Training a class membership function where there are no samples

William W. Armstrong, Olaf Christ

Abstract

Some two-class classification tasks have training sets where the samples of different classes don't intermix and are a certain minimum distance from each other. In this case one can construct a membership function that slowly decreases from a maximum value on the samples of one class to a minimum value on the samples of the other class. A slower decrease means better generalization. Neural nets need something to appropriately shape the membership function between classes when samples are lacking. We use new method exploiting Lipschitz bounds on the learned function. The MNIST data with resolution reduced to a 14x14 retina serves as an example. The method translates into simple, high-speed hardware.

The basics of adaptive logic networks

Adaptive logic networks (ALNs) were originally conceived at Bell Telephone Laboratories in 1967 and further developed at the Université de Montréal. They were originally networks that passed Boolean signals through various layers. In 1995, when confronted with a challenging real-world application in control, a new way of doing credit assignment was found that used floating point values. Unlike the chain rule, credit was assigned to very few nodes in the first layer of the net [Armstrong and Thomas(1997)]. The first layer of an ALN computes several inner products between an input vector and some weight vectors and adds a constant. The graph is a hyperplane or "linear piece". That is where the similarity with most other neural nets ends: all higher level layers of an ALN consist of two-input maximum and minimum operators only. After a possibly erroneous output, a single linear piece in the first layer whose weights have to be changed is found, starting from the root, by choosing the input with the greater value at a maximum node or the lesser value at a minimum node. No squashing functions are needed because the maximum and minimum operations are non-linear. The network architecture is not set at the beginning of training: the net grows to fit the problem by splitting linear pieces in two.

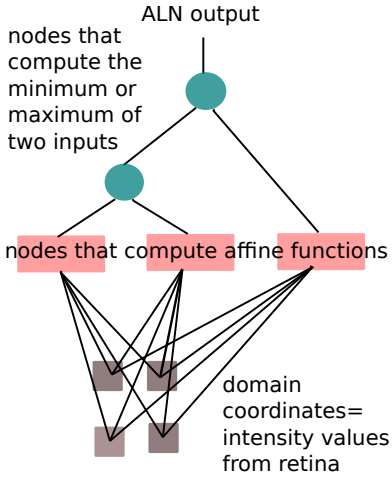

Figure 1: How an ALN computes.

ALNs have been used in various applications, notably for forecasting electrical power demand and power production at windfarms a few days ahead of time [Armstrong(2006)]. They have been used for controlling walking prostheses for persons with incomplete spinal cord injuries [Aleksandar Kostov(1995)]. For a long time classification was ignored because it seemed that a system based on regression had no business trying to learn how to generalize from labeled samples to places where there were no samples. Other authors have worked on this problem [Uwe Moenks(2010)].For low dimensional problems, jitter could synthesize additional samples, but in high dimensions the number required made that approach infeasible. The usual neural net can use regularization to cause the function to be smooth between classes, but with no guarantees in particular locations. This is an advantage of ALNs: the learned functions can be made to satisfy Lipschitz conditions everywhere, without fail.

The ALN computation structure is shown in Figure 1. The inputs $X = (x_1, \ldots, x_n)$ to an ALN are n real numbers, implemented in floating point. These feed

into a first layer of nodes which compute affine functions $y = w_0 + \sum_{i=1}^{n} w_i x_i$. The graphs of the functions are hyperplanes. The outputs of the affine functions feed into a tree whose nodes are two-input maximum and minimum operators. The hyperplanes intersect and cut each other off where one becomes greater than the other to form a continuous function graph of linear pieces. The ALN is still called a "logic network" since maximum and mimimum operators in fuzzy logic act like the OR and AND operations of Boolean logic.

A simple proof shows that this computing structure can compute any continuous function to any degree of accuracy uniformly on any compact set of Euclidean n-space. The graph of a minimum of affine functions is a convex-up dome-shape and a maximum of appropriate domes can approximate any continuous function. Proving that a finite number of domes is enough uses the compactness of the domain. For an alternative proof, one could instead use a minimum of maxima of affine functions, the latter having a bowl-shape. A more efficient fit requires that bowls be used for convex-down parts of the function and domes for convex-up parts. Saddle surfaces can be synthesized by a maximum of domes or a minimum of bowls.

Given any input n-vector X, there is exactly one affine function computing the ALN output for X. We say it is the active affine function or the responsible leaf node for that input. It is found by an analog of backpropagation in other neural networks which uses the chain rule. The ALN, after computing an output for X, follows the ALN tree from the root towards the inputs, choosing the branch on the side of the greater input at a maximum node or the lesser input at a minimum node. In the case of a (rare) tie, we choose, say, the left input. This is the ALN way of doing credit assignment. It eliminates the vanishing gradient problem of neural networks using the chain rule for apportioning credit. An ALN requires doing enough evaluations of leaf nodes to be able to compare two inputs at some nodes to see which is greater. However the number of evaluations can be reduced in two ways: 1. alpha-beta pruning and 2. not evaluating leaves whose centroids are distant from X. Distant linear pieces are likely to be cut off by other linear pieces and thus have no influence on the value of the ALN output. Although it may provide significant advantages for some applications, assigning a bit of credit to many nodes in the typical neural net is very inefficient compared to ALN's targeting of one active linear piece.

A training step presents an input X to the ALN, computes the ALN output, finds the active affine function and changes its coefficients. It uses a different way of computing the affine function, namely it tracks the centroid $C = (C_0, ...C_n)$ of the samples activating that leaf node using exponential smoothing. The representation of the affine function for training is $y = C_0 + \sum_{i=1}^{n} w_i(x_i - C_i)$. With a learning rate of 0.2, say, the adaptation routine tries to correct 1/5 of the error, the ALN output minus the desired output, by distributing parts of the correction to the output centroid C_0 and $w_1, ..., w_n$. The process is linear regression, with the difference that as weights of linear pieces change, the set of samples active on any linear piece may change, which changes the least-squares problem to be solved. This is why we can't use a standard mathematical method.

The reason for using the term "adaptive" for an ALN is that the entire structure adapts to the data during training. The structure grows. Any first layer node can split into two nodes with the identical node function to the one that split. They are connected by a new maximum or minimum node. The growth mechanism examines the convexity of the part of the data for which the affine function is active and replaces it with a minimum or maximum node accordingly. There are two choices of criterion for deciding to split a leaf node after it has had time to train but the result is not satisfactory: 1. the mean square training error is greater than a user-chosen threshold, or 2. the mean square training error is greater than a noise variance estimate of the samples activating that leaf node. Estimating the local noise variance is done by looking at the differences of desired values of closest pairs in the training set, which should have roughly double the variance of the noise, and correcting later for the difference of location of the two samples using the weights of a linear piece being trained. The latter method makes a validation set unnecessary. Training stops when splitting stops.

We come to the most important feature of ALNs for the purpose of this article – the imposition of maximum and minimum partial derivative constraints on the learned function on each axis. This is done during training steps: if the training algorithm sends any weight outside a prescribed bound, the weight is set at the bound. The bounds define a Lipschitz condition which will be globally satisfied by the learned function. The bounds can be changed dynamically. Being able to impose a Lipschitz condition on learned functions is important in many applications where large deviations from hoped-for behaviour of the learned function between training samples could be harmful. In this article, we are going to use these constraints to perform classification.

Setting up the MNIST data

The MNIST data consists of a training set of 60,000 samples of handwritten digits and a test set of 10,000 samples. Separate files contain the corresponding labels, 0 to 9, of the samples. Each sample has integer values of intensity from 0 to 255 on the elements of a 28x28 square grid. We changed this to a 14x14 grid by adding the intensities of 2x2 blocks to reduce training time. This did not appear to do much harm to a human's ability to classify the digits. We found it beneficial to adjust the intensities so all images had the same average.

Generating a class membership function in the absence of data

In this paper we use partial derivative constraints to shape the membership function between samples. We let 1 be the desired value of all the target class samples, and -1 the desired value of all the rest of the samples. We are assuming that the classes , subsets of $\{0, ..., 1020\}^{196}$, have closest neighbors separated by a minimum positive L1 distance $d_{interclass}$. The L1 distance is the sum of absolute values of differences of coordinates of domain vectors of two samples. Between the closest samples, an ideal membership function would decrease from 1 at the target samples to minus 1 at the non-targets. For each unit of absolute difference of a coordinate change then, the membership function should decrease by $s = 2/d_{interclass}$. For example, in the MNIST data, the closest pair of classes are digit 1 and digit 7, and $d_{interclass} = 3353$, so s is very close to 0.0006. In the simplest case, a hyperplane with slope s might just separate the two classes and provide a membership function.

Since we want to fit the space between samples, we set up an ALN with certain fixed linear pieces to start. We use one piece at a constant 0.95 level and one at the -0.95 level. The root is a minimum operator connected to the constant piece at 0.95. Its other input comes from from a maximum operator connected to the other fixed piece and to a linear piece which can grow into an ALN tree. Where the unrestricted ALN's linear pieces are below the -0.95 level or above the 0.95 level, they will not be active for any sample. When the growing part of the of the ALN separates the two classes perfectly, all training samples will be fitted by the ALN with an error of 0.05. In Figure 2, samples B and C will attract active linear pieces of the ALN (the darker polyline). Samples A and D attract the constant pieces, hence will have no effect. The reason for placing the constant pieces at a small distance from the samples is that then when a piece is attracted by a sample, it is pulled, say 20% closer to the sample, and if it reaches or crosses the + or -0.95 level, it does so definitely and won't just slowly converge to a limit. The faint lines at 45 degrees show Lipschitz bounds which emanate from every point in the ALN graph.

If we don't know the distances between classes, there is a solution: allow the weight bound to increase during training. If it isn't fitting all samples, then the RMS error will be above 0.05 and the loosening of the bound must continue. When the RMS error equals or at least is close to 0.05, we will have fitted all or most of the training set and the ALN is ready for testing.

Results on the MNIST data

Ten ALN membership functions were created, one for each digit 0 to 9 as the target class. The second class consisted of the digits of all non-target classes. Here are the results on the training and test sets of correctly identifying the correct class (target or non-target). Software is on GitHub [Armstrong(2020)]

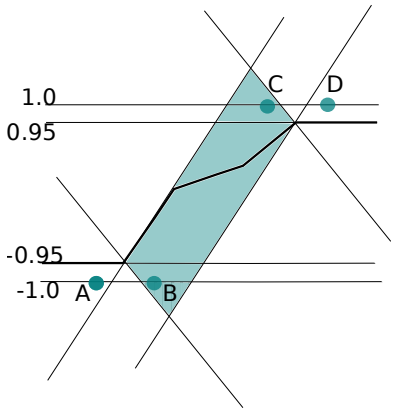

Figure 2: Dark area is allowed for the growing ALN

Target	Train (%)	Test (%)
0	99.9683	99.9600
1	99.9600	99.9600
2	99.9783	99.9800
3	99.9483	99.9400
4	99.9833	99.9900
5	99.9900	99.9800
6	99.9750	99.9900
7	99.9517	99.9400
8	99.9800	99.9900
9	99.9750	99.9800
Average	99.9710	99.9710

It is easy to see that generalization has been very good. To the best of our knowledge, these results are among the best, if not the best, ever obtained. This means a low weight bound has been preserved and a good membership function has been generated in each case. The final weight bounds were larger than the minimum possible because training was not stopped using the RMS error at 0.05. In fact, that level of RMS error and 100% correctness of training was never attained this way. In trying to explain this, beyond blaming bad images or labels, we came up with a new idea: using only a dome to separate the classes.

Convex Classification

In some of our tests, we have classified the MNIST data with a special ALN structure to perform what we call convex classification. As was done above, one class, the target class, had desired value 1 and all the others had desired value minus 1; the ALN had two constant linear pieces at levels -.95 and +.95. However, in contrast to the above, the growing part of the ALN used only minimum nodes in layers above the first layer. The non-constant part of the ALN grows as part of a dome bounded by the two constant pieces. The target samples pull the linear pieces they activate up until they

are above the top constant piece. The non-target samples pull their pieces down to below the lower constant level. After a perfect training, the decision boundary at ALN output level 0 is a section of a convex set, which is convex, and separates the target class from the others in the domain. Convex classes which are disjoint can be separated by a hyperplane, say one which bisects the direction vector between two of the closest samples. If MNIST classes were convex, we would need only one affine function to distinguish each pair of classes, 45 in the case of MNIST data.

We define a convex a classification problem as one where any convex combination (positive weights that sum to 1) of images of one class is not equal to member of a different class. Applying this idea to the MNIST data, gave inferior results. Why? We can think of samples of digit "5" which, in a convex combination, look like a "6" because they have gaps on the lower left side at different heights. Two narrow figure "8"'s with small holes could easily fuse into a wide "1". One notable case of non-convexity is zeros that have holes at different places. These classes are non-convex because convex combinations of samples that have voids at disjoint places may not have a void. Samples having gaps at the same place can form a convex class. Readers can perhaps determine if any of their classification problems are convex.

Convex classification can be implemented in very simple hardware that has a good chance of beating any other neural network for speed of classification. We can just look at the convex dome part of the ALN which computes a function f and forget the two constant pieces which play no rôle in determining if $f(X) > 0$. So after training, we can transform the ALN into a Boolean logic tree by applying the following formulas starting at the root: $max(g, h) > c$ iff $g > c$ OR $h > c$, $min(g, h) > c$ iff $g > c$ AND $h > c$. At a leaf, we have $y = w_0 + \sum_{i=1}^{n} w_i x_i > 0$. That is, we have a perceptron [Minsky and Papert(1969)]. Such a hardware implementation would be extremely fast. In the case of convex classification, we have all minima in the function f. So $f > 0$ if and only if all scalar products at leaves are positive. In hardware, say a GPU, if all the products are done in parallel, the time of evaluation of the ALN would be that of one scalar product plus a very small time to decide if none are negative. In the general non-convex case, there are groups of affine functions whose signs, interpreted as TRUE or FALSE, have to satisfy some Boolean condition. This would still lead to very fast execution of classification.

Dealing with huge numbers of classes

Suppose we have a vision system on a given domain which has many ALN recognizers, one for each class, where the classes are not intermixed and are at a certain minimum distance from each other. Consider an arbitrary hyperplane that cuts the domain in the middle of the samples. Then, given an input X, some ALNs may have $f > 0$ entirely on the opposite side of the hyperplane, so X is not in the class of that ALN. Further some linear pieces of ALNs still in contention may lie on the other side and may not need to be evaluated because of alpha-beta pruning or other optimization. By choosing axis-parallel hyperplanes and forming a decision tree, we can place X inside a box in the domain where only a few ALNs and a few linear pieces can be active. For example, in a 196-dimensional domain such as the one we are using, the greatest number of hyperplanes coming together at a point is 197. That is the maximm number of affine functions that have to be evaluated for an ALN and any X if we have a decision tree that divides the domain into small enough boxes. That is well within the reach of today's GPU's. So after a fast evaluation of the decision tree, a GPU would need one thread for each element of a (monochromatic) retina, plus one, to be able to classify an image in the time of one scalar product, done in parallel, plus a very short time for some additional levels of logic gates. It is hard to imagine any classification hardware that could do the job faster, a fitting tribute to Frank Rosenblatt, the originator of the perceptron.

Conclusion

The method we have presented allowed us to achieve a low error rate on the MNIST dataset, and is generally applicable to other data where the classes don't intermix and are at a minimum L1 distance from each other. ALNs can provide excellent generalization by placing constraints on the rate of change of the learned learned class membership function. The implementation in hardware using perceptrons and logic gates would be simple and extremely fast.

References

[Aleksandar Kostov(1995)] Aleksandar Kostov, e. a. 1995. Machine Learning in Control of Functional Electrical Stimulation Systems for Locomotion. *IEEE Trans. on Biomedical Engineering* .

[Armstrong(2006)] Armstrong, W. W. 2006. Use of piecewise linear functions in forecasting. In *Artificial Intelligence in Energy Systems and Power*. ICSC. ISBN 3-906454-36-3.

[Armstrong(2020)] Armstrong, W. W. 2020. NANO. https://github.com/Bill-Armstrong/NANO.git. Machine learning software in C++.

[Armstrong and Thomas(1997)] Armstrong, W. W.; and Thomas, M. M. 1997. Adaptive logic networks. In Fiesler, E.; and Beale, R., eds., *Handbook of Neural Computation*, chapter C1.8. Institute of Physics Pub. and Oxford U. Press. ISBN 0 7503 0312 3.

[Minsky and Papert(1969)] Minsky, M. L.; and Papert, S. A. 1969. *Perceptrons: An Introduction to Computational Geometry*. MIT. ISBN 0 262 13043 2. Rev. 1988.

[Uwe Moenks(2010)] Uwe Moenks, Denis Petker, V. L. 2010. Fuzzy-Pattern-Classifier Training with Small Data Sets. In *Information Processing and Management of Uncertainty in Knowledge-Based Systems 2010*. Springer.