields
- E.g., Let \( \ell = 1, x = x, z = z, K(x, z) = \sin(x - z) \)
- By Fourier expansion,
  \[
  \sin(x-z) = a_0 + \sum_{n=1}^{\infty} a_n \sin(n x) \sin(n z) + \sum_{n=1}^{\infty} a_n \cos(n x) \cos(n z)
  \]
  for Fourier coefficients \( a_0, a_1, \ldots \)
- This is the dot product of two infinite sequences of nonlinear functions:
  \( \{\phi_i(x)\}_{i=0}^{\infty} = [\sin(x), \cos(x), \sin(2x), \cos(2x), \ldots] \)
- I.e., there are an infinite number of features in this remapped space!

Types of Kernels

- Gaussian
  \[
  K(x', x) = \exp\left(-\frac{||x' - x||^2}{2\sigma^2}\right)
  \]
  (a) \( \sigma^2 = 2 \)
  (b) \( \sigma^2 = 0.5 \)
  (c) \( \sigma^2 = 0.25 \)
  (d) \( \sigma^2 = 0.1 \)

- Hyperbolic tangent:
  \[
  K(x', x) = \tanh(2x' \cdot x + 1)
  \]
  (not a true kernel)

- Other
  Also have ones for structured data: e.g., graphs, trees, sequences, and sets of points
  In addition, the sum of two kernels is a kernel, the product of two kernels is a kernel
  Finally, note that a kernel is a similarity measure, useful in clustering, nearest neighbor, etc.

Support Vector Machines

Finding a Hyperplane

Can show that if data linearly separable in remapped space, then get maximum margin classifier by minimizing \( w \cdot w \) subject to \( r^T (w \cdot x + b) \geq 1 \)

Can reformulate this in dual form as a convex quadratic program that can be solved optimally, i.e., won’t encounter local optima:

\[
\text{maximize } \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{N} \alpha_i \alpha_j r^i r^j K(x^i, x^j)
\]

s.t.
\[
\alpha_i \geq 0, i = 1, \ldots, m
\]
\[
\sum_{i=1}^{N} \alpha_i r^i = 0
\]

After optimization, label new vectors with decision function:
\[
f(x) = \text{sgn}\left(\sum_{i=1}^{N} \alpha_i r^i K(x, x^i) + b\right)
\]
(Note only need to use \( x^i \) such that \( \alpha_i > 0 \), i.e., support vectors)

Can always find a kernel that will make training set linearly separable, but beware of choosing a kernel that is too powerful (overfitting)
Support Vector Machines
Finding a Hyperplane (cont'd)

If kernel doesn’t separate, can soften the margin with slack variables $\xi$:

$$\begin{align*}
\text{minimize} & \quad \|w\|^2 + C \sum_{i=1}^{N} \xi_i \\
\text{s.t.} & \quad r^i ((x_i \cdot w) + b) \geq 1 - \xi_i, \quad i = 1, \ldots, N \\
& \quad \xi_i \geq 0, \quad i = 1, \ldots, N
\end{align*}$$

The dual is similar to that for hard margin:

$$\begin{align*}
\text{maximize} & \quad \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j r^i r^j K(x_i, x_j) \\
\text{s.t.} & \quad 0 \leq \alpha_i \leq C, \quad i = 1, \ldots, N \\
& \quad \sum_{i=1}^{N} \alpha_i r^i = 0
\end{align*}$$

Can still solve optimally

Support Vector Machines
Finding a Hyperplane (cont'd)

If number of training vectors is very large, may opt to approximately solve these problems to save time and space

Use e.g., gradient ascent and sequential minimal optimization (SMO)

When done, can throw out non-SVs