A backpropagation network package

The documentation

Marko Grönroos (Email: magi@utu.fi)
Christian Lehtinen (Email: jcleh@utu.fi)

Department of Computer Science of the University of Turku, 1993

Contents:

Chapter 1. Classes and class operators of the BPN package 2
1.1. Unit 2
1.2. Layer 2
1.3. BPN 3
1.4. TrainSet 5
1.4.1. FullTrainSet 5
1.4.2. ArrayTrainSet 6
Chapter 2. Using the package 7
2.1. Training patterns 7
2.2. Network definition 7
2.3. Training 8
2.4. Testing 8
Chapter 3. Examples 10
3.1. Character recognition with no bias 10
3.2. Character recognition with no bias and linear output layer 10
3.3. XOR with no bias 11
3.4. Training with use_allonce flag set on, no bias 11
3.5. Character recognition with bias 11

Chapter 1. Classes and class operators of the BPN package

1.1. Unit

```
class Unit {
  protected:
    float  output,
           error;
```

output is the output value of the unit and ***error*** the calculated error.

```
floatARR  weights,
          weight_deltas,
          last_delta;
```

weights is a float array that contains the inputs to the unit and ***last_delta*** contains the deltas of the respective weights from the last iteration turn. 'floatARR' is a parameterized (generic) array of floating point items. The bias (threshold) is the last item in the weights-array. ***weight_deltas*** is used when the use_allonce-flag is set on (see class BPN). It stores the deltas of the weight array.

```
public:
  void   make (int n);
```

make is the constructor of the Unit class. The use of parameterized array classes prevents the use of normal construction. The parameter is the number of source units that connect to an unit.

```
  void   init ();
```

init initializes the unit. This could be done in the constructor, but is made separate to get some more genericity. It initializes the weights with random values.

```
ARRAY(Unit)
```

This creates a parameterized class called '***UnitARR***', which is used by the Layer and BPN classes.

1.2. Layer

```
class Layer {
  protected:
```

```
    UnitARR   units;
```

units is the array of units in a layer. Should be self-explanatory.

```
    int  transf_func;
```

transf_func is a flag that indicates which transfer function should be used in this layer. Two transfer functions are implemented: *TF_SIGMOID* and *TF_LINEAR*. It is set with the function BPN::lsize and can be seen in the layer printout as letter 'S' (sigmoid) or 'L' (linear). Only the last layer can be made linear in this implementation for efficiency purposes.

```
  public:
    void      make    (int n, int prevn);
```

make constructs the layer. The parameter '*n*' is the number of units in the layer and the second parameter '*prevn*' the number of units in the previous layer.

```
    void      init    ();
```

init initializes all the units in the layer.

```
    ostream&  print   (ostream& out);
```

prints the layer to the given output stream.

```
    void      propagate   (Layer& previous, int use_bias);
```

propagates the signal from the given layer. This first parameter MUST be of the same size as the size given in the constructor. The second parameter is a flag that tells whether to use bias or not.

```
    float     compouterr  (floatARR& desired);
```

computes the error at the last layer. This must be used only for the last layer. The parameter is a float array that contains the desired output for the propagated pattern.

```
    void      errorprop   (Layer& upper);
```

computes the error for all other but not (ever) for the last layer. The parameter is a reference to the next upper layer, where from the error is to be propagated backwards.

```
    void      nulldeltaweights ();
```

Sets the weight deltas to 0.

1.3. BPN

```
class BPN {
  protected:
    LayerARR  layers;
```

```
    float     alpha,
              eta;
```

'***layers***' contains the layers in the network and ***alpha*** and ***eta*** are the learning parameters of the network.

```
  public:
    int       use_bias;
    int       use_allonce;
```

use_bias is a flag that indicates whether (true) or not (false) the bias value should be used in calculations. Yes, it's public, and it's used by directly setting it to 0 or 1. The default is true. ***use_allonce***-flag indicates whether or not to use the learning method where all patterns are scanned before weight changes. The default is false.

```
              BPN     (int n, float alpha, float eta);
```

The constructor of the entire network. The first parameter, '*n*', is the number of layers in the network. The other parameters are the learning parameters of the network.

```
    void      init    ();
```

Initializes the network. This MUST be used AFTER the sizes of layers have been set.

```
    void      lsize   (int n, int siz);
    void      lsize   (int n, int siz, int tfunc);
```

lsize sets the size (second parameter) of the given layer. The optional third parameter is the transfer function (*TF_SIGMOID* or *TF_LINEAR*). The third parameter must be used only for the last layer, and it can be omitted if the default, TF_SIGMOID, is to be used. This function MUST be used for each of the layers in the network.

```
    ostream&  print   (ostream& out);
```

Prints out the output values of all the layers to the given output stream.

```
    void      propagate ();
```

Propagates the signal present at the input layer to the output layer.

```
    float     errorprop (floatARR& desired);
```

Backpropagates the errors between propagated signal and the given
desired output pattern back to the input layer. Does not adjust the weights.

```
    void      nulldeltaweights ();
```

Sets the weight deltas to 0.

```
    void      adjweights  ();
```

Adjusts the weights of the network.

```
void     train   (TrainSet& set, int iterations);
```

Trains the network with the given training set for the given number of iterations.

```
void     test    (TrainSet& set, int number);
```

Tests the network with the training pattern of the given training set which is located in the place given in the second parameter.

```
void     iterate(TrainSet& set, int times);
```

Iterates the network so that after propagating the last pattern in the given training set to the output layer, the operation shifts the values at the input layer one step to left and then copies the output value to the rightmost input node. This is then repeated for *times* times. Especially useful for extrapolation applications.

```
void     save    (char* filename);
```

Saves the weight matrix of the network into a file.

```
void     load    (char* filename);
```

Loads the weight matrix of the network from a file. Observe that the network size MUST be the same as the size in the saved file.

1.4. TrainSet

1.4.1. The basic TrainSet

TrainSet is a base class that is never used as such, but always inherited (a deferred class). It contains the basic description for training set interface, which it's descendants then apply for different types of training sets.

```
class TrainSet {
  public:
    int  inputs,
         outputs;
```

inputs and *outputs* are the number of input and output values in the training pattern. These are usually give as parameters to the constructors in the descendant classes.

```
float    input   (int pattern, int i);
float    output  (int pattern, int i);
```

The functions *input* and *output* are access functions for values in the training set. They fetch the *i*th input or output value from the selected *pattern*. These are deferred functions and always redefined in descendant classes.

1.4.2. FullTrainSet

```
class FullTrainSet : public TrainSet {
  private:
    floatARRARR ins,
                outs;
```

ins and ***outs*** are matrices that contain the corresponding input-output training patterns (of type floatARR).

```
  public:
       FullTrainSet   (int patts, int inputs, int outputs,
                       char* filename);
```

Patts is the number of training patterns and *inputs* and *outputs* are the number of input and output sizes of the patterns. The last parameter is the name of the file that contains the patterns.

```
    void   print  (ostream& out);
```

Prints the training set to the given output stream.

```
    void   mutate (int errcnt);
```

Mutates the inputs of the training set by given number of errors. The number of errors equals to the hamming distance between normal and mutated patterns.

1.4.3. ArrayTrainSet

This class was made to make it possible to use plain data sets as training sets, without the need to first convert their source files to classical training set files. The class simply reads the set from file to an array, and then calculates the values for input and output requests relatively. It also calculates the number of training patterns from the size of the set and sizes of inputs and outputs.

```
class ArrayTrainSet {
     floatARR  value;
  public:
    ArrayTrainSet  (int inputs, int outputs, char* filename);
```

Chapter 2. Using the package

2.1. Training patterns

The training patterns should be written to a separate file so that first comes the input pattern, then the output pattern. Values are separated by spaces, tabulator characters or linefeeds. The number of values in the file must match *patterns*(inputs + outputs)*, where patterns is the number of training patterns.

Example: xor.tra

```
0  0   0
0  1   1
1  0   1
1  1   0
```

2.2. Network definition

First we construct the network by defining the number of layers it has and it's learning parameters:

```
BPN  aNetwork  (number_of_layers, alpha, eta);
```

Then we set the size of each of the layers. Here we assume to have a three-layered network.

```
aNetwork.lsize (1, number of input units);
aNetwork.lsize (2, number of hidden units);
aNetwork.lsize (3, number_of_output_units, transfer_function);
```

The definition of the transfer function in the last layer can be omitted if the desired transfer function is sigmoid. The setup of the layer sizes must be done before initialization. Now the structure of the network has been defined. Next we have to initialize the weights, etc. This must be done.

```
aNetwork.init ();
```

If one wishes not to use the bias (threshold) value, the optional *use_bias*-flag should be set off:

```
aNetwork.use_bias = 0;
```

If one wishes to use a learning method where all the patterns are propagated before weight changes, this flag should be set to true.

```
aNetwork.use_allonce = 1;
```

2.3. Training

The input patterns are stored in a TrainSet. The file size MUST match the parameters of this constructor call.

```
FullTrainSet aTrainSet (Number of patterns in the file,
            number of inputs in each pattern,
            number of outputs in each pattern, "filename.tra");
```

Then we train the network with the training set for 2000 iterations. The *train* operation prints the current ***total squared error*** of the network to the standard output during the training so that one can follow the learning interactively. It also prints out the test for each pattern after every 50 iterations.

```
aNetwork.train   (aTrainSet, 2000);
```

After the training we usually wish to save the network.

```
aNetwork.save ("filename.wgh");
```

2.4. Testing

First we load a training set or a test set that we wish to apply to the network. If we don't have a trained network, we can load it's weights from a file with the load operation.

```
TrainSet aTrainSet (patterns, inputs, outputs, "filename.tra");
aNetwork.load ("filename.wgh");
```

Then we can test the network with a pattern in the traning set that we select. The test-operator doesn't automatically print the results so we have to do it by hand. If we wish to do the testing for all the patterns in the set, we must do it with a for-loop.

```
aNetwork.test (aTrainSet, number of a pattern in the TrainSet);
aNetwork.print(cout);
```

If we have only a training set and we wish to easily create a test set, we can do it with the mutate-operation of the TrainSet class. It automatically makes random changes to all the patterns in the training set. Observe that it is dependant on the random number generator of the machine architecture. For example in Unix it always generates the same errors in every execution, unless the srand (int seed) - function is ran to change the random number seed.

```
aTrainSet.mutate (4);
```

The training set can be printed out with the print operator.

Chapter 3. Examples

These are the answers to the backpropagation network exercises in the neural network course.

3.1. Character recognition with no bias

Training:

```
BPN network (3, 0.2, 0.1);
network.lsize (1, 35);
network.lsize (2, 20);
network.lsize (3, 36, TF_LINEAR);
network.init ();
network.use_bias = 0;
FullTrainSet trainset (36, 7*5, 36, "merkit.tra");
network.train (trainset, 10000);
network.save ("3.2.wgh");
```

Testing:

```
FullTrainSet trainset (36, 7*5, 36, "merkit.tra");
network.load ("3.2.wgh");
trainset.mutate (4);
for (int i=1; i<=36; i++) {
  network.test (trainset, i);
  network.print (cout);
}
```

, which is ran with different number of errors (here 4).

3.2. Character recognition with no bias and linear output layer

Training:

```
BPN network (3, 0.2, 0.1);
network.lsize (1, 35);
network.lsize (2, 20);
network.lsize (3, 36, TF_LINEAR);
network.init ();
network.use_bias = 0;
TrainSet trainset (36, 7*5, 36, "merkit.tra");
network.train (trainset, 10000);
network.save ("3.2.wgh");
```

Testing as in 3.1, except that with our parameters the network didn't learn so much that it would be sensible to run it with any errors.

3.3. XOR with no bias

```
BPN network (3, 0.5, 0.2);
network.lsize (1, 2);
network.lsize (2, 4);
network.lsize (3, 1);
network.init ();
network.use_bias = 0;
TrainSet trainset (4, 2, 1, "xor.tra");
network.train (trainset, 10000);
```

With this setup (4 hidden neurons) the network learned the patterns in 3000 iterations.

3.4. Training with use_allonce flag set on, no bias

As in 3.1. except

```
network.use_allonce = 1;
```

It learned very slowly and poorly. After 2000 iterations the error was about 13 (alpha=0.5, eta=0.2). Only about half of the characters were recognized, while others were about 0.00's in the output layer.

3.5. Character recognition with bias

Training:

```
BPN network (3, 0.7, 0.2);
```

(Network sizes as in 3.1)

```
network.use_bias = 1;
FullTrainSet trainset (36, 7*5, 36, "merkit.tra");
network.train (trainset, 10000);
network.save ("3.5.wgh");
```

Learns the patterns adequetly in about 1000 iterations.
Testing as in 3.1.