CB NOTE: 215
T. Degener
14.10.1992

Electromagnetic split-off recognition with Artificial Neural Networks

Thomas Degener
Ruhruniversität Bochum
Institut für Experimentalphysik I
D-4630 Bochum
e-mail: thomas@tau.ep1.ruhr-uni-bochum.de

Artificial Neural Networks are used to identify fluctuations in electromagnetic showers

1. Introduction

The first step when analyzing data of the Crystal Barrel calorimeter is to run the 'BCTRAK' part of the CB[1] off-line software package which groups neighboring crystals, with an energy deposit, to form clusters. These clusters are searched for local maxima, the PEDs[2], and it is assumed that every PED corresponds to a particle (in most cases photons) that has hit the calorimeter at this location. This assumption is correct in many cases but some PEDs are just fluctuations in an electromagnetic shower: this is called an electromagnetic split-off. Especially PEDs coming from low energy photons are hard to distinguish from split-offs because a 4C kinematic fit will almost ever fit an extra low energy photon.A lot of work has been done on this problem already and two methods, SMART[3] and DOLBY C[4], were developed. Section 2. gives a brief review on SMART, for more details see [1] . Using this methods, however, a model is introduced. A self organizing method like an ANN[5] as a model-free estimator is a totally different approach to that problem. This paper introduces the basic principles and terminology of ANN and describes how they were applied to the split-off problem.

1. CB: Crystal Barrel detector
2. PED: particle energy deposit
3. by Jürgen Salk
4. by Nigel Hessey
5. ANN: Artificial Neural Network

2. SMART

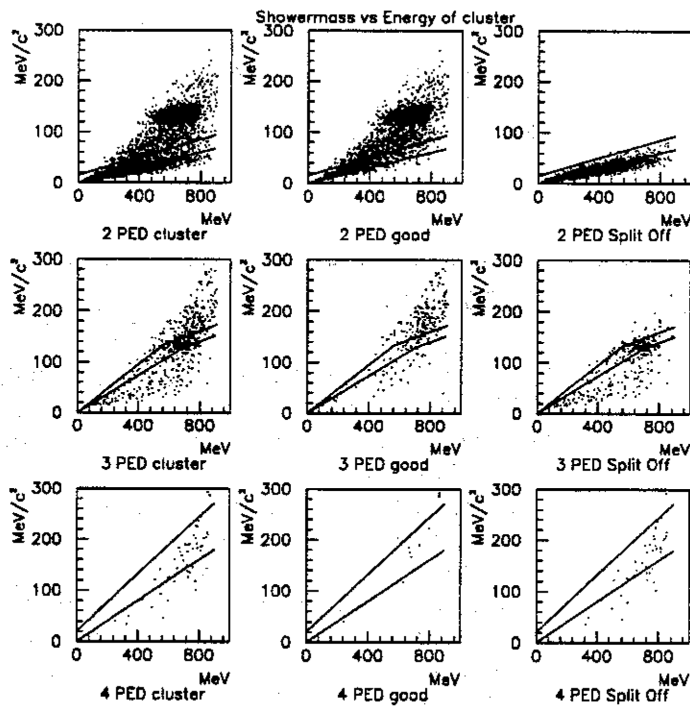
This method consists of a linear cut in the two-dimensional space given by the following quantities:

Energy of a cluster:.$E_{Cl} = \sum_i E_i$

Invariant showermass: $S_m = \sqrt{(\sum_i E_i)^2 - (\sum_i \vec{p}_i)^2}$

Fig.1 shows how the parameter space looks like for clusters with different number of PEDs.

FIGURE 1. Showermass vs. Energy ($2\pi_0\eta$ - Monte Carlo data)



The column in the middle shows the MC[1]-simulated showermass vs. energy for clusters containing only real PEDs and in the right column those containing at least one split-off.The lines correspond to the (linear) SMART cuts. Remember that in case of real data it is not possible to distinguish a priori between a split-off and a real PED. In this case

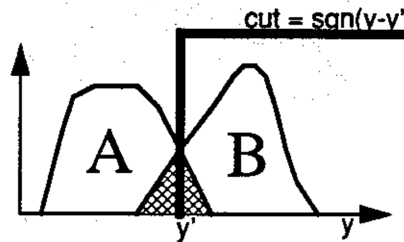
1. Monte Carlo (simulation)

one has a superposition of both sides shown in the left column. It is clearly visible that there are regions containing split-offs and real PEDs.A total separation of this classes in the space of cluster energy and showermass is therefore impossible.

3. The Bayesian limit

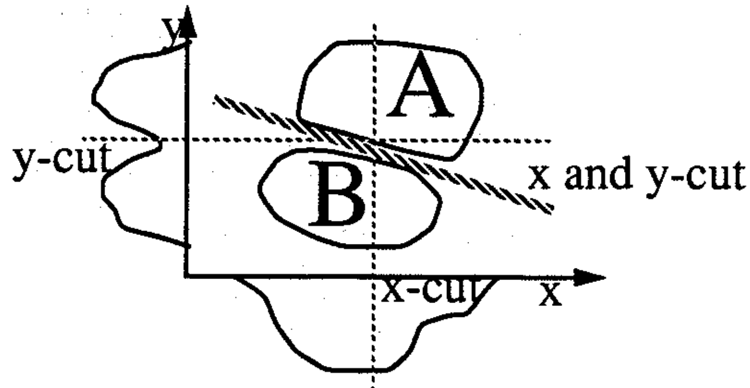
Perfect separation of two classes A and B is often impossible in principle no matter what kind of properties of the classes one investigates.Suppose there is only one property y known characterizing the classes.Then the situation is like in Fig. 2.There are elements or in terms of physics events of both classes in the hatched area.In this area a perfect separation is impossible.

FIGURE 2. One-dimensional class separation problem: The classes A(y) and B(y) aren't separable



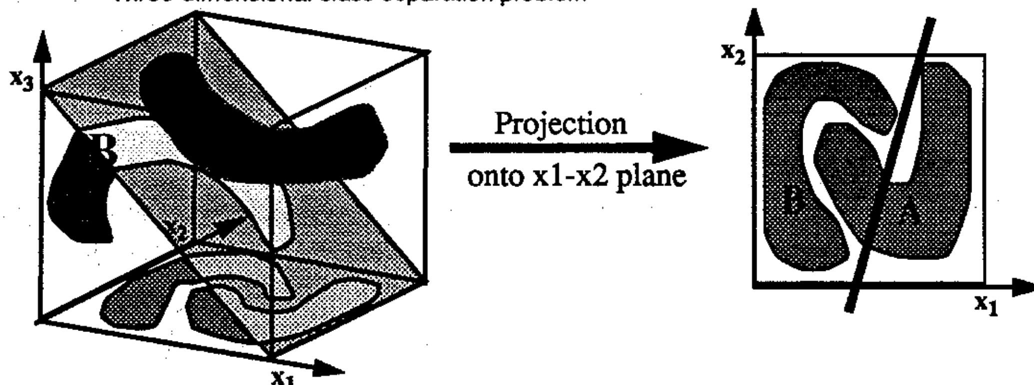
The best separation of the two classes with respect to y is reached if a cut is set in the middle of this region. This optimum of class separation is called the Bayesian limit.In this very simple one dimensional example the Bayesian limit is just a linear cut represented by a step function. 0 is assigned to elements of class A and 1 to class B. But in most cases however a one-dimensional linear cut is not sufficient.Take an additional parameter x into account (Fig. 3).

FIGURE 3. Two-dimensional class separation problem: A two dimensional step function (xy-cut) separates the classes



A linear cut just with respect to x would lead to an even worse result, but a linear cut in the xy-plane can separate the two classes perfectly! Again it can be represented as a step function where the argument is some linear combination of x and y. Fig. 4 shows on the left side a three-dimensional linear separable class separation problem and on the right its projection onto the x_1-x_2-plane. The two classes A and B can be separated by

FIGURE 4. Three-dimensional class separation problem



the grey plane, whereas the projection is no longer separable by a simple linear cut. The aim of every cut one applies to data is to come as close as possible to the Bayesian limit. Bear in mind that even finding the relevant properties of two classes A and B can be a difficult task, if it turns out that many properties contribute. Then it may be hopeless to find a series of cuts just by visual inspection because of the high dimensionality of the parameter space. Furthermore, a model is introduced by application of a series of sequential arbitrary cuts and almost ever a huge amount of background events (bias) remains and the Bayesian limit is missed by far. It would be best to find an estimator that yields a probability for class membership from parameter space directly.

For the problem given in Fig. 3 an optimal solution can be accomplished by using a step function with a two dimensional argument.Now I will discuss a similar but more complicated example where the structure of the solution will turn out to be of the same nature like an ANN. In Fig. 5.a) a two dimensional class separation problem is solved by using a highly non linear cut.Another way to accomplish an optimal cut F is building it up through an ensemble of step functions. Fig. 5.b) to Fig. 5.f) show how the cut function F is generated. A diagrammatic representation of F is given in Fig. 6. This looks very much like a feed forward ANN, where the input variables x_1 and x_2 are fed into the 'input nodes', get multiplied by a weight along the arrows, summed up at the entrance of a 'hidden node' and so on.The bias weights are just added. Adding bias weights gives the possibility to perform cuts that do not necessarily go through the origin. For ANN the original step function is normally replaced by a sigmoidal function (section 4.1.2). There are two reasons for doing so: The first is that some 'learning rules' (section 4.1.6) require a differentiable node function.The second reason is that for overlapping classes (Fig. 2 on page 3),the sigmoidal node function can be used to approximate the probability of an element belonging to one class or another.

FIGURE 5. Two-dimensional class separation problem: An optimal cut can be performed by using a highly non linear cut (a) or by building it up through an ensemble of linear cuts. The indices are chosen to fit to Fig. 6

a) non linear cut

b) $U_1 = \mathrm{sgn}(w_{31}x_1 + w_{32}x_2 + w_{30})$
$F = U_1$

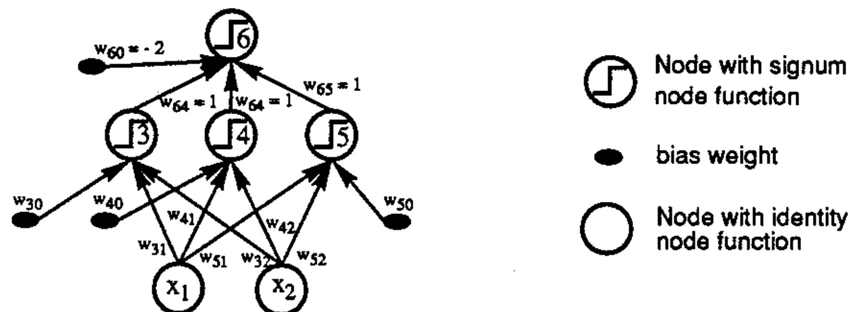
c) $U_2 = \mathrm{sgn}(w_{41}x_1 + w_{42}x_2 + w_{40})$
$F = U_1 + U_2$

d) $U_3 = \mathrm{sgn}(w_{51}x_1 + w_{52}x_2 + w_{50})$
$F = U_1 + U_2 + U_3$

e) $F = U_1 + U_2 + U_3 + w_{60}$
$w_{60} = -2$

f) Final result
$F = \mathrm{sgn}(U_1 + U_2 + U_3 + w_{60})$

FIGURE 6. Diagrammatic representation of the cut function F in Fig. 5



4. Artificial Neural Networks ANN

This chapter introduces the terminology of ANN (without trying to be complete). The network mostly used in HEP[1] is the Feed Forward Back Propagation ANN[2]. Other network structures are the linear and non linear associative memory, the Perceptron, the Hopfield ANN and the Boltzmann machine[3]. The main disadvantages of ANN to algorithmic methods are:
- It is hard to understand how the solution of a problem was found.
- Sometimes ANN have an unpredictable behavior because of their non linear nature
- ANN give an approximation instead of an exact solution.

On the other hand there are many advantages:
- ANN are self-organizing and allow the selection of classes without the application of sequential cuts.
- ANN act as a model free estimator (no previous knowledge has to be used).
- Learning by examples instead of programing an already known solution.
- ability to generalize to unknown inputs.
- ANN are suitable for massive parallel processing.

4.1 Components of an ANN

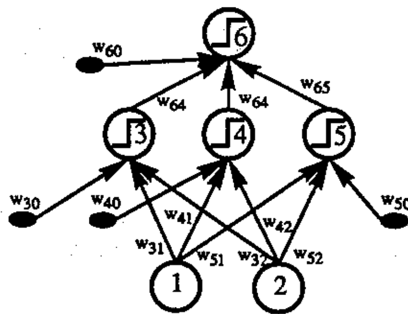
Two example graphs of ANNs are shown in Fig. 7.

4.1.1 Set of processing units

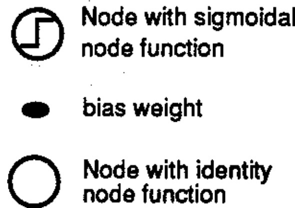
An ANN comprises a set of processing units called nodes. Every node can have an input and/or an output.There are three different kinds of nodes in a network:
Input node: Receives input only from external sources
Hidden node: Input and output only to/from other nodes in the network
Output node: Gives its output to external targets
Often one arranges a network in layers such that no node of a layer receives input or sends its output to a node of its own layer.In this case one has, depending on the nature of the nodes, input-, hidden- and output layers.
If the signal flow is straight through the network,i.e. from input- to hidden- to output layer then this is called a Feed Forward Neural Network.If the network contains loops then this is called a Feedback structure.

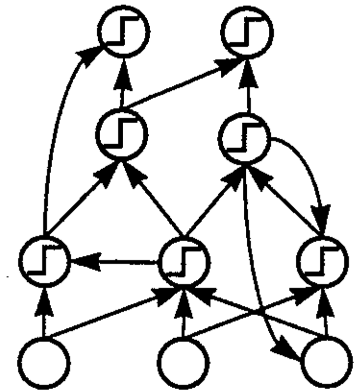
1. HEP: High Energy Physics
2. for a summary of recent ANN applications in HEP see [2]
3. they all differ in some of the aspects given in section 4.1

FIGURE 7. Examples for the layout of Neural networks



A feed forward ANN with one hidden layer





A more general ANN with feedback.

4.1.2 Node function

Also called activation function in literature.
This function defines how to calculate the output of a node from its inputs.There are excitatory or inhibitory connections in a network. This only makes sense if the node functions are restricted to monotonous node functions,i.e. uniform changes in the input of a node causes uniform changes in the output of it.Examples for node functions are:

Signum function: $$\operatorname{sgn}(x) = \begin{cases} 1 & \text{for} \quad x > 0 \\ 0 & \text{for} \quad x \leq 0 \end{cases} \quad \text{(EQ 1)}$$

Sigmoidal function: $$f(x) = \frac{1}{1+e^{-x}} \quad \text{(EQ 2)}$$

or $$f(x) = \frac{1}{2}(1 + \tanh(x)) \quad \text{(EQ 3)}$$

4.1.3 Network topology

The topology of a network is defined by the set of connection weights w_{ij}.. The weights define how strong a node is coupled to another one.A possible range of values is:

$$w_{ij} \in (-1, 1)$$

Representations of the network topology are:

1. Connection matrix $\mathcal{W}$
   The elements of the matrix are the weights.In a program it can be represented as a two-dimensional array.
2. Interlinked list
   Dynamical storing of actually existing weights.This is realized by complex pointer structures.For large networks this is absolutely necessary which is one of the reasons why a programming language like C is widely used for encoding neural networks, as pointers are not available in e.g. FORTRAN.

4.1.4 Network input function

The input function defines how to calculate from the output o_i of a previous node and from the connection weights w_{ij} the input a_i to node i.Often used input functions are:
Sum of weighted outputs:

$$\text{Input to node j: } a_j = \sum_i w_{ij} o_i \quad \text{(EQ 4)}$$

or

$$\text{Input to node j:} a_j = \sum_i w_{ij} o_i + w_{0j} \quad \text{(EQ 5)}$$

Product of weighted outputs:

$$\text{Input to node j: } a_j = \prod_i w_{ij} o_i \quad \text{(EQ 6)}$$

Maximum of weighted outputs:

$$\text{Input to node j:} a_j = max_i (w_{ij} o_i) \quad \text{(EQ 7)}$$

Minimum of weighted outputs:

$$\text{Input to node j:} a_j = min_i (w_{ij} o_i) \quad \text{(EQ 8)}$$

4.1.5 Network transfer function

One distinguishes:

1. Work phase (static ANN)
   The network does not change anymore i.e. all weights are fixed.The transfer function maps an input vector $\vec{a} = (a_1, a_2, \ldots a_k)$ from an external source onto an

   output vector $\vec{o} = (o_1, o_2, \ldots o_l)$ to an external target,i.e.:

   transfer function F: $\vec{o} = F(\vec{a})$
2. Training phase (dynamic ANN)

In this case the output of the network depends on previous input vectors as well,i.e.:

$$\text{transfer function F:} \quad \vec{o} = F(\vec{a}(t_0), \vec{a}(t_1), \ldots \vec{a}(t_j)) \qquad \text{(EQ 9)}$$

where the t_j 's denote the numbers of iteration.

4.1.6 Learning rule

During the training phase a set of learning examples (training set), i.e. external input vectors, is used to change the connection weights according to a learning algorithm. The aim is to change the transfer function in a way such that the training set is adapted,i.e. to a specific input vector of the training set the networks gives the desired output vector. If the correct target vectors to their corresponding input vectors of the training set are known (or a criterion exists that decides wether the result is desired or not) one can integrate this knowledge into the learning rule. This is called supervised learning. If no assumption on the network's output is used (for example Hebb's rule), then this is called unsupervised learning.In this case one hopes that the networks organize themselves according to the structure of the input data. This structure can be clusters, principal components, prototypes, and/or other typical features.A very good working learning rule for supervised learning, which is used here, is the Backpropagation algorithm (see section 5. on page 10). Some other learning rules are: Delta-rule, Metropolis algorithm, Vector quantization [3] . Principally the learning should stop when the weights do not change anymore or only very little.Then one says that the network has converged[1].

4.1.7 Convergence test

A way to test for convergence during learning is to store the weights after some iteration k., i.e. present k times an input vector to the network, calculate an output vector and change the weights according to the learning rule. Then do j extra iterations and store the weights for every iteration. If for all iterations the weight change is smaller than a small number ε, then go back to iteration k and 'freeze' the network otherwise continue learning.

The convergence test used here is simpler and less CPU consuming and is used in case of supervised learning: Stop learning and save the weights after k iterations. Then present another j patterns (j input vectors and their target vectors) to the network and calculate an error (section 4.1.8) for every corresponding output vector **without** changing the weights meanwhile. If for all j patterns these errors are smaller than a value ε (called the error bound), then 'freeze' the network. Otherwise continue learning at iteration k.

4.1.8 Error function

In case of supervised learning,e.g. the target vectors $\vec{t}$ are known, an error function has to be defined that tells how good the network performs at a specific time. This can be for example a sum of squared, absolute errors:

$$E = \frac{1}{2}\sum_j (t_j - o_j)^2 \qquad \text{(EQ 10)}$$

1. This does not mean exact mathematical convergence

But other error functions are possible as well. If the correct target vectors are unknown E can be replaced by a function that gives a measure of quality for the output of the network. For unsupervised learning no error function is needed.

4.1.9 Performance

Once the network has converged one likes to know how good the network does in separating classes. For a performance test of the network a different data set, that is 'unknown' to the network, has to be used. Not surprisingly the network usually does its task much better on the training set than on a unknown data set. If the number of extra iterations for the convergence test equals the number of training examples, then it is obvious that the error is smaller than the error bound (see section 4.1.7) for all outputs and all patterns. In this case the network has learned the training set by heart! But this must not be the best solution of the problem because especially when the training set is small, the network starts to pick up the noise of the training set as well. The use of a 'unknown' data set for a performance test shows how good the network generalizes.

5. The Backpropagation algorithm

BP[1] uses the derivative of the node function for gradient descent, therefore the node function must be differentiable. Let the network input function be a weighted sum input plus a bias weight, then the input a_j to the jth node is

$$a_j = w_{j0} + \sum_i w_{ji} o_i \qquad \text{(EQ 11)}$$

Where o_i is the output of a previous node.This is passed through the node function f_j to produce the output of the jth node

$$o_j = f_j(a_j) \qquad \text{(EQ 12)}$$

The change in weight is taken to be proportional to the contribution of that weight on the total error E (see equation 10 on page 9).The constant of proportionality α is called the **learning rate**[2]

$$\Delta w_{ji} = -\alpha \frac{\partial E}{\partial w_{ji}} \qquad \text{(EQ 13)}$$

By the chain rule this gives

1. BP: Back Propagation algorithm

2. equation 13 is known as the Delta-rule when it is only applied to the weig[illegible] connecting to the output layer. BP is a generalization of the Delta-rule.

(EQ 14)

$$\Delta w_{ji} = -\alpha \frac{\partial E}{\partial w_{ji}}$$

$$= -\alpha \frac{\partial E}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}$$

$$= -\alpha \Lambda_j o_i$$

where the derivative of E with respect to a_j has been replaced by Λ_j.Using the chain rule again gives

(EQ 15)

$$\Lambda_j = \frac{\partial E}{\partial a_j}$$

$$= \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial a_j}$$

$$= \frac{\partial E}{\partial o_j} f'(a_j)$$

Suppose the error function E has been defined as a sum of squared errors and the jth node is an output node then the change in weight for this node is

$$\Delta w_{ji} = -\alpha (t_j - o_j) \cdot f'(a_j) o_i \qquad \text{(EQ 16)}$$

For the sigmoidal function given in equation 2 on page 7 this can be written more explicitly as

$$\Delta w_{ji} = -\alpha (t_j - o_j) \cdot o_i \cdot f(a_j) \cdot (1 - f(a_j)) \qquad \text{(EQ 17)}$$

If the jth node is a hidden node and the indices from k to l denote nodes to which the jth node is connected then the weight change is given by

$$\Delta w_{ji} = -\alpha o_i \cdot f'(a_j) \cdot \frac{\partial E}{\partial o_j}$$

$$= -\alpha o_i \cdot f'(a_j) \cdot \sum_{m=k}^{l} \frac{\partial E}{\partial a_m} \frac{\partial a_m}{\partial o_j}$$

$$= -\alpha o_i \cdot f'(a_j) \cdot \sum_{m=k}^{l} \Lambda_m w_{mj}$$

(EQ 18)

According to this change in weights the Backpropagation algorithm works as follows:

1.Initialize the weights w_{ji} with random values.
2.Repeat until w_{ji} and w_{kj} have settled:
  2.1.Pick pattern from the training set (input and target vector).
  2.2.Propagate the input vector through the network to get an output vector.
  2.3.Calculate an error and update the weights according to

$$w_{ji}(t+1) = w_{ji}(t) + \Delta w_{ji}(t)$$

where Δw_{ji} is given by the above equations depending on the node and t denotes the number of the iteration.

6. Networks for split-off recognition

This chapter presents the first resulting ANN for split-off recognition.The training data set used were MC generated $3\pi_0$ data. For performance checks data were taken from a $2\pi_0\eta$ MC and from another $3\pi_0$ MC (different from the training set). All networks have just one output node with the sigmoidal node function given in section 4.1.2. The output of the networks is always in the range between 0 and 1, being a measure for the probability of a PED to be a real photon. In the training set 0 was assigned to a cluster containing at least one split-off and 1 to a cluster containing real PEDs only.

6.1 First ANN for clusters with two PEDs

The network shown in Fig. 8 has been designed to check wether there is a split-off classifier that is as good as the one found by SMART when using the same properties. Its input x is just a simple function of the invariant showermass and the energy of a cluster.

Input for ANN: $$x = 3\left(\frac{S_m}{E_{Cl}} - 0{,}16\right) \quad \text{(EQ 19)}$$

The reason for this choice is to scale the input data to small values around zero to make use of the bigger gradient of the sigmoidal node function around zero.This causes larger weight changes at the beginning and usually faster convergence.

FIGURE 8. First network for 2 PED cluster and its input data

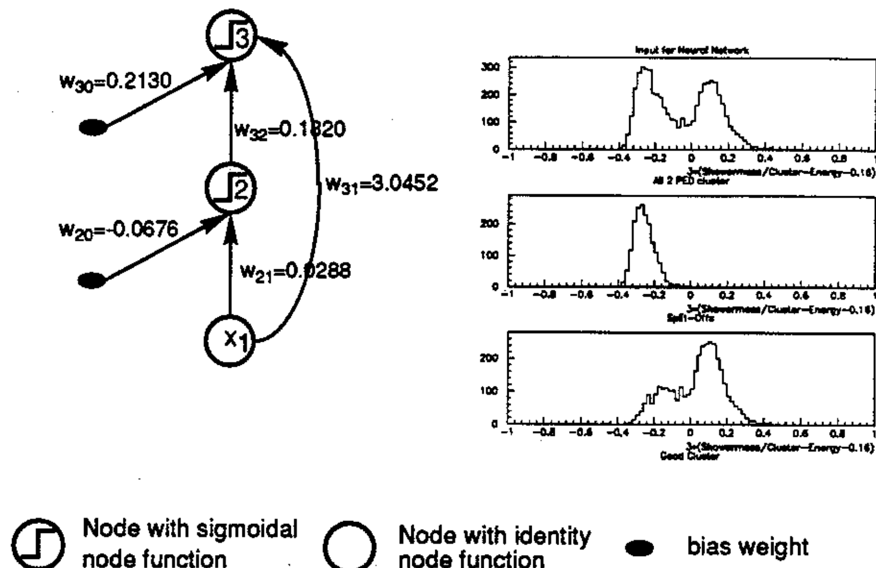
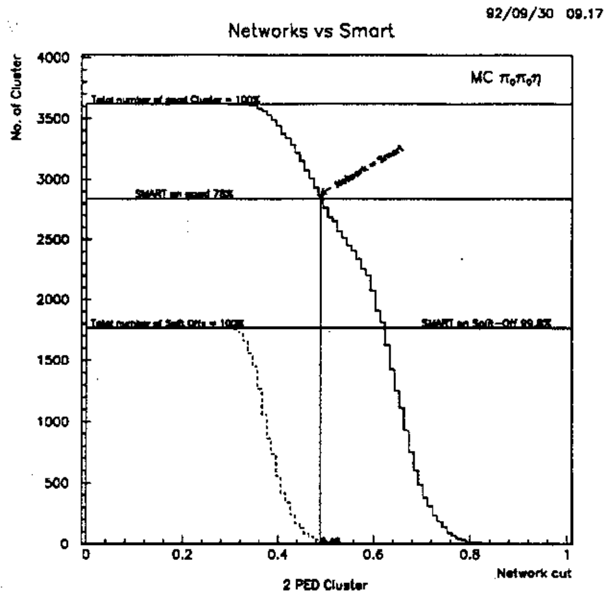


Fig. 8 shows the layout of the network with its weights which turned out when the network fulfilled the convergence criterion.This required to pass 90 patterns (input and target vectors from a training set that consists of 2000 'real' and 2000 split-off cluster) with an error bound of 0.49 (see section 4.1.7 on page 9).This error bound seems to be large, but remember that there are only two different target values (0 and 1) in the training set.Of course an error bound of more or equal 0.5 would have been a stupid choice because it is obvious that the network would have converged to some weights such that the output value is always around 0.5 and no class separation would have been gained at all! The training was stopped every 10 iterations (i.e. after 10 training examples) to test for convergence. It fulfilled the convergence criterion very quickly after 950 iterations.The training used less than a second of CPU time on an IBM RS/6000-320-h workstation.After training the network the ANN simulation was recompiled with fixed weights. Then the resulting ANN C-function was put into the CB- off-line software to check its performance on a reference data set.

FIGURE 9. Performance of first network for 2 PED Cluster



The split-off 'probability' of a cluster is found in a single procedure call. The result is shown in Fig. 9. The diagram may be interpreted in the following way: The x-axis gives the cut which was applied to the value of the output node i.e. the value which is assumed to be the border between clusters containing a split-off and clusters without split-off PEDs. If the networks output is smaller than this value the cluster is supposed to be a cluster containing a split-off otherwise a cluster without split-off's. On the y-axis is the number of clusters that survive this cut, i.e. all clusters with a smaller value were rejected. There are two curves for clusters with split-offs (lower) and without split-offs (upper) respectively. The upper curve represents the efficiency of the cut, the lower curve the background that remains in the data. For example at x=0.4 about 5% of good clusters are cut and about 15% split-offs remain. If the cut value is 0 or less it is clear that no cluster is rejected at all, because the networks output is always greater than 0. If the cut value is bigger than 0.3 the network starts to reject clusters containing split-off PEDs. Further increase leads to less split-offs in the remaining dataset but unfortunately the network starts to reject good clusters as well.At 0.47 the network has about the same performance like the upper SMART-cut (vertical line), nearly all split-offs are recognized but 32% of the good clusters are identified wrongly as a split-off. Fig. 9 shows that using an ANN for cutting must not result in a fixed cut that a user applies. The user can use it like a smooth regulator instead. If absolute purity is required then switch to 0.6 with the drawback of loosing half of the good clusters in this case. This principle is integrated in SMART as well by using two cuts an upper and a lower one. Every point below the lower SMART cut is definitely a split-off.Points between the two cuts are referred to be ambiguous. The intention for this is to 'repair' clusters which definitely contain a split-off.The use of an ANN is even more versatile! The purity of the accepted event sample can be adjusted by only <u>one</u> cut to the value of the output node and the user is free to optimize this value for his special needs!

6.2 Second ANN for clusters with two PEDs

In order to improve the network's performance more parameters (input nodes) and 7 hidden nodes were added:

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} 3\,(\frac{S_m}{E_{Cl}} - 0{,}16) \\ (\frac{2N_{xtal}}{35} - 1) \\ 2\left(\frac{E_{PED1}}{E_{Cl}} - 0{,}5\right) \\ 4\,(\alpha_{PED1} - 0{,}93) \\ 2\left(\frac{E_{PED2}}{E_{Cl}} - 0{,}2\right) \\ 4\,(\alpha_{PED2} - 0{,}94) \end{pmatrix}$$

With the following properties:

E_{PED1} : PED_1-central energy

E_{PED2} : PED_2-central energy

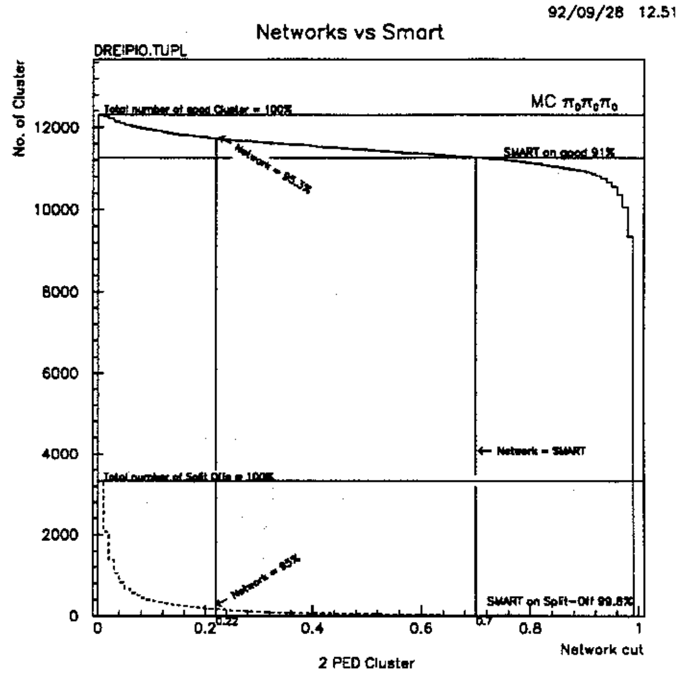
α_{PED1} : angle to next PED of PED1

α_{PED2} : angle to next PED of PED2

N_{xtal} : Number of crystals in the cluster

This time the network was able to pass a stricter convergence criterion, that was to pass 140 patterns with an error bound of 0.49 (see section 4.1.7 on page 9). And in fact the performance is somewhat better than for the network with only one input.Fig. 10 shows the result in a diagram of the same type as Fig. 9 for $3\pi_0$ MC data. For a cut value of 0.7 the network has about the same performance like the upper SMART-cut. But a cut value of 0.22 is a better choice because nearly as many clusters containing split-off PEDs are rejected but a lot more of good clusters are kept.

FIGURE 10. Performance of second network for 2 PED cluster



7. Summary and Outlook

The results obtained with the networks described above look very promising.It is possible to apply the ANN technique to the split-off problem. The performance, i.e. the quality of the classifier, turned out to be at least as good as obtained with other methods. The major advantages up to now are the greater flexibility and ease for the user when applying a cut.There has been no check on systematic errors yet, but for the first network the cut is basically the same as the SMART-cut. Therefore the systematic errors are the same as well. Once a good working network structure is found it is easy to suit the network to slightly different situations,e.g. when using 'in flight' data instead of data 'at rest' one just has to load other weights generated with a 'in flight' data training set.

At the moment networks are under investigation that work on the basis of a matrix of energy deposits in a cluster. Such a network represents a really model-free estimator, as no correlation (= knowledge) is put into the input layer. The same network might work for other purposes as well, e.g. hadronic split-offs, identifying merged π_0, etc.
For hadronic split-offs the major problem will be that there does not exist a satisfying Monte Carlo simulation for hadronic showers.A possibility might be to use real data for training and a suitable measure of quality for the output of the network (section 4.1.8 on page 9).
A further advantage would be the possibility of down-loading the weight matrix into an ANN hardware trigger box for on-line applications.

Appendix A. The Aspirin/Migraines software tool

Although it is quite easy to develop the routines for a simple ANN simulation, a network compiler named Aspirin was used to create the above networks. Aspirin is an ANN parser that uses a declarative language to describe feed forward neural networks to a certain complexity.It generates C-code that can be included into an application[1] or into a already made computer program that simulates that network. The main reason for using such a parser is that many different network structures can be generated and tested in a short time without struggling with complicated pointer structures in C, because the actual code of the network can be treated as a black box. Migraines is a tool to move through such a network interactively,i.e. one can have a look at the weights at any desired time (iteration) and dump them to a file or any other feature of the network. Aspirin is available free of charge via anonymous ftp:

ftp.cognet.ucla.edu
in the directory:
alexis
in the file:
am6.tar.Z.

Russell Leighton
The MITRE Corporation
7525 Colshire Dr.
McLean, Va. 22102-3481
INTERNET: russ@dash.mitre.org
leighton@mitre.org

The following list gives some hints how to create, train and test a network like in section 6. with Aspirin, for a detailed description of Aspirin/Migraines see the User's manual.

1. **Network description**

   Create a network description file called <MyNet>.aspirin. For the network in section 6.2 this looks like:

```
DefineBlackBox splitoffnet
 {
  OutputLayer-> splitOff
  InputSize-> 6
  Components->
   {
     PdpNode splitOff
      {
       InputsFrom-> hidden and $INPUTS
      }
     PdpNode hidden [8]
      {
       InputsFrom-> $INPUTS
      }
   }
 }
```

All words in bold type are reserved by the Aspirin language.

PdpNode: Node with sigmoidal activation function

There are many options, for example:
A black box can have arbitrarily many layers (Components). A layer can be two dimensional. Three different node functions (PdpNode = default) and a user defined

1. for example the CB off-line software

node are supported.The example network is fully connected, but Aspirin supports tessellation as well, i.e. a node can receive input from a rectangular area of a previous layer.

2. **The bpmake utility**

   To create a complete ANN simulation execute the shell script bpmake with no arguments. Then the Aspirin parser generates C code, which is compiled and linked to a standard simulation and the Migraines interface. The name of the executable simulation will be <MyNet> see step 1.)

3. **Data file and data format file**

   Create a data format file <name>.df and at least one file that contains the actual data. The data format file will tell the simulation what kind of data is used (e.g. ASCII, binary...), the order of input and target pattern and the file where to find them. Furthermore you can cause the data to be rescaled, normalized or a filter function to be applied to etc....When starting the simulation the whole data file(s) is read into memory.

   Example for a data format file:

```
ReadFile(splitoff.data Ascii)
ReadFile(good.data Ascii)
```

   This means that there are two data files both containing data in ASCII format in the order: input target input target...
   When the simulation starts a pattern (input and target) is taken from the first file then from the second and so on.

4. **Training**

   Train the network by typing:

```
<MyNet> -l -s <num1> -t <num2 num3 num4> -d <name.df>-a <α> -I <num5>
```

   This executes <Mynet> with the following options:
   -l : learn
   -s <num1> : save the weights every <num1> iterations (to a file Network.save)
   -t <num2 num3 num4> : test for convergence every <num2> iterations with the criterion: must pass <num3> iterations with <num4> error bound
   -d : data format file <name>.df
   -a : with learning rate <a>
   -I : truncate learning after <num5> iterations

   When the networks fulfills the convergence criterion the simulation terminates and the state of the network is dumped to a file Network.Finished

5. **Performance check**

   One possibility is to create another ('unknown') data file and the corresponding data format file and do the test by typing:

```
<MyNet> <name.Finished> -d <name.df> -f <num1> -p
```

   This executes <Mynet> with the following options:
   <name.Finished> : load weights from a dump file into the network
   -d : data format file <name.df>
   -f <num1> : (forward) take <num1> input and output pattern from the data file and propagate them just forward through the network (no learning!).
   -p : print the output and target values to the standard output

   Adding '> filename' will redirect the output of the simulation to a file

Another possibility is to link the network to the program that generates the data[1] without the Migraines interface and with fixed[2] weights.For this purpose add the keyword **Static->** in the network description file <name>.aspirin.Then compile the network step by step:

First type

```
aspirin <name>.aspirin <filename> -c backprop
```

and then

```
cc -fsingle -I<include.dir> -g <-qsource> -c <filename.c>
```

The first command generates a file <filename>.c and a file <filename>.h which contains C-code.Then the <filename.c> file is given to the C-compiler to produce executable code in a file <filename>.o that will be linked to the application. When the <filename>.h file is included in the source code then a number of network- and black box[3] control functions are available to initialize the network, to set and get the input and output, to propagate data through it etc...

1. In this case the CB off-line software.
One could have done that for training as well without adding Static->
2. i.e. once the weights are loaded into the simulation they are fixed
3. A network can consist of several black boxes (e.g. a network of networks)

[1] J. Salk, split-off recognition in pure neutral events, CB note 182,Bochum 1991

[2] B. Denby, Tutorial on Neural Network applications in high energy physics, Fermi National Accelerator Laboratory 1992

[3] C. Peterson, T. Rögnvaldsson, An Introduction to Artificial Neural Networks, Proceedings, 1991 CERN School of Computing, CERN May 1992