# Convolutional Adaptive Logic Networks: A First Approach

**Nima Mohajerin** [1] **Monroe M. Thomas** [1] **Oz Solomon** [1]

## Abstract

Adaptive Logic Networks (ALNs) are binary trees of min/max operators and linear functions. They have a simple structure which can be extended during training which adds additional parameters where the fit to the target function is poor. The simple structure allows for fast evaluation, which makes them suitable for implementation in edge devices. ALNs are universal approximators for continuous functions defined on a compact set, which makes them great candidates to be used in deep learning. In this work, we revisit the theory of ALNs and propose a simple architecture to use them with Convolution layers, which we train and test against the CIFAR-10 dataset. Our results demonstrate that the new Convolution ALNs (CALNs) can perform on par with state-of-the-art CNNs. We hope this work, accompanied by code for running ALNs in modern deep learning frameworks, opens new horizons for the community and interested researchers.

## 1. Introduction

Deep Neural Networks (DNNs) have become one of the most powerful tools to numerically model many tasks with high accuracy. However, deploying them on low-computation devices, such as edge devices, remains a challenge. The reason is that, depending on the number of parameters and layers, a DNN can be quite expensive in terms of memory and computation. Therefore, it is natural to investigate methods for compression and acceleration in DNNs. Researchers in (Cheng et al., 2017) divide such methods into four categories: parameter pruning and quantization (Liang et al., 2021), low-rank factorization (Wen et al., 2016), transferred/compact convolutional filters (Zhai et al., 2016), and knowledge distillation (Gou et al., 2021). A higher level categorization, given in (Liang et al., 2021), categorizes methods for Convolutional Neural Networks (CNN) acceleration into three categories: Network Structure, Network Optimization and Hardware. In this article, we look at the compression and acceleration in DNNs from an alternative point of view.

To solve a Machine Learning (ML) problem, choosing the right model and architecture is fundamental. In the context of non-parametric models, architectural parameters (such as the number of convolution layers) are tuned as part of hyper-parameter optimization which can result in a model with improved accuracy. However, this tuning step is usually taken later in the model training; after some initial promising results are obtained for a given task. Therefore, the most common practice is to over-parameterize DNN architectures mainly because it is extremely difficult to find the right architecture and number of parameters for any given task. As a result, an immense amount of time and effort is spent on methods to prevent overfitting. Regularization, expanding datasets, early-stopping, etc., are examples for coping with overfitting in over-parameterized models. On the contrary, models that grow over the course of training, e.g., Genetic Programming (Ahvanooey et al., 2019) (GP), do not suffer from over-parameterization, but are more prone to become trapped in local minima. In such models, decisions for growing the network on the fly are central to the training process.

Adaptive Logic Networks (ALNs) are another example of this approach. They have been around for more than two decades (Armstrong & Thomas, 1996; Gorodnichy et al., 1997). Despite being relatively old in the ML community, ALNs have simple and computationally efficient self-growing structures and can be appealing for use in practical applications. In this work, we would like to investigate the modelling capability of ALNs in the context of Deep Learning and DNNs both in terms of accuracy and computation speed. We will revisit ALNs and describe their training algorithm from a practical perspective, and then propose Convolutional ALNs (CALNs), which use convolutional layers as feature extractors for ALNs. It will be demonstrated that for some small and medium size problems, CALNs are comparable with state-of-the-art CNNs both in terms of accuracy and inference speed. Given the relatively small amount of research in ALNs, there are many available avenues to conduct further research on ALNs in the Deep Learning context. Therefore, we are making our codebase public for any one who is interested in conducting

[1]Microsoft Applied Science Group. Correspondence to: Monroe M. Thomas <monroe.thomas@microsoft.com>.

research in this domain[1]. We will also present some ideas on possible avenues for future research in this direction.

## 2. Adaptive Logic Networks

ALNs are revisited in this section. We first explain the forward execution in ALNs through a simple example and then describe the tree decomposition as well as how to leverage it for speeding up the evaluation.

### 2.1. ALN Tree - Forward Execution

An ALN, in its original form, is a binary tree of AND/OR logical gates (Armstrong & Thomas, 1996). The leaf nodes are Linear Threshold Units (LTUs) and the output of the ALN is either 0 or 1. In the context of function approximation, the nodes are either $\max$ or $\min$ operators, represented by $\vee$ and $\wedge$, respectively, and the leaves are linear weighted sums of the input values, a.k.a. Linear Units (LUs). In equation (1) an LU's mapping is given for an $N$-dimensional input, $\mathbf{x} \in \mathbb{R}^N$, where $\mathbf{w}_i \in \mathbb{R}^N$ and $b_i \in \mathbb{R}$ are the input, weight and bias terms, respectively,

$$l_i = \mathbf{w}_i^T \mathbf{x} + b_i. \tag{1}$$

In this setting, the ALN output is real valued. In (Armstrong & Thomas, 1996) it is shown that ALNs are universal approximators for any continuous function defined on a compact set.

Figure 1 shows a simple ALN tree and the function it represents graphically. In this example, the ALN has three input LUs ($l_0, l_1, l_2$) which are connected through two $\max$ nodes. The input to the ALN is $x \in \mathbb{R}$. Each LU (yellow node) is a line (dashed blue) whose equation is given in the figure. To calculate the output of the ALN given an input, $x$, the LU values, corresponding to $x$, are passed through the $\max$ nodes (green) down to the root, $y$. This particular ALN represents a function of $x$ depicted in (2). The corresponding surface is illustrated in figure 1 as the bottom border of the yellow region (three black dotted lines).

$$y = (l_0 \vee l_1) \vee l_2 = \begin{cases} -x - 2 & x \leq -1 \\ 0.5(x+1) & -1 \leq x \leq 1 \\ 2x - 1 & x \geq 1 \end{cases}. \tag{2}$$

From this simple example, it is clear that ALNs divide the input space into small regions and associate a line to each region. Theoretically, the number of LUs over the input space can be arbitrarily large and therefore, the function represented by the ALN arbitrarily smooth.

[1]Codebase URL

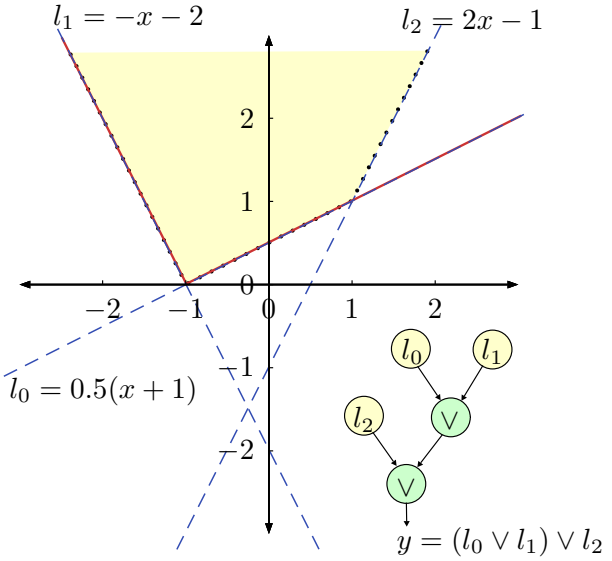


*Figure 1.* A simple ALN. The red border illustrates the function represented by $l_0 \vee l_1$. The ALN output $y$ is represented by the bottom border of the yellow region (black dots).

### 2.2. Lazy Evaluation

An interesting characteristic of ALNs is the decomposition of trees into smaller ALNs which can leverage what is referred to as *lazy evaluation* in (Armstrong & Thomas, 1996). To understand this concept, first note that an ALN with more than one leaf node can be written as combinations of multiple ALNs. Figure 2 illustrates a simple example. In this figure, the ALN tree on the left ($y$) can be decomposed into two ALNs ($y_1$ and $y_2$) on the right. Such decomposition breaks down a large ALN into smaller ones some of which may not need to be computed for some given inputs. For instance, consider the ALN function represented by the tree on the left side of the figure 2,

$$y = (l_0 \wedge l_1) \vee (l_2 \wedge l_3),$$

where the input space is $\mathbb{R}$. For any input, $x \in \mathbb{R}$ where $y_1 > l_2$, there is no need to calculate $l_3$, because,

$$\forall x \in \mathbb{R}, y_2 \leq l_2,$$

and since we assume $y_1$ is larger than $l_2$, then it is obvious that $y = y_1$. It is clear that by implementing a few if-then statements, in practice we can skip evaluating large subtrees which improves the tree evaluation time significantly.

### 2.3. Training ALNs

In this section, the ALN training algorithm implemented in the shared codebase is explained. Note that this algorithm is based on (Armstrong & Thomas, 1996); however,

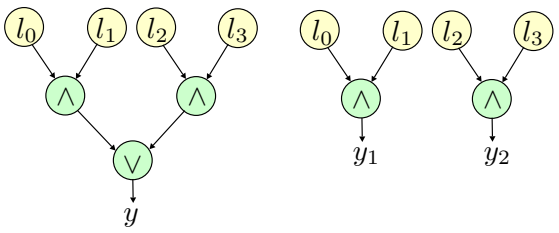


*Figure 2.* ALN trees can be decomposed to smaller ALNs. This helps to implement *lazy evaluation* (refer to text).

our explanation is slightly different. In supervised training, where the goal is to approximate an unknown function $y^* : \mathbb{R}^M \rightarrow \mathbb{R}$, we need a dataset of $N$ samples, $\mathcal{D} = \{(\mathbf{x}_i, y_i^*), i = 1, \ldots, N\}$, where $\mathbf{x}_i \in \mathbb{R}^M$ is the observed input and $y_i^*$ is the corresponding observed output. To facilitate understanding the algorithm, the following definitions are useful.

**Definition 2.1.** We call the $i^{\text{th}}$ LU, $l_i$, *responsible for input* $\mathbf{x}$, when the ALN output at $\mathbf{x}$ is equal to the LU's value at that point, i.e., $y = l_i(\mathbf{x})$.

**Definition 2.2.** The set of all points in the given dataset, $\mathcal{D}$, for which the $i^{\text{th}}$ LU, $l_i$, is responsible, is called the *responsibility set* of $l_i$, and is represented by $\mathcal{S}_i$. Formally,

$$\mathcal{S}_i = \{(\mathbf{x}_j, y_j^*) \in \mathcal{D} | y_j = l_i(\mathbf{x}_j)\}. \tag{3}$$

**Definition 2.3.** The mean point of the $i^{\text{th}}$ LU's responsibility set, $\mathcal{S}_i$, is called the centroid of $l_i$,

$$\begin{aligned} C_i =& (\mathbf{x}_{c,i}, y_{c,i}), \\ \mathbf{x}_{c,i} =& \frac{1}{|\mathcal{S}_i|} \sum_{\mathbf{x}_j \in \mathcal{S}_i} \mathbf{x}_j, \\ y_{c,i} =& \frac{1}{|\mathcal{S}_i|} \sum_{y_j \in \mathcal{S}_i} y_j, \end{aligned}$$

where $|.|$ means the cardinality of a set.

**Definition 2.4.** A *dormant* LU is the one whose responsibility set is empty. Note that a dormant LU does not activate over the input domain. For instance, in the following ALN,

$$l_0 = x, l_1 = x + 1, y = l_0 \vee l_1,$$

$l_0$ is a dormant LU because for all $x \in \mathbb{R}, l_1 > l_0 \Rightarrow y = l_1$.

□

Similar to many other ML training algorithms, training ALNs is an iterative process. There are two major steps that we may want to execute (one or both) at each iteration. We shall refer to them as the *fit* step and *split* step. The fit step alters parameters of LUs so that the fitness of each LU over its responsibility set improves. The responsibility set for an LU may grow or shrink during the fit step as the LU and its neighbours rotate and translate relative to each other as fitness improves. In the split step some (or all) of the LUs may be split into two LUs, causing the ALN tree to grow.

It is possible that a particular input may belong to more than one LU's responsibility set if the point lies precisely at the intersection of these LUs. This has no effect on the output the ALN, but does force a choice to be made regarding which LU to update. In our publicized codebase, we choose to update only one such LU at each step time. This changes the point of intersection between the responsible LUs, greatly increasing the likelihood that after the update, the input will belong only to a single responsibility set.

#### 2.3.1. Fit

In general, fitting a hyper-plane to a set of points is straightforward. Formulated as Least-Squares (LS) problem, one can easily derive a closed form solution using Maximum Likelihood with a Gaussian noise prior (Bishop & Nasrabadi, 2006). However, there are two major reasons for not using the standard curve-fitting methods for ALNs. The first reason is that the responsibility set for each LU changes over the course of training. This is because LUs may translate, rotate, and split during training and a particular point that belongs to the responsibility set of an LU may later not belong to it. Therefore, having the best fit for LU parameters at each iteration is a premature optimization and should be avoided.

The second reason stems from the different error sensitivity between the LUs bias and weights. Using the Mean Squared Errors (MSE) measure, the error sensitivity is equal to the gradient of the LU's output with respect to its parameters. Without loss of generality and for simplicity we consider a 2D case where the error sensitivity for an LU is depicted in (4),

$$y = wx + b, \nabla_w y = x, \nabla_b y = 1. \tag{4}$$

Equation (4) simply shows that the LU's output gradient w.r.t. its weight ($w$) depends on the input while it is a fixed value w.r.t. the bias ($b$). Normalizing input values is a standard practice in most ML algorithms which improves the accuracy as well as the behaviour of the activation functions and avoids numerical instabilities. Therefore, in most cases[2] $\nabla_b y$ is larger than $\nabla_w y$. Geometrically, this means relatively larger translations of hyper-planes over the input space than rotation, which in practice, may result in dormant LUs and also dramatic changes over the responsibility sets of LUs.

As ALNs divide the input space to subsets, it is intuitive

[2]Even if the input is binary, it is quite rare that all of the inputs in a training batch are equal to one.

**Algorithm 1** LU ($l_i$) parameter update

**Input:** $\mathbf{v}_i$, $C_i$, $\mathbf{y}_i$, $\mathbf{X}_i = \{\mathbf{x}_j | (\mathbf{x}_j, y_j^*) \in \mathcal{S}_i\}$, $\alpha = |\mathcal{S}_i|$
($\mathbf{y}_i$ is the output of the ALN for all inputs $\mathbf{x}_j \in \mathbf{X}_i$)
**Learning rates:** $\lambda$, $\lambda_c = \frac{\alpha^2}{\alpha - 1}\lambda$

**for** $\mathbf{x}_j$ in $\mathbf{X}_i$ **do**
  Calculate error, $e = y_j - y_j^*$.
  Calculate average distances:
  $$\begin{cases} \mathbf{d}_x = \frac{1}{\alpha}\sum(\mathbf{x}_j - \mathbf{x}_{c,i}) \\ d_y = \frac{1}{\alpha}\sum(y_j - y_{c,i}) \end{cases}.$$
  Update variance, $\mathbf{v}_i \leftarrow \mathbf{v}_i + \lambda_c(\mathbf{d}_x^2 - \mathbf{v}_i)$.
  Update centroids, $$\begin{cases} \mathbf{x}_{c,i} \leftarrow \mathbf{x}_{c,i} + \lambda_c \mathbf{d}_x \\ y_{c,i} \leftarrow y_{c,i} + \lambda_c d_y \end{cases}.$$
  $\mathbf{w}_i \leftarrow \mathbf{w}_i - \lambda e \boldsymbol{\nu}$, where $\boldsymbol{\nu} = [\frac{d_x}{v_i}]$.
  $b_i = y_{c,j} - \mathbf{w}_i^T \mathbf{x}_{c,j}$.
**end for**

to improve the LU hyper-planes fitness by adjusting their rotation and then their translation rather than entirely rely on gradient values to update the weights and biases. One way to achieve this is by incorporating the centroids in the update algorithm. For a general LU, represented by equation (1), the parameter update algorithm is given in Algorithm 1.

The most important aspect of this algorithm is the fact that the bias term, $b_i$ is *calculated* after updating the weight, $\mathbf{w}_i$. The bias calculation leverages the fact that the centroid point lies on the hyper-plane. Note that the centroid is also updated because $\mathcal{S}_i$ may change at each update. Another noticeable factor is the weight update rate, $\lambda\boldsymbol{\nu}$. This value consists of a scalar learning rate, $\lambda$, and a vector $\boldsymbol{\nu} \in \mathbb{R}^N$, which is the element-wise division of the sample distance, $\mathbf{d}_x$, and variance, $\mathbf{v}_i$:

$$\boldsymbol{\nu} = \mathbf{d}_x \oslash \mathbf{v}_i. \tag{5}$$

Equation (5) implements a mechanism to rotate the hyper-plane, $l_i$, towards the direction in which the inputs are farther from the centroid. Note that the variance and centroid (mean) are updated iteratively, rather than being calculated for each batch, to avoid relying extensively on transient responsibility sets.

#### 2.3.2. Split

The binary tree that ALNs implement can (and should) grow. Although there can be other variants for the growth mechanism, here we discuss the one implemented in our public codebase.

At a split step, there are two decisions to be made for splitting LUs: which LUs to split and how to connect the new LUs to the old ones. Keeping track of the distribution of the points in each LU's responsibility set gives a means to both ends. First, we select LUs which are not good fits for their responsibility set distribution. The measure for fitness is a design choice. For instance, we can use MSE for each LU and its responsibility set. Then for each split candidate, a new LU is created by cloning the candidate parameters and connecting the new LU to it, using a $\min$ or a $\max$ operator. If the responsibility set distribution around the candidate LU is convex, the connecting operator will be $\max$, otherwise it will be $\min$.

Figure 3 illustrates this concept via a simple example. In this figure, two snapshots of a training process are shown, where the goal is to fit an ALN to the data samples (dark yellow dots). The top plot shows the ALN when it consists of two LUs that are connected via a $\min$ operator; the blue line shows the output of the ALN over the input domain. Paying attention to the error plot for each LU, one can observe that the convexity of the error distribution around $l_0$ is dominant while the error is distributed in a concave manner around $l_1$. Therefore, cloning these two LUs, the first new LU (cloned from $l_0$), namely $l_2$, should be connected to $l_0$ via a $\max$ operator and the other new LU, namely $l_3$, via a $\min$ operator to $l1$. After the split, we run another fit step and the ALN output over the input space is depicted in the bottom plot. The newly added nodes, $l_2$, $l_3$, $\vee$ and $\wedge$ are illustrated as hatched.

To conclude this section, we would like to summarize the hyper-parameters and important design choices that are discussed in this section either explicitly or implicitly.

- **Learning rate**, which similar to other ML algorithms should be chosen carefully. A scheduler may also improve the convergence speed.
- **Initial tree structure**, meaning how many initial LUs the tree should have and through what operators they should be connected.
- **Split step frequency**, that is, how often we should split versus fit the LUs. We recommend a warm-start/cool-down procedure.
- **LU fitness criteria**, here we used MSE. However, MSE can be sensitive to outliers. One may use Mean Absolute Error (MAE) or other measures.

## 3. Convolutional ALNs for Image Classification

In this section, we assess ALN performance on image classification task. First, we use the MNIST dataset to perform a computational assessment of ALNs. Afterwards, we propose to use convolution layers as feature extractors for ALNs, hence the name Convolutional ALNs (CALNs). We

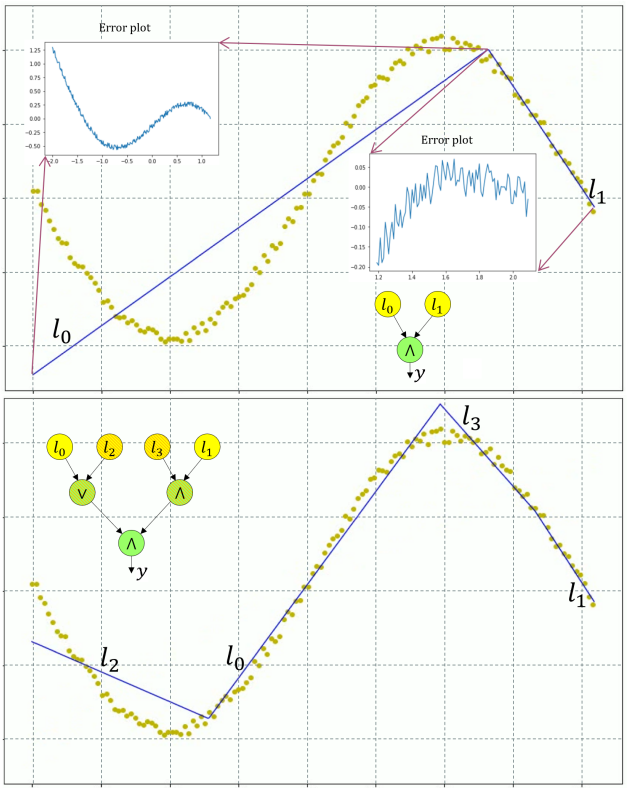


*Figure 3.* Illustration of the concept behind the split mechanism for ALNs in this work. The two plots represent two snapshots of an ALN being trained on the data samples (dark yellow points). The top one is prior in time to the bottom one. The ALN tree structure at the time of the snapshots is also shown over each plot.

train and evaluate CALNs on CIFAR-10 dataset to demonstrate the effectiveness of using Convolution operators with ALNs.

### 3.1. MNIST

To investigate the applicability of ALNs to image classification, we begin by using them on the MNIST dataset, where the inputs are vectors of $28 \times 28 = 784$ real values. Because ALN outputs are scalar, we need to train one ALN for each class. To stay with the traditional formulation, we use MSE as the criterion for training. The main goal for this task is to assess the forward computation speed of ALNs. Therefore, the ALN training and evaluation is purely implemented in C++. We also do not apply any augmentation technique. The results are summarized in Table 1.

From Table 1 it is evident that ALNs are superior on computation speed over a typical CNN while having similar accuracy. Note that the Eval time is per sample and ALNs are abour 14 times faster than CNNs. The tests are done on Intel(R) Xeon(R) W-2145 CPU under Windows 10 OS. It

*Table 1.* Comparison between an ALN with a simple CNN on MNIST. No augmentation is used.

| NETWORK | ACCURACY | EVAL TIME (MS) |
|---|---|---|
| ALN | 97.5± 0.5 | 0.0445 |
| CNN | 98.3± 0.5 | 0.624 |

is also worthwhile to mention that this is not an exhaustive test and the values are approximate, however, it motivates further investigation of ALNs, as will be discussed next.

### 3.2. CIFAR-10

Although useful as a toy dataset, MNIST does not reflect many challenges that exist in image classification tasks. To assess ALNs further, in this section we present raining and evaluation results for the CIFAR-10 dataset. CIFAR-10 inputs are 32-by-32 color images; therefore, the input dimension is $3 \times 32 \times 32 = 3072$, almost 4 times the MNIST input size. As our first attempt, we train and evaluate pure ALNs on CIFAR-10, using data augmentation. Figure 4 illustrates the F1-Scores per class for multiple experiments, each having slightly different hyper-parameter settings. However, the results are fairly poor.

One explanation is that ALNs do not have any mechanism to capture spatial dependencies among the pixel values of an image, whereas the convolution operator does. A solution is to employ convolution layers as feature extractors for ALNs. We call this architecture Convolutional ALNs or CALNs. It is arguable that CALNs basically replace Fully Connected layers (FCs) with ALNs in CNNs. Although this interpretation is valid, ALNs contribute slightly more than a mere substitution to FCs. In fact, our experiments show that CALNs provide equal or slightly better accuracy on CIFAR-10.

The experiments being presented here are based on three

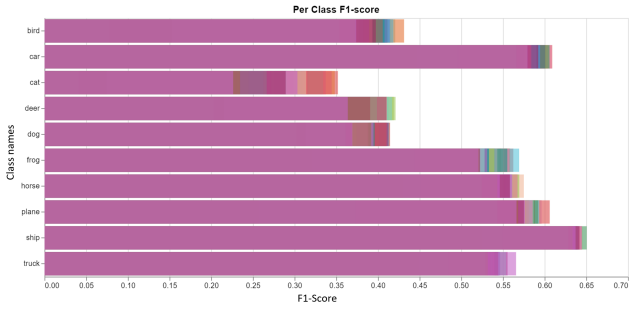


*Figure 4.* Per class F1-Scores from pure ALN approach on CIFAR-10 image classification. Each color shows one experiment.

*Table 2.* Assessment of CALNs accuracy on CIFAR-10 in comparison with CNNs.

| NETWORK | BEST ACCURACY | MEAN ACCURACY | N PARAMS. | MEAN EVAL. TIME (MS) |
|---|---|---|---|---|
| RESNET13+ALN | 95.55 | 95.40 | 2.8M | 13.94 MS |
| RESNET14 | 95.51 | 95.31 | 2.79M | 13.45 MS |
| RESNET18 | 95.68 | 95.53 | 11.2M | 19.83 MS |

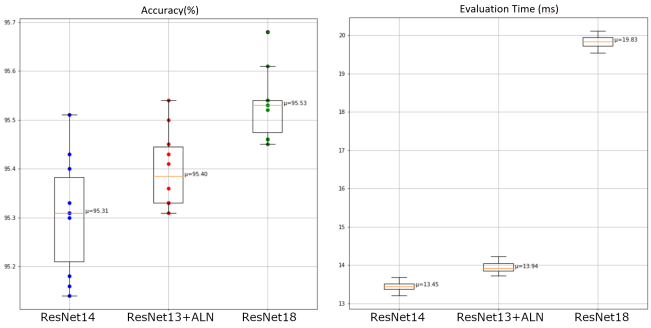


*Figure 5.* Per class F1-Scores from pure ALN approach on CIFAR-10 image classification. Each color shows one experiment.

networks:

- ResNet13+ALN, which uses 13 2D convolutions (3 residual blocks) connected to an ALN (1 ALN per class),
- ResNet14, uses the same backbone as above connected to one FC with 10 outputs,
- ResNet18, uses standard ResNet18 backbone, connected to one FC with 10 outputs.

The results are presented in Table 2. Note that the networks are trained from random weights, meaning pre-trained models were not used. The results show slightly better performance of the ResNet13+ALN network, on average, comparing to the ResNet14 output. For each architecture, we ran multiple experiments and the corresponding box-and-whiskers plot is illustrated in Figure 5. The immediate observation from Figure 5 is the linear improvement of accuracy in going from ResNet14 to ResNet13+ALN and then ResNet18 while the execution time grows exponentially. Note also that a single FC layer is merely a matrix multiplication, and is much faster than ALN. It is not immediately clear that using ALNs in this setting is always a win; more experiments are required to be conducted on larger datasets. However, it is evident that CALNs performance, both in terms of accuracy and speed, is comparable with CNNs.

## 4. Conclusion and Future Work

ALNs are binary trees of $\min$ and $\max$ operators on linear mappings of the input which can grow and adapt during training. In this work, we revisited ALN architecture. We propose a Convolutional ALN, which uses convolution operators as feature extractors for ALNs and, hopefully, provide some motivation for future works on ALNs. We have also made our code publicly available at (URL).

To the best of our knowledge this is the first attempt of using ALNs in Deep Learning. Therefore, in this section we would like to discuss some of the challenges in using ALNs and propose preliminary.

The current implementation of ALNs require *target* values for the ALN output. Therefore, ALNs should only be used as the final stage of an architecture. In addition, ALNs conceptually fit an indefinite number of hyper-planes over some points. If these points change drastically during training, the presented algorithms for adapting and growing ALNs will struggle, and may even be unable to find a useful tree structure. One immediate solution may be to entirely depend on the gradients, allowing the autograd procedure to find the derivative for us. The downside of using autograd, as described earlier, is that the pure gradient-based update of the LU parameters can lead to many dormant LUs. It may be possible to implement a *sweep-and-remove* procedure to remove dormant LUs, however, this will increase training time.

A shortcoming of the current formulation of the ALNs is the inability of removing, or, merging existing LUs. In the current implementation, ALNs will only grow; they never shrink. To use ALNs as additional operators in arbitrary places of a Deep Learning architecture, a shrinking mechanism seems imperative.

The current codebase also suffers from a degradation in performance when using Python code to iterate over each ALN used in a network, which represents a serial bottleneck in evaluation of training. Finding a way to present the structure of multiple ALNs (with varying numbers of LUs and min/max operators) in parallel to remove this bottleneck could greatly increase CALNs performance. This improvement is particularly important as it enables CALNs (and other variants) to be trained on larger datasets.

## Acknowledgement

We would like to acknowledge W. W. (Bill) Armstrong for his tireless work in developing the theory of ALNs over the decades, and for releasing some of his recent work and papers under an MIT license. Bill's unpublished paper "Adaptive Logic Networks" (2016, with Kenneth O. Cogger) provided many valuable insights and motivations for our present effort. Bill's work can be found at https://github.com/Bill-Armstrong/NANO and https://github.com/Bill-Armstrong/Deep-Learning-ALN/blob/master/ALNpaper.pdf.

## Source code

Source code used to conduct the experiments in this paper may be found at https://github.com/microsoft/ConvolutionALNs

## References

Ahvanooey, M. T., Li, Q., Wu, M., and Wang, S. A survey of genetic programming and its applications. *KSII Transactions on Internet and Information Systems (TIIS)*, 13 (4):1765–1794, 2019.

Armstrong, W. W. and Thomas, M. M. Adaptive logic networks. *Handbook of Neural Computation*, 1:8, 1996.

Bishop, C. M. and Nasrabadi, N. M. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

Cheng, Y., Wang, D., Zhou, P., and Zhang, T. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282*, 2017.

Gorodnichy, D. O., Armstrong, W. W., and Li, X. Adaptive logic networks for facial feature detection. In *International Conference on Image Analysis and Processing*, pp. 332–339. Springer, 1997.

Gou, J., Yu, B., Maybank, S. J., and Tao, D. Knowledge distillation: A survey. *International Journal of Computer Vision*, 129(6):1789–1819, 2021.

Liang, T., Glossner, J., Wang, L., Shi, S., and Zhang, X. Pruning and quantization for deep neural network acceleration: A survey. *Neurocomputing*, 461:370–403, 2021.

Wen, W., Wu, C., Wang, Y., Chen, Y., and Li, H. Learning structured sparsity in deep neural networks. *Advances in neural information processing systems*, 29, 2016.

Zhai, S., Cheng, Y., Zhang, Z. M., and Lu, W. Doubly convolutional neural networks. *Advances in neural information processing systems*, 29, 2016.