Neural Networks

Neuron Model and Network Architectures.

Perceptron.

Course web page has been moved to:

Neuron Models and Network Architectures - Outline

• Single-input neuron
• Transfer functions
• Multiple-input neuron
• Layer of neurons
• Multilayer network
• An illustrative example

Notation

• scalars - small italic letters: a, b, c
• vectors - small bold non-italic letters: a, b, c
• matrices - capital BOLD non-italic letters: A, B, C

Single-Input Neuron

• scalar input \( p \) * scalar weight \( w = wp \)
• bias (offset) \( b \)
  • it's like a weight except that it has a constant input of 1
  • a neuron may have or may not have a bias
  • \( n \) = net input (net input)
  • \( f \) = transfer function (activation function)
  • \( a \) = neuron output

\[
a = f(wp+b)
\]

• \( w \) and \( b \) are adjustable scalar parameters of the neuron (by a learning rule)
• \( f \) is chosen by the designer

Step Transfer Functions

• A particular transfer function is chosen to satisfy some specification of the problem that the neuron is attempting to solve
• Hard limit (step) transfer function - used in perceptrons

\[
a = \begin{cases} 
1 & \text{if } n \geq 0 \\
0 & \text{if } n < 0
\end{cases}
\]

\[
a = f(wp+b)
\]

Linear Transfer Functions

\[
a = \text{purelin}(n)
\]

\[
a = wp + b
\]

• used in ADALINE networks
Sigmoid Transfer Functions

- Log-sigmoid and tan-sigmoid transfer functions
- Used in multilayer networks trained with backpropagation

\[ a = \frac{1}{1 + e^{-z}} \]

- Output: between 0 and 1

\[ a = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \]

- Output: between -1 and 1

- Transfer functions - summary; see the handout
- To experiment with a single-input neuron: try nnd2n1!

Multiple-Input Neuron

- A neuron with R inputs

\[ a = f(Wp + b) \]

- Weight indices convention:
  - 1st index - neuron destination
  - 2nd index - source of the signal fed to the neuron
  - e.g. \( w_{ij} \) represents the connection to the 1st neuron from the 2nd source

Multiple-Input Neuron - Abbreviated Notation

- You can tell immediately if the variable is a scalar, a vector or a matrix & what its dimensionality is
- To experiment with 2-input neuron: try nnd2n2!

A Layer of Neurons

- Each layer has its own weight matrix \( W \), a bias vector \( b \), a net input vector \( a \) and an output vector \( a' \)
- To distinguish between the layers - use superscripts
- \( W^1 \) - weight matrix of the 1st layer, \( S_2 \) - neurons in the 2d layer, etc.
- 1 output and 2 hidden layers

Network Architectures

- 1 neuron, even with many inputs may not be sufficient...

A Layer of Neurons

- Weight matrix \( W \) now has \( S \) rows

Multilayer Network

- Each layer has its own weight matrix \( W \), a bias vector \( b \), a net input vector \( a \) and an output vector \( a' \)
- To distinguish between the layers - use superscripts
- \( W_1 \) - weight matrix of the 1st layer, \( S_2 \) - neurons in the 2d layer, etc.
- 1 output and 2 hidden layers
Perceptron - Outline

- Perceptron’s neuron model
- Investigation of Perceptron’s boundaries
- Perceptron’s learning rule
- Example
- Capabilities and limitations

Single-Neuron Perceptron

- 2 inputs \( p_1 \) and \( p_2 \) (may have one or more inputs)
- Each input is weighted with a weight \( w_{11} \) and \( w_{12} \)
- An additional weight \( b \) (bias) associated with the neuron
- The sum of the weighted inputs, together with an input of 1 transmitted through the bias, is sent to a step function
- 2 possible outputs: 0 and 1 (or –1 and 1)

Single-Neuron Perceptron - Investigation of the Boundaries

- 4. The decision boundary is:
  \[ n = w_2 p_2 + b = w_1 p_1 + w_2 p_2 + b = 0 \]
  (or \( w_2 p_2 + b = w_1 p_1 + w_2 p_2 + b = 0 \) in 2-D space)
  I.e. a line in the input space

- 5. Draw the decision boundary:
  \( p_2 = 0 \Rightarrow p_1 = 1 \) (\( p_1 \) intersect)
  \( p_1 = 0 \Rightarrow p_2 = 1 \) (\( p_2 \) intersect)

Perceptron Learning Rule – Derivation.

- Supervised learning:
  - A set of training examples: \( [p_1, t_1], [p_2, t_2], \ldots, [p_n, t_n] \)
  - Random initial weights \( w \) and \( b \)
  - After each example, the learning rule adjusts the weights and biases to move the network output closer to the target output.
  - Goal: classify all examples correctly.

  Simple test problem & experimentation with possible rules

- Infinite number of boundaries
Test Problem - Staring Point
- A perceptron with 2 inputs and no bias
- Random initial weight vector
  \[ w = [1 \quad -0.8] \]
- Presenting the patterns to the perceptron
  \[ \{ p_1, p_2 \}, t_1 = 1 \]
  \[ a = \text{hardlim}(w \cdot p) = \text{hardlim}( \begin{bmatrix} 1 & -0.8 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix}) = \text{hardlim}(-0.6) = 0 \]
  Incorrect classification!

Test Problem - Tentative Learning Rule
- we need to alter the weight vector so that it points more toward \( p_1 \)
  so that in the future it has a better chance of classifying it correctly
  - add \( p_1 \) to \( w \) - repeated presentations of \( p_1 \) would cause the direction of \( w \) to approach the direction of \( p_1 \)
  - tentative learning rule (rule 1):
    \[ w^{(n+1)} = w^{(n)} + p^T \]
    \[ w^{(n+1)} = w^{(n)} + p_1^T \]
    \[ w^{(n+1)} = w^{(n)} + p_1^T = [1 \quad -0.8] [1 \quad 2] = [2 \quad 1.2] \]

Test Problem - Second Input Vector
- we would like to move the weight vector \( w \) away from the input
  => rule 2:
  \[ w^{(n+1)} = w^{(n)} - p^T \]
  \[ w^{(n+1)} = w^{(n)} - p_2^T = [2 \quad 1.2] - [-1 \quad 2] = [3 \quad -0.8] \]

Test Problem - Second Input Vector
- we would like to move the weight vector \( w \) away from the input
  => rule 2:
  \[ w^{(n+1)} = w^{(n)} - p^T \]
  \[ w^{(n+1)} = w^{(n)} - p_2^T = [2 \quad 1.2] - [-1 \quad 2] = [3 \quad -0.8] \]

Unified Learning Rule
- covers all combinations of output and target values (0 and 1):
  \[ \text{if} \ t = 1 \text{ and } a = 0, \ w^{(n+1)} = w^{(n)} + p^T \]
  \[ \text{if} \ t = 0 \text{ and } a = 1, \ w^{(n+1)} = w^{(n)} - p^T \]
  \[ \text{if} \ t = a, \ w^{(n+1)} = w^{(n)} \]
- define:
  \[ e = t - a \]
  \[ \text{if} \ a = 1, \ w^{(n)} = w^{(n)} + p^T \]
  \[ \text{if} \ a = 0, \ w^{(n)} = w^{(n)} - p^T \]
- unified rule:
  \[ w^{(n+1)} = w^{(n)} + e p^T = w^{(n)} + (t-a) p^T \]

A Layer of Perceptrons
- \( f \) is a step function:
  \[ w_0^{(n+1)} = w_0^{(n)} + e p^T \]
  \[ w_0^{(n+1)} = w_0^{(n)} + e p^T \]
- in matrix form for a layer of perceptrons:
  \[ W^{(n+1)} = W^{(n)} + e P^T \]
  \[ b^{(n+1)} = b^{(n)} + e \]
Perceptron Learning Law – Summary
1. Initialize weights (including biases) to small random values
2. Choose a random input-output pair \( \{ p, t \} \) from the training set
3. Let the network to operate on the input to generate output \( a \)
4. Compute the output error \( e = t - a \)
5. Update weights:
   - Add a matrix \( \Delta W \) to the weight matrix \( W \), which is proportional to the product \( ep^T \) between the error vector and the input:
   \[
   W_{new} = W_{old} + \Delta W = W_{old} + ep^T
   \]
   - Add a vector \( \Delta b \) to the bias vector \( b \), which is proportional to the error vector:
   \[
   b_{new} = b_{old} + \Delta b = b_{old} + e
   \]
6. Choose another random pair and do the correction again
7. Continue until the stopping criteria is satisfied:
   - all examples are correctly classified or a maximum number of epochs is reached

Perceptron – Stopping Criteria
Stopping criteria is checked at the end of each epoch:
- epoch - one pass through the training set (i.e. each training example is passed once) involving weight adaptation
- the numbering starts from 1: epoch 1, epoch 2, etc.
To check if all examples are correctly classified at the end of the epoch:
- all training examples are passed again, the actual activation is calculated and it is compared with the target activation.
- note: this does not count for another epoch as there is no weight adaptation

An Example - Apple/Banana Sorter
- A produce dealer has a warehouse that stores a variety of fruits. He wants a machine that will sort the fruit according to the type...
- There is a conveyer belt on which the fruit is loaded... it is then passed through a set of sensors, which measure 3 properties of the fruit: shape, texture and weight.
- the sensors are somehow primitive :-) :
  - shape sensor: \(-1\) if the fruit is round, \(1\) - if it is more elliptical
  - texture sensor: \(-1\) if the surface is smooth, \(1\) - if it is rough
  - weight sensor: \(-1\) if the fruit is \( \geq 500\)g, \(1\) - if \( < 500\)g
- The sensor outputs will then be input to a NN...
- NN’s purpose:
  - fruit type recognition,
  - so that it is directed to the correct storage bin
- For simplicity - only 2 kinds of fruit

Apple/Banana Example – First Iteration
Applying \( p_1 \):
\[
\begin{bmatrix}
    a_1 \\
    a_2 \\
    a_3 
\end{bmatrix} = \text{hardlim}(W \begin{bmatrix}
    p_1^T + b \end{bmatrix}) = \text{hardlim}(0.5 \begin{bmatrix}
    -1 & 1 & 0.5 \\
    -1 & 1 & 0.5
\end{bmatrix}) = \begin{bmatrix}
    -1 \\
    -1
\end{bmatrix}
\]
\[
a = \text{hardlim}(-0.5) = 0
\]
\[
e = t_1 - a = 1 - 0 = 1
\]
Updating the weights:
\[
W_{new} = W_{old} + ep^T = 0.5 \begin{bmatrix}
    -1 & 1 & 0.5 \\
    -1 & 1 & 0.5
\end{bmatrix} = \begin{bmatrix}
    -0.5 & 0 & -1.5 \\
    -0.5 & 0 & -1.5
\end{bmatrix}
\]
\[
b_{new} = b_{old} + e = 0.5 + 1 = 1.5
\]

Apple/Banana Example – Second Iteration
Applying \( p_2 \):
\[
\begin{bmatrix}
    a_1 \\
    a_2 \\
    a_3 
\end{bmatrix} = \text{hardlim}(W \begin{bmatrix}
    p_2^T + b \end{bmatrix}) = \text{hardlim}(\begin{bmatrix}
    1 & -1.5 & -1.5 \\
    1 & -1.5 & -1.5
\end{bmatrix}) = \begin{bmatrix}
    1 \\
    1
\end{bmatrix}
\]
\[
a = \text{hardlim}(2.5) = 1
\]
\[
e = t_2 - a = 1 - 0 = 1
\]
\[
W_{new} = W_{old} + ep^T = \begin{bmatrix}
    0.5 & -1 & -1.5 \\
    1 & 1 & -1
\end{bmatrix} = \begin{bmatrix}
    -0.5 & 0 & -1.5 \\
    -1.5 & 1 & -1
\end{bmatrix}
\]
\[
b_{new} = b_{old} + e = 1.5 + 1 = 2.5
\]
* End of epoch; check if the stopping criteria is satisfied
Apple/Banana Example - Check

To experiment with the perceptron learning rule:
- Matlab: nnd4pr

a = hardlim(Wp + b) = hardlim(\[ \begin{bmatrix} 1 & 5 & -1 & -0.3 \end{bmatrix} \] * 0.5)

b = hardlim(b) = 1, if b > 0; 0, otherwise.

• How many epochs were necessary to train the perceptron?

Another Example

The following training set is given:

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 1</td>
<td></td>
</tr>
<tr>
<td>0 1 1 0</td>
<td></td>
</tr>
<tr>
<td>1 1 0 1</td>
<td></td>
</tr>
<tr>
<td>1 1 1 0</td>
<td></td>
</tr>
<tr>
<td>0 0 1 0</td>
<td></td>
</tr>
<tr>
<td>1 0 1 1</td>
<td></td>
</tr>
</tbody>
</table>

a) Train by hand a perceptron with bias on this training set. Assume that all initial weights (including the bias of the neuron) are 0. Show the set of weights (including the bias) at the end of each iteration. Apply the examples in the given order. Use the hardlim step function: hardlim(n) = 1, if n >= 0; hardlim(n) = 0, otherwise. Stopping criteria: patterns are correctly classified.

b) How many epochs were needed? Is the training set linearly separable?

Capability and Limitations

• The output values of a perceptron can take only 2 values: 0 and 1 (or -1 & 1)
• Proof of convergence: If the training examples are linearly separable, the perceptron learning rule is guaranteed to converge to a solution (i.e. a set of weights that correctly classify the examples) in a finite number of steps.
• Linear decision boundary – examples are separated by a linear boundary (line, hyperplane)

Layer of Perceptrons - Capabilities

• Each perceptron defines one decision boundary, i.e. one line
• A layer of perceptrons can classify input vectors into many categories
• Maximum number of categories 2^S, where S is the number of perceptrons in the layer
• A layer of perceptrons can be used to solve more difficult linearly separable problems

Question

• Can this problem be solved by a perceptron network?

For Homework

• Solve this classification problem with the perceptron rule. Apply each input vector in order, for as many repetitions as it takes to ensure that the problem is solved. Draw a graph of the problem only after you have found a solution. Use the following initial weights and bias:

```
W(0) = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, b(0) = \begin{bmatrix} 1 \end{bmatrix}
```

```
class1: p1 = \begin{bmatrix} 1 \\ p2 \\ 1 \end{bmatrix}, p4 = \begin{bmatrix} 1 \\ p3 \\ 2 \end{bmatrix}
class2: p1 = \begin{bmatrix} 2 \\ 1 \\ 2 \end{bmatrix}, p2 = \begin{bmatrix} 2 \\ 0 \end{bmatrix}
class3: p1 = \begin{bmatrix} -1 \\ 0 \\ -2 \end{bmatrix}, p2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}
class4: p1 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}, p2 = \begin{bmatrix} -2 \\ -2 \end{bmatrix}
```