II. Neural Network Learning

A. Neural Network Learning

Supervised Learning

- Produce desired outputs for training inputs
- Generalize reasonably & appropriately to other inputs
- Good example: pattern recognition
- Feedforward multilayer networks
Part 4A: Neural Network Learning

Feedforward Network

Typical Artificial Neuron

Typical Artificial Neuron
Equations

Net input:

\[ h_i = \left( \sum_{j=1}^{n} w_{ij} s_j \right) - \theta \]

\[ h = Ws - \theta \]

Neuron output:

\[ s_i' = \sigma(h_i) \]

\[ s' = \sigma(h) \]

Single-Layer Perceptron

Variables
Single Layer Perceptron Equations

Binary threshold activation function:

\[ a(h) = \Theta(h) = \begin{cases} 1, & \text{if } h > 0 \\ 0, & \text{if } h \leq 0 \end{cases} \]

Hence, \[ y = \begin{cases} 1, & \text{if } \sum_j w_j x_j > \theta \\ 0, & \text{otherwise} \end{cases} \]

\[ w_j x_j > \theta \iff w \cdot x > \theta \iff v > \theta \]

2D Weight Vector

\[ w \cdot x = |w||x| \cos \phi \]
\[ \cos \phi = \frac{v}{|v|} \]
\[ w \cdot x = |w||x| \]
\[ w \cdot x > \theta \iff |w||x| > \theta \iff |v| > \theta \]

N-Dimensional Weight Vector

normal vector

separating hyperplane
Goal of Perceptron Learning

- Suppose we have training patterns $x^1, x^2, ... , x^P$ with corresponding desired outputs $y^1, y^2, ... , y^P$.
- where $x^p \in \{0, 1\}^n, y^p \in \{0, 1\}$.
- We want to find $w, \theta$ such that $y^p = \Theta(w \cdot x^p - \theta)$ for $p = 1, ..., P$.

Treating Threshold as Weight

Let $x_0 = -1$ and $w_0 = \theta$.

\[ h = w_0 x_0 + \sum_{j=1}^n w_j x_j \]

\[ y = \Theta(h) \]

\[ h = \sum_{j=1}^n w_j x_j - \theta \]

\[ y = \Theta(h - \theta) \]

\[ h = \Theta(h) \]

\[ y = \Theta(h - \theta) \]
Augmented Vectors

\[
\tilde{w} = \begin{pmatrix} \theta \\ w_1 \\ \vdots \\ w_n \end{pmatrix}, \quad \tilde{x}^p = \begin{pmatrix} -1 \\ x_1^p \\ \vdots \\ x_n^p \end{pmatrix}
\]

We want \( y^p = \Theta(\tilde{w} \cdot \tilde{x}^p) \), \( p = 1, \ldots, P \)

Reformulation as Positive Examples

We have positive \((y^p = 1)\) and negative \((y^p = 0)\) examples

Want \( \tilde{w} \cdot \tilde{x}^p > 0 \) for positive, \( \tilde{w} \cdot \tilde{x}^p \leq 0 \) for negative

Let \( z^p = \tilde{x}^p \) for positive, \( z^p = -\tilde{x}^p \) for negative

Want \( \tilde{w} \cdot z^p \geq 0 \), for \( p = 1, \ldots, P \)

Hyperplane through origin with all \( z^p \) on one side

Adjustment of Weight Vector
Outline of Perceptron Learning Algorithm

1. initialize weight vector randomly
2. until all patterns classified correctly, do:
   a) for \( p = 1, \ldots, P \) do:
      1) if \( x^p \) classified correctly, do nothing
      2) else adjust weight vector to be closer to correct classification

Weight Adjustment

\[
\tilde{w}' \cdot z^p = (\tilde{w} + \eta z^p) \cdot z^p
\]

Improvement in Performance

\[
\tilde{w}' \cdot z^p = (\tilde{w} + \eta z^p) \cdot z^p
= \tilde{w} \cdot z^p + \eta z^p \cdot z^p
= \tilde{w} \cdot z^p + \eta \| z^p \|^2
> \tilde{w} \cdot z^p
\]
Perceptron Learning Theorem

- If there is a set of weights that will solve the problem,
- then the PLA will eventually find it
- (for a sufficiently small learning rate)
- Note: only applies if positive & negative examples are linearly separable

NetLogo Simulation of Perceptron Learning

Run Perceptron-Geometry.nlogo

Classification Power of Multilayer Perceptrons

- Perceptrons can function as logic gates
- Therefore MLP can form intersections, unions, differences of linearly-separable regions
- Classes can be arbitrary hyperpolyhedra
- Minsky & Papert criticism of perceptrons
- No one succeeded in developing a MLP learning algorithm
Hyperpolyhedral Classes

Credit Assignment Problem
How do we adjust the weights of the hidden layers?

NetLogo Demonstration of Back-Propagation Learning

Run Artificial Neural Net.nlogo
Adaptive System

Gradient

\[ \nabla F = \left( \frac{\partial F}{\partial P_1}, \frac{\partial F}{\partial P_2}, \ldots, \frac{\partial F}{\partial P_m} \right) \]

\( \nabla F \) points in direction of maximum local increase in \( F \)

Gradient Ascent on Fitness Surface
Gradient Ascent by Discrete Steps

Gradient Ascent is Local But Not Shortest

Gradient Ascent Process

\[ \dot{P} = \eta \nabla F(P) \]

Change in fitness:

\[ \dot{F} = \frac{dF}{dt} = \sum_{i=1}^{n} \frac{\partial F}{\partial P_i} \frac{dP_i}{dt} = \sum_{i=1}^{n} (\nabla F)_i \dot{P}_i \]

\[ F = \nabla F \cdot \dot{P} \]

\[ \dot{F} = \nabla F \cdot \eta \nabla F = \eta \| \nabla F \|^2 \geq 0 \]

Therefore gradient ascent increases fitness (until reaches 0 gradient)
General Ascent in Fitness
Note that any adaptive process $P(t)$ will increase fitness provided:

$$0 < F = \nabla F \cdot P = \|\nabla F\| P \cos \varphi$$

where $\varphi$ is angle between $\nabla F$ and $P$

Hence we need $\cos \varphi > 0$

or $|\varphi| < 90^\circ$

General Ascent on Fitness Surface

Fitness as Minimum Error
Suppose for $Q$ different inputs we have target outputs $t^1, \ldots, t^Q$

Suppose for parameters $P$ the corresponding actual outputs are $y^1, \ldots, y^Q$

Suppose $D(t, y) \in [0, \infty)$ measures difference between target & actual outputs

Let $E^q = D(t^q, y^q)$ be error on $q$th sample

Let $F(P) = -\sum_{q=1}^Q E^q(P) = -\sum_{q=1}^Q D[t^q, y^q(P)]$
Gradient of Fitness

\[ \nabla F = - \sum_{i} E^i = - \sum_{i} \nabla^i \]

\[ \frac{\partial E^i}{\partial P_k} = \frac{\partial}{\partial P_k} D(t^i, y^i) = \sum_{j} \frac{\partial D(t^i, y^i)}{\partial y_j} \frac{\partial y_j}{\partial P_k} \]

\[ = \frac{d D(t^i, y^i)}{d y^i} \frac{\partial y_j}{\partial P_k} \]

\[ = \nabla_{y^i} D(t^i, y^i) \frac{\partial y_j}{\partial P_k} \]

\[ \nabla_y D(t^i, y^i) \frac{\partial y_j}{\partial P_k} \]

Jacobian Matrix

Define Jacobian matrix \( J = \left[ \frac{\partial y_j}{\partial P_i} \right] \)

Note \( J \in \mathbb{R}^{n \times m} \) and \( \nabla D(t^i, y^i) \in \mathbb{R}^{m} \)

Since \( (\nabla^i E) = \frac{\partial E^i}{\partial P_i} = \sum_{j} \frac{\partial D(t^i, y^i)}{\partial y_j} \frac{\partial y_j}{\partial P_i} \)

\[ \therefore \nabla^i E = (J)^T \nabla D(t^i, y^i) \]

Derivative of Squared Euclidean Distance

Suppose \( D(t, y) = \| t - y \|^2 = \sum (t_i - y_i)^2 \)

\[ \frac{\partial D(t, y)}{\partial y_j} = \frac{\partial}{\partial y_j} \sum (t_i - y_i)^2 = \sum \frac{\partial (t_i - y_i)^2}{\partial y_j} \]

\[ = \frac{d(t_i - y_i)^2}{d y_j} = -2(t_i - y_i) \]

\[ \therefore \frac{d D(t, y)}{d y} = 2(y - t) \]
Gradient of Error on $q^{th}$ Input

\[
\frac{\partial E^q}{\partial P_k} = \frac{dD(t^q, y^q)}{dy^q} \frac{\partial y^q}{\partial P_k} = 2(y^q - t^q) \frac{\partial y^q}{\partial P_k}
\]

\[
\nabla E^q = 2J^q (y^q - t^q)
\]

Recap

\[
P = \eta \sum_j \left( J^j \right)^T (t^q - y^q)
\]

To know how to decrease the differences between actual & desired outputs, we need to know elements of Jacobian, \( \frac{\partial y^j}{\partial P_k} \), which says how jth output varies with kth parameter (given the qth input).

The Jacobian depends on the specific form of the system, in this case, a feedforward neural network.

Multilayer Notation
Notation

- $L$ layers of neurons labeled 1, ..., $L$
- $N_l$ neurons in layer $l$
- $s^l$: vector of outputs from neurons in layer $l$
- input layer $s^1 = x^q$ (the input pattern)
- output layer $s^L = y^q$ (the actual output)
- $W^l$: weights between layers $l$ and $l+1$
- Problem: find out how outputs $y^q_i$ vary with weights $W^l_{jk}$ ($l = 1, \ldots, L-1$)

Typical Neuron

Error Back-Propagation

We will compute $\frac{\partial E^q}{\partial W^l_{jk}}$ starting with last layer ($l = L-1$) and working back to earlier layers ($l = L-2, \ldots, 1$)
Delta Values

Convenient to break derivatives by chain rule:

$$\frac{\partial E}{\partial W_{ij}} = \frac{\partial E}{\partial h_i} \frac{\partial h_i}{\partial W_{ij}}$$

Let \( \delta_i = \frac{\partial E}{\partial h_i} \)

So \( \frac{\partial E}{\partial W_{ij}} = \delta_i \frac{\partial h_i}{\partial W_{ij}} \)

Output-Layer Neuron

Output-Layer Derivatives (1)

$$\delta_i = \frac{\partial E}{\partial h_i} = \frac{\partial}{\partial h_i} \sum (s_i - t_i)^2$$

$$= \frac{d(s_i - t_i)}{dh_i} = 2(s_i - t_i) \frac{dx_i}{dh_i}$$

$$= 2(s_i - t_i) \sigma'(h_i)$$
Output-Layer Derivatives (2)

\[ \frac{\partial h_i^l}{\partial W_{ij}^{l-1}} = \frac{\partial}{\partial W_{ij}^{l-1}} \sum_k W_{ik}^{l-1} s_j^{l-1} = s_j^{l-1} \]

\[ \therefore \frac{\partial E_q}{\partial W_{ij}^{l-1}} = \delta_i^l s_j^{l-1} \]

where \( \delta_i^l = 2(s_i^l - t_i^q) \sigma'(h_i^l) \)

Hidden-Layer Derivatives (1)

Recall \( \frac{\partial E_q}{\partial W_{ij}^{l+1}} = \delta_i^l \frac{\partial h_i^l}{\partial W_{ij}^{l+1}} \)

\[ \delta_i^l = \frac{\partial E_q}{\partial h_i^l} = \sum_j \frac{\partial E_q}{\partial h_j^{l+1}} \frac{\partial h_j^{l+1}}{\partial h_i^l} = \sum_j \delta_j^{l+1} \frac{\partial h_j^{l+1}}{\partial h_i^l} \]

\[ \frac{\partial h_i^{l+1}}{\partial h_i^l} = \frac{1}{2} \sum_k W_{ik}^{l+1} s_k^l - W_{ik}^l \frac{d}{dh_i^l} W_{ik}^l \]

\[ \therefore \delta_i^l = \sum_k \delta_j^{l+1} W_{ik}^l \sigma'(h_i^l) = \sigma'(h_i^l) \sum_k \delta_j^{l+1} W_{ik}^l \]
Hidden-Layer Derivatives (2)

\[
\frac{\partial h_i}{\partial W_{ij}} = \frac{\partial}{\partial W_{ij}} \sum_k W_{ki}^l s_k^{l-1} = \sum_k \frac{\partial W_{ki}^l}{\partial W_{ij}} s_k^{l-1} = s_j^{l-1}
\]

\[
\therefore \frac{\partial E}{\partial W_{ij}} = \delta_i^l s_j^{l-1}
\]

where \( \delta_i^l = \sigma'(h_i^l) \sum \delta_k^{l+1} W_{ki}^l \)

---

**Derivative of Sigmoid**

Suppose \( s = \sigma(h) = \frac{1}{1 + \exp(-ah)} \) (logistic sigmoid)

\[
D_a s = D_a \left[ \frac{1}{1 + \exp(-ah)} \right] = \left[ -\frac{1}{1 + \exp(-ah)} \right] D_a (1 + e^{-ah})
\]

\[
= -\left(1 + e^{-ah}\right)^{-1} (-ae^{-ah}) = ae^{-ah} \left(1 + e^{-ah}\right)
\]

\[
= \left(1 + e^{-ah}\right)^{-1} e^{-ah} = e^{-ah} \left(1 + e^{-ah} \right) + \left(1 + e^{-ah} \right)
\]

\[
= e^{-ah} \left(-1 \right)
\]

---

**Summary of Back-Propagation Algorithm**

Output layer: \( \delta_i^L = 2 \sigma'(1 - s_i^L) (s_i^L - t_i^L) \)

\[
\frac{\partial E}{\partial W_{ij}^{L-1}} = \delta_i^L s_j^{L-1}
\]

Hidden layers: \( \delta_i^l = \alpha s_i^l \sum \delta_k^{l+1} W_{ki}^l \)

\[
\frac{\partial E}{\partial W_{ij}^l} = \delta_j^{l+1} s_i^l
\]
Part 4A: Neural Network Learning

Output-Layer Computation

\[ \Delta W_{L-1}^{L} = \eta \delta_i s_j^{L} \]

\[ \delta_i^L = 2\alpha \epsilon_i^L (1 - s_i^L) (t_i^L - s_i^L) \]

Hidden-Layer Computation

\[ \Delta W_{l}^{l+1} = \eta \delta_i^{l+1} s_j^{l} \]

\[ \delta_i^{l+1} = \alpha \epsilon_i^{l+1} \sum \delta_j^{l} W_{ij}^{l+1} \]

Training Procedures

- **Batch Learning**
  - on each epoch (pass through all the training pairs),
  - weight changes for all patterns accumulated
  - weight matrices updated at end of epoch
  - accurate computation of gradient
- **Online Learning**
  - weight are updated after back-prop of each training pair
  - usually randomize order for each epoch
  - approximation of gradient
- Doesn’t make much difference
Summation of Error Surfaces

Gradient Computation in Batch Learning

Gradient Computation in Online Learning
Part 4A: Neural Network Learning

Testing Generalization

Problem of Rote Learning

Improving Generalization
A Few Random Tips

- Too few neurons and the ANN may not be able to decrease the error enough
- Too many neurons can lead to rote learning
- Preprocess data to:
  - standardize
  - eliminate irrelevant information
  - capture invariances
  - keep relevant information
- If stuck in local min., restart with different random weights

Run Example BP Learning

Beyond Back-Propagation

- Adaptive Learning Rate
- Adaptive Architecture
  - Add/delete hidden neurons
  - Add/delete hidden layers
- Radial Basis Function Networks
- Recurrent BP
- Etc., etc., etc…
Deep Belief Networks

- Inspired by hierarchical representations in mammalian sensory systems
- Use “deep” (multilayer) feed-forward nets
- Layers self-organize to represent input at progressively more abstract, task-relevant levels
- Supervised training (e.g., BP) can be used to tune network performance.
- Each layer is a Restricted Boltzmann Machine

Restricted Boltzmann Machine

- Goal: hidden units become model of input domain
- Should capture statistics of input
- Evaluate by testing its ability to reproduce input statistics
- Change weights to decrease difference

Unsupervised RBM Learning

- Stochastic binary units
- Assume bias units: \( x_0 = y_0 = 1 \)
- Set \( y_i \) with probability: \( \sigma \left( \sum_j w_{ij} x_j \right) \)
- Set \( x_i \) with probability: \( \sigma \left( \sum_j w_{ij} y_j \right) \)
- Set \( y_i' \) with probability: \( \sigma \left( \sum_j w_{ij} x_j' \right) \)
- After several cycles of sampling, update weights based on statistics: \( \Delta w_{ij} = \eta \left( \langle y_i x_j \rangle - \langle y_i' x_j' \rangle \right) \)
Training a DBN Network

- Present inputs and do RBM learning with first hidden layer to develop model
- When converged, do RBM learning between first and second hidden layers to develop higher-level model
- Continue until all weight layers trained
- May further train with BP or other supervised learning algorithms

What is the Power of Artificial Neural Networks?

- With respect to Turing machines?
- As function approximators?

Can ANNs Exceed the “Turing Limit”?  

- There are many results, which depend sensitively on assumptions; for example:
  - Finite NNs with real-valued weights have super-Turing power (Siegelmann & Sontag ’94)
  - Recurrent nets with Gaussian noise have sub-Turing power (Maass & Sontag ’99)
  - Finite recurrent nets with real weights can recognize all languages, and thus are super-Turing (Siegelmann ’99)
  - Stochastic nets with rational weights have super-Turing power (but only P/POLY, BPP/log”) (Siegelmann ’99)
  - But computing classes of functions is not a very relevant way to evaluate the capabilities of neural computation
A Universal Approximation Theorem
Suppose $f$ is a continuous function on $[0,1]^n$. Suppose $\sigma$ is a nonconstant, bounded, monotone increasing real function on $\mathbb{R}$. For any $\varepsilon > 0$, there is an $m$ such that there exist $a \in \mathbb{R}^m$, $b \in \mathbb{R}^n$, $W \in \mathbb{R}^{m \times n}$ such that if

$$F(x_1, \ldots, x_n) = \sum_{i=1}^m a_i \left( \sum_{j=1}^n W_{ij} x_j + b_i \right)$$

[\text{i.e., } F(x) = a \cdot \sigma(Wx + b)]

then $|F(x) - f(x)| < \varepsilon$ for all $x \in [0,1]^n$.

(see, e.g., Haykin, N.Nets 2/e, 208–9)

One Hidden Layer is Sufficient
• Conclusion: One hidden layer is sufficient to approximate any continuous function arbitrarily closely.

The Golden Rule of Neural Nets
Neural Networks are the second-best way to do everything!