# A Component for Real-Time Machine Learning

© William Ward Armstrong, PhD

October 16, 2024

**Abstract**

A component for real-time machine learning systems is described. The goal of the component is to perform a classification task by using training data to create hyperplanes to partition the space of all measurements (e. g. intensities of all pixels of any possible image) into blocks which have either a high concentration of the target class and a low concentration of the non-target class, or vice versa. The example described herein is in the context of a *Learning Feature Tree (LF-tree)* which natually organizes the splitting hyperplanes as a tree hierarchy for training, although alternative organizions may be more appropriate for execution on parallel hardware. On each block, the LF-tree stores a linear function which is an estimate of the probability of membership in the target class at each point of the block. When all blocks are taken together, their linear functions form a *discontinuous* membership function over the whole measurement space. By overlapping and weighting the outputs of several LF-trees, the discontinuities can be removed and the accuracy increased. Ideally both the hyperplane parameters and the linear function parameters are developed based on training data without human intervention. The trees are all independent of each other, so they could all be created ( and exploited) in a parallel system in little more than the maximum time required to create (or exploit) a single LF-tree. The trees start small and grow to fit the training data. The block whose linear function was used in producing an ouput can at the same time produce, internal to the system, an "engram" identifying the block. If, at any later time, a positive or negative reinforcement is received by the system, it can use the engram and other information collected at the same time to change the function of the block to improve performance. This is called *forward propagation* of credit assignment.

## 1 References

1. D. Kahaner, C. Moler, S Nash, Numerical Methods and Software, Prentice-Hall, 1989.

2. https://github.com/Bill-Armstrong/Real_Time-Machine-Learning and other programs.

## 2 Embodiments of the component

At this time, the only embodiment of the component is a C++ program running on a single computer. This could be translated into software running on a single computer with a GPU, or software running on many computers perhaps connected through a public network. Other embodiments could be as an FPGA or ASIC. The main resource requirement is enough memory to store the parameters of hyperplanes and linear functions of blocks. To achieve real-time potential, hardware embodiments are essential. For example, the demo program to learn to classify MNIST handwritten numeral images can use 200 LF-trees and take about 1/2 hour to run on an i7 Windows computer. It can obtain about 96% correct classifications on the MNIST test set. The tree-building part of a CUDA program using a large GPU, would be *at least* 200 times faster because all the trees would be built at the same time. Beyond that, we show below how parts of a single tree-building can be made up to 78 times faster, so training on a large number of classes in a fraction of a second may be possible even with current and affordable hardware.

Any form of parallel execution can result in several orders of magnitude speedup compared to the software version. In many situations, the software calls for several hundred multiply-add operations to be executed sequentially. For example, a feature multiplies some or all of the intensities of pixels in an image with stored constants and adds the products. A much faster version would use a tree-hierarchy of multiply-adds. One thousand of these operations could then be done roughly in the time of ten of them. This is a one-hundredfold speedup of tree operation. In the case of MNIST data, the speedup doing this for features would be 78 times.

It is extremely important that the trees have blocks of samples that don't coincide. Some samples must be different in different blocks and must bestochastically independent of other samples, leading to improved accuracy. Since the trees are not interdependent, all of them can be executed in parallel. So for example, a large system of trees, some doing completely different pattern classification tasks, others doing the same task but on different data, can all be done in parallel. The task of learning to classify handwritten characters of the MNIST data could be completed in under 30 seconds after the data is loaded, judging by the demo program, even without the many possible optimizations of the trees themselves. With all optimizations done on fast hardware, the MNIST data could possibly be learned in a fraction of a second.

There will be tasks that require the outputs of trees as inputs for other trees, and in that case, trees will have to be executed sequentially. At this time, we don't know how to put this all together to get the system to reason like a human. It is encouraging that the component works on different classification tasks with just changing a definition of what is wanted. For example with no human help, the system which classifies images of numerals 0 to 9, can find what is common to 3's and 8's and pick those out in a test set.

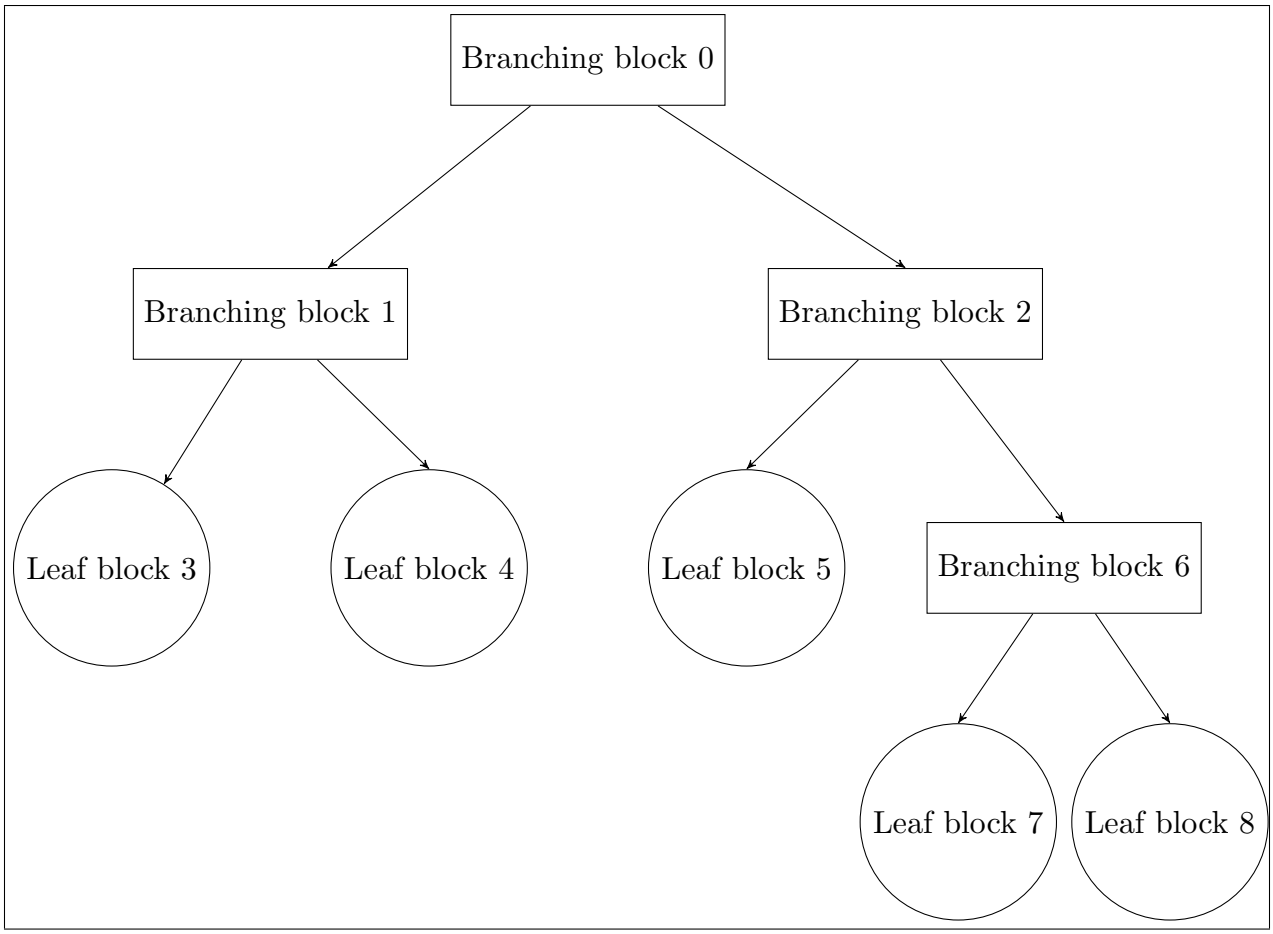


Figure 1: Two types of blocks in an LF-tree. Arrows point to children of a split. When Leaf block 6 split, it created Leaf blocks 7 and 8 and became Branching block 6. Blocks 7 and 8 are at the end of the list of blocks and may in turn be split or made final during the same call of growTree.

## 3 The pattern-classification method

Consider two ideal membership functions: one which is the probability of a sample being a target, the other the probability of being a non-target sample. The latter class can encompass a large collection of classes lumped together as in the case of MNIST data. Each digit 0 - 9 can be a target for some trees and in that case, all the other digits are non-targets. If there were images of cursive letters, lions or petunias in the training set, they would be in the non-target class. The target membership function gives the probability, at any point of a block, of a target being observed relative to all possible samples that could occur there. This is the kind of target membership function discussed in this document.

Using the component, one can add a new target class by simply creating LF-trees for it and putting them into the existing collection of LF-trees. The only connection between trees is that a sample can find its way down one tree to a non-zero linear function, and down a second tree too. For example a picture of a

violin could be in one tree as an example of a valuable instrument and in another tree as a currenly missing item sought by police. The same picture always goes to the same place in the measurment space. There is no blurring by a convolution operation. This points out that sometimes the blocks in an LF-tree should contain other information to link them to blocks of other trees. Given a sample and estimates of the values of the membership functions of relevant classes at the location of the sample, one can choose the class whose mambership function value is greatest to classify the sample.

The job of a Learning Feature Tree is to train on data to learn the estimated class-membership function of its class. The MNIST data has handwritten numerals from 0 to 9 on a 28x28 grid where intensities range from 0 to 255. The demo program can use, for example, seventy LF-trees with groups of seven trees being devoted to each one of the numerals. The weighted average of the membership functions of the seven is used to increase the accuracy of classification. By weighting the linear functions with weights that approach zero at block boundaries, the *Weighted Overlapping Linear Functions* can be averaged to form, in theory at least, a continuous function, as long as the block boundaries of different functions don't coincide. We call this the *WOLF* approximation technique.

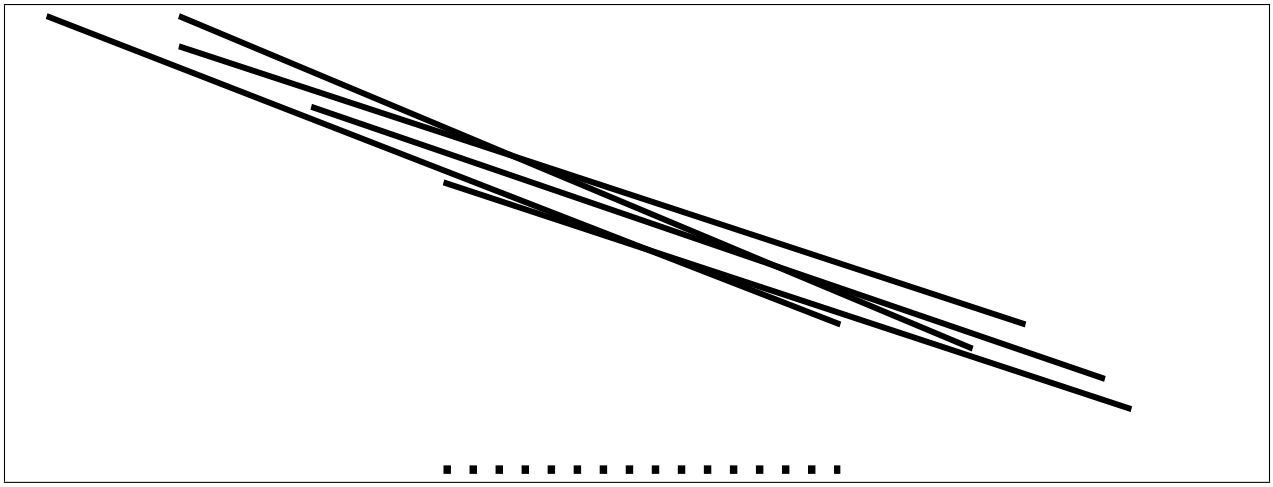

Figure 2: Averaging several overfits on different data with the WOLF technique can produce a good fit.

Figure 2 shows five LF-trees at work. It shows the simple case of a one-pixel image where the intensity of the pixel (the X-axis) determines the probability of a class. The dark line segments represent five blocks from five trees whose height (the Y-axis) estimates the probability of the target class among all classes. The endpoints of each line segmant are the two boundaries of its block. Any sample X in the dotted interval, when evaluated by features in one of the trees, leads to one leaf block in that tree and its linear function. The values of these five linear functions are averaged. It would be too confusing to show the discontinuous average function or the weighted average which hides the discontinuities. We can imagine them this way: the average without weighting is a linear function graph above the dotted inteval in the middle of the mass

of line segments. The linear function in each block is computed using a different set of samples, and because many are assumed statistically independent, this can improve accuracy of the fit. When weighting is used, the fit in the dotted interval is made up of three parts. The two parts at the ends depend on the width of the interval within which the weight on one of the bottom two pieces (in this case) goes to zero. When weighted, the unweighted average of the five functions is bent up at the left and right as two of the linear functions reach their domain boundaries. The bent parts continue going straight for a short distance until they get close to another block boundary.

Omitted are other blocks belonging to other trees to the left and right of the dotted interval which have a zero value in the dotted interval. The reader should try to imagine the situation in two or three dimensions, with blocks in the form of convex polygons or polyhedrons, to get an idea of what happens in higher dimensions. Finally, the smoothing can vary linearly with the distance from the block boundary, which results in a continuous membership function, or it can be cubic, making the function continuously differentiable, at least in theory.

There may be a few sample points where the boundaries of all functions coincide, all weights are zero and the weighted average is indeterminate. Then additional work must be done to classify a sample. In that case, the function can be evaluated by taking two points close to and in line with the sample point in the interior of all involved blocks (i.e. not on any boundaries). A linear extrapolation then gives the value of the sample's membership function.

In earlier work, the author placed a high value on generating continuous membership functions. Continuity just says that a very small change in pixel intensities is unlikely to change the class of what is seen. There is another way to get continuity which adds considerable complexity: by looking at the intersections of linear functions on neighboring blocks and using those intersections to define block boundaries. We called this the ALN approach from earlier work. It results in a continuous membership function. However, there is a large penalty involved in finding which block a sample belongs to. The component presented here derives its speed from the idea that *the requirement of continuity in the theoretical sense can be loosened for the membership function.* We note that weighting is not used during tree growth, only during classifications, and even then it could be dropped in favor of faster execution in most cases.

We view back-propagation systems like convolutional neural networks as putting a lot of emphasis on learning a continuous membership function. The smoothness of the function provides a form of interpolation where there are missing samples. Unfortunately, the pattern recognition task becomes hidden and inscrutable behind a large convolution and multple layers of neurons with linear functions and the required non-linearies. We have employed a convolution too when there are obvious correlations between neighboring pixels of an image. However, our convolutions are not applied to the samples presented for training or execution, but

rather on the features that refer to the pixels of the sample. That way, there is zero time or hardware penalty for improved accuracy through exploiting correlations.

The main problem confronting designers of improved LF-tree systems is automatic feature design based on training data. The demo program has one general feature design method, described in detail below, which works on the MNIST data and should work on other problems. The example in the reinforcement routine shows that the automatic feature design works to detect features of images like the two bumps on the right of images of 3's and 8's. Once one has an automatic feature design process, to specify the problem to be solved on any given data, all the user does is write a short expression in the reinforcement routine describing the reinforcement to be given. The demo program should next be tested on a full training set of handprinted letters to see how the presented automatic method carries over to other data.

The performance will likely be less than perfect. This is the case for the MNIST data which reaches only about 96% correct recognition. One possible path to higher accuracy is better defining the problem by having more data. That way, the measurement space can be more finely divided, and blocks which have mixtures of targets and non-targets can be split more finely; more blocks consisting of all targets or all non-targets can be created. Clearly, the method will work to any degree of accuracy desired if there are enough samples.

Fake samples can be created from existing samples by using jitter: just changing the pixel intensities of an image a bit and keeping the same class label. It is interesting that jitter just increases the variance on all axes for jittered samples that still belong in the same block under consideration. One other way is to impose distortions on the training images which should not require a change to the class label. The obvious but probably most costly way, is to collect more data from the original source.

Another approach that might help is creating features that analyse the image of a digit looking for strokes, loops etc. Asking the question: "Is this image a 2, 3, 5, 7, 8, 9, or 0" would be like asking if there is a horizontal stroke at the top. This was learned to 98% correct. That could be applied after the imperfect 96% classification to check whether the initial classification is right or what the decision should be. We envision more such tests being applied, just like a human would look for features like loops and strokes in a hard-to-interpret image of a numeral.

# 4 A flatness criterion based on noise variance

While this idea is not implemented in the software demo, it has been used successfully in practice, and it could be added to the splitting criteria to stop unnecessary splits. The noise variance can be estimated by the method given below. If the data is not well fitted by a linear function, then the mean square error will be greater than the noise variance estimate.

Suppose we have a function $f$ which is defined on convex block in n-space. Suppose there are $m$ noisy samples of $f$ and the noise has mean zero with respect to $f$ and constant variance VAR over the domain. We do a linear least-squares fit $F$ to approximate $f$. Then we can estimate VAR by first finding for each sample the closest other sample. If a sample $j$ is $(X_{1j}, y_{1j})$ and one closest to it is $(X_{2j}, y_{2j})$ and the samples happen to be at the *same place* in the domain, then their difference would have variance twice VAR, assuming stochastic independence of the noise of the two samples. It is unlikely that two samples end up at the same place, but we can correct for the difference of place by using the differences with $F$ of the sample values. In other terms, we slide one of the samples along the linear fit and change the sample value as we go along. This approximates the idea of getting two samples at the same place. Subtracting these, we get a set of values which, if the samples are generated by a single linear function and the least squares fit is perfect, have mean zero and twice the variance VAR.

$$VAR \approxeq 1/2m \sum_{j=1}^{m} ((y_{1j} - F(X_{1j}) - (y_{2j} - F(X_{2j}))^2$$

If the data is generated by a single linear function then this variance estimate should be close to the mean MSE of the samples with respect to the least squares estimate. If the data is not well modeled by a single linear function, then the MSE should be significantly larger than the VAR estimate because it has to have some larger values to accomodate bends of the ideal function and its samples. If the piece is not flat enough, its block should be split in two. To make this more precise, it could formulated using a statistical F-test.

# 5 Detailed description of a software embodiment

The routine growTree() grows one LF-tree at each call. The tree starts with a single leaf block. It is (often) the block of all training samples. In the case of MNIST data, the measurement space is a 784-dimensional hypercube. Since the same data is used in all trees, the numbers of the images, 0, 1, ... 59999 are inserted in the list attached to the first leaf node. That list will be split in two and put on the lists of two child leaf bocks by the first feature developed as described below. The centroid of the set of all samples can also be precomputed. The centroid is a vector formed by the mean intensity values of each of the 784 pixels in a MNIST image. The block 0 centroid is shifted for each tree by multiplying the actual centroid by a positive factor equal to 0.4 plus the current number of the tree divided by the total number of trees to be created. So the factor stays between 0.4 and 1.4. These bounds are not magical and can be changed. By changing the starting "centroids" of all trees in this way, different trees have different samples in overlapping blocks

as they continue growing. This is what increases the accuracy of classification when more trees are used on the same target class. Another trick we could do is take one tree and just shift it a small distance in each axis. If many shifted trees are required, the shift becomes greater and the danger is that some blocks that were filled with samples in the original tree are shifted where there are no samples.

We assume that the entire training data is given all at once. It is possible to give inputs intermixed with processing, so the first split is based on a smaller number of samples which would generate an adequate feature for processing many inputs yet to come. Developing rules for this is research yet to be done.

We make another assumption in the demo software that is not necessary, namely that the reinforcement is given without delay to the learner. Ideally, images should be presented in a random order and at the start random exploratory actions should be taken. This is not done in the demo for the sake of keeping execution time down, but a demo could be enhanced to allow random order and longer delays in getting reinforcement feedback. Any action taken could be recorded through an engram of the block, perhaps also with the action, the time the action was taken, and any other useful information. Instead of the centroid, any image in the block can serve as an engram. Then when some agent, a trainer, later says, in effect (pointing at a copy of the image or even just giving the time when the action of attempted classification was done) "that image was not correctly classified by your action", at that moment, the system recalls the action and the block involved and applies a change to the linear function that was responsible for the action. If an average probability is involved in choosing the action, all trees involved are changed. The change could be as simple as lowering the value of a constant or could involve changing the weights of the linear functions. As engram, we then have to add to the event list record the actual sample received because that sample contains the correct $x$ values for all active sensors and the $y$-vale output.

If we modeled the whole process, and the teacher often takes over an hour to give the reinforcement, then it will take over an hour to train the system. A functioning learning system, say in a robot, would contain a list of events with actions, times, and engrams and any useful information collected at the time of the action. There will be a lot of data there, but it can be reduced in volume by choosing a new retina, where the sensors are the engrams of other trees and the intensities are the probabilities.These "secondary retinas" have as their pixel values, the probabilities associated with the values of the linear functions of a block. The retinas could be reduced to a minimal size by the same "variance-based" process used for the visual retina which is described in the next paragraph.

The learning process creates a least-squares solution to fit the data of a block. The solution uses the mean value of all reinforcements, which we store as the "output centroid". There are 784 sensors numbered from 0 to 783, and centroid vector index 784 stores that mean value. The other sensors each have a weight attached defining the linear function of the block. Computing the weight involves computing a denominator

which turns out to be the variance of the intensity values of the samples attached to the block.

We compute for each pixel (i.e.sensor) intensity (or value on an axis of the initial block)

1. $En$ the number of samples on the block,
2. $Ex$ the mean value of intensity of those samples,
3. $Ex2$ the mean of the squares of the intensities,
4. $Ey$ the mean value of reinforcements ( 0 or 1 in our example),
5. $Exy$ the mean value of intensity times reinforcment.

Calulus then allows us to derive the least-squares value for the weight (slope) for each sensor as

$$W = \frac{(Exy - ExEy)}{(Ex2 - ExEx)}$$

The same idea can be found in the reference of Kahaner, Moler and Nash on the normal equations in matrix notation. We don't have to worry about zero denominators because we eliminate all sensors from consideration if their values are "deemed constant" if the samples in a block don't vary enough on some axis to generate a non-zero variance (i.e. a non-zero denominator). Actually, we can set the limit for sensors that are "deemed non-constant" as higher than zero. This eliminates more sensors and accelerates the computation of features. Our least-squares solution will fail if there are not at least as many samples as variables. When we get to small blocks, samples are in short supply, so it may not be possible to have weights, just a constant value as linear function of the block. To overcome the short supply of samples, one can generate new samples by simple distortions of existing ones or several means already mentioned.

We note that if the values of $Ex$ etc. are computed using the centroid as origin of the values so that $Ex = 0$ , the formula for the weight on the sensor simplifies as follows:

$$W = \frac{Exy}{Ex2}.$$

The program has a parameter that requires a block to have at least a certain number of samples to be able to split. The final act of the routine growTree on a block is to decide to make the block final (there will be no further changes to it). This it does if there are too few samples for the block to split; otherwise it passes the block to the createFeature() routine.

In the routine createFeature(), which is the most important of all routines in the program, the targets and non-targets are first counted. If the block has all targets or all non-targets, it is made final and the

value of C[nSensors] is set to 1.0 or 0.0 accordingly. That is, the value of the linear function of the block is a constant.

Features multiply the intensities of pixels by constants assigned to those pixels. As an experiment we tried convolving the features. The factors on individual sensors were weighted and combined with those of neighboring sensors. This seems reasonable to do since neighboring sensors in the image have highly correlated values. It seemed to produce a little improvement in generalization, but of course this would not apply in cases where the pixel values are uncorrelated as in "secondary retinas".

Next, the numbers of target and non-target values on each active sensor are obtained and summed so the means of targets and non-targets on each sensor can be computed. The differences of the two means are an indication of how well the sensor can separate targets and non-targets.

The maximum and minimum sensor differences are determined. The minimum can be negative. Then to eliminate the least useful sensors, some with sensor differences near zero are removed. A features vector is created with the remaining sensors and associated factors being the sensor differences divided by the maximum ( or the minimum). The factors are renormalized so the absolute values sum to 1.0.

Now we examine the various placements of hyperplanes to divde the block. We look at the minimum and maximum values of the feature for targets and for non-targets. By choosing one of these, we can sometimes chop off part of the block that contains only targets or only non-targets and form a simple new block. This is in keeping with the idea of separating target- and non-target samples into different blocks. Each placement of the hyperplane is determined by a constant, FTvalue. The feature vector merely determines the direction vector normal to the hyperplane. Tge FTvalue determines an appropriate placement.

The routine createFeature ends with passing the block along to the routine splitSB to split the sample block in two. The routine splitSB creates two new blocks and copies the vector of active sensors to each of the chilren. It distributes the list of image indices attached to the original block to the lists of the children. The distribution follows according to the sign of the feature vector together with the FTvalue. As the samples are distributed, the computation of the centroids of the children is carried out. The routine ends by turning the block that was split which, up to now, has been a leaf in the Learning Feature Tree, into a non-leaf node which points to the two leaf children (Figure 1). Finding the right block for a sample is done in the Find routine. When results of classification are needed, another Find routine also computes the weight.

Finally we note that when the two new blocks are created, they are placed at the end of the list of all blocks. The for-loop processing all blocks has an upper bound that is not fixed but grows as new blocks are created. This means that the call to growTree which produced the new children by splitting will also process them. In this way, the entire tree grows with just one call to growTree.

## 6 Making it fast enough for real-time applications

There are several operations of the component that can be accelerated. For example, any time a long sequence of multiplication products are added, as in a feature calculation, this can be structured to be done in parallel as a tree. If there are 1024 pairs of numbers to be multiplied and all the 1024 results added, then one can do the mulplication of pairs in parallel for any way of grouping into 512 pairs. Then the results are grouped into 256, and those into 128 and so on. The tree computation is done in the time of ten multiply-adds instead of 1024. The speedup is by a factor of 100. The downside is a need for a large number of hardware muliply-adders.

The operation to find a block given an image may require 10 features, say, to be computed, one after the other. However one can calculate all of the tree's features simultaneously at the start and do the branchings to find the block much faster since the left-or-right decisions are already made. This requires enough hardware.

The computation of the centroids of the values on each sensor in the case of MNIST data, use as many adds as there are samples. Again a tree structure can be used. All the sensors can be done in parallel. In fact, all the values $Ex2$ etc. used in the least-squares computation can be computed using parallelism.

Expert designers of software and hardware systems will be able to optimize training and execution times on a wide variety of platforms.

## 7 Building complex real-time machine learning systems

The WOLF approximation method produced results that were 96% correct on the MNIST test set. If it has to read 8 handwritten numerals and gets each one right with probability 0.96, then it will get a ZIP+4 or 9-digit postal code right less than 70% of the time. The more digits to read, the less the likelihood of success. To overcome this deficiency, we have suggested using jitter or transformations to generate new artificial data and get closer to 100%.

To make further progress, we should also think about how humans recognize numerals. Certainly it is not just pattern-matching on the original image. In difficult cases, the human will probably analyze parts of the image to find strokes, loops etc. By asking the question: "Is the image a 3 or an 8", we can get the system to recognize a common feature of 3 and 8. Detecting a horizontal stroke at the top can also be done by Learning Feature Trees by choosing as target set 2, 3, 5, 7, 8, 9, 0. Instead of the human constructing features, the system derives them automatically from the training samples. The final classification could be the result of several such processes, each with a certain probability of a successful result, and could be closer to 100% and possibly better than a human in some cases.

The state of a complex real-time system at any time is stored in the features, the linear functions of blocks, the values like $Ex2$ and the event-list collection of engrams and associated data. At this point, we have no idea how to take advantage of solid theory already developed for choosing sequences of actions in reinforcement learning. Are there algorithms like solving the Bellman equation, value iteration or Q-learning? Can an android robot learn to keep a short rod upright in the palm of a hand while running and jumping?

# 8 Forward propagation of credit assignment

A system based on back-propagation of credit assignment operates as follows: an image arrives, a convolution is done on the image, it is passed through a neural net, the error is evaluated, the chain rule is used to find out the inflluence of each of maybe thousands of coefficients on the output. Then many weights are adjusted a bit so that the error on the current image is reduced. Weights enter the optimization process in a way that prevents too-rapid changes. A validation set is used for stopping training. A major problem from our point of view is that the chain rule cannot be applied unless the original input image is present and all the nodes retain their inputs from the image. If another image is presented to the system, the possibility of training on the previous pattern is lost. If that information is to be preserved, it would be a very complicated process. In contrast, the centroids or other samples in blocks are very simple engrams. Forward propagation of the information that can later be used for adjustment of weights seems to be the best approach.

# 9 A plethora of research topics

The component has been demonstrated with batch learning, where all the training data has been presented to it at the start. It is clear that taking smaller chunks at a time would not be significantly different from what has been shown as long as the chumks of data are good statistical samples of the whole of the data. But what if the component has to be trained on single samples? This is important even for updating linear function values of components after batch training has concluded.

At the end of batch training, the system could retain the values of $En, Ex, Ex2, Exy, Ey$ as vectors in all blocks and for all sensors. These could be updated using exponential smoothing as follows: For all domain sensors, i.e. pixels in the image, do this iterative step using a small positive "learning rate" $\lambda$:

$newEn = oldEn + 1$

$newEx = (1 - \lambda)oldEx + \lambda(x - C)$ where $x$ is the intensity of the sensor and $C$ is the centroid of its values, which is equal to $oldEx$

$newEx2 = (1 - \lambda)oldEx2 + \lambda(x - C)^2$

$newExy = (1 - \lambda)oldExy + \lambda(x - c)y$ where $y$ is the reinforcement given

$newEy = (1 - \lambda)oldEy + \lambda Ey$.

Clearly in this case, we need as engram the actual image and not just the centroid, unless of course, the linear function is a constant. How to split a block is an issue. If there are enough samples in a block, the same approach to splitting the block by creating a new feature as in batch mode training could be used.

## 10 Can back-prop networks be recast as Learning Feature Trees?

Yes, with slight modifications. If one creates training samples by submitting input samples to a back-prop network and recording that network's output, then one can use those samples to train LF-trees. The zero-and-one limits of the value used for membership functions would be eliminated. One would start with one block and split it in two at the centroid of the data if the linear function does not fit the data well enough according to the F-test suggested in a previous section. The direction of the hyperplane can be determined by splitting the block into two parts with maximally, or at least significantly, different slopes (weights). During this process, one can check to see if the weights do not exceed desired bounds that might have led to dangerous excursions of output value of the original back-prop network. Details of regression type machine learning can be found in different machine learning projects by the author on GitHub under the name Bill-Armstrong. The method could result in a copier, checker and accelerator of back-prop systems.

## 11 Conclusions

The software demo illustrates that the Learning Feature Tree can learn to accurately recognize patterns in a framework of reinforcement learning. Forward propagation of the engrams and other associated data makes it possible to have training occur later than the action of determining or guessing what the images or other stimuli or patterns in an event list represent. For robots, the fact that features can be learned from data means that human intervention will not always be required for the robot to learn new tasks by itself.