Acta Scientific COMPUTER SCIENCES

Volume 4 Issue 12 December 2022

Research Article

A Multi-hyperplane Separation Method to Train Shallow Classifiers

Ákos Hajnal[1,2]*

[1]*Institute for Computer Science and Control (SZTAKI), Eötvös Loránd Research Network (ELKH), Laboratory of Parallel and Distributed Systems, Hungary*

[2]*Óbuda University, John von Neumann Faculty of Informatics, Hungary*

***Corresponding Author:** Ákos Hajnal, Institute for Computer Science and Control (SZTAKI), Eötvös Loránd Research Network (ELKH), Laboratory of Parallel and Distributed Systems, Hungary.

Received: September 23, 2022

Published: November 08, 2022

Abstract

This paper presents a novel approach to train classifiers implemented as a shallow neural network. The proposed solution is based on the original Perceptron algorithm but extends to the multi-hyperplane case. Consequently, it allows of solving not only linearly separable problems. Besides simplicity, the advantage of the method is its tolerance to imbalanced data, which can occur in practice. The applicability of the method has been demonstrated on several artificial and on real-life datasets.

Keywords: Machine Learning; Artificial Neural Networks; Shallow Neural Network; Classification; Perceptron; Multi-Hyperplane Separation; Mistake-Driven Algorithm

Abbreviations

AI: Artificial Intelligence; ANN: Artificial Neural Network; CNN: Convolutional Neural Network; ML: Machine Learning; SGD: Stochastic Gradient Descent

Introduction

Use of Artificial Neural Networks (ANN) is a common means to solve various tasks in Machine Learning (ML). Variants of the (Stochastic) Gradient Descent (SGD)-based learning algorithms have already proved their applicability and efficiency in various fields since the Backpropagation [1,2] that can also be used to train deep neural networks and Convolutional Neural Networks (CNNs).

Although the results achieved are beyond question, AI experts often face inherent problems when designing and training neural networks. Beyond technical issues (finding the proper hyperparameters, data normalization, data balancing, vanishing/exploding gradients, local optimums, regularization, determining loss functions), the main concern with such “black-box” models produced by general optimization techniques is the lack of interpretability, trust, and uncertainty in robustness (e.g., [3]).

Another main approach to training neural networks is based on a geometrical perspective. The Perceptron [4] and Support Vector Machines (SVM) [5] are prominent examples for such a concept. Neurons can be considered and treated as hyperplanes, which need to be “properly” aligned, crossing between points representing data in a multi-dimensional space.

Classification in supervised learning categorizes a set of data into classes. Each data in the training set is explicitly labelled with a discrete value corresponding to a class (or category) that it belongs to. Binary classification means a classification task that has only two class labels: “positive” and “negative”.

The geometrical approach to classification can be interpreted as separability problems. The Perceptron algorithm [6] is applicable to find a hyperplane separating positive and negative data points; SVM computes such a hyperplane with maximal margin. A notable advantage of these methods, besides their clear interpretation, is that they are concerned with the width of the gap and shape of separation surface (marked out by supports), and less influenced by the volumes of data points falling farther. It results in better tolerance to imbalanced data that can occur in practice.

When the data is not linearly separable, single-hyperplane methods are not applicable [7] but try to find approximate solutions [9]. SVMs use feature space mapping ("kernel trick") to avoid this problem [8].

The paper proposes a novel heuristic to separate classes of data with several hyperplanes, by training neurons in a shallow neural network. It can be considered as an extension of the original Perceptron algorithm to the multi-neuron case, but keeping its simplicity, tolerance to imbalanced data, also the potential for application in parallel training (e.g., in "extreme classification" [10]). We provide empirical validation for convergence, however, without a formal proof or guarantee for finite-step termination.

The paper is structured as follows. In Section 5, we introduce the architecture of a two-layer neural network and then the method how to train weights and biases of the neurons. In Section 6, we present empirical results of the proposed method on examples and on a larger dataset. Finally, Section 7 concludes the paper.

Materials and Methods

In this section, we first define the neural network architecture used for binary classification, enumerating specialties (activation function, fixed output neuron concept), and providing a geometrical interpretation. Then, we present the training method applicable to determine hidden layer neuron weights and biases.

Shallow neural networks with a single output neuron

Neurons in artificial neural networks perform a simple computation: $z = \underline{w}\cdot\underline{x}+b$ (where $\underline{w}$ is the weights vector, $\underline{x}$ is the input, · is the dot product operator, and b is the bias of the neuron), which value is transformed further by a (nonlinear) activation function: a(z) (e.g., ReLU, Sigmoid, Tanh, Signum, Step) to produce the final the output of the neuron.

Shallow neural networks consist of three layers: the input layer that represents the different fields of a single input (called features) but without any neuron in it; the hidden layer, contains a number of, let us name it: n, neurons in it; and finally the output layer contains a single neuron (in case of binary classification) that produces the result (prediction) of the entire neural network. All input features are connected to all hidden layer neurons, and all hidden layer neuron outputs are connected to the output neuron, respectively (fully connected/dense, feedforward network).

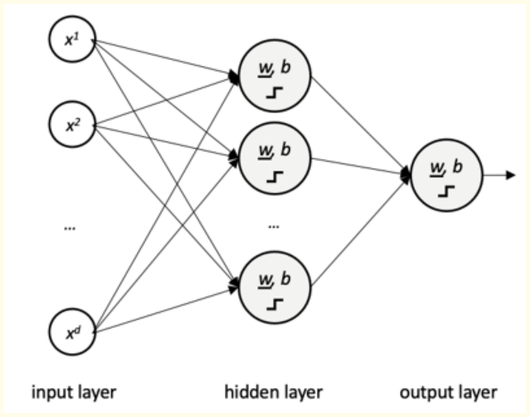

Figure 1: Shallow neural network with a single output neuron.

From a geometrical point of view, a neuron directly corresponds to a hyperplane H (defined by equation: $\underline{w}\cdot\underline{x}+b=0$) in a multi-dimensional space, where the dimension is defined by the size of the input ($\underline{x}$). For this reason, we shall use terms "neurons" and "hyperplanes" interchangeably in the rest of the paper. Value z computed by the neuron for input $\underline{x}$ is a value proportional to the signed distance of point $\underline{x}$ from H, that is, the sign of z indicates that in which half-space the given point lies wrt. H.

We will use the Step function as activation function (a(z)=1, if z>0, 0 otherwise) in every neuron throughout this paper. Considering the hidden layer, the hidden layer therefore represents a mapping of every input data $\underline{x}$ to a bit-vector of size n. The ith bit is 1 if $\underline{x}$ lies on the positive side of H_i corresponding to the ith neuron (defined by normal vector $\underline{w}$), or 0, otherwise. This bit-vector can be considered as the coordinates of a vertex of an n-dimensional unit hypercube (n-cube). From this perspective, the hidden layer corresponds to a mapping from input space points to hypercube vertices. Such a mapping is illustrated in figure 2.a, assuming three hidden layer neurons (n=3). White/black colored nodes illustrate vertices onto which (one or more) positive/negative inputs had been mapped, respectively; gray ones represent vertices with no data mapped to it at all (undefined, don't care).

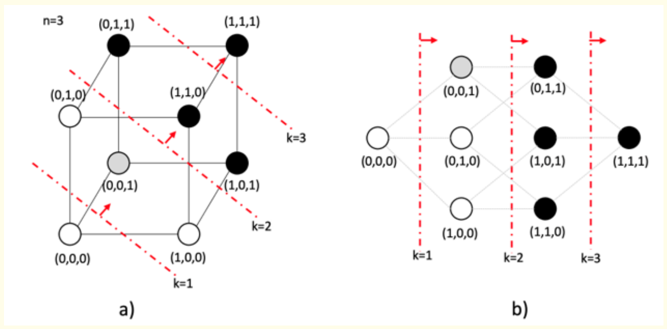

Figure 2: a) n-cube with positive (white)/negative (black)/ undefined (gray) vertices. b) n-cube with horizontally aligned main diagonal.

If there exist a number of n hyperplanes such that, with a proper alignment, they can realize a mapping to positive and negative vertices of the n-cube that are linearly separable, the neural network can accurately classify the series of data. The goal of the training process is thus to determine such hyperplanes (consequently, neuron weights and biases).

Unfortunately, in general, it is not known how many hidden neurons we will need at minimum (optimum) to solve a specific problem; therefore, n remains a hyperparameter.

Furthermore, as training hidden layer neurons and the output neuron simultaneously is a difficult problem, we fix the hyperplane corresponding to the output neuron to a specific alignment, and use another hyperparameter, referred to as: k ($0< k \leq n, k \in \mathbb{Z}$), that determines its offset from the origin. We set all the weights of the output neuron to constant 1 and the bias value to: -k + 0.5. Geometrically, such a hyperplane is perpendicular to the main diagonal, and to see where it slices the hypercube, let us consider figure 2.b. It shows the same hypercube in figure 2.a but with rotated main diagonal (horizontal).

If k=1, all vertices except the origin are on the positive side of the hyperplane of the output neuron; if k=2, the origin and vertices corresponding to coordinates: (1,0,0), (0,1,0), (0,0,1) get to the negative half-space, and so on. Considering the individual coordinates as "votes" (0: "no", 1: "yes"), k defines a "voting rule", where hidden layer neurons represent voters (with equal voting right). k=1 means that a single upvote is enough to judge positively (predict 1); in the case of k=n, a single downvote is enough to decide negatively (veto); and assuming odd n, k=(n+1)/2 corresponds to the majority rule. This alignment can be applied in any dimension n.

The nkPerceptron heuristic

At given hyperparameters, n and k, the heuristic presented in this section aims at determining hidden layer hyperplanes, iteratively, using a method similar to the original Perceptron algorithm. (At values n=1 and k=1, it directly reduces to it.) It follows the same mistake-driven update concept: on wrong predictions, input features (vector) are added to or subtracted from neuron weights, depending on whether it was a false negative or a false positive prediction, respectively.

Having multiple hidden layer neurons, the identification of which neuron(s) are to update is not straightforward. As an example, let us have 10 neurons in the hidden layer and voting rule: "5 votes to pass" (n=10, k=5). Assume that for a positive sample we get four upvotes and six downvotes, so the prediction will be false negative. We known that at least one of the six downvoters is corrupt, but we don't know which one(s).

We may update all the neurons voted (seemingly) incorrectly, but it has the risk that we update also ones that worked correctly. We may select only one and leave potential further updates for later iterations. We chose the latter option.

It is also non-trivial, if there is more than one incorrect vote, which voter to select. We found no straightforward answer to this question, and we decided to choose: the one that needs the least update, i.e., the hyperplane nearest to the input on the wrong side. Intuitively, this decision was motivated by "stability" reasons.

Finally, we note that that instead of computing Euclidean distances, we used simply the dot product value ($\underline{w} \cdot \underline{x} + b$). The reason was empirical: we experienced faster convergence in terms of iterations (and in in computation time).

The pseudo code of the method is shown below:

Inputs:

training_set: $[(\underline{x}_1,y_1), (\underline{x}_2,y_2), \ldots,(\underline{x}_T,y_T)]$: $\underline{x}_i \in \mathbb{R}^d$, $y_i \in \{+1,-1\}$

N: hidden layer size

K: voting threshold ($1<K\leq N$)

E: number of epochs to run

Globals:

W: matrix of hidden layer neuron weights $[\underline{w}_1, \underline{w}_2, \ldots, \underline{w}_N]$, all initialized to $\underline{0}$

$\underline{b}$: vector of hidden layer neurons biases $[b_1, b_2, \ldots, b_N]$, all initialized to 0

```
for epoch in 1 .. E
    for ($\underline{x}_i$, $y_i$) in training_set
        if there exists $\underline{w}_i$ in $\underline{W}$ s.t. $\underline{w}_i = \underline{0}$ && $b_i = 0$ then
            $\underline{w}_i := y_i * \underline{x}_i$
            $b_i := y_i$
            continue
        $\underline{v} := W\underline{x}_i + \underline{b}$
        $p$ := number of positive elements in $\underline{v}$
        if $y_i = 1$ && $p \geq k$ || $y_i = -1$ && $p < k$
            continue
        $\underline{v} := \underline{v} * y_i$
        $i$ := index of the maximum negative value in $\underline{v}$
        $\underline{w}_i := \underline{w}_i + y_i * x_i$
        $b_i := b_i + y_i$
    end for training_set
end for epoch
```

The algorithm runs for E epochs (line 1). For each input data we get its features as a vector ($\underline{x}_i$) and its label, with value: y_i=+1: if positive, y_i =-1: if negative (line 2). One may consider shuffling training set before each epoch. All weights and biases of the hidden layer neurons are initialized to 0. If there is uninitialized neuron, we assign the input data as weights and label +1/-1 as bias correspondingly, then we take the next input (lines 3–6). We calculate the predictions of the hidden layer (line 7). We count positive votes (line 8), and if the result of the voting is correct (≥k positive), we continue with next sample without any update of the network (lines 9–10). Note that we do not use activation function in the hidden layer, but implicitly infer from the sign of the dot product ($\underline{w}_i\underline{x}$+b). Also, we count positive-negative votes without using an output neuron. In case of false positive or false negative predications, we multiply hidden layer outputs by +1 or -1 correspondingly to the input label. Consequently, all "incorrect" outputs will be negative (line 11). Finally, we select the neuron that produces the largest negative value (minimal absolute valued negative) (line 12), and update its weights and bias with the input and the label (lines 13–14).

Results and Discussion

In this section, we evaluate the proposed algorithm on some simple and on some non-trivial problems. First, we present artificially created datasets, which carry specific challenges, then we try to apply the method on a real-life dataset, namely, on the MNIST dataset [11].

As mentioned earlier, the optimal values of hyperparameters of n and k for a given problem are difficult to determine in general. In the following examples, we use the minimum values we (humans) could see, compute, or guess in advance to challenge the heuristic.

The first example is the classical XOR problem, as shown in figure 3.a (top), which is not solvable by a single Perceptron. The problem can however be solved with two hidden layer neurons. The heuristic correctly finds a solution (in 19 epochs, with four training data elements) with hyperparameters: n=2 and k=1. The results are illustrated in figure 3.b (bottom), where we can see the how data points are mapped onto the two-dimensional unit square by two hyperplanes, and how they are separated by the output neuron corresponding to value k=1.

In figure 3.b we can see the "XOR cube problem". This problem might be challenging even for humans to see that it can be solved with three hidden layer neurons (there are more than one solution, with different orientations). The heuristic successfully finds a solution for this problem as well (in 38 epochs, 8 input data).

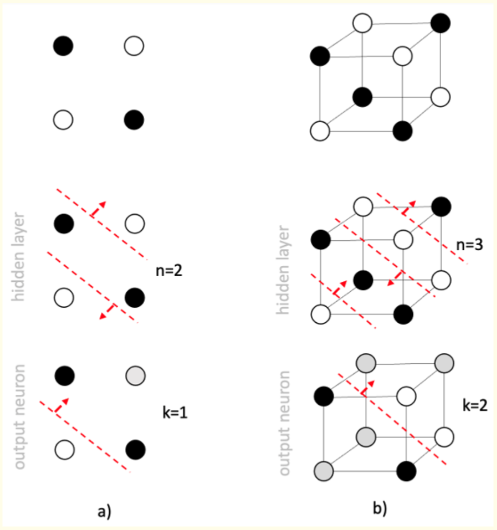

Figure 3: a) XOR problem. b) XOR cube

In the following examples, the challenge is to find hyperplanes between positive and negative point sets (represented, surrounded by white and black squares) with a narrow gap between them, as illustrated in figure 4. Boxes were 10 unit wide, the gap was 1 unit wide, and we generated 1,000 input data, with appropriate labels in the corresponding boxes. The method finds the appropriate hyperplanes in cases a) (with hyperparameters: n=2, k=2) and b) (with hyperparameters: n=4, k=4), but fails for case c).

The latter case happened because, using two hyperplanes (one vertically, one horizontally, crossing the middle point) we obtain a mapping that just results in the XOR problem (but with points instead of boxes, as in Figure 3.a), which would require another shallow network to solve, and so the whole problem cannot be solved with a single shallow network.

We also note, narrowing the gap in examples 4.a and 4.b, the algorithm requires more and more iterations to find the proper hyperplanes. The increase of iterations can also be observed at applying the Perceptron algorithm on linearly separable problems with small gaps.

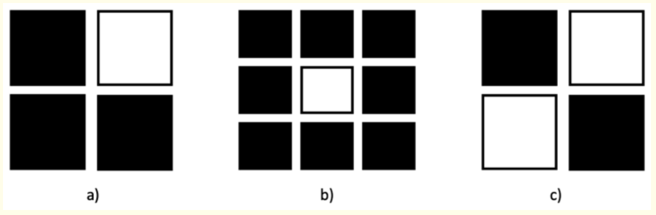

Figure 4: a) L-shape. b) white-in-the-middle. c) "evil" XOR problem (failed).

Finally, we used the MNIST dataset (containing 70 thousand input data, 28x28 pixel grayscale images of handwritten digits) to train ten individual neural networks, each trying to recognize a specific digit (e.g., is it 7 or some other digit?). We used n=30 hidden layer neurons and k=15 output neuron value. The training was successful in all the cases, reaching 100% accuracy on the training set (in about 120 epochs in the worst case). In figure 5, we show the accuracy improvement over epochs of the individual trainings, where label "Cat0" refers to the network trying to recognize digit 0, "Cat1" for digit 1, etc. In the chart we can see learning curves similar to the ones we would experience at other machine learning tasks, and they show convergence.

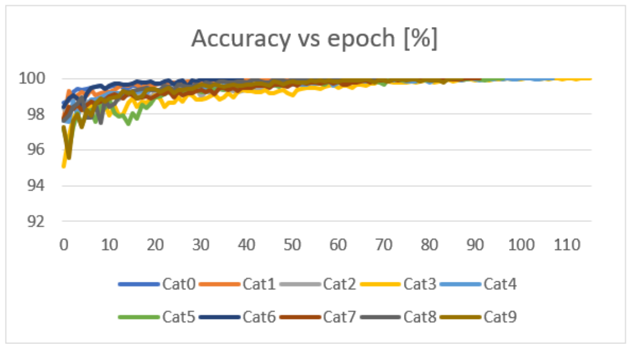

Figure 5: Accuracy [%] vs. epoch charts of individual neural networks learning specific digits.

Conclusion

This paper presented a non-SGD-based machine learning method applicable to train shallow neural networks that can be used classification problems. The proposed solution build on the original idea of the Perceptron algorithm but extends it to the multi-hyperplane separation case. The main advantage of this heuristic beyond its simplicity is that it tolerates imbalanced datasets. The effectiveness of the method has been demonstrated on several non-trivial cases, including artificial ones and a real-life dataset.

Regularization, that is, how to improve the method to perform accurately on the test data as well (not seen during training), is out of the scope in this paper, and serves as a future work.

Conflict of Interest

The author declares that there is no financial or any other conflict of interest.

Bibliography

1. Robbins H and Monro S. "A Stochastic Approximation Method". *The Annals of Mathematical Statistics* 22.3 (1951): 400-407.
2. Haskell B C. "The Method of Steepest Descent for Non-linear Minimization Problems". *Quarterly of Applied Mathematics* 2.3 (1944): 258-261.
3. Rumelhart D., *et al*. "Learning representations by back-propagating errors". *Nature* 323 (1986): 533-536.
4. Su J., *et al*. "One pixel attack for fooling deep neural networks". *IEEE Transactions on Evolutionary Computation* 23.5 (2019): 828-841.

5. Rosenblatt F. “The perceptron: A probabilistic model for information storage and organization in the brain”. *Psychology Review* 65 (1958): 386-407.

6. Cortes C and Vapnik V. “Support-vector networks”. *Machine Learning* 20.3 (1995): 273-297.

7. Rosenblatt F. “Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms”. Spartan Books, Washington, D.C., (1962).

8. Minsky ML and Papert SA. “Perceptrons”. MIT Press, Cambridge, MA, (1969).

9. Sch**ö**lkopf B and Smola A. “Learning with Kernels”. MITPress, Cambridge, MA, (2002).

10. Gallant S I. “Perceptron-based learning algorithms”. *IEEE Transactions on Neural Networks* 1.2 (1990): 179-191.

11. Varma M. “Extreme classification”. *Communications of the ACM* 62.11 (2019): 44-45.

12. LeCun Y., *et al*. “The MNIST dataset of handwritten digits”. (1999).