Deep Target Algorithms for Deep Learning

Anonymous Author(s)
Affiliation
Address
email

Abstract

There are many algorithms for training shallow architectures, such as peceptrons, SVMs, and shallow neural networks. Backpropagation (gradient descent) works well for a few layers but breaks down beyond a certain depth due to the well-known problem of vanishing or exploding gradients, and similar observations can be made for other shallow training algorithms. Here we introduce a novel class of algorithms for training deep architectures. This class reduces the difficult problem of training a deep architecture to the easier problem of training many shallow architectures by providing suitable targets for each hidden layer without backpropagating gradients, hence the name of deep target algorithms. This approach is very general, in that it works with both differentiable and non-differentiable functions, and can be shown to be convergent under reasonable assumptions. It is demonstrated here by training a four-layer autoencoder of non-differentiable threshold gates and a a 21-layer neural network on the MNIST handwritten digit dataset.

1 Introduction

Learning in deep architectures is a fundamental problem in machine learning, engineering, and neuroscience, with a long history.

1.1 Brief Historical Perspective

In necessarily very schematic terms, the problem goes back at least as far as the work of Hebb who proposed correlation-based learning rules as a possible solution in the late 1940s. Ten years later or so, Rosenblatt introduced the perceptron learning algorithm [17, 18] for shallow classification networks consisting of one layer of threshold gates with linearly separable data. This algorithm was extended by Widrow and Hoff who introduced the Delta rule, essentially gradient descent, for shallow one-layer differentiable networks [21]. Minsky and Papert in their influential 1969 book [15] proved a number of interesting results about perceptrons, including the perceptron cycling theorem [5, 10] to partially address the non-linearly separable case. They also raised major concerns regarding the possibility of finding good learning algorithms for multi-layer perceptrons. Their challenge was taken up in the mid 1980s by the Parallel Distributed Processing group which introduced and developed the backpropagation learning algorithm [19], basically an application of the chain rule to compute the gradient at each level of a multi-layer feedforward neural network, as well as the first autoencoder circuits. While backpropagation enables training networks with more than one layer, it is also well known to fail beyond a certain depth, depending on implementation details, due to the problem of vanishing or exploding gradients [13]. In the early 1990s and early 2000s additional shallow one-layer learning algorithms were developed in connection with SVMs, kernel methods, and maximum margin optimization [20]. Beginning in the mid 2000s, the problem of how to train deep architectures has regained a central role in AI and machine learning [4] with the development of new applications and approaches, including semi-supervised approaches to initialize the majority

of the weights based on stacks of autoencoder networks that can be trained in unsupervised ways using the input data only [11, 12, 8].

1.2 Proposed Approach

If we consider this brief perspective and examine the methods that have been somewhat successful (i.e. the perceptron algorithm, the backpropagation algorithm, the SVM algorithm, and the stacked-autoencoders approach), they all have one thing in common in that they provide targets for each layer that needs to be trained. Furthermore, when a target is provided for a layer, we know how to train this layer by using one of these algorithms. Thus we propose a new approach where the goal is to identify suitable targets for each hidden layer in a deep architecture. This approach is very general and can be used for both differentiable and non-differentiable (e.g. threshold gate) transfer functions. We begin by presenting the general framework and notation.

2 General Framework and Notation

A deep feed-forward architecture $\mathcal{A}(n_0, \ldots, n_l, \mathcal{F}_1, \ldots, \mathcal{F}_l)$ with l layers (Figure 1) is described by providing

- The size $n_0, \ldots n_l$ of the l layers. The size of the input layer is n_0 and the size the output layer is n_l.
- The classes of functions associated with each layer. $\mathcal{F}_j$ represent the class of functions allowed in layer j. If F is in $\mathcal{F}_j$, then F is a function from $\mathbb{F}_{j-1}^{n_{j-1}}$ to ${\mathbb{F}_j}^{n_j}$. Thus we assume that the activity vector in layer j is a vector over ${\mathbb{F}_j}^{n_j}$, where $\mathbb{F}_j = \mathbb{R}$ or $\mathbb{F}_j = \{0, 1\}$ correspond to the most standard cases. Note that restricted connectivity between layers can be incorporated into the definition of $\mathcal{F}_j$.

An instantiation $\mathcal{A}(n_0, \ldots, n_l, F_1, \ldots, F_l)$ of the architecture is defined by the l functions $F_1, \ldots, F_l$ of the proper size connecting the various layers. For instance, F_1 is a function from $\mathbb{F}_0^{n_0}$ to $\mathbb{F}_1^{n_1}$ and F_l is a function from $\mathbb{F}_{l-1}^{n_{l-1}}$ to $\mathbb{F}_l^{n_l}$. We let $W = F_l \circ \ldots \circ F_2 \circ F_1$ denote the overall transformation. It will be useful to isolate a particular layer j and write

$$W = F_l \circ \ldots \circ F_{j+1} \circ F_j \circ F_{j-1} \circ \ldots \circ F_1 = A_{j+1} \circ F_j \circ B_{j-1} = A_{j+1} F_j B_{j-1} \tag{1}$$

where $A_{j+1} = F_l \circ \ldots \circ F_{j+1}$ and $B_{j-1} = F_{j-1} \circ \ldots \circ F_1$.

A training set is a set of input-output pairs $(x_1, y_1), \ldots, (x_m, y_m)$ where for every i, $x_i \in \mathcal{X} \subset {\mathbb{F}_0}^{n_0}$ and $y_i \in \mathcal{Y} \subset {\mathbb{F}_l}^{n_l}$. Given a distance or distortion or dissimilarity function Δ^l defined over $\mathbb{F}_l^{n_l}$, the learning problem is the problem of minimizing the overall distortion E

$$\min E = \min_W \sum_{i=1}^{m} \Delta^l(W(x_i), y_i) \tag{2}$$

where the minimization is carried over all possible functions W allowed by the classes $\mathcal{F}_1, \ldots, \mathcal{F}_l$. It will be useful to have a distortion function available for each layer, in which case we let Δ^j denote the distortion function defined on layer j. For an instantiation $\mathcal{A}(n_0, \ldots, n_l, F_1, \ldots, F_l)$, the activity h_i^j of layer j for input x_i is defined by $h_i^j = F_j \circ \ldots \circ F_1(x_i)$. Obviously $h_i^l = W(x_i)$ is the overall output vector produced by the instantiation applied to input vector x_i.

To be concrete, in the typical cases $\mathbb{F}_j$ is $\mathbb{R}$ or $\{0, 1\}$, and the distortion function Δ^j is the squared Euclidean distances, the Hamming distance, or the relative entropy. Connectivity issues aside, the functions $F_1, \ldots, F_l$ between the layers can come from any of the following classes: (1) unrestricted Boolean functions; (2) restricted Boolean functions, such as threshold gates; (3) artificial neural networks; (4) any other class of differentiable functions. We specifically allow mixed architectures where different layers use different classes of functions, or non-differentiable architectures (e.g. threshold gates).

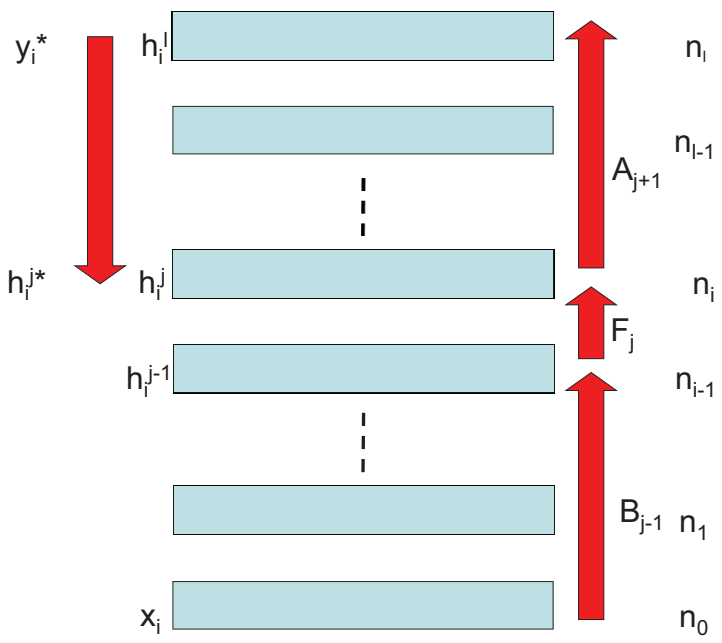

Figure 1: Deep architecture and deep target algorithms. The algorithm visits the various layers according to some schedule and optimizes each one of them. This is achieved by the deep target algorithms which is capable of providing suitable targets $(h_i^j)^*$ for any layer j and any input x_i, assuming that the rest of the architecture is fixed. The targets are used to modify F_j.

3 Deep Target Algorithms

3.1 Overview of the Learning Algorithm (Outer Loop)

The key assumption is the availability of an algorithm Θ for optimizing any layer or unit, while holding the rest of the architecture fixed, provided that we can specify a suitable target for that layer or unit. The optimization by Θ may be complete or partial, and takes place with respect to an error measure Δ^j specific to layer j in the architecture (or even specific to a subset of units in the case of an architecture where different units are found in the same layer). An exact optimization algorithm Θ is obvious in the unconstrained Boolean case. For a layer of threshold gates, Θ can be the perceptron algorithm, which is exact in the linearly separable case, or an SVM maximum margin algorithm. For a layer of artificial neurons or other continuous differentiable functions, Θ can be the delta rule or gradient descent, which in general performs only partial optimization.

Under these assumptions the training algorithm proceeds according to the following outer loop:

1. Cycle through the layers and possibly the individual units in each layer according to some schedule. Examples of relevant schedules include successively sweeping through the architecture layer by layer from the first layer to the top layer. Other schedules are discussed below.
2. During the cycling process, for a given layer or unit, identify suitable targets, while holding the rest of the architecture fixed.
3. Use the algorithm Θ to optimize the corresponding function F_j.

The most important point left to address of course is how the targets are identified. While targets are obvious for the output layer l, it is less obvious how to provide suitable targets for deeper layers. This is achieved by the deep target family of algorithms described below in on-line fashion, i.e. for an individual input x_i with a corresponding overall output target y_i.

3.2 Deep Target Algorithm (Inner Loop)

Consider any layer j, with $1 \leq j \leq l$. Recall that $W = A_{j+1}FjB_{j-1}$. We assume that both A_{j+1} and B_{j-1} are fixed. The input x_i produces an activation vector $B_{j-1}(x_i) = h^i_{j-1}$ and our goal is to find a suitable vector target in layer j. For this we take a sample $\mathcal{S}^j_i$ of vectors v^j in layer j. This sampling can be carried in different ways, for instance: (1) using only the values h^j_i over the training set; (2) using random values generated around the vectors h^j_i, i.e. around the hidden activities produced in the hidden layer j (e.g. using gaussian perturbations or sampling from the binomial or multinomial distributions associated with standard neural network transfer functions); (3) sampling more or less uniformly over $\mathbb{F}^{n_j}_j$; and (4) exhaustively, for instance in the case of a short binary layer. Given such a set of samples v^j, we compute the corresponding outputs $A_{j+1}(v^j)$. We then select as the target the vector v_j that produces an output closest to the ideal target y_i. Thus

$$(h^j_i)^* = \arg \min_{v^j \in \mathcal{S}^j_i} \Delta^l(y_i, A_{j+1}(v^j)) \tag{3}$$

If there are several optimal vectors in $\mathcal{S}^j_i$, then one can select one of them at random, or use Δ^j to control the size of the learning step. For instance, by selecting a vector v^j that not only minimizes the output distortion but also minimizes the distortion $\Delta^j(v^j, h^j_i)$, one can ensure that the target is as close as possible to the current layer activity, and hence minimize the corresponding perturbation. As with other learning and optimization algorithms, these algorithmic details can be varied during training, for instance by progressively reducing the size of the learning steps as learning progresses. Obviously for the output layer l one can use the target y_i directly instead of using this sampling approach. [Note: that the algorithm described above is the natural generalization of the algorithm introduced in [2] for the unrestricted Boolean autoencoder, specifically for the optimization of the lower layer.]

3.3 Possible Refinement

In many situations, there will be only one x_i in the training set such that $B_{j-1}(x_i) = h^{j-1}_i$. Even when multiple training vectors produce the same hidden activity the algorithm above can be used on-line. However, in the non-injective case, a slightly modified procedure may be preferable. To see this, define the cluster

$$\mathcal{C}^0(h^{j-1}_i) = \{x_k \in \mathcal{X} : B_{j-1}(x_k) = h^{j-1}_i\} \tag{4}$$

Since all the training vectors in $\mathcal{C}^0(h^{j-1}_i)$ map to the same hidden activity $h^{j_1}_i$, any individual information about each of these training vectors is lost in the architecture past the layer $j-1$. Now consider the corresponding set of output targets

$$\mathcal{T}^l(h^{j-1}_i) = \{y_k \in \mathcal{Y} : x_k \in \mathcal{C}^0(h^{j-1}_i)\} \tag{5}$$

The entire set $\mathcal{T}^l(h^{j-1}_i)$ ought to be taken into consideration when deriving the corresponding target $(h^j_i)^*$. The same sampling procedure as above can be used to produce a sample $\mathcal{S}^j_i$ and derive the target:

$$(h^j_i)^* = \arg \min_{v^j \in \mathcal{S}^j_i} \sum_{y_k \in \mathcal{T}^l(h^{j-1}_i)} \Delta^l(y_k, A_{j+1}(v^j)) \tag{6}$$

Again if the optimum is achieved by more than one vector v^j, the same remarks as above apply. [Note: that the algorithm described above is the natural generalization of the algorithm introduced in [2] for the unrestricted Boolean autoencoder, specifically for the optimization of the upper layer.]

In short, in all cases one is able to provide a target $(h^j_i)^*$ for h^{j-1}_i for any training input x_i and any layer j in the architecture, while holding the rest of the architecture fixed. This target can be used by the algorithm Θ to optimize the corresponding function F_j.

3.4 Additional Remarks

- Depending on the schedule in the outerloop, the sampling approach and optimization algorithm used in the inner loop, as well as other implementation details, the description above provides a family of algorithms, rather than a single algorithm.
- Adjustable learning rates can be used with different adjustment rules for different learning phases [6, 7].
- The proposed approach can easily be combined with backpropagation. For instance, targets can be provided for every other layer, rather than for every layer, and backpropagation used to train pairs of adjacent layers. It is also possible to interleave the layers over which backpropagations is applied to better stitch the shallow components together (e.g. use backpropagations for layers 3,2,1 then 4,3,2, etc).
- Examples of interesting schedules for the outerloop include a single pass from layer 1 to layer l, alternating up-and down passes along the architecture, cycling through the layers in the order 1,2,1,2,3,1,2,3,4, etc, and their variations.
- Similarly to backpropagation, the proposed approach can be combined with all the other "tricks" used in machine learning such as weight sharing, momentum, second order methods, and so forth.
- The exact complexity of the approach depends on its implementation details, in particular on the cycling method implemented in the outer loop and the sampling method implemented in the inner loop. The complexity however can be kept low, for instance using batches of training example that share the same sampling, and involves only forward passes to determine the targets. In particular, targets can be identified for multiple inputs at once to save on computations. If the set of possible targets $\mathcal{S}^j$ is the same for a batch of training examples, then we can update F_j using the entire batch but only a single computation of $A_{j+1}(v^j)$ for all $v^j \in \mathcal{S}^j$.
- In practice the algorithms converge, at least to a local minima of the error function. In general the convergence is not monotonic (Figure 2), with occasional uphill jumps that can be beneficial in avoiding poor local minima. Convergence can be proved mathematically in several cases.For instance, if the optimization procedure can map each hidden activity to each corresponding target over the entire training set, then the overall training error is guaranteed to decrease or stay constant at each optimization step and hence it will converge to a stable value. In the unrestricted Boolean case (or in the Boolean case with perfect optimization), with exhaustive sampling of each hidden layer the algorithm can also be shown to be convergent. Finally, it can also be shown to be convergent in the framework of stochastic learning and stochastic component optimization.
- The DT algorithms lend themselves to parallel implementations.
- The DT algorithms can be applied to recurrent or recursive architectures [1, 9, 3] by converting them into deep architectures, by unfolding them in time or in space.

4 Data and Experimental Results

4.1 Training Networks with Threshold Gate Units

Here we demonstrate that a deep target approach is capable of directly training a multi-layer perceptron with threshold gate units and Hamming distance error.

$$E = \sum_{i=1}^{m} \Delta^l(y_i, W(x_i)) \tag{7}$$

$$\Delta^j(a,b) = \sum_{k=1}^{n_j} \mathbb{1}_{a_k \neq b_k} \tag{8}$$

The task is to train a four layer autoencoder of binary data. The input and output layers have $n_0 = n_4 = 100$ units each and there are three hidden layers of 30, 10, and 30 units. All units in

layer j are fully connected to the n_{j-1} units in the layer below, plus a bias term. The weights for F_j are initialized randomly from the uniform distribution $U(-\frac{1}{\sqrt{n_{j-1}}}, \frac{1}{\sqrt{n_{j-1}}})$ except for the bias terms which are all zero.

The training data consists of 10 clusters of 100 examples each for a total of $m = 1000$. The centroid of each cluster is a random 100-bit binary vector with each bit drawn independently from the binomial distribution with $p = 0.5$. An example from a particular cluster is generated by starting from the centroid and introducing noise – each bit has an independent probability 0.05 of being flipped. The test data consists of an additional 100 examples drawn from each of the 10 clusters.

The distortion function Δ^j for all layers is the Hamming distance, and the optimization algorithm Θ is 10 iterations of the perceptron algorithm with a learning rate of 1. The gradient is calculated in batch mode using all 1000 training examples at once.

For the second layer with $n_2 = 10$, the set of potential targets $\mathcal{S}_i^2$ for input x_i is simply the set of all possible activations of the hidden layer $\mathcal{S}_i^2 = \{0, 1\}^{10}$ because $2^{10} = 1024$ samples leads to a manageable amount of computation. For other layers where $n_j > 10$, $\mathcal{S}_i^j$ comprises all activation vectors h_i^j, plus a set of 1000 random binary vectors where each bit is independent and 1 with probability 0.5.

Updates to the layers are made on a schedule that cycles through the layers in sequential order: $1, 2, 3, 4$. One update of all four layers constitutes an epoch. The trajectory of the training error is shown in Figure 2.

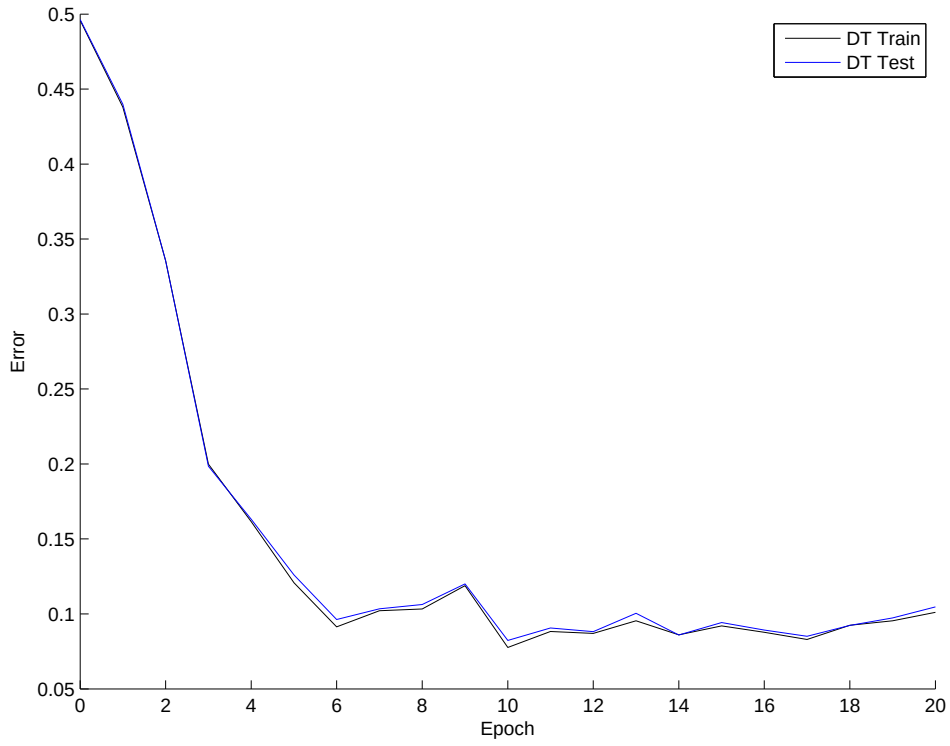

Figure 2: A deep target (DT) algorithm is used to train a simple autoencoder network of threshold gates thus purely comprised of non-differentiable transfer functions. The y axis correspond to the average Hamming error per component and per example.

4.2 Training Networks with Sigmoidal Units

Here we use the deep target approach to train a standard neural network classifier with the sigmoid transfer function in the lower layers, soft-max transfer function in the last layer, and cross-entropy error for multi-class output.

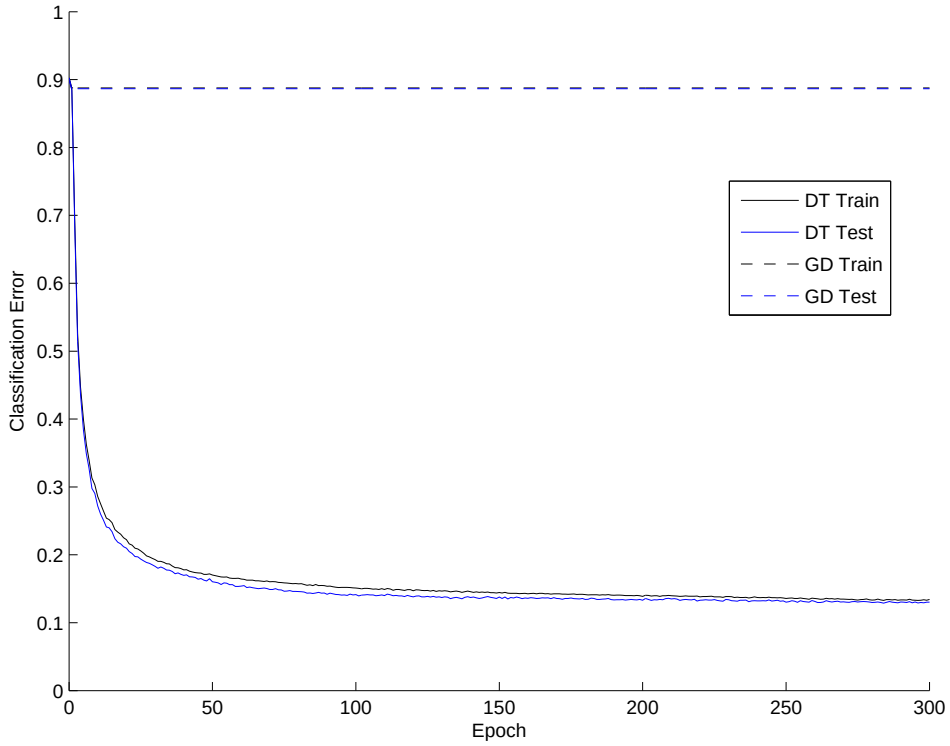

Figure 3: A deep target (DT) algorithm is used to train a neural network with 14 hidden layers on the MNIST dataset. Gradient descent (GD) with backpropagation is unable to update the weights after the first couple epochs due to machine precision limitations.

$$E = \sum_{i=1}^{m} \Delta^l(y_i, W(x_i)) \tag{9}$$

$$\Delta^l(a,b) = -\sum_{k=1}^{n_l} a_k \log b_k \tag{10}$$

$$\Delta^{j\neq l}(a,b) = -\sum_{k=1}^{n_j} a_k \log b_k + (1-a_k)\log(1-b_k) \tag{11}$$

We use the cross-entropy error as the distortion function Δ^j and gradient descent as the partial optimization algorithm Θ. Specifically, updates to the weights in F_j are made with a simple step down the gradient of the cross-entropy error. The learning rate is $0.1(\frac{1}{n_{j-1}})$.

The outer loop of the algorithm makes updates in batches of 100. Updates are made to all layers sequentially $1, 2, ..., l$ before moving on to the next batch, and one cycle through the training data constitutes an epoch. The primary advantage of batch operations is that we can compute targets for the entire batch at once. The set of possible targets $\mathcal{S}^j$ is the same for all examples x_i in the batch – $\mathcal{S}^j$ comprises all batch activation vectors h_i^j, plus a set of 1000 random binary vectors where each bit is independent and 1 with probability 0.5. The weights in F_j are updated once per batch by using the gradient of the average error.

We apply our algorithm to a benchmark classification task. The MNIST handwritten digit dataset contains 28-by-28 pixel greyscale images [14]. There are 60000 training examples and 10000 test examples.

The architecture is a feed-forward network with 14 layers and the following numbers of units in each layer: 784 (input), 1000, 800, 500, 300, 300, 300, 300, 300, 100, 100, 100, 100, 100, 10. Each unit is fully connected to the n_{j-1} units in the layer below it, plus a bias term. The weights for $\mathbb{F}_j$ are initialized randomly from the uniform distribution $U(-\frac{4\sqrt{3}}{\sqrt{n_{j-1}}}, \frac{4\sqrt{3}}{\sqrt{n_{j-1}}})$ except for the bias terms which are all initially zero. The parameter $4\sqrt{3}$ helps ensure that perturbations to the input produce perturbations in the output. The trajectory of the classification error is plotted in Figure 3.

For comparison, we also attempted to train the same architecture using gradient descent with backpropagation. We initialized the weights in the same way, and used MATLAB code by Carl Rasmussen for minimization with conjugate gradients [16]. This was done in batch mode so that 3 linesearches were done for each batch of 1000 training examples, and one cycle through the entire training set constitutes an epoch.

Computations were done in MATLAB on a server with 8 processors (MATLAB can automatically take advantage of multiple cores for some operations). The two algorithms were trained for 300 epochs. In real time, the deep target algorithm ran for 96 hours and the gradient descent algorithm ran for 54.

In a similar experiment, we used the same deep target algorithm to train an architecture of 21 layers. Each of the 20 hidden layers has 300 units. Again we plot the trajectory of the classification errors (Figure 4).

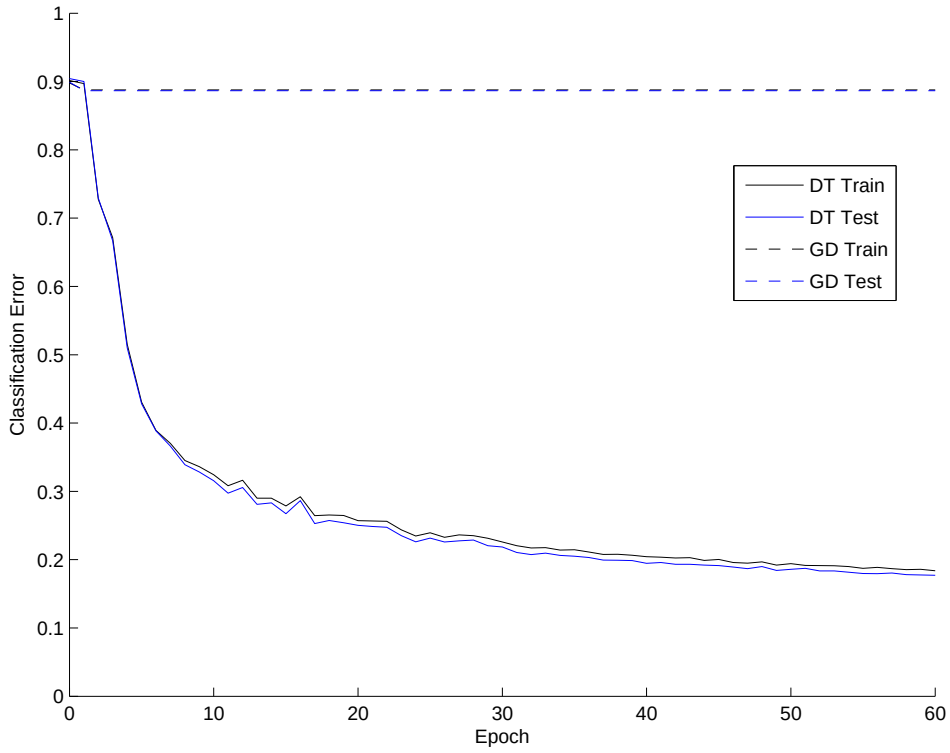

Figure 4: A deep target (DT) algorithm is used to train a neural network with 21 layers on the MNIST dataset. Each of the 20 hidden layers has 300 units. Gradient descent (GD) with backpropagation again fails.

5 Discussion

We have described a general class of algorithms, the deep target algorithms, that can be used to train essentially any feedforward architecture. We have demonstrated this capability in two cases where gradient descent with backpropagation fails: (1) when the transfer function is non-differentiable; and (2) when the number of layer is too large to allow the backpropagation of gradients.

The accuracy of the MNIST classifiers trained with deep target algorithms in Figure 3 and Figure 4 is not impressive in itself, as similar networks with only three layers are known to achieve 2% test error. However these experiments do show that the DT approach is capable of training large networks, where backpropagation is useless, by providing targets to all, or a sufficient subset, of the hidden layers. Impressive accuracy results ought to be sought in situations where shallow architectures cannot achieve such results, and this requires very difficult learning problems and/or very strong architectural constraints (e.g. local connectivity). Work along these lines is currently in progress [Anonymous].

In the MNIST experiments each layer is updated individually using deep targets, but as mentioned earlier, the DT framework also allows updating multiple layers at a time. We can train any subset of adjacent layers simply by providing a set of inputs and targets, then training this section of the network with a multi-layer optimisation algorithm Θ such as backpropagation. In fact, we have observed this strategy to be useful for additional training of the upper layers of the 14 layer MNIST classifier in figure 3. The network has a test error of 13% after 300 iterations of updating each layer individually; if we continue training all layers together with backpropagation for another 200 iterations (in effect training the top layers), this test error drops to 3.7%.

In short, we believe that DT algorithms have the potential for addressing some of the most important problems in deep learning and, as described in Section 3, many interesting directions remain to be explored.

References

[1] P. Baldi. Gradient descent learning algorithms overview: A general dynamical systems perspective. *IEEE Transactions on Neural Networks*, 6(1):182–195, 1995.

[2] P. Baldi. Boolean autoencoders and hypercube clustering complexity. *Designs, Codes, and Cryptography*, 2012.

[3] P. Baldi and G. Pollastri. The principled design of large-scale recursive neural network architectures–DAG-RNNs and the protein structure prediction problem. *Journal of Machine Learning Research*, 4:575–602, 2003.

[4] Yoshua Bengio and Yann LeCun. Scaling learning algorithms towards AI. In L. Bottou, O. Chapelle, D. DeCoste, and J. Weston, editors, *Large-Scale Kernel Machines*. MIT Press, 2007.

[5] H.D. Block, S.A. Levin, and American Mathematical Society. *On the boundedness of an iterative procedure for solving a system of linear inequalities*. American Mathematical Society, 1970.

[6] Léon Bottou. Online algorithms and stochastic approximations. In David Saad, editor, *Online Learning and Neural Networks*. Cambridge University Press, Cambridge, UK, 1998.

[7] Léon Bottou. Stochastic learning. In Olivier Bousquet and Ulrike von Luxburg, editors, *Advanced Lectures on Machine Learning*, Lecture Notes in Artificial Intelligence, LNAI 3176, pages 146–168. Springer Verlag, Berlin, 2004.

[8] Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio. Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*, 11:625–660, February 2010.

[9] P. Frasconi, M. Gori, and A. Sperduti. A general framework for adaptive processing of data structures. *IEEE Transactions on Neural Networks*, 9(5):768–786, 1998.

[10] A. Gelfand, L. van der Maaten, Y. Chen, and M. Welling. On herding and the perceptron cycling theorem. *Advances of Neural Information Processing Systems (NIPS)*, 23:694–702, 2010.

[11] G.E. Hinton, S. Osindero, and Y.W. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006.

[12] G.E. Hinton and R.R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504, 2006.

[13] H. Larochelle, Y. Bengio, J. Louradour, and P. Lamblin. Exploring strategies for training deep neural networks. *The Journal of Machine Learning Research*, 10:1–40, 2009.

[14] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278 –2324, November 1998.

[15] M. Minsky and S. Papert. *Perceptron*. MIT Press, 1969.

[16] Carl Edward Rasmussen and Christopher K. I Williams. *Gaussian processes for machine learning*. MIT Press, Cambridge, Mass., 2006.

[17] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[18] F. Rosenblatt. *Principles of neurodynamics*. Spartan Book, 1962.

[19] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing. Vol 1: Foundations*. MIT Press, Cambridge, MA, 1986.

[20] B. Scholkopf and A. J. Smola. MIT Press, Cambridge, MA, 2002.

[21] B. Widrow and M.E. Hoff. Adaptive switching circuits. 1960.