Neural Networks

(continued)
Perceptron

- A perceptron is the simplest type of ANN

\[ \sum_{i=0}^{n} w_i x_i \]

- \( x_1, \ldots, x_n \): inputs
- \( w_1, \ldots, w_n \): weights
- \(-w_0\): threshold
- \( x_0 = 1 \): additional constant input

Perceptron learning

- Given a training set \( X = \{(x, \text{out}_x)\} \)
  - \( x \) is the input vector
  - \( \text{out}_x \) is the desired output value (i.e., -1 or 1)
- Determine a weight vector \( w \) that makes the perceptron produce the correct output (-1 or 1) for every training instance
- The hypothesis space is \( H = \{ w \mid w \in \mathbb{R}^{n+1} \} \)
- How to learn: idea
  - If a training instance \( x \) is correctly classified, then no (weight) update is needed
  - If \( \text{out}_x = 1 \) but the perceptron outputs -1, then the weight \( w \) should be updated so that \( \sum w_i x_i \) is increased
  - If \( \text{out}_x = -1 \) but the perceptron outputs 1, then the weight \( w \) should be updated so that \( \sum w_i x_i \) is decreased
Perceptron training rule

Perceptron training rule \((X, \eta)\)
- **initialize** \(w\) (w \(\leftarrow\) an initial (small) random value)
- repeat
  - for each training instance \((x, out_x) \in X\)
    - compute the real output \(o_x = sgn(w \cdot x)\)
    - if (\(out_x \neq o_x\))
      - for each \(w_i\)
        - \(w_i \leftarrow w_i + \eta(out_x - o_x)x_i\)
      - end for
    - end if
  - end for
- until all the training instances in \(X\) are correctly classified
- return \(w\)

**Examples**
- \(x\) is correctly classified, \(out_x - o_x = 0\)
  - \(\Rightarrow\) no update
- \(o_x = -1\) but \(out_x = 1\), \(out_x - o_x > 0\)
  - \(\Rightarrow\) \(w_i\) is increased if \(x_i > 0\), decreased otherwise
  - \(\Rightarrow\) \(w \cdot x\) is increased
- \(o_x = 1\), but \(out_x = -1\), \(out_x - o_x < 0\)
  - \(\Rightarrow\) \(w_i\) is decreased if \(x_i > 0\), increased otherwise
  - \(\Rightarrow\) \(w \cdot x\) is decreased

---

Perceptron training rule - limitation

- The perceptron training rule is proven to converge if
  - the training instances in \(X\) are linearly separable
  - a sufficiently small \(\eta\) is used
- The perceptron training rule does not converge if the training instances are not linearly separable
  - A perceptron cannot correctly classify this training set!
- We need to use gradient descent or the delta rule
  - Converge toward a best-fit approximation of the target function regardless if the training examples are linearly separable
Gradient descent and delta rule

- Let’s consider a linear unit

\[ o = w \cdot x \]

No threshold, differentiable function

- Training error

\[ E(w) = \frac{1}{2} \sum_{x \in X} (out_x - o_x)^2 \]

The error \( E \) depends only on \( w \) because the training set \( X \) is fixed

Gradient descent – Illustration

- The \( w_0, w_1 \) plane is the hypothesis space
- The error \( E(w_0, w_1) \) is function of the 2 weights
- For linear units, the error surface is parabolic with single global minimum
- The arrow shows the negated gradient at one point, the steepest descent along the error surface
- The weights should be updated along the negated gradient
Gradient descent

- The gradient $\nabla E$ of error $E$ is a vector that
  - specifies the direction that produces the steepest increase in $E$
  - has length proportional to the steepness of the hill

$$\nabla E(w) = \begin{bmatrix} \frac{\partial E}{\partial w_0} & \frac{\partial E}{\partial w_1} & \cdots & \frac{\partial E}{\partial w_n} \end{bmatrix}$$

$n+1$ is the number of the weights in the network (i.e., $n+1$ is the length of $w$)

- The direction that produces the steepest decrease is the negative of $\nabla E$

- The training rule for gradient descent is $w \leftarrow w + \Delta w$
  where $\Delta w = -\eta \nabla E(w)$

- Written in his components form

  $$w_i \leftarrow w_i + \Delta w_i, \quad \Delta w_i = -\eta \frac{\partial E}{\partial w_i}, \quad i = 0, \ldots, n$$

Derivation of gradient descent

- Each component $\frac{\partial E}{\partial w_i}$ of $\nabla E(w)$ can be obtained as follows

  $$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{k \in K} (out_k - o_k)^2 =$$

  $$= \frac{1}{2} \frac{\partial}{\partial w_i} \sum_{k \in K} (out_k - o_k)^2 =$$

  $$= \frac{1}{2} \sum_{k \in K} 2(out_k - o_k) \frac{\partial}{\partial w_i}(out_k - o_k) =$$

  $$= \sum_{k \in K} (out_k - o_k) \frac{\partial}{\partial w_i}(out_k - w \cdot x)$$

  $$\frac{\partial E}{\partial w_i} = \sum_{k \in K} (out_k - o_k)(-x_{i,k})$$

  where $x_{i,k}$ denotes the $i$-th input of the linear unit for example $x$

- The variation of the $i$-th weight is

  $$\Delta w_i = \eta \sum_{k \in K} (out_k - o_k) \cdot x_{i,k}, \quad i = 0, \ldots, n$$
Gradient descent algorithm

Searches through the space of possible network weights, iteratively reducing the error $E$ between the training example target values and the network outputs

**Gradient_descent** $(X, \eta)$
Initialize $w$ ($w_i \leftarrow$ an initial (small) random value)
while stopping criterion not satisfied
  initialize each $\Delta w_i$ to 0
  for each training instance $(x, out_x) \in X$
    compute the network output $o_x$
    for each weight $w_i$
      $\Delta w_i \leftarrow \Delta w_i + \eta(out_x - o_x)x_i$
    end for
  end for
  for each weight $w_i$
    $w_i \leftarrow w_i + \Delta w_i$
  end for
end while
return $w$

- Stopping criterion: # of iterations (epochs), threshold error, etc.
- This algorithm can be applied whenever
  - The hypothesis space contains continuously parameterized hypothesis (i.e., the weights in a linear unit)
  - The error can be differentiated with respect to these hypothesis parameters

Stochastic (incremental) gradient descent

- Problems with gradient descent
  - Convergence to local minimum can sometimes be quite slow
  - If there are multiple local minima in the error surface the convergence to global minimum is not assured
- Stochastic gradient descent
  - Approximates gradient descent by updating weights incrementally instead after having processed all the training examples

**Stochastic_gradient_descent** $(X, \eta)$
Initialize $w$ ($w_i \leftarrow$ an initial (small) random value)
while stopping criterion not satisfied
  initialize each $\Delta w_i$ to 0
  for each training instance $(x, out_x) \in X$
    compute the network output $o_x$
    for each weight $w_i$
      $w_i \leftarrow w_i + \eta(out_x - o_x)x_i$
    end for
  end for
end while
return $w$

Stochastic gradient descent can sometimes avoid falling into local minima because it uses the various gradients of $E$ rather than overall gradient of $E$
Learning thresholded perceptron units

- \( o_x = w \cdot x \): output of unthresholded linear unit
- \( o_x' = \text{sgn}(w \cdot x) \): output of thresholded perceptron unit
- Suppose we want to train the perceptron to fit training examples with target values of \( \pm 1 \) for \( o_x' \)
- We can use the same training examples to train the linear unit, using the delta rule
  - if the linear unit can be trained to fit the target values perfectly, then the perceptron will perfectly fit them as well, as \( \text{sgn}(1) = 1 \) and \( \text{sgn}(-1) = -1 \)
  - when the target values cannot be fit perfectly, the perceptron will fit them whenever they have the correct sign

Remarks on perceptron learning

We have considered two similar algorithms for iteratively learning perceptron weights

- The perceptron training rule
  - updates weights based on the error in the thresholded perceptron output
  - converges after a finite number of iterations to a hypothesis that perfectly classifies the training data, provided the examples are linearly separable
- The delta rule
  - updates weights based on the error in the unthresholded linear combination of inputs
  - converges asymptotically toward the minimum error hypothesis, possibly requiring unbounded time, but converges regardless of whether the training data are linearly separable
Multi-layer Artificial Neural Networks

- As we have seen, a perceptron can only express a linear decision surface
- A multi-layer ANN learned by the backpropagation (BP) algorithm can represent highly non-linear decision surfaces
- Example: speech recognition
  - distinguishing among 10 possible vowels, all spoken in the context of “h...d”
  - the input is two numerical parameters $F_1$ and $F_2$ obtained from a spectral analysis of the sound
  - the 10 network outputs correspond to the 10 possible vowel sounds

Multi-layer Artificial Neural Networks

What type of unit should we adopt as the basis for multilayer networks?

- Perceptron: not differentiable -> can’t use gradient descent
- Linear Unit: multi-layers of linear units still produce only linear function
- Sigmoid Unit:
  - differentiable threshold function
  - output is non linear function of the input

The derivative is easily expressed
Backpropagation algorithm

- The BP learning algorithm is used to learn the weights of a multi-layer ANN having fixed structure (i.e., fixed set of neurons and interconnections).
- The BP algorithm employs gradient descent to minimize the squared error between the actual output values and the desired output ones, given the training instances.

\[ E(w) = \frac{1}{2} \sum_{x \in X} \sum_{i \in \text{inputs}} (o_{ix} - o_{ix})^2 \]

- Error surface can have multiple local minima.
  - Guarantee toward some local minimum.
  - No guarantee to the global minimum.

---

Backpropagation algorithm

- Backpropagation consists of two phases:
  - **Signal forward phase**
    - The input signals (i.e., the input vector) are propagated forwards from the input layer to the output layer (through the hidden layers) and output is calculated.
  - **Error backward phase**
    - Since the desired output value for the current input vector is known, the error is computed.
    - Starting at the output layer, the error is propagated backwards through the network, layer by layer, to the input layer.
    - The error backpropagation is performed by recursively computing the local gradient of each neuron. Weights are updated.

Signal forward phase
- Network activation

Error backward phase
- Output error computation
- Error propagation and weights updated
Let’s use this 3-layer ANN to illustrate the details of the BP learning algorithm

- **m** input signals \( x_j \) \((j=1..m)\)
- **l** hidden neurons \( z_q \) \((q=1..l)\)
- **n** output neurons \( y_i \) \((i=1..n)\)
- \( w_{qj} \) is the weight of the interconnection from input signal \( x_j \) to hidden neuron \( z_q \)
- \( w_{iq} \) is the weight of the interconnection from hidden neuron \( z_q \) to output neuron \( y_i \)
- \( o_q \) is the output value of hidden neuron \( z_q \)
- \( o_i \) is the output of the output neuron \( y_i \)

### Back-propagation algorithm

**Backpropagation** \((X, \eta)\)

1. Initialize all network weights to some small random value (e.g., between -0.05 and 0.05)
2. Until the termination condition is met, do
   1. For each training example \(<(x_1, \ldots, x_m), \text{out}>\), do
      - **Propagate the input forward through the network**
        1. Input the instance \((x_1, \ldots, x_m)\) to the network and compute the network outputs \( o_i \)
      - **Propagate the errors backward through the network**
        1. For each output unit \( y_i \), calculate its error term
           \[ \delta_i \leftarrow o_i(1-o_i)(\text{out}_i-o_i) \]
        2. For each hidden unit \( z_q \), calculate its error term
           \[ \delta_q \leftarrow o_q(1-o_q) \sum_i w_{qi} \delta_i \]
        3. For each network weight \( w_{jk} \) do
           \[ w_{jk} \leftarrow w_{jk} + \Delta w_{jk} \text{ where } \Delta w_{jk} = \eta \delta_j x_{jk} \]

\( o(1-o) \): derivative of sigmoid
**BP algorithm – Forward phase (1)**

- For each training instance $x$
  - The input vector $x$ is propagated from the input layer to the output layer
  - The network produces an actual output $a_x$
    (i.e., a vector of $a_{x_i}, i=1..n$)
- Given an input vector $x$, a neuron $z_q$ in the hidden layer receives an input
  $$Net_q = \sum_{j=1}^{m} w_{qj} x_j$$
  ... and produces a (local) output of
  $$o_q = \sigma(Net_q) = \sigma\left(\sum_{j=1}^{m} w_{qj} x_j\right)$$
where $\sigma$ is the sigmoid function

**BP algorithm – Forward phase (2)**

- The input for a neuron $y_i$ in the output layer is
  $$Net_i = \sum_{q=1}^{l} w_{iq} o_q = \sum_{q=1}^{l} w_{iq} \sigma\left(\sum_{j=1}^{m} w_{qj} x_j\right)$$
- Neuron $y_i$ produces an output of the network
  $$o_i = \sigma(Net_i) = \sigma\left(\sum_{q=1}^{l} w_{iq} o_q\right) = \sigma\left(\sum_{q=1}^{l} w_{iq} \sigma\left(\sum_{j=1}^{m} w_{qj} x_j\right)\right)$$
- The vector $o$ of output values $o_i (i=1..n)$ is the actual network output, given the input vector $x$
BP algorithm – hidden and output error

- According to the error equations $\delta_i$ and $\delta_q$ in the algorithm, the error signal of a neuron in a hidden layer is different from the error signal of a neuron in the output layer.
- Because of this difference, the derived weight update procedure is called the generalized delta learning rule.
- The error signal $\delta_q$ of a hidden neuron $z_q$ can be determined in terms of
  - the error signals $\delta_i$ of the neurons $y_i$ (i.e., that $z_q$ connects to) in the output layer.
  - the weights $w_{iq}$ of the connections.
- The important feature of the BP algorithm: the weights update rule is local.
  - To compute the weight change for a given connection, we need only the quantities available at both ends of that connection!

BP algorithm – generalization

When a network has more than one hidden layer, the general form of the BP update rule is

$$\Delta w_{ab} = \eta \delta_a x_b$$

- $b$ and $a$ refer to the two ends of the $(b \rightarrow a)$ connection (i.e., from neuron (or input signal) $b$ to neuron $a$).
- $x_b$ is the output of the hidden neuron (or the input signal) $b$.
- $\delta_a$ is the error signal of neuron $a$. 
BP algorithm issues – Learning rate

- $\eta$ significantly affects the effectiveness and convergence of the BP learning algorithm
- A large value of $\eta$ could speed up the convergence, but might result in overshooting or local minima
- A smaller value of $\eta$ may take a very long time for the training
- Usually chosen experimentally for each problem
- Good values of the learning rate at the beginning of the training may not be as good in later time (of the training)
- Use an adaptive (dynamic) learning rate

Adding Momentum

- Original weight update rule for BP: $\Delta w_{jk}(n) = \eta \delta_j x_{jk}$
- Adding momentum $\alpha$

  $\Delta w_{jk}(n) = \eta \delta_j x_{jk} + \alpha \Delta w_{jk}(n-1), \quad 0 < \alpha < 1$

- Help to escape a small local minima in the error surface
- Speed up the convergence
BP– Number of hidden neurons

- The size of a hidden layer is a fundamental question for the application of multilayer feed-forward ANNs to real-world problems.
- In practice, it is very difficult to determine a sufficient number of neurons to achieve the desired accuracy.
- The size of a hidden layer is usually determined experimentally—trial and test.
- A recommendation:
  - Begin with the size of hidden nodes ~ a relatively small fraction of the dimensionality of the input layer.
  - If the network fails to converge to a solution, add more hidden nodes.
  - If it does converge, you may try fewer hidden nodes.

Termination Conditions for BP

- The weight update loop may be iterated thousands of times in a typical application.
- The choice of termination condition is important because:
  - Too few iterations can fail to reduce error sufficiently.
  - Too many iterations can lead to overfitting the training data.

Termination Criteria:
- After a fixed number of iterations (epochs).
- Once the error falls below some threshold.
- Once the validation error meets some criterion.
**ANNs – Expressive capabilities**

- **Boolean functions**
  - Every Boolean function can be represented by an ANN with a single hidden layer

- **Continuous functions**
  - Every bounded continuous function can be approximated, with arbitrarily small error, by an ANN with one hidden layer [Cybenko, 1989; Hornik et al., 1989]
  - Any function can be approximated to arbitrary accuracy by an ANN with two hidden layers [Cybenko, 1988]

**Generalization and Overfitting**

- Continuing training until the training error falls below some predetermined threshold is a poor strategy since BP is susceptible to overfitting
  - Need to measure the generalization accuracy over a validation set (distinct from the training set)

- Two different types of overfitting
  - Generalization error first decreases, then increases, even the training error continues to decrease
  - Generalization error decreases, then increases, then decreases again, while the training error continues to decrease
Two Kinds of Overfitting Phenomena

Techniques for Overcoming the Overfitting Problem

- Weight decay
  - Decrease each weight by some small factor during each iteration
  - This is equivalent to modifying the definition of $E$ to include a penalty term corresponding to the total magnitude of the network weights
  - The motivation for the approach is to keep weight values small, to bias learning against complex decision surfaces

- Cross validation
  - The algorithm uses both training and validation sets
  - checks the error on the validation set
  - Outputs the weights that produced the lowest error during the all training

- $k$-fold cross-validation
  - Cross validation is performed $k$ times, each time using a different partitioning of the data into training and validation sets
  - Each time the number $i$ of iterations that produces the lowest error is recorded
  - The results of the $k$ runs are averaged obtaining $\bar{i}$
  - The net is trained for $\bar{i}$ iterations on all the data
ANNs – Advantages vs. Disadvantages

- **Advantages**
  - Massively parallel in nature
  - Fault (noise) tolerant because of parallelism

- **Disadvantages**
  - No clear rules or design guidelines for arbitrary applications
  - No general way to assess the internal operation of the network (therefore, an ANN system is seen as a “black-box”)
  - Difficult to predict future network performance (generalization)

Artificial neural networks – When?

- Input is high-dimensional discrete or real-valued
- The target function is real-valued, discrete-valued or vector-valued
- Possibly noisy data
- The form of the target function is unknown
- Human readability of result is not (very) important
- Long training time is accepted
- Short classification/prediction time is required