F20DL and F21DL: Part 2, Machine Learning
Lecture 8. Supervised Learning: Neural Networks

Katya Komendantskaya

Plan for today...

Finish Neural nets and finish the course.

- Linear and non-liner separation of data
- Deep Neural net architectures
- Backpropagation learning

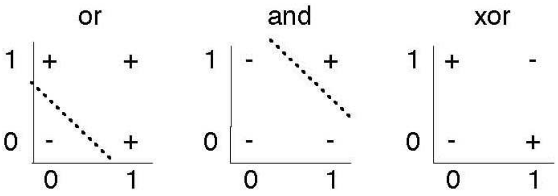
Plan for today...

Finish Neural nets and finish the course.

- Linear and non-liner separation of data
- Deep Neural net architectures
- Backpropagation learning
- Start with a demo: real-time neuron training

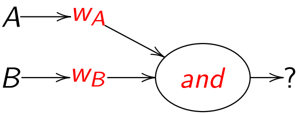
Historical uses of Neural nets: Perceptron

Neural nets doing logic [McCulloch and Pitts, 1943]:



A	B	A and B	A or B	A xor B
true	true	true	true	false
true	false	false	true	true
false	true	false	true	true
false	false	false	false	false

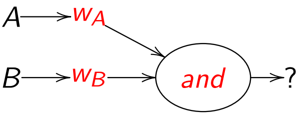
Perceptron for and



Input features and target features:

A	B	A and B
true	true	true
true	false	false
false	true	false
false	false	false

Perceptron for and

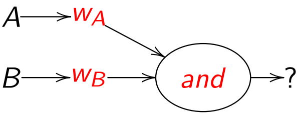


Input features and target features:

A	B	A and B
true	true	true
true	false	false
false	true	false
false	false	false

Now train the network: will it be able to learn the correct (linear) function $\theta + w_A \times A + w_B \times B$ to simulate and?

Perceptron for and

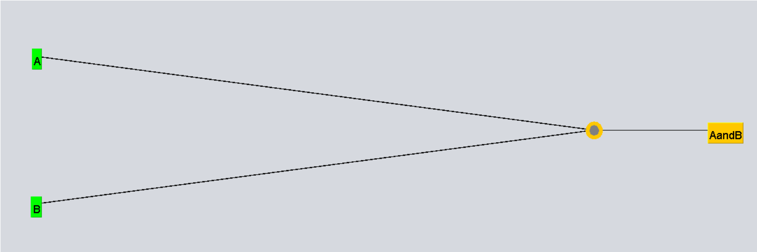


Input features and target features:

A	B	A and B
true	true	true
true	false	false
false	true	false
false	false	false

Now train the network: will it be able to learn the correct (linear) function $\theta + w_A \times A + w_B \times B$ to simulate and?
On the board: e.g. $-0{,}9 + 0{,}5 \times A + 0{,}5 \times B$

Typical Weka output, Multilayer Perceptron



Typical Weka output

```
Linear Node 0
    Inputs    Weights
    Threshold    -0.6401869158878506
    Attrib A    0.5607476635514019
    Attrib B    0.5280373831775702
Class
    Input
    Node 0

    inst#     actual  predicted      error
        1       1         0.724     -0.276
        2       0         0.196      0.196
        3       0         0.164      0.164
        4       0        -0.364     -0.364
```

Typical Weka output

(for logistic outputs only)

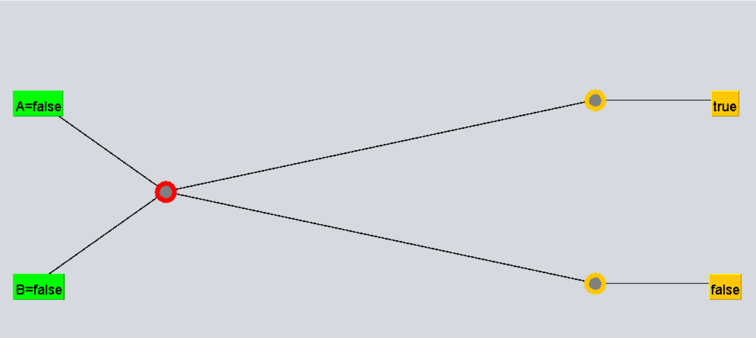
```
=== Predictions on training set ===

    inst#     actual  predicted error prediction
        1     1:True     1:True       0.929
        2    2:False    2:False       0.929
        3    2:False    2:False       0.929
        4    2:False    2:False       1
```

```
 a b   <-- classified as
 1 0 | a = True
 0 3 | b = False
```

Typical Weka output, Multilayer Perceptron

(Nominal version, logistic functions)



Weka output, note the sigmoid nodes

```
Sigmoid Node 0
    Inputs    Weights
    Threshold    0.024509736809619657
    Attrib a=False    0.01062088595083302
    Attrib b=False    0.00490194736192285
Sigmoid Node 1
    Inputs    Weights
    Threshold    -0.024509736809619903
    Attrib a=False    -0.010620885950833422
    Attrib b=False    -0.004901947361923509
Class True
    Input
    Node 0
Class False
    Input
    Node 1
```

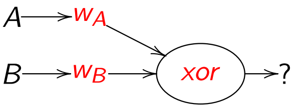
A few observations

- Note: Weka's forming two output neurons for nominal data sets is accidental (an implementation decision)
- One can have one sigmoid neuron just as well (demo)

A few observations

- Note: Weka's forming two output neurons for nominal data sets is accidental (an implementation decision)
- One can have one sigmoid neuron just as well (demo)
- Epochs versus iterations: iteration is one run of the algorithm (one weight update), an epoch is one run of the algorithm over all training instances.
  - So, in your manual computation, you will have 3 iterations of training (3 weight updates)
  - But only one epoch (as we run over 3 examples just once)

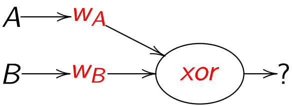
Perceptron for xor



Input features and target features:

A	B	A xor B
true	true	false
true	false	true
false	true	true
false	false	false

Perceptron for xor

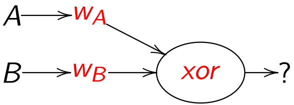


Input features and target features:

A	B	A xor B
true	true	false
true	false	true
false	true	true
false	false	false

Now train the network: will it be able to learn the correct (linear) function $\theta + w_A \times A + w_B \times B$ to simulate xor?

Perceptron for xor



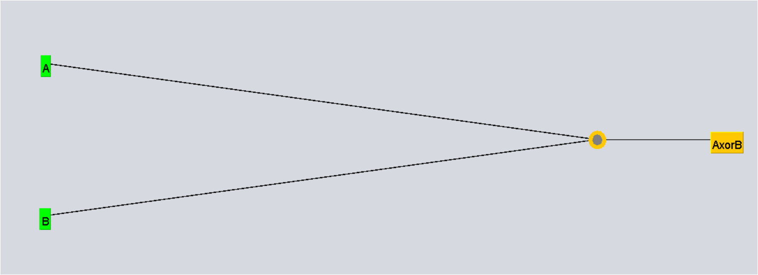
Input features and target features:

A	B	A xor B
true	true	false
true	false	true
false	true	true
false	false	false

Now train the network: will it be able to learn the correct (linear) function $\theta + w_A \times A + w_B \times B$ to simulate xor?
On the board: None exists.

Typical Weka output, Multilayer Perceptron

(Numeric Version, linear function)



Typical Weka output

```
Attributes:    3
               A
               B
               AxorB
Test mode:     4-fold cross-validation

=== Classifier model (full training set) ===

Linear Node 0
    Inputs     Weights
    Threshold     0.5454545454545454
    Attrib A     -0.2363636363636328
    Attrib B     -0.1090909090909090 7
Class
    Input
    Node 0
```

Typical Weka output

```
Time taken to build model: 0 seconds

=== Predictions on test data ===

    inst#     actual  predicted      error
        1      0          2            2
        1      0          2            2
        1      1         -1           -2
        1      1         -1           -2
```

Typical Weka output, Multilayer Perceptron

```
=== Predictions on training set ===

    inst#     actual  predicted error prediction
        1    2:False     1:True    +   0.502
        2     1:True     1:True        0.505
        3     1:True     1:True        0.508
        4    2:False     1:True    +   0.51


=== Confusion Matrix ===

 a b   <-- classified as
 2 0 | a = True
 2 0 | b = False
```

Limitation of Linear classifiers

- ... Was first acknowledged when Perceptron failed to classify XOR
- But it is a general problem that arises for the whole class of algorithms called Linear Classifiers

Linearly Separable

- A classification is linearly separable if there is a hyperplane where the classification is *true* on one side of the hyperplane and *false* on the other side.

Linearly Separable

- A classification is linearly separable if there is a hyperplane where the classification is *true* on one side of the hyperplane and *false* on the other side.
- The hyperplane is defined for the predicted value :

$$f(w_0 + w_1 \times X_1 + \cdots + w_n \times X_n) = 0{,}5$$

  This separates the predictions $> 0{,}5$ and $< 0{,}5$.
- For the sigmoid function, this occurs when

$$w_0 + w_1 \times X_1 + \cdots + w_n \times X_n = 0$$

Linearly Separable

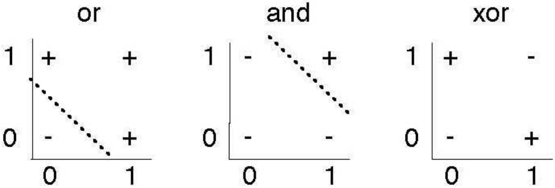
- A classification is linearly separable if there is a hyperplane where the classification is *true* on one side of the hyperplane and *false* on the other side.

Linearly Separable

- A classification is linearly separable if there is a hyperplane where the classification is *true* on one side of the hyperplane and *false* on the other side.
- linearly separable implies the error can be arbitrarily small

Linearly Separable

- A classification is linearly separable if there is a hyperplane where the classification is *true* on one side of the hyperplane and *false* on the other side.
- linearly separable implies the error can be arbitrarily small
- Example when this does not hold: the XOR problem



Example of non-linear separable set:

Holiday preferences:

Culture	Fly	Hot	Music	Nature	Likes
0	0	1	0	0	0
0	1	1	0	0	0
1	1	1	1	1	0
0	1	1	1	1	0
0	1	1	0	1	0
1	0	0	1	1	1
0	0	0	0	0	0
0	0	0	1	1	1
1	1	1	0	0	0
1	1	0	1	1	1
1	1	0	0	0	1
1	0	1	0	1	1
0	0	0	1	0	0
1	0	1	1	0	0
1	1	1	1	0	0
1	0	0	1	0	0
1	1	1	0	1	0
0	0	0	0	1	1
0	1	0	0	0	1

Variants in Linear Separators

Which linear separator to use can result in various algorithms:

- Perceptron
- Logistic Regression
- Support Vector Machines (SVMs)
- ...

Solution for linearly non-separable data?

Add more layers to your networks to get rid of the linearity problem!

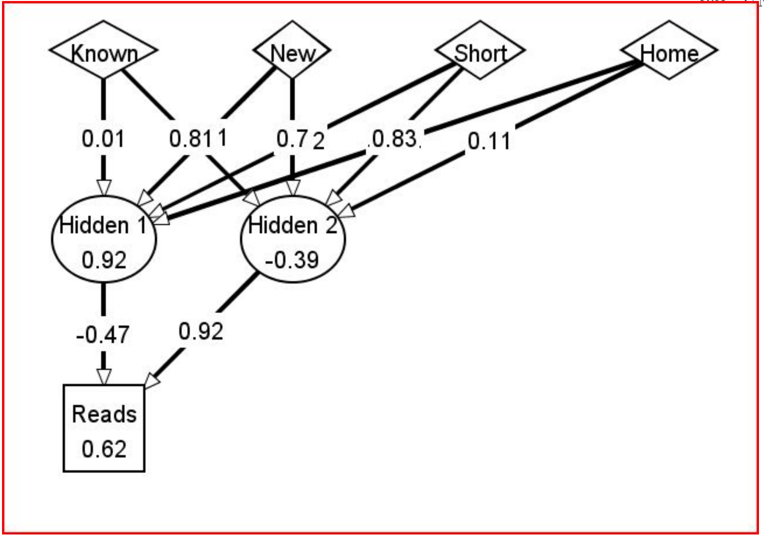
Note on joining neurons:

- Neurons working in parallel – for capturing several different features

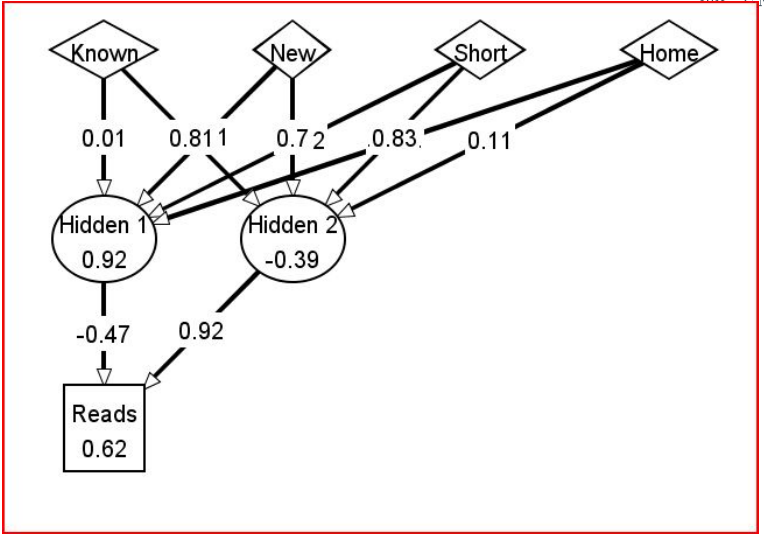
Note on joining neurons:

- Neurons working in parallel – for capturing several different features
- Neurons joined sequentially – for capturing more complex activating and learning functions; and for capturing nonlinear feature dependencies.

Neural networks used for classification



Neural networks used for classification



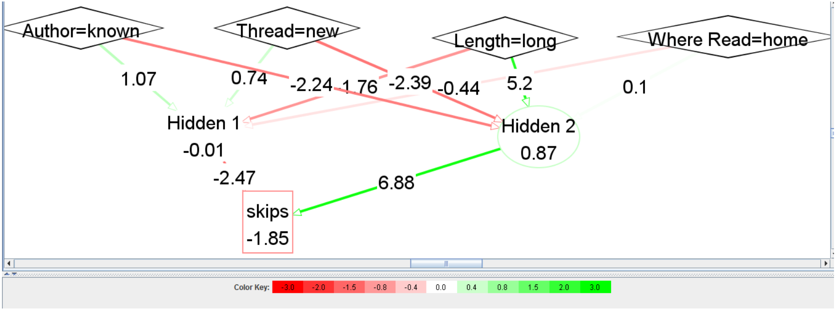
Quick recap: No of dimensions this data "lives in"?

Example: this network learns a linear classifier:

as a method, will not work for non-linearly separable data

Example of joining neurons: mail classification

one hidden layer added: can be used for non-linearly separable data



There are 10 parameters to be learned. Therefore, the hypothesis space is a 10-dimensional real space. Each point in this space corresponds to a function that predicts a value for "skips".

How can Neural Nets solve this?

- Multi-layered networks are like cascaded squashed linear functions.

From lecture on Linear Regression

For classification, one uses **squashed linear function** of the form

$$f(X_1, \ldots, X_n) = G(w_0, +w_1 X_1 + \ldots + w_n X_n)$$

where G is **an activation function** from real numbers to $[0, 1]$.

How can Neural Nets solve this?

- Multi-layered networks are like cascaded squashed linear functions.

From lecture on Linear Regression

For classification, one uses **squashed linear function** of the form

$$f(X_1, \ldots, X_n) = G(w_0, +w_1 X_1 + \ldots + w_n X_n)$$

where G is **an activation function** from real numbers to $[0, 1]$.

- Each of the hidden neurons is a squashed linear function of its inputs.

How can Neural Nets solve this?

- Multi-layered networks are like cascaded squashed linear functions.

From lecture on Linear Regression

For classification, one uses **squashed linear function** of the form

$$f(X_1, \ldots, X_n) = G(w_0, +w_1X_1 + \ldots + w_nX_n)$$

where G is **an activation function** from real numbers to $[0, 1]$.

- Each of the hidden neurons is a squashed linear function of its inputs.
- Output neurons can be linear (for regression) or sigmoid (for classification) functions.

How can Neural Nets solve this?

- Multi-layered networks are like cascaded squashed linear functions.

From lecture on Linear Regression

For classification, one uses **squashed linear function** of the form

$$f(X_1, \ldots, X_n) = G(w_0, +w_1 X_1 + \ldots + w_n X_n)$$

where G is **an activation function** from real numbers to $[0, 1]$.

- Each of the hidden neurons is a squashed linear function of its inputs.
- Output neurons can be linear (for regression) or sigmoid (for classification) functions.
- Learning by neural networks — is adjustment of the weights such that the prediction error is minimized.

More on NN learning:

Given:

- values for parameters: network architecture (incl. activation functions), learning rate, target error, number of iterations, etc..

More on NN learning:

Given:

- values for parameters: network architecture (incl. activation functions), learning rate, target error, number of iterations, etc..
- values for input features
- set of examples

More on NN learning:

Given:

- values for parameters: network architecture (incl. activation functions), learning rate, target error, number of iterations, etc..
- values for input features
- set of examples

Need to:

- predict a value for each target feature
- that is, adjust parameters (=weights)

More on NN learning:

Given:

- values for parameters: network architecture (incl. activation functions), learning rate, target error, number of iterations, etc..
- values for input features
- set of examples

Need to:

- predict a value for each target feature
- that is, adjust parameters (=weights)

Back-propagation learning

is a gradient descent search through the parameter space to minimize the sum-of-squares error.

Formulae it uses

Because we will cascade squashed linear functions (of which some may be sigmoid), all we need to remember is our two old formulae for linear and sigmoid functions (from lecture on Linear Functions):

Formulae it uses

Because we will cascade squashed linear functions (of which some may be sigmoid), all we need to remember is our two old formulae for linear and sigmoid functions (from lecture on Linear Functions): Given a linear function $f(X_1, \ldots, X_n) = G(w_0, +w_1 X_1 + \ldots + w_n X_n)$

Weight update for sum-of-squares error (no "G")

$$w_i := w_i + \eta \times \delta \times val(e, X_i)$$

with $\delta = (val(e, Y) - pval^{\overline{w}}(e, Y))$

Weight update when "G" is a sigmoid (logistic) function σ

$$w_i := w_i + \eta \times \delta \times pval^{\overline{w}}(e, Y) \times [1 - pval^{\overline{w}}(e, Y)] \times val(e, X_i)$$

Formulae it uses

Because we will cascade squashed linear functions (of which some may be sigmoid), all we need to remember is our two old formulae for linear and sigmoid functions (from lecture on Linear Functions): Given a linear function $f(X_1, \ldots, X_n) = G(w_0, +w_1 X_1 + \ldots + w_n X_n)$

Weight update for sum-of-squares error (no "G")

$$w_i := w_i + \eta \times \delta \times val(e, X_i)$$

with $\delta = (val(e, Y) - pval^{\overline{w}}(e, Y))$

Weight update when "G" is a sigmoid (logistic) function σ

$$w_i := w_i + \eta \times \delta \times pval^{\overline{w}}(e, Y) \times [1 - pval^{\overline{w}}(e, Y)] \times val(e, X_i)$$

- Red parts measure the change in error estimations when w_i varies (in other words – red parts are given by derivative of the error function);
- Remember $pval^{\overline{w}}(e, Y) = \sigma \sum_i (w_i \times val(e, X_i))$

Formulae it uses

Because we will cascade squashed linear functions (of which some may be sigmoid), all we need to remember is our two old formulae for linear and sigmoid functions (from lecture on Linear Functions): Given a linear function $f(X_1, \ldots, X_n) = G(w_0, +w_1 X_1 + \ldots + w_n X_n)$

Weight update for sum-of-squares error (no "G")

$w_i := w_i + \eta \times \delta \times val(e, X_i)$

with $\delta = (val(e, Y) - pval^{\overline{w}}(e, Y))$

Weight update when "G" is a sigmoid (logistic) function σ

$w_i := w_i + \eta \times \delta \times pval^{\overline{w}}(e, Y) \times [1 - pval^{\overline{w}}(e, Y)] \times val(e, X_i)$

- Red parts measure the change in error estimations when w_i varies (in other words – red parts are given by derivative of the error function);
- Remember $pval^{\overline{w}}(e, Y) = \sigma \sum_i (w_i \times val(e, X_i))$

1. **Algorithm** BackPropagationLearner(X,Y,E,n_h,η)

1. **Algorithm** BackPropagationLearner(X,Y,E,n_h,η)
2. **Inputs:** input $X = \{X_1, \ldots X_n\}$ and output $Y = \{Y_1, \ldots, Y_k\}$ set of examples E, n_h - number of hidden neurons, η - learning rate.

1. **Algorithm** BackPropagationLearner(X,Y,E,n_h,η)
2. **Inputs:** input $X = \{X_1, \dots X_n\}$ and output $Y = \{Y_1, \dots, Y_k\}$ set of examples E, n_h - number of hidden neurons, η - learning rate.
3. **Outputs:** hidden neuron weights $hw[1 : n, 1 : n_h]$; output weights $ow[0 : n_h, 1 : k]$

1. **Algorithm** BackPropagationLearner(X,Y,E,n_h,η)
2. **Inputs:** input $X = \{X_1, \ldots X_n\}$ and output $Y = \{Y_1, \ldots, Y_k\}$ set of examples E, n_h - number of hidden neurons, η - learning rate.
3. **Outputs:** hidden neuron weights $hw[1 : n, 1 : n_h]$; output weights $ow[0 : n_h, 1 : k]$
4. **Local:** for each hidden neuron, value $hid[0 : n_h]$ and error $hErr[1 : n_h]$; for each output neuron, predicted value $out[1 : k]$ and error $oErr[1 : k]$.

1. **Algorithm** BackPropagationLearner(X,Y,E,n_h,η)
2. **Inputs:** input $X = \{X_1, \dots X_n\}$ and output $Y = \{Y_1, \dots, Y_k\}$ set of examples E, n_h - number of hidden neurons, η - learning rate.
3. **Outputs:** hidden neuron weights $hw[1 : n, 1 : n_h]$; output weights $ow[0 : n_h, 1 : k]$
4. **Local:** for each hidden neuron, value $hid[0 : n_h]$ and error $hErr[1 : n_h]$; for each output neuron, predicted value $out[1 : k]$ and error $oErr[1 : k]$.
5. initialise hw and ow randomly, $\backslash\backslash$ σ *is a sigmoid function*

1. **Algorithm** BackPropagationLearner(X,Y,E,n_h,η)
2. **Inputs:** input $X = \{X_1, \ldots X_n\}$ and output $Y = \{Y_1, \ldots, Y_k\}$ set of examples E, n_h - number of hidden neurons, η - learning rate.
3. **Outputs:** hidden neuron weights $hw[1 : n, 1 : n_h]$; output weights $ow[0 : n_h, 1 : k]$
4. **Local:** for each hidden neuron, value $hid[0 : n_h]$ and error $hErr[1 : n_h]$; for each output neuron, predicted value $out[1 : k]$ and error $oErr[1 : k]$.
5. initialise hw and ow randomly, $\backslash\backslash$ σ *is a sigmoid function*
6. **repeat** $\{$
7.   **for each example** e **in** E **do** $\{$
8.   **for each** $h \in \{1, \ldots, n_h\}$ **do** $\{$
9. $hid[h] := \sigma(\Sigma_{i=0}^{n}(hw[i, h] \times val(e, X_i)))$ $\}$

1. **Algorithm** BackPropagationLearner(X,Y,E,n_h,η)
2. **Inputs:** input $X = \{X_1, \ldots X_n\}$ and output $Y = \{Y_1, \ldots, Y_k\}$ set of examples E, n_h - number of hidden neurons, η - learning rate.
3. **Outputs:** hidden neuron weights $hw[1:n, 1:n_h]$; output weights $ow[0:n_h, 1:k]$
4. **Local:** for each hidden neuron, value $hid[0:n_h]$ and error $hErr[1:n_h]$; for each output neuron, predicted value $out[1:k]$ and error $oErr[1:k]$.
5. initialise hw and ow randomly, \\ σ *is a sigmoid function*
6. **repeat** $\{$
7. **for each example** e **in** E **do** $\{$
8. **for each** $h \in \{1, \ldots, n_h\}$ **do** $\{$
9. $hid[h] := \sigma(\Sigma_{i=0}^{n}(hw[i,h] \times val(e, X_i)))$ $\}$

Algorithm BackPropagationLearner(X,Y,E,n_h,η)

Inputs: input $X = \{X_1, \dots X_n\}$ and output $Y = \{Y_1, \dots, Y_k\}$ set of examples E, n_h - number of hidden neurons, η - learning rate.

Outputs: hidden neuron weights $hw[1:n, 1:n_h]$; output weights $ow[0:n_h, 1:k]$

Local: for each hidden neuron, value $hid[0:n_h]$ and error $hErr[1:n_h]$; for each output neuron, predicted value $out[1:k]$ and error $oErr[1:k]$.

initialise hw and ow randomly, \\\\ *σ is a sigmoid function*

repeat $\{$

    for each example e **in** E **do** $\{$

        for each $h \in \{1, \dots, n_h\}$ **do** $\{$

            $hid[h] := \sigma(\Sigma_{i=0}^{n}(hw[i,h] \times val(e, X_i)))$ $\}$ \\\\ $pval(h,e)$

1. **Algorithm** BackPropagationLearner(X,Y,E,n_h,η)
2. **Inputs:** input $X = \{X_1, \ldots X_n\}$ and output $Y = \{Y_1, \ldots, Y_k\}$ set of examples E, n_h - number of hidden neurons, η - learning rate.
3. **Outputs:** hidden neuron weights $hw[1:n, 1:n_h]$; output weights $ow[0:n_h, 1:k]$
4. **Local:** for each hidden neuron, value $hid[0:n_h]$ and error $hErr[1:n_h]$; for each output neuron, predicted value $out[1:k]$ and error $oErr[1:k]$.
5. initialise hw and ow randomly, $\backslash\backslash$ *σ is a sigmoid function*
6. **repeat** $\{$
7. **for each example** e **in** E **do** $\{$
8. **for each** $h \in \{1, \ldots, n_h\}$ **do** $\{$
9. $hid[h] := \sigma(\Sigma_{i=0}^{n}(hw[i,h] \times val(e, X_i)))$ $\}$
10. **for each** $o \in \{1, \ldots, k\}$ **do** $\{$
11. $out[o] := \sigma(\Sigma_{h=0}^{n}(ow[i,h] \times hid[h]))$
12. $oErr[o] := out[o] \times (1 - out[o]) \times (val(e, Y_o) - out[o])$ $\}$

1. **Algorithm** BackPropagationLearner(X,Y,E,n_h,η)

2. **Inputs:** input $X = \{X_1, \dots X_n\}$ and output $Y = \{Y_1, \dots, Y_k\}$ set of examples E, n_h - number of hidden neurons, η - learning rate.

3. **Outputs:** hidden neuron weights $hw[1:n, 1:n_h]$; output weights $ow[0:n_h, 1:k]$

4. **Local:** for each hidden neuron, value $hid[0:n_h]$ and error $hErr[1:n_h]$; for each output neuron, predicted value $out[1:k]$ and error $oErr[1:k]$.

5. initialise *hw* and *ow* randomly, \\\\ *σ is a sigmoid function*

6. **repeat** $\{$

7.   **for each example** e **in** E **do** $\{$

8.   **for each** $h \in \{1, \dots, n_h\}$ **do** $\{$

9. $hid[h] := \sigma(\Sigma_{i=0}^{n}(hw[i,h] \times val(e, X_i))) \}$

10.   **for each** $o \in \{1, \dots, k\}$ **do** $\{$

11. $out[o] := \sigma(\Sigma_{h=0}^{n}(ow[i,h] \times hid[h]))$

12. $oErr[o] := out[o] \times (1 - out[o]) \times (val(e, Y_o) - out[o]) \}$

1. **Algorithm** BackPropagationLearner(X,Y,E,n_h,η)
2. **Inputs:** input $X = \{X_1, \ldots X_n\}$ and output $Y = \{Y_1, \ldots, Y_k\}$ set of examples E, n_h - number of hidden neurons, η - learning rate.
3. **Outputs:** hidden neuron weights $hw[1 : n, 1 : n_h]$; output weights $ow[0 : n_h, 1 : k]$
4. **Local:** for each hidden neuron, value $hid[0 : n_h]$ and error $hErr[1 : n_h]$; for each output neuron, predicted value $out[1 : k]$ and error $oErr[1 : k]$.
5. initialise hw and ow randomly, \\\\ *σ is a sigmoid function*
6. **repeat** $\{$
7.   **for each example** e **in** E **do** $\{$
8.   **for each** $h \in \{1, \ldots, n_h\}$ **do** $\{$
9. $hid[h] := \sigma(\Sigma_{i=0}^{n}(hw[i,h] \times val(e, X_i)))\ \}$ \\\\ $pval(h, e)$
10.   **for each** $o \in \{1, \ldots, k\}$ **do** $\{$
11. $out[o] := \sigma(\Sigma_{h=0}^{n}(ow[i,h] \times hid[h]))$ \\\\ $pval(o, e)$
12. $oErr[o] := out[o] \times (1 - out[o]) \times (val(e, Y_o) - out[o])\ \}$

1. **Algorithm** BackPropagationLearner(X,Y,E,n_h,η)

2. **Inputs:** input $X = \{X_1, \dots X_n\}$ and output $Y = \{Y_1, \dots, Y_k\}$ set of examples E, n_h - number of hidden neurons, η - learning rate.

3. **Outputs:** hidden neuron weights $hw[1 : n, 1 : n_h]$; output weights $ow[0 : n_h, 1 : k]$

4. **Local:** for each hidden neuron, value $hid[0 : n_h]$ and error $hErr[1 : n_h]$; for each output neuron, predicted value $out[1 : k]$ and error $oErr[1 : k]$.

5. initialise hw and ow randomly, \\\\ *σ is a sigmoid function*

6. **repeat** $\{$

7.   **for each example** e **in** E **do** $\{$

8.   **for each** $h \in \{1, \dots, n_h\}$ **do** $\{$

9. $hid[h] := \sigma(\Sigma_{i=0}^{n}(hw[i, h] \times val(e, X_i)))\ \}$

10.   **for each** $o \in \{1, \dots, k\}$ **do** $\{$

11. $out[o] := \sigma(\Sigma_{h=0}^{n}(ow[i, h] \times hid[h]))$

12. $oErr[o] := out[o] \times (1 - out[o]) \times (val(e, Y_o) - out[o])\ \}$ \\\\ *σ error*

1. **Algorithm** BackPropagationLearner(X,Y,E,n_h,η)

2. **Inputs:** input $X = \{X_1, \ldots X_n\}$ and output $Y = \{Y_1, \ldots, Y_k\}$ set of examples E, n_h - number of hidden neurons, η - learning rate.

3. **Outputs:** hidden neuron weights $hw[1:n, 1:n_h]$; output weights $ow[0:n_h, 1:k]$

4. **Local:** for each hidden neuron, value $hid[0:n_h]$ and error $hErr[1:n_h]$; for each output neuron, predicted value $out[1:k]$ and error $oErr[1:k]$.

5. initialise hw and ow randomly, \\\\ *σ is a sigmoid function*

6. **repeat** $\{$

7.   **for each example** e **in** E **do** $\{$

8.   **for each** $h \in \{1, \ldots, n_h\}$ **do** $\{$

9. $hid[h] := \sigma(\Sigma_{i=0}^{n}(hw[i,h] \times val(e, X_i)))$ $\}$

10.   **for each** $o \in \{1, \ldots, k\}$ **do** $\{$

11. $out[o] := \sigma(\Sigma_{h=0}^{n}(ow[i,h] \times hid[h]))$

12. $oErr[o] := out[o] \times (1 - out[o]) \times (val(e, Y_o) - out[o])$ $\}$

13.   **for each** $h \in \{0, \ldots, n_h\}$ **do** $\{$

14. $hErr[h] := hid[h] \times (1 - hid[h]) \times \sum_{o=0}^{k}(ow[h,o] \times oErr[o])$

1. **Algorithm** BackPropagationLearner(X,Y,E,n_h,η)
2. **Inputs:** input $X = \{X_1, \ldots X_n\}$ and output $Y = \{Y_1, \ldots, Y_k\}$ set of examples E, n_h - number of hidden neurons, η - learning rate.
3. **Outputs:** hidden neuron weights $hw[1:n, 1:n_h]$; output weights $ow[0:n_h, 1:k]$
4. **Local:** for each hidden neuron, value $hid[0:n_h]$ and error $hErr[1:n_h]$; for each output neuron, predicted value $out[1:k]$ and error $oErr[1:k]$.
5. initialise hw and ow randomly, \\\\ *σ is a sigmoid function*
6. **repeat** $\{$
7. **for each example** e **in** E **do** $\{$
8. **for each** $h \in \{1, \ldots, n_h\}$ **do** $\{$
9. $hid[h] := \sigma(\Sigma_{i=0}^{n}(hw[i,h] \times val(e, X_i)))$ $\}$
10. **for each** $o \in \{1, \ldots, k\}$ **do** $\{$
11. $out[o] := \sigma(\Sigma_{h=0}^{n}(ow[i,h] \times hid[h]))$
12. $oErr[o] := out[o] \times (1 - out[o]) \times (val(e, Y_o) - out[o])$ $\}$
13. **for each** $h \in \{0, \ldots, n_h\}$ **do** $\{$
14. $hErr[h] := hid[h] \times (1 - hid[h]) \times \sum_{o=0}^{k}(ow[h,o] \times oErr[o])$ where backpropagation of error from output to hidden layer happens

Algorithm BackPropagationLearner(X,Y,E,n_h,η)
Inputs: input $X = \{X_1, \ldots X_n\}$ and output $Y = \{Y_1, \ldots, Y_k\}$ set of examples E, n_h - number of hidden neurons, η - learning rate.
Outputs: hidden neuron weights $hw[1:n, 1:n_h]$; output weights $ow[0:n_h, 1:k]$
Local: for each hidden neuron, value $hid[0:n_h]$ and error $hErr[1:n_h]$; for each output neuron, predicted value $out[1:k]$ and error $oErr[1:k]$.
initialise hw and ow randomly, \\ *σ is a sigmoid function*
repeat $\{$
  for each example e **in** E **do** $\{$
    for each $h \in \{1, \ldots, n_h\}$ **do** $\{$
      $hid[h] := \sigma(\Sigma_{i=0}^{n}(hw[i,h] \times val(e, X_i)))$ $\}$
    for each $o \in \{1, \ldots, k\}$ **do** $\{$
      $out[o] := \sigma(\Sigma_{h=0}^{n}(ow[i,h] \times hid[h]))$
      $oErr[o] := out[o] \times (1 - out[o]) \times (val(e, Y_o) - out[o])$ $\}$
    for each $h \in \{0, \ldots, n_h\}$ **do** $\{$
      $hErr[h] := hid[h] \times (1 - hid[h]) \times \sum_{o=0}^{k}(ow[h,o] \times oErr[o])$

      for each $i \in \{0, \ldots, n\}$ **do** $\{$
        $hw[i,h] := hw[i,h] + \eta \times hErr[h] \times val(e, X_i)$ $\}$

Algorithm BackPropagationLearner(X,Y,E,n_h,η)
Inputs: input $X = \{X_1, \ldots X_n\}$ and output $Y = \{Y_1, \ldots, Y_k\}$ set of examples E, n_h - number of hidden neurons, η - learning rate.
Outputs: hidden neuron weights $hw[1 : n, 1 : n_h]$; output weights $ow[0 : n_h, 1 : k]$
Local: for each hidden neuron, value $hid[0 : n_h]$ and error $hErr[1 : n_h]$; for each output neuron, predicted value $out[1 : k]$ and error $oErr[1 : k]$.
initialise hw and ow randomly, \\ σ *is a sigmoid function*
repeat {
    for each example e **in** E **do** {
        for each $h \in \{1, \ldots, n_h\}$ **do** {
            $hid[h] := \sigma(\Sigma_{i=0}^{n}(hw[i, h] \times val(e, X_i)))$ }
        for each $o \in \{1, \ldots, k\}$ **do** {
            $out[o] := \sigma(\Sigma_{h=0}^{n}(ow[i, h] \times hid[h]))$
            $oErr[o] := out[o] \times (1 - out[o]) \times (val(e, Y_o) - out[o])$ }
        for each $h \in \{0, \ldots, n_h\}$ **do** {
            $hErr[h] := hid[h] \times (1 - hid[h]) \times \sum_{o=0}^{k}(ow[h, o] \times oErr[o])$

            for each $i \in \{0, \ldots, n\}$ **do** {
                $hw[i, h] := hw[i, h] + \eta \times hErr[h] \times val(e, X_i)$ }
            for each $o \in \{1, \ldots, k\}$ **do** {
                $ow[h, o] := ow[h, o] + \eta \times oErr[o] \times hid[h]$ } } } }

1. **Algorithm** BackPropagationLearner(X,Y,E,n_h,η)

2. **Inputs:** input $X = \{X_1, \ldots X_n\}$ and output $Y = \{Y_1, \ldots, Y_k\}$ set of examples E, n_h - number of hidden neurons, η - learning rate.

3. **Outputs:** hidden neuron weights $hw[1:n, 1:n_h]$; output weights $ow[0:n_h, 1:k]$

4. **Local:** for each hidden neuron, value $hid[0:n_h]$ and error $hErr[1:n_h]$; for each output neuron, predicted value $out[1:k]$ and error $oErr[1:k]$.

5. initialise hw and ow randomly, \\\\ σ *is a sigmoid function*

6. **repeat** $\{$

7.   **for each example** e **in** E **do** $\{$

8.   **for each** $h \in \{1, \ldots, n_h\}$ **do** $\{$

9. $hid[h] := \sigma(\Sigma_{i=0}^{n}(hw[i,h] \times val(e, X_i)))\ \}$

10.   **for each** $o \in \{1, \ldots, k\}$ **do** $\{$

11. $out[o] := \sigma(\Sigma_{h=0}^{n}(ow[i,h] \times hid[h]))$

12. $oErr[o] := out[o] \times (1 - out[o]) \times (val(e, Y_o) - out[o])\ \}$

13.   **for each** $h \in \{0, \ldots, n_h\}$ **do** $\{$

14. $hErr[h] := hid[h] \times (1 - hid[h]) \times \sum_{o=0}^{k}(ow[h,o] \times oErr[o])$

15.   **for each** $i \in \{0, \ldots, n\}$ **do** $\{$

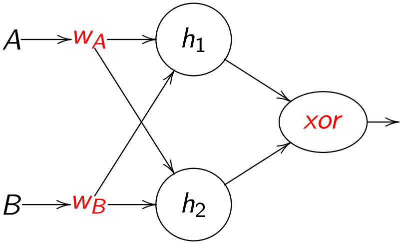
16. $hw[i,h] := hw[i,h] + \eta \times hErr[h] \times val(e, X_i)\ \}$

17.   **for each** $o \in \{1, \ldots, k\}$ **do** $\{$

18. $ow[h,o] := ow[h,o] + \eta \times oErr[o] \times hid[h]\ \}\ \}\ \}\ \}$

19. **until** termination.

Linear inseparable data problem – SOLVED



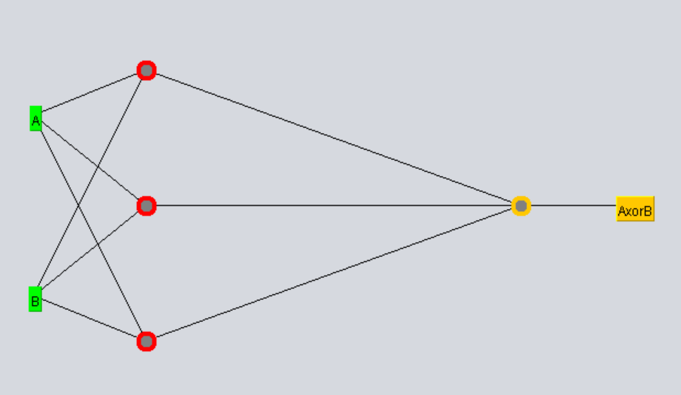
Input features and target features:

A	B	A xor B
true	true	false
true	false	true
false	true	true
false	false	false

Demo...

Typical Weka output...

For the network: (Note the three hidden nodes)

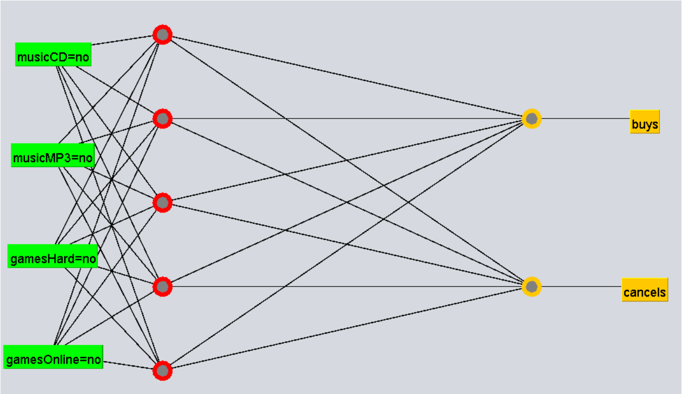


Typical Weka output...

```
Linear Node 0
    Inputs    Weights
    Threshold    1.1843021531056706
    Node 1    2.6416772902929098
    Node 2    -2.5814290683635264
    Node 3    -2.630503995513653
Sigmoid Node 1
    Inputs    Weights
    Threshold    -3.4118173052519554
    Attrib A    -2.2379377251180195
    Attrib B    2.9634175660215147
Sigmoid Node 2
    Inputs    Weights
    Threshold    -1.114705675412895
    Attrib A    -2.399963442729726
    Attrib B    -0.5388576617560114
Sigmoid Node 3
    Inputs    Weights
    Threshold    -2.393498304903795
    Attrib A    1.4958255155150815
    Attrib B    2.868341374969347
Class    Node 0
```

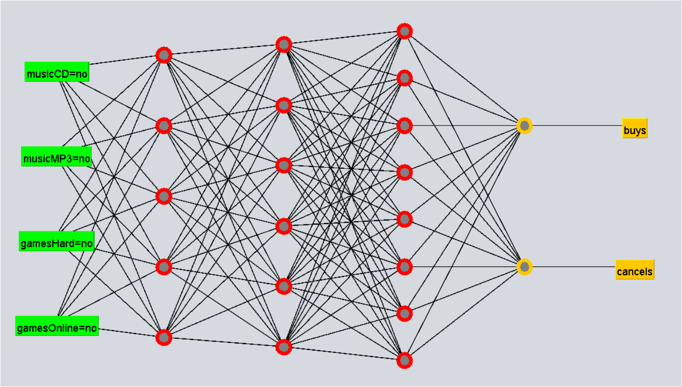
Our **Customer Preferences** data set, in Weka

One hidden layer with 5 neurons (and sigmoid output)



Our **Customer Preferences** data set, in Weka

Three hidden layers with 5,6,8 hidden neurons (and sigmoid output)



In Summary, the algorithm

- uses backpropagation

In Summary, the algorithm

- uses backpropagation
- repeats evaluation for all examples

In Summary, the algorithm

- uses backpropagation
- repeats evaluation for all examples
- minimises the error - by iterating through all of the examples.

In Summary, the algorithm

- uses backpropagation
- repeats evaluation for all examples
- minimises the error - by iterating through all of the examples.

Homework: practice using Neural nets with various parameters

- learning rate
- initialisation of parameters
- stopping criterion (number of iterations, target errors)
- activation functions
- number of layers; and number of neurons in every hidden layer
- number of features
- number of output classes
- make feature values more interesting than 0 and 1

In Summary, the algorithm

Further Reading

Our Weka textbook: §6.4 pages 232-241, §11.4 pages 469-472

Some generic advice:

- The neural net size will depend on the size of your data: the input layer will be as big as many features you have; the output layer – as big as you have “labels”/classes.

Some generic advice:

- The neural net size will depend on the size of your data: the input layer will be as big as many features you have; the output layer – as big as you have "labels"/classes.
- Often the software you use will determine these parameters at the time you load your data.

Some generic advice:

- The neural net size will depend on the size of your data: the input layer will be as big as many features you have; the output layer – as big as you have "labels"/classes.
- Often the software you use will determine these parameters at the time you load your data.
- You will still have to provide: number of hidden layers and their size, learning parameters (learning rate, learning function); and preferred activation functions;
- Extra layers are needed to handle "non-linearly separable" data; and to make classification more precise.

Some generic advice:

- The neural net size will depend on the size of your data: the input layer will be as big as many features you have; the output layer – as big as you have "labels"/classes.
- Often the software you use will determine these parameters at the time you load your data.
- You will still have to provide: number of hidden layers and their size, learning parameters (learning rate, learning function); and preferred activation functions;
- Extra layers are needed to handle "non-linearly separable" data; and to make classification more precise.
- How many layers will work best for your example? – is determined experimentally. You will notice that after some point adding more layers no longer improves accuracy but still consumes time; may even lead to overfitting

Some generic advice:

- Whether the data is linearly separable or not does not depend on the size of the set or the number of features – see XOR example. Generally, just using 1-2 extra layers is a good rule of thumb.

Some generic advice:

- What should I do if the accuracy is low? – You will need to understand where the problem lies.

Some generic advice:

- What should I do if the accuracy is low? – You will need to understand where the problem lies.
  1. May be you need to tune Neural net parameters (number of layers, learning rate, etc)

Some generic advice:

- What should I do if the accuracy is low? – You will need to understand where the problem lies.
    1. May be you need to tune Neural net parameters (number of layers, learning rate, etc)
    2. May be your data is badly split: e.g. your training examples have little in common with your testing examples. So, you are training or testing on non-representative sets

Some generic advice:

- What should I do if the accuracy is low? – You will need to understand where the problem lies.
  1. May be you need to tune Neural net parameters (number of layers, learning rate, etc)
  2. May be your data is badly split: e.g. your training examples have little in common with your testing examples. So, you are training or testing on non-representative sets
  3. May be your feature extraction is not representative: e.g. your features are "gender" and "nationality" when you are trying to determine customer preferences – such features are not enough!

Some generic advice:

- How many features can one have? – you can have many, if you have plenty of data (many tools will require some ratio between the number of features and the number of training examples). ... Could be hundreds of features; but generally, people avoid adding too many, as excessive feature increase may affect ability to learn efficiently.

Some generic advice:

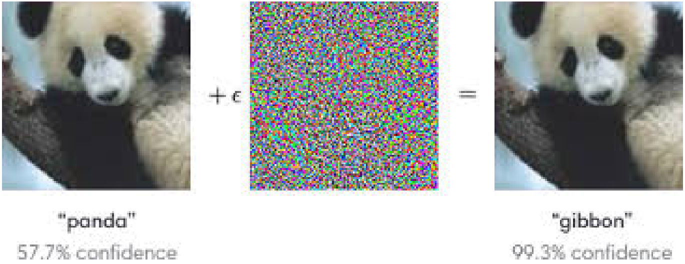
- How many features can one have? – you can have many, if you have plenty of data (many tools will require some ratio between the number of features and the number of training examples). ... Could be hundreds of features; but generally, people avoid adding too many, as excessive feature increase may affect ability to learn efficiently.
- Feature "values" do not have to be binary; in fact, often it is un-natural for them to be binary. Our example was: feature "email short"? One would better reformulate and have a feature "number of lines".

Research topics in Neural nets

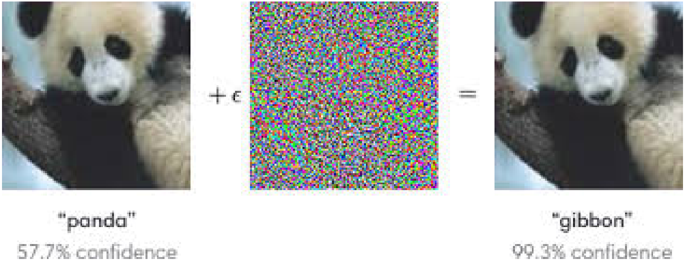
Weaknesses of Neural nets

- not easily conceptualised
- prone to error
- prone to adversarial attack

Research topics in Neural nets



Research topics in Neural nets



- Verification needed: many issues with safety (autonomous devices, cars), security (adversarial attacks)
- Problem: – even to state verification conditions!
- Current methods: Neurons to Logic (*á la* McCulloch and Pitts), Automated Theorem proving, SMT solvers

Your homework; Test 4, Part 2

- Load the small emotion recognition set to Weka (the numerical version); and the corresponding test set from Test 3. Choose:
  - Multilayer Perceptron as a classifier, training on the training set only (no cross validation)
  - GUI = True option: you will see the graphical interface
  - set the number of hidden layers to 0
  - **Use "More options" $\longrightarrow$ "output predictions" $\longrightarrow$ Plaintext"** - (it is really crucial)
  - Learning rate: 0.2
  - momentum: 0.2
  - training time = 500
  - weka.classifiers.functions.MultilayerPerceptron -L 0.2 -M 0.2 -N 500 -V 0 -S 0 -E 20 -H 0 -G -R
- Check the network's architecture, be ready to answer questions
- Check the performance of the network; and the weights it computes as a result.

Your homework; Test 4, Part 2

- Repeat the same experiment, with the same settings, but now use the logistic, instead of numeric, data sets, attached to Test 3.
- Notice and explain any differences in the neural nets, and the algorithm outputs.
- Be ready to answer questions.

Make predictions using the neural net

Note that your test set is, as before:

Test set:

1. Test 1: "a Happy face with noise":
   `White , Black , Black , White , Happy`
2. Test 2: "a Happy face with a beard":
   `Black, Black, White, Black, Happy`

... numeric or logistic version

Test 4, Part 3

- Take the same settings for Multilayer Perceptron as in Test 4, Part 2
- Take again Numeric and Logistic representations of the small emotion recognition set; and the test sets
- For each, vary the following:
  - Number of hidden neurons: 1, 2, 5 (in one hidden layer)
  - Number of hidden layers: 3 (in hidden layer 1) and 5 (in hidden layer 2)
  - Number of hidden layers: 3 (in hidden layer 1), 5 (in hidden layer 2) and 7 (in hidden layer 3)
  - Note how the network architecture varies in the course of these experiments
  - Note all parameters learned by the network
  - Note accuracies
  - Be ready to answer questions

Course summary: has the plan worked?

We had 4 weeks of lectures, covering major groups of ML methods

- Bayesian Probabilities,
- Unsupervised learning (Clustering) and
- three major Supervised Learning Methods:
  - Decision trees,
  - Linear Regression,
  - Neural Nets.

Course summary: has the plan worked?

My goals were:

- Give you “simple enough” material so that you can understand every little detail as your “own”.

Course summary: has the plan worked?

My goals were:

- Give you “simple enough” material so that you can understand every little detail as your “own”.
- Expose you to the challenges of the area:
  - Theoretical – its strong rootings in Linear Algebra, Probability Theory and Statistics;
  - and Practical – Search spaces are too big, complexities too high – therefore much of work in the area is about finding good parameters and heuristics for some local kinds of problems.

Course summary: has the plan worked?

My goals were:

- Give you "simple enough" material so that you can understand every little detail as your "own".
- Expose you to the challenges of the area:
  - Theoretical – its strong rootings in Linear Algebra, Probability Theory and Statistics;
  - and Practical – Search spaces are too big, complexities too high – therefore much of work in the area is about finding good parameters and heuristics for some local kinds of problems.
- Give you a lot of practice – hence many demos, practical tests, and big Coursework assignments
  - Tests were to support your understanding of lectures, and prepare you for CW2-3.
  - In CW2-3 it was crucial for you to get an experience with data of real-life industrial size
  - Some problems: complexity of algorithms, redundancy of features are not really seen on small data sets,
  - it was crucial for you to see them.

The end

When you use ML in your future work, I hope that your clear knowledge of "simple things" will support you, and help you to have a firm ground when you need to tackle harder problems.

Thanks for your attention,

questions, hard work, enthusiasm...

The end

When you use ML in your future work, I hope that your clear knowledge of "simple things" will support you, and help you to have a firm ground when you need to tackle harder problems.

Thanks for your attention,

questions, hard work, enthusiasm...

- Good luck with CW3!
- Any questions – please ask by email and/or in the lab
- Next week Thursday – revision lecture (by Diana and myself jointly); Thursday labs are on for any final help with test or CW3.
- Next week Friday – free
- Final CW3 interviews – one Thursday after