A Greedy Learning Approach for Multi-Layer Perceptrons

Achim G. Hoffmann

School of Computer Science & Engineering

University of New South Wales

Sydney, 2052 NSW, Australia

E-mail: achim@cse.unsw.edu.au

Abstract

Neural networks as a general mechanism for learning and adaptation became increasingly popular in recent years. Mainly due to the development of the backpropagation learning procedure which allowed to train Multi-Layer Perceptrons. Unfortunately, back propagation is well-known for its particularly low learning speed. Thus, it is desirable to have more efficient learning mechanisms. In this paper a new learning algorithm for multi-layer Perceptrons is presented. It will be argued that by using a four-layer Perceptron instead of a three-layer Perceptron substantial improvements in the learning speed can be obtained.

1 Introduction

In the past few years there was a strongly increasing interest in neural network-like models of computation ([5]). One of the most important algorithms in today's neural networks is the back propagation algorithm for multi-layer Perceptrons. Unfortunately, back propagation is very slow with respect to its convergence toward an unknown target classification function. Therefore, faster learning algorithms are very desirable in order to speed-up the design and development of systems which contain multi-layer Perceptrons as subsystems. This paper presents a new *fast* learning algorithm for multi-layer Perceptrons which is based on the classical Perceptron algorithm for learning a single linear threshold function.

The organization of the paper is as follows. In section 2 the concept of the Perceptron-based networks is recalled. Section 3 presents the new training algorithm. Section 4 contains the conclusions and future work.

2 Pattern classification by neural networks

The most frequently used kind of pattern description is the description by a set of features. Each pattern is described by its respective values for each of the chosen features. E.g. cars can be described by features as color, weight, height, length, width, maximal speed, etc.

Formally speaking, an ordered set of features, or a vector of features, is considered. For each feature x_i a prespecified range r_i of values is admissible. This defines the complete set of possible representations of patterns. More precisely: Given is a feature vector $X = \langle x_1, \ldots, x_n \rangle$ and a range $r_i \subseteq \Re$ for each feature $x_1, \ldots, x_n$.

Thus, the set P of possible patterns is given by the Cartesian product of the ranges of all features. I.e. $P = r_1 \times \ldots \times r_n \subseteq \Re^n$ P is also called the *feature space*. The patterns can also be regarded as points in the n-dimensional Euclidean space.

2.1 Perceptrons

The Perceptron is a simple computing unit (also called a neuron) which computes a linear threshold function of an vector $\mathbf{X}$. A linear threshold function f is given as follows:

Input: Training examples $\{(X^1,C^1),\ldots,(X^n,C^n)\}$ where X^k is a vector with $x_0^k = 0$ and other components, $x_1^k,\ldots,x_n^k$ assuming values in $\{-1,0,+1\}$. $C^k \in \{-1,+1\}$ is the desired response.
Output: $W = \langle w_0,w_1,\ldots,w_n\rangle$ is a vector of pocket weights where w_0 is the threshold.
Temporary data: π is a vector of Perceptron weights, $\langle\pi_0,\ldots,\pi_n\rangle$.
run_π is the number of consecutive correct classifications using Perceptron weights π.
run_W is the number of consecutive correct classifications using pocket weights W.

```
      begin
(1)     set π = ⟨0,...,0⟩, run_pi=run_W=num_okπ=num_okW=0
(2)     Choose a training example (X^k, C^k)
(3)     if π correctly classifies (X^k, C^k)
          then
(3a)      run_π=run_π+1
          if run_π > run_W+1
              then
(3aa)         W := π, run_W :=run_π
          endif
          goto step 2
        else
(3b)      then  W := W + X^k C^k
          goto step 2
        endif
      end.
```

Figure 1: The pocket Perceptron learning algorithm: An extended version of the classical Perceptron learning algorithm.

$$f(\mathbf{X}) = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} w_i x_i < \theta \\ -1 & \textit{otherwise.} \end{cases}$$

I.e. if the weighted sum of the feature values exceeds a certain threshold θ, the function value switches from 0 to 1. Their geometrical interpretation is in the case of a two-dimensional feature space a separating line dividing the plane into two classes. In the n-dimensional feature space the separating linear decision function can be thought of a hyperplane dividing the feature space into two separate areas.

To the end of the 1950s, Rosenblatt [4] developed his so-called *Perceptron* algorithm, which computes an appropriate linear threshold function for a given set of patterns. The algorithm (an actually extended version of the original Perceptron algorithm is given in Figure 1) successively adjusts the weights of a candidate weight vector in order to fit the presented training patterns. If there exists a linear threshold function classifying all presented patterns correctly, then the candidate weight vector will eventually do so.

Unfortunately, not always is the distribution of patterns in the feature space such that a linear threshold function exists. In simple cases, this is due to noise in the feature values. In other cases, it is due to the fact that a linear threshold function is simply not adequate for separating the respective pattern classes. In such cases, more appropriate and in most cases more complex separation functions have to be employed.

Linear threshold functions are quite limited in their ability to divide a feature space into two different areas. For many classes of sets of patterns an appropriate separation cannot be achieved by a linear threshold function. Even reasonable approximations may not exist as linear threshold functions.

Minsky and Papert [3] studied in their very influential book *Perceptrons* the limitations of linear threshold functions in depth. They proved severe limitations of linear threshold functions, e.g. the inability to compute the XOR function.

2.2 Multi-Layer Perceptron

However, one way to overcome the unability of the standard Perceptron to compute the XOR function is to use multiple Perceptrons. Each Perceptron then discriminates a subset of the patterns. The various subsets have to be merged together in order to allow a proper discrimination of the entire pattern set (Figure 2).

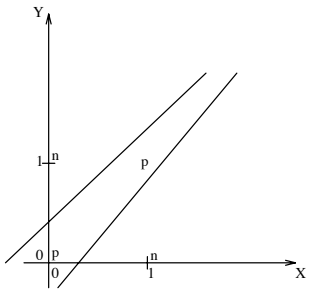

Figure 2: Merging the area above the upper line with the area below the lower line can be done by a third Perceptron which gets as input the output of the two Perceptrons which implement the solid lines.

Unfortunately, the training of a combination of Perceptrons appears much harder than the training of a single Perceptron, since it is not clear how to distribute the feedback from each example over the various neurons.

2.3 Backpropagation

Nevertheless, in 1986 Rumelhart, McClelland & Williams, see [5], introduced in the context of multi-layer Perceptrons (MLPs) the *generalized delta rule*, which became famous as the backpropagation algorithm.[1] It is a general learning rule, which allows to find even in an MLP an appropriate set of threshold values and input weights for each neuron.

Similarly to the ordinary Perceptron learning algorithm, the operation of the backpropagation in MLPs is also based on presented patterns and iteratively adjusting the weights and thresholds of each neuron depending on whether the produced output is as desired or not. The backpropagation became possible by the introduction of the sigmoidal output function for each Perceptron in the MLP. The sigmoidal function allows to backpropagate an occurring error in the output signal to the units of earlier layers in the network. This amounts to a decreasing error in the output signal for the next time.

The generalized delta rule compares the actual value of the output signal with the desired value and evaluates the difference in order to modify the weights and threshold of the output unit such that the difference will be reduced the next time. Further, this difference is also used in order to be *back-propagated* to the previous units which contributed to the deviation of the output signal - hence the name *backpropagation rule*.

The backpropagation algorithm may get stuck in some local optimum. Although, this seems to be not too bad in practice, backpropagation exhibits yet another problem: *Its learning speed is particularly slow.*

3 Speeding-up learning by using an additional layer

In this section a new algorithm for training a multi-layer Perceptron is presented. It will be shown that the new approach requires a fourth layer in order to be capable of learning arbitrarily complex functions in a greedy manner. The Perceptron pocket algorithm [1] and some variations of it are used for the independent training of each single neuron. This algorithm is always advisable for training a single Perceptron if no linear threshold function can be found which classifies all training examples correctly.

The basic idea is to train one neuron at a time starting with the neurons of the first hidden layer. After the first hidden layer has completely been trained the second layer is trained. Finally the single output neuron on the fourth layer is trained. See Figure 3 for an example of a four-layer Perceptron. It contains two hidden layers. In section 3.2 it is shown that this approach is capable to learn arbitrary functions if the two hidden layers contain a sufficient number of neurons. This compares to the proved result that the three layer Perceptron is capable to generate any particular (even continuous) function if the weights and thresholds are properly adjusted and sufficiently many neurons are on the hidden layer (see e.g. [2]).

[1] Actually, Werbos [6] had developed the generalized delta rule already in 1974 without putting it into the context of MLPs.

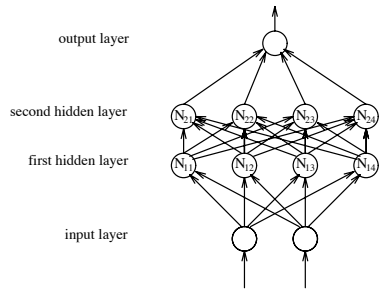

Figure 3: An example for a four layer Perceptron. It has two hidden layers of neurons. Each neuron computes a linear threshold function.

Remark: Let $\nu \in [0..1]$ be a user-defined parameter which allows that the resulting four layer Perceptron may only classify a fraction ν of the entire set of training examples correctly.

procedure *add_neuron_to_first_hidden_layer(E)*; where E is a set of training examples.
begin
(1) initialization of variables: $E_t = E$.
(2) **call** *perceptron_pocket*(E_t).
(3) assign the resulting weight vector W to a new neuron N and add N to the first hidden layer.
(4) Let X^+ (X^-) be the set of positive (negative) examples for which N outputs 1.
(5) **if** $(\frac{\max(|X^+|,|X^-|)}{|X^+|+|X^-|} < \nu)$
(5a) **then call** *add_neuron_to_first_hidden_layer*($X^+ \cup X^-$).
endif
(6) Let X^+ (X^-) be the set of positive (negative) examples for which N outputs 0.
(7) **if** $(\frac{\max(|X^+|,|X^-|)}{|X^+|+|X^-|} < \nu)$
(7a) **then call** *add_neuron_to_first_hidden_layer*($X^+ \cup X^-$).
endif
end.

Figure 4: Training the first hidden layer of Perceptrons.

3.1 The greedy Perceptron algorithm

3.1.1 Training the first hidden layer

After training the first neuron N_1 with the Perceptron pocket algorithm the sample space is divided by a hyperplane H_1. As long as each of the emerging subspaces contain both positive and negative examples another neuron N_i on the first hidden layer is required in order to impose another division H_i of the respective subspace and simultaneously on the entire feature space. I.e. N_i is trained again by the Perceptron pocket algorithm with only the examples in the respective subspace as training sample. The algorithm for training the first hidden layer is shown in Figure 4 on a rather abstract level.

3.1.2 Training the second hidden layer

After the training of all neurons of the first hidden layer has been completed, the training of the second hidden layer can begin: The training happens again by the Perceptron pocket algorithm. This time it is required for a training vector to be stored in the pocket that it minimizes the number of misclassified examples at least on either side of the respective hyperplane. I.e. if objects are only misclassified on one side of the hyperplane the multi-layer Perceptron has to train another neuron which is responsible for the proper classification of those examples. See Figure 7. This is easily achieved by modifying the original Perceptron pocket algorithm slightly as follows, which we call the *one_sided_perceptron_pocket_algorithm*:

The training vector is only modified if the training vector misclassifies an actually positive example as negative. The resulting vector from a run of this algorithm is attempted to be improved by running another version of the pocket algorithm, the *purity_ratchet_perceptron_pocket_algorithm*:

procedure *build_second_hidden_layer(E)*; where E is a set of training examples.
begin
(1) initialize $E_t = E$.
do
(2) **call** *one_sided_perceptron_pocket*(E).
% See section 3.1.2
(3) **call** *purity_ratchet_perceptron_pocket*(E).
% with the pocket vector initialized to the output vector of step 2.
(4) assign the resulting weight vector W to a new neuron N and add N to the second hidden layer.
(5) Remove all positive examples from E_t which are correctly classified by the output signal of N.
(6) **until** $E_t = \emptyset$
end.

Figure 5: Training the second hidden layer of Perceptrons.

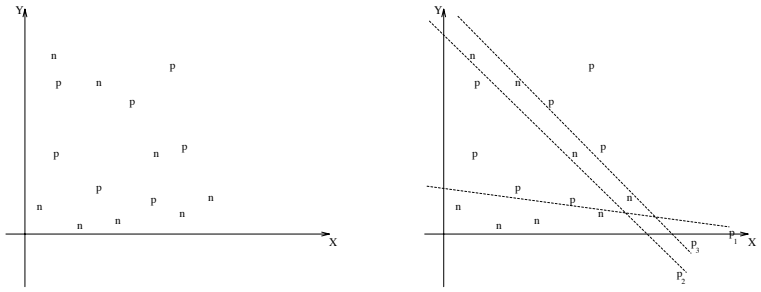

Figure 6: Dividing the feature space by hyperplanes until any pair of differently classified examples is divided by at least one hyperplane. These hyperplanes are implemented by the first layer of hidden neurons.

The pocket algorithm from Figure 1 is modified such that the pocket vector is only replaced by the current training vector if the training vector classifies more positive examples correctly **and** *if the fraction of negative examples among all positively classified examples is less than* $1 - \nu$.

How the second hidden layer of neurons is trained using the above mentioned modifications of the pocket Perceptron algorithm is shown in Figure 5.

3.1.3 Training the output layer Perceptron

The output layer Perceptron is simply trained by the standard Perceptron algorithm providing all examples to the input of the network. Thereby, the output Perceptron gets only the transformed input signals through hidden layer one and two as input.

3.2 The completeness of the greedy Perceptron algorithm

In the following it is proved that the presented algorithm always results in a four-layer Perceptron which classifies the entire training set correctly, if ν is set to 1. I.e. the following theorem can be proved:

Theorem 1 *Let X be an arbitrary finite and consistent set of training examples, where $X \subseteq P$ for some feature space P. Let $\nu = 1$. Then the greedy Perceptron algorithm presented in subsection 3.1 applied to X will finally find a four-layer Perceptron which classifies the entire set X correctly.*

Proof: After the completion of the training of the first hidden layer there is no pair of differently classified (one negative and one positive) examples in the sample which is not divided by at least one of the linear decision functions implemented by the neurons of the first hidden layer. See Figure 6 (right side) for an example.

As a consequence, a correct classification function of the complete set of examples can be obtained by a boolean function of the outputs of the first hidden layer. I.e. each singleton area (area which is not divided

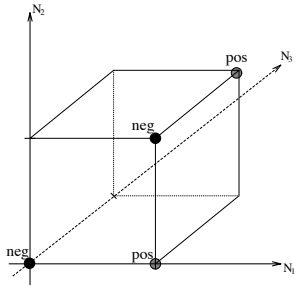

Figure 7: The feature space provided by the output values of three neurons on the first hidden layer of the previous figure is shown. All positive (negative) examples in the previous figure are mapped onto the two respectively indicated points in the three dimensional space above through the processing of the first hidden layer. A plane can be implemented by the second hidden layer which separates the two positive points from the two negative points in the figure. (In this case a three-layer Perceptron would actually suffice.)

by any hyperplane imposed by a neuron of the first hidden layer) can be 'selected' by demanding a suitable output value for some or all of the neurons on the first hidden layer. By using the algorithm described the neurons on the second layer are at least trained to perform this function or possibly cover a larger number of examples correctly. The output neuron then implements basically a logical 'or' function among the output values of the neurons of the second hidden layer. Each of the second hidden layer neurons reponds to one area of positive examples in the feature space.

□

4 Conclusions and future work

The presented algorithm is a fairly straightforward way to construct a feedforward neural network which classifies the training examples either 100% correctly or classifies at least a prespecified fraction ν of the training examples correctly. Some important open questions are the following:

How is the algorithm's performance if the number of nodes per hidden layer is restricted ?

How is its performance if a severe limit in the number of iterations of the simple Perceptron pocket algorithm is imposed ? (The algorithm will still find a correct classification but possibly with a significantly greater number of neurons in the hidden layers.)

How can the strategy be heuristically modified if the number of neurons per layer is restricted ?

References

[1] S. I. Gallant. Perceptron-based learning algorithms. *IEEE Transactions on Neural Networks*, 1(2):179–191, June 1990.

[2] R. Hecht-Nielsen. Kolmogorov's mapping neural network existence theorem. In *Proceedings of IEEE first International Joint Conference of Neural Networks*, volume III, pages 11–14, 1987.

[3] M. Minsky and S. Papert. *Perceptrons*. MIT Press, Cambridge, MA, 1969.

[4] F. Rosenblatt. Two theorems of statistical separability in the perceptron. In *Proceedings of the Symposium on the Mechanization of thought*, pages 421–456, London, 1959. Her Majesty's Stationary Office.

[5] D. E. Rumelhart, J. L. McClelland, and the PDP Research Group. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition,I & II*. MIT Press, Cambridge, MA, 1986.

[6] P. J. Werbos. *Beyond regression: New tools for prediction and analysis in the behavioral sciences*. PhD thesis, 1974.