Continuously Constructive Deep Neural Networks

Ozan İrsoy and Ethem Alpaydın

***Abstract*— Traditionally, deep learning algorithms update the network weights, whereas the network architecture is chosen manually using a process of trial and error. In this paper, we propose two novel approaches that automatically update the network structure while also learning its weights. The novelty of our approach lies in our parameterization, where the depth, or additional complexity, is encapsulated continuously in the parameter space through control parameters that add additional complexity. We propose two methods. In tunnel networks, this selection is done at the level of a hidden unit, and in budding perceptrons, this is done at the level of a network layer; updating this control parameter introduces either another hidden unit or layer. We show the effectiveness of our methods on the synthetic two-spiral data and on three real data sets of MNIST, MIRFLICKR, and CIFAR, where we see that our proposed methods, with the same set of hyperparameters, can correctly adjust the network complexity to the task complexity.**

***Index Terms*— Constructive learning, deep learning, neural networks.**

I. Introduction

DEEP learning involves multiple layers of nonlinear information processing [1]. This allows learning architectures that implement functions as repeated compositions of simpler functions, thereby learning the layers of abstraction with better generalization and representation capacity. Recent advances in efficient training of deep neural networks using larger data sets, GPUs, or better optimization techniques enabled their application to many problems, ranging from speech and vision to natural language processing, possibly combining information from multiple sources [2].

Though deep learning is helpful, having too many layers may be problematic. First, when there are more layers/units/weights, space and computational complexity are higher. Second, with more free parameters, there is a higher risk of overfitting, which needs to be combated using regularization methods, such as weight sharing, weight decay, and dropout. Third, when the network is deep, there is the problem of vanishing/exploding gradients when the error is backpropagated over many layers [3], and one relatively simple mechanism is to introduce the gating mechanisms that allow passing signals with minor modification, as in LSTM or GRU for sequence learning [4], [5].

Starting from the 1990s, there have been many approaches to optimize the network architecture, ranging from the early incremental methods of adding hidden units one by one [6], or starting from a large network and pruning it [7], to more complex recent approaches, such as evolutionary algorithms [8], reinforcement learning [9], and boosting-style methods [10].

This paper, here, has a similar goal of learning the network architecture from data. The main difference in this paper is that instead of searching over a discrete space of all architectures, we parameterize our models in such a way that the notion of complexity, or depth, itself is continuous, making the model end-to-end differentiable and allowing gradient descent to search over the architectures in addition to their parameters.

We propose two methods for continuously constructing deep neural networks. In tunnel networks, inspired from highway networks [11], associated with each hidden unit is a continuous parameter, and if this parameter is not active, the unit just copies its input to its output bypassing the nonlinearity. We start with a network with many layers with all its hidden layers inactive; during gradient descent, the parameter may change, and the corresponding unit becomes active; this implies adding a new layer of nonlinearity effectively increasing the depth of the network.

In our second method of budding perceptrons, inspired by budding trees [12], there is a parameter associated with each layer, indicating whether further nonlinear processing is needed. Initially, we start with a single layer, and during learning with gradient descent, when needed, this parameter may become active, which causes the creation of another full layer, effectively increasing the depth of the network.

We start by a survey of algorithms that adapt network structure during learning in Section II, after which we explain our two methods, in detail, in Section III. Our experiments are discussed in Section IV, where we first show our didactic results on the synthetic, 2-D two-spiral data and, then, on the larger, real-world data sets of MNIST, MIRFLICKR, and CIFAR, also comparing our methods with baseline constructive methods. Our results indicate that both methods can grow networks automatically to learn these problems; with the same set of hyperparameters, the complexity of the learned network matches with the complexity of the underlying task. For simple problems, small networks are constructed, and the depth increases, as the complexity of the task it faces increases. We conclude this paper in Section V.

II. Related Work

The optimization of the network structure is typically done manually using trial and error. A straightforward way of searching for the architecture is to treat the number of layers and units as hyperparameters and optimize over these hyperparameters [13], [14].

Manuscript received March 30, 2018; revised September 21, 2018 and January 10, 2019; accepted May 12, 2019. Date of publication June 24, 2019; date of current version April 3, 2020. *(Corresponding author: Ozan İrsoy.)*

O. İrsoy is with Bloomberg LP, New York, NY 10022 USA (e-mail: oirsoy@bloomberg.net).

E. Alpaydın is with the Department of Computer Engineering, Boğaziçi University, 34342 Istanbul, Turkey (e-mail: alpaydin@boun.edu.tr).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNNLS.2019.2918225

Early work on searching for the optimal network focused on incremental methods. Most such methods fix the depth to one, so the problem reduces to a search over the number of hidden units [6], [15], [16]. Pruning methods start with a large network and remove units/connections if they are deemed unnecessary [7], [17], [18]. After training a network, weights may be set to zero if the change in generalization error is small. Weight decay is equivalent to L2 regularization and other regularization, and the Bayesian techniques are also used for model selection in neural networks [19], [20].

There are also hybrid methods that allow both addition and deletion of hidden units and layers [21], especially for radial-basis-function networks, where hidden units have local responses [22]. Finding the right network can be defined as a search in the space of possible networks with operators that add/remove units/layers, and different search methods can be used here [23]. One can also use evolutionary search [8], [24].

In neural architecture search (NAS), reinforcement learning is used, whereby a controller network that generates candidate neural network descriptions is trained to optimize the expected accuracy [9]. More recently, a boosting-style method, named AdaNet, is used to incrementally grow architectures while learning the parameters [10]; AdaNet also provides data-dependent generalization guarantees of their method.

III. Proposed Methods

A. Tunnel Networks

Our first approach involves defining a parameter, $g \in [0, 1]$, for each hidden unit that determines how much it acts as a nonlinear unit versus how much it just copies its input to its output without performing any transformation on it

$$y(x) = g \cdot \sigma(w^T x + b) + (1 - g) \cdot x. \tag{1}$$

If $g = 0$, the unit just copies its input to its output; if $g = 1$, the output is given by the output of the nonlinear $\sigma(w^T x + b)$ with its internal parameters of w and b, namely, the weights and the biases. We have a layer of such units, which takes the (elementwise) convex combination of a perceptron layer and the identity mapping.

We can consider g as a gating parameter that chooses between the identity and the nonlinear transformation. We can also interpret it as a complexity control parameter that chooses between the simpler identity and the more complex nonlinear hidden unit. As long as it is 0, we have a simple model, but if g gets larger during learning, the nonlinearities start to kick in, the network becomes more complex, and one can interpret this as growing a neural network constructively.

To favor simplicity, we use an objective function that penalizes positive values for g

$$J(X) = \sum_{x \in X} E(x) + \lambda \sum_{l} g_l \tag{2}$$

where x is an instance from the data set X, $J(\cdot)$ is the overall objective function to be minimized, $E(\cdot)$ is the error (loss) function over an instance (e.g., cross entropy), l is the index of each unit in all layers, and $\lambda \geq 0$ is the tradeoff parameter adjusted using cross validation. During stochastic gradient descent (SGD), we update all w_l and b_l in the network and also the g_l values. During learning, just as it is possible that g_l moves away from 0 and adds a nonlinear hidden unit, it is also possible that because of regularization, it moves back to 0, effectively pruning it back.

Tunnel networks are inspired from highway networks [11], where g is a function of x and as such works as an input-dependent gating model

$$g(x) = \sigma_g\left(v_g^T x + c_g\right). \tag{3}$$

We can consider our tunnel network as a special case of highway networks, where the gating is constant. In highway networks, a negative initial bias for c_g is used, such as -2 or -4 [11], which results in starting values for $g(x)$, which are close to 0; hence, each individual layer starts close to the identity. As $g(x)$ values move away from 0 when v_g and c_g are updated, the network response becomes nonlinear. A highway network where the $g(x)$ values are regularized to be close to 0 (by having negative c_g and very small v_g) would work similar to our tunnel network.

In Network Slimming [25], a similar multiplicative scaling factor is applied to each convolutional channel, whereas in tunnel networks, we apply it to each unit. In [26], a similar approach is used to select other substructures, such as neurons, groups, or residual blocks. Incomplete dot product [27] is another approach to dynamically adjust and limit the incoming contributions to each unit. Adaptive computation graphs is yet another architecture that is similar to highway networks, in which the computation paths are dynamically chosen as a function of the input [28]; in this regard, they can be seen as an extension of the highway networks.

Another related architecture is the deep residual network [29], where there is no explicit gating and the transformation and the input copy are summed directly, that is, in 1, both g and $(1 - g)$ terms are set to 1 (for a residual block of a single-layer perceptron). Deep residual networks have won several competitions, including ImageNet and COCO challenges [29], indicating the strength of methods that can adjust network complexity to that of data during training.

B. Budding Perceptrons

Our second approach is inspired by the budding tree that we proposed before, and it is a tree where complexity is softly parameterized [12]. Unlike a regular tree, where a node is either a decision node or a leaf node, in a budding tree, each node m has a leafness parameter $\gamma_m \in [0, 1]$ and is both a leaf node with weight γ_m and an internal node with weight $1 - \gamma_m$. In the beginning, the root is a leaf (its γ is equal to 1), but when it is updated (using gradient descent) to get smaller, its two children are created, effectively growing the tree. The error function has a regularization term to penalize γ_m that are smaller than 1.

In the budding perceptron, we propose here, the approach is the same, except that we have perceptron layers in a network instead of nodes of a tree. Every perceptron layer has a complexity parameter γ_m that is initially 1, and if γ_m gets smaller, another full perceptron layer is created next to it.

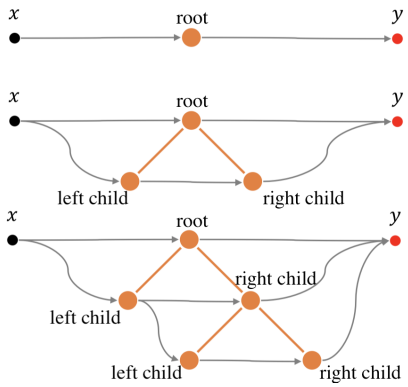

Fig. 1. Illustration of information flow in budding perceptron. (Top) Initially, budding perceptron has a single root node, which makes it equivalent to a multilayer perceptron with one hidden layer. (Middle) When the root node decides to split into two children, another path of information flow is introduced, which is of two perceptron layers. At this stage, the architecture is equivalent to a convex combination of a one-layer and a two-layer perceptron networks. (Bottom) Process can continue to introduce depth by further splitting of the children.

More formally, a budding perceptron network uses a tree structure to implement the composition of individual perceptron layers to make up a deep network. Its recursive definition is as follows:

$$y_m(x) = (1 - \gamma_m) \cdot y_{mr}(y_{ml}(x)) + \gamma_m \cdot \sigma(w_m x + b_m) \quad (4)$$

where ml and mr are the left and right children of m. $\gamma \in [0, 1]$ is the complexity parameter, and σ is the nonlinearity function used in the hidden layers. If γ_m is 1, we only have the current perceptron layer; however, for $\gamma_m < 1$, its children layers are also created, and every parent becomes a convex combination of a perceptron and the composition of two similar functions implemented by its children (see Fig. 1).[1]

Observe that the above-mentioned parameterization defines an infinite complete binary tree since every node in the tree has two children. Therefore, the parameter space contains feedforward networks of all depths. In practice, since all γ_m values start from 1, and during training, only finitely many of them will assume values that are smaller, the recursion is guaranteed to end when a node with $\gamma = 1$ is encountered, and its children will not be evaluated.

We add a regularizer term to prefer smaller networks

$$J(X) = \sum_{x \in X} E(x) + \lambda \sum_{m \in T} (1 - \gamma_m) \quad (5)$$

where T denotes the set of all nodes in the tree.

During gradient descent, we update all the perceptron weights, w_m and b_m, and also γ_m. Initially, a single root node is equivalent to a single layer of perceptron with its γ equal to 1. During training, when its γ is updated to get smaller, two perceptron layers are created as children (with their own γ_m initially 1), making up a deeper network. Then, in turn, those two layers can spawn new layers, effectively adding more and more layers. It is important to note that during gradient descent, γ_m values move away from 1, adding new layers, and then, as the network weights converge, γ_m may move back to 1, effectively eliminating previously added layers.

Note that with the above-mentioned definition, there is no information passed on from parent weights (e.g., w_m) to weights of the children (e.g., w_{ml}). This means that when a leaf is split into two nodes, w_{ml} and w_{mr} need to be learned from scratch. This might slow down convergence considerably and impact learning dynamics negatively. To help with information propagation when splitting, we share (tie) the weights of one of the children with the parent.

Note that unlike the tunnel networks and similar work that were mentioned in Sections II and III-A, budding perceptrons can both grow (without bound, beyond its initial configuration) and shrink during training, which sets it apart from the pruning-based approaches.

C. Measuring Network Complexity

Since we have notions of complexity which are soft, discrete measures, such as the depth or the number of hidden units, are not directly applicable. For instance, a ten-layer tunnel network that has all g values equal to 0 implements an identity function, whereas the same size tunnel network with g values equal to 1 is a multilayer perceptron with depth 10.

To this end, we propose soft criteria for measuring the complexity of tunnel networks, highway networks, and budding perceptrons that we use in our experiments. For tunnel networks, it is simply the layerwise (or total) sum of all g values

$$s_l = \sum_{k}^{K} g_{lk} \quad \text{for layer } l \in L \text{ and } s = \sum_{l \in L} s_l \quad (6)$$

where l indexes the layer out of the set L of all layers and k indexes a hidden unit out of the K hidden units of layer l.

For highway networks, $g_{lk}(x)$ replaces the constant g_{lk}, and since it is a function of the input, we compute it as the average over all input instances (in the validation set)

$$s_l(X) = \sum_{k}^{K} \frac{1}{|X|} \sum_{x \in X} g_{lk}(x) \quad \text{and } s(X) = \sum_{l \in L} s_l(X). \quad (7)$$

For budding perceptrons, in addition to the tree size (node counts), we define a *soft tree size* that incorporates the leafness parameters γ_m as well

$$s_m = 1 + (1 - \gamma_m)(s_{ml} + s_{mr}) \text{ for node } m \text{ and } s = s_{\text{root}}.$$

This soft size gives us a notion of effective depth of a budding perceptron network since every node itself acts as a single (nonlinear) perceptron layer.

Note that the soft criteria that we propose here do not correspond to memory or computational complexity; if the gating value is nonzero, the corresponding unit or layer still needs to be stored and computed. It is, however, an indication of what percentage of the large network effectively participates in generating the overall approach and as such is an indication of how effective is the pruning/constructive method.

[1]We apologize for the possible confusion. In the tunnel network, $g_l = 0$ is the simpler alternative, and $g_l = 1$ is the more complex, whereas in the budding perceptron, $\gamma_m = 1$ is the simpler alternative, i.e., a perceptron, and $\gamma_m = 0$ is the complex alternative of two children perceptrons. This is because we wanted to stick to the definitions in the original models of highway networks and budding trees from which we are inspired.

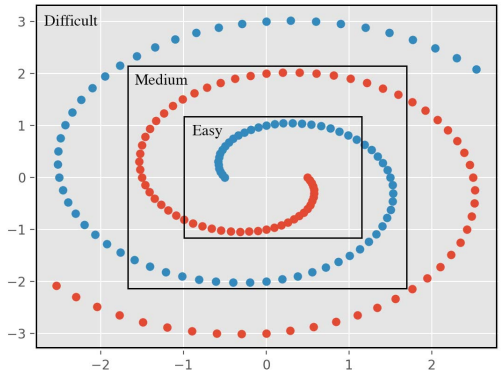

Fig. 2. Three different variants of the two spirals data set in increasing complexity. Different colors show different class labels.

IV. Experiments

A. Results on Two Spirals

1) Data: We use the two-class spirals data set [30] for didactic experimentation. Our claim is that our constructive methods can adjust the complexity of the learned network to the complexity of the problem. To test this, we created three variants, namely, easy, medium, and hard (see Fig. 2), where the easy task is linearly separable, and the medium and hard variants need deeper networks.

2) Training: First, it is important to stress that we use the same hyperparameters for different variants of the two spirals and we want to see if the constructive algorithm can find the right complexity just by changing the data.

We train all models with SGD using the Adam update rule, which modifies learning rates and momentum rates for each parameter during training [31]. We use a binary cross-entropy objective function $[\sum_x \mathbf{1}(r(x)=1)\log(y(x)) + \mathbf{1}(r(x)=0)(1-\log(y(x))$, where $r(x) \in \{0,1\}$ is the true label of x] on top of a sigmoid output layer. We do not employ minibatching for the two spirals data, but we use the purely online setting. We schedule the learning rate and determine the length of training as follows. Training continues until there is no improvement on the development set (which is the same as the training set for this synthetic data set) for 20 epochs after which we use 0.3th of the original learning rate. Next time, in which there is no improvement for 20 epochs, we switch to 0.1th of the original learning rate. Finally, training stops when there is no further improvement for another 20 epochs.

We assign ten hidden units to each layer of tunnel and highway networks and budding perceptrons. For tunnel networks, we use a total (maximum) number of ten layers. We use a penalty of 0.001 for λ in L1 regularizers. For learning rate, we pick the smallest one from the set {0.0001, 0.0003, 0.001, 0.003} that yields zero training error, which results in 0.003 for tunnel networks and 0.001 for budding perceptrons. We employ a standard L2 regularization penalty of 10^{-5} on network weights, which is typically useful for avoiding parameter explosions.

We tried two approaches for making learning rate dependent on depth. In the first case, we simply use the same learning rate on all layers. We found that this typically results in similar behavior for all layers (in terms of soft size growth during training), and the layers are not distinguished. To better differentiate the layer behavior, we assigned decreasing learning rates to each layer, where each layer has 3/4th of the learning rate of the previous layer. This results in a more incremental behavior, where lower layers grow more quickly and higher layers adapt slowly. Furthermore, as we will see in the experiments, the learning dynamics, in this case, encourages lower layers to be used early on, and if higher layers are not necessary, they are not used at all, allowing them to be actually pruned from the network after training.

All networks use rectified linear units (ReLu) $(\max\{0, x\})$ for their elementwise nonlinearities [32]. Since all of the models that we evaluate require input and output dimensionality to be the same, all architectures have an additional linear projection layer at the very beginning. For this instance, since we have ten hidden units and the input space is 2-D, we have a 2×10 matrix that linearly projects an input instance to a 10-D space (which is also learned during backpropagation).

3) Results: Our plots using tunnel networks are shown in Fig. 3, and budding perceptrons are shown in Fig. 4.

In the top row of Fig. 3, we see the training error rates for easy, medium, and difficult cases, respectively. In all cases, tunnel networks are able to converge to an error of zero. However, as the problem becomes more difficult, zero error is reached more slowly—it takes 2, 23, and 91 epochs to reach zero error for easy, medium, and difficult cases, respectively.

In the middle row, we plot the soft sizes for layers, each layer in a separate color. We see that early on, the constructive method introduces more layers than necessary, but as learning proceeds, those extra ones are pruned back. The peak is reached around where the models reach zero error; from then on, the objective improves mostly by reducing the regularization term, that is, simplifying the network. We believe that this is an interesting end result of defining complexity using continuous parameters. There are no distinct, separate phases of structure growing and pruning nor structure learning versus parameter learning; the network structure is learned, by growing or pruning, coupled with the learning of the parameters. We also observe that typically, the model uses earlier layers more; this is because we set our learning rates in a decreasing fashion. In experiments where we set the learning rates equally, we observed that all layers are used in similar amounts.

Finally, the bottom row shows the total complexity. We see that as the problem becomes more difficult, the model grows more in terms of its soft size. This suggests that the models indeed grow networks of different complexities depending on the complexity of the problem itself, with the same set of hyperparameters. For instance, soft sizes of the tunnel network are around 2, 10, and 15 for the easy, medium, and difficult tasks, respectively.

In the top row of Fig. 4, we see the training errors of budding perceptrons on easy, medium, and difficult tasks, respectively. Similar to tunnel networks, budding perceptrons can reach zero error, and the time to reach there increases with difficulty (about 2, 30, and 125 epochs, respectively).

The middle and bottom rows show the hard and soft sizes of the budding perceptrons. Similar to tunnel networks, trees

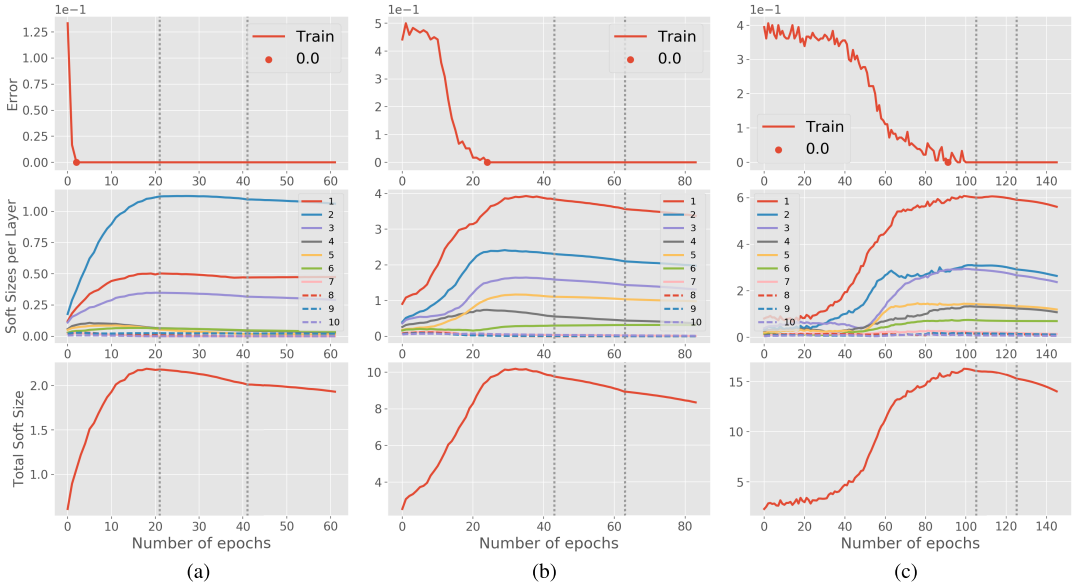

Fig. 3. Results of tunnel networks on different variants of two spirals. We show error rate (top) and total g values for each layer (middle) and for all layers (bottom). In all, vertical bars show where learning rate shrinkage occurs. During training, complexity increases but may start decreasing back again. (a) Easy. (b) Medium. (c) Difficult.

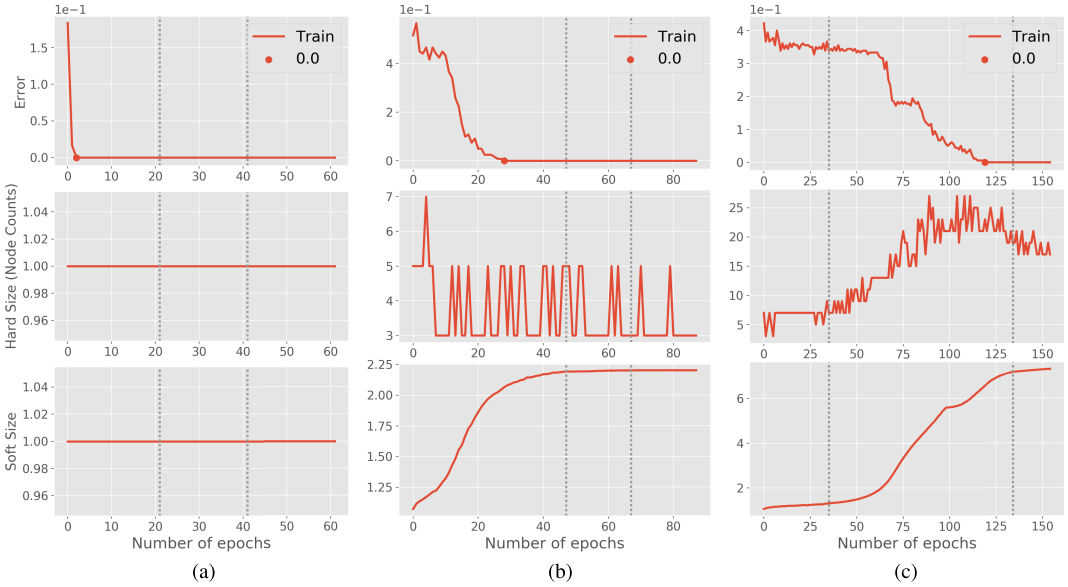

Fig. 4. Results of budding perceptrons on the different variants of two spirals. We show error rate (top), hard tree sizes (middle), and soft tree sizes (bottom). (a) Easy. (b) Medium. (c) Difficult.

grow to different sizes depending on the complexity of the problem. Budding perceptron uses the smallest possible size of 1 node on the easiest task, whereas it uses soft sizes of roughly 2.2 and 7 on the medium and hard variants.[2]

We see the discriminants learned for the different variants using the budding perceptron in Fig. 5. We observe that the decision boundary (implemented by the same constructive model with the same set of hyperparameters) becomes more complex as needed, as the problem becomes harder (and in the easiest case, it is linear as expected).

[2]Note that because of our parameterization, the smallest depth that we can have is 1 for budding perceptrons even though the problem can be solved with a depth of 0. A parameterization that started with the identity mapping at the root node (instead of a nonlinear perceptron) would allow the model to assume 0 depth.

Tunnel networks also find similar boundaries adapted to the task. We have also tested highway networks on the two spirals data set. We cannot include those figures here because of lack of space, but we find that the highway network proper requires almost twice as many units as tunnel networks; when a regularization term is added (so that highway units act as tunnel units), the number of units becomes comparable.

B. Results on Digit Recognition

1) Data: Next, we evaluate our models on MNIST that contains 60 000 training and 10 000 test examples of 28×28 handwritten digits from ten classes. We randomly partition the original training set into training and validation sets with a ratio of 5:1. We also define the easier binary task of separating digits 0 from 1, discarding instances from all other classes.

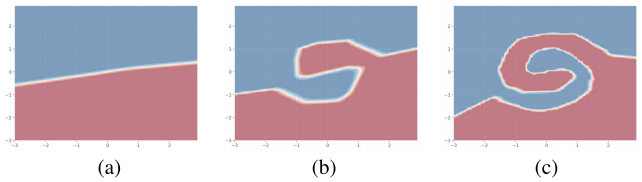

Fig. 5. Discriminants learned by budding perceptrons on two spirals data. (a) Easy. (b) Medium. (c) Difficult.

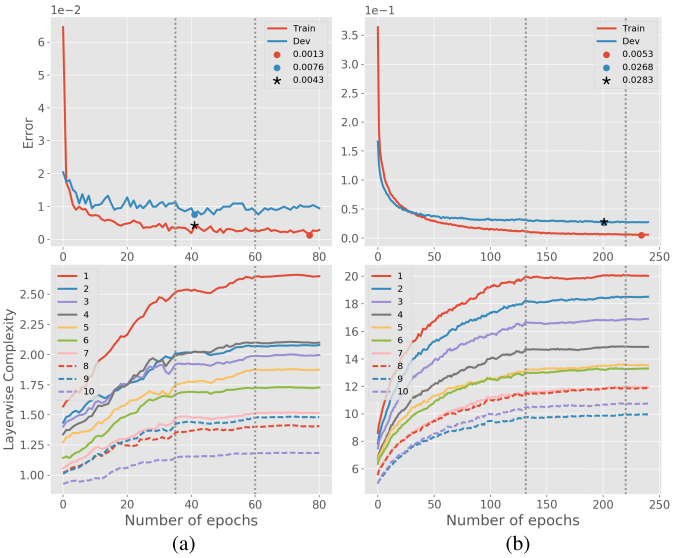

Fig. 6. Results on MNIST using highway networks. Left and right columns: easy task (two classes) and original task (ten classes). Top and bottom rows: error rates and layerwise complexities for each layer. Training and validation errors (shortened as dev for development) are shown in red and blue curves. The best training and validation errors are marked with a dot. The test set error rate of the best performing model (on the validation set) is marked with an asterisk. (a) Two class. (b) Ten class.

2) Training: We use the standard multiclass cross entropy with a softmax output layer. We use 100 hidden units for each layer. Learning rate and L1 regularizer coefficient are fixed to 0.0003 and 0.001, respectively. An additional dropout regularizer is used at the input layer with a probability of 0.25 [33]. All other hyperparameters are the same as before.

3) Results: Our plots using highway networks, tunnel networks, and budding perceptrons are shown in Figs. 6–8, respectively. We perform early stopping, i.e., the best model is chosen over all iterations based on the validation error rate and its test performance is evaluated. Training and validation errors are shown in red and blue curves, respectively. Best training and validation errors are marked with a dot. The test set error rate of the best performing model is marked with an asterisk. Note that for budding perceptrons, our size measure is at the scale of depth (soft count of the number of layers), whereas for tunnel networks and highway networks, it is at the scale of hidden units (soft count of the number of hidden units for each layer).

The top row of Fig. 6 shows the training and validation errors with highway networks. On the easy 0 versus 1 task, it is possible to reach almost zero test error—the highway network gets 0.43% (top left). On the ten-class task, training error is also very close to zero, but it reaches a test error of 2.83% suggesting overfitting. The bottom row shows the soft sizes of highway networks for individual layers. On both the easy

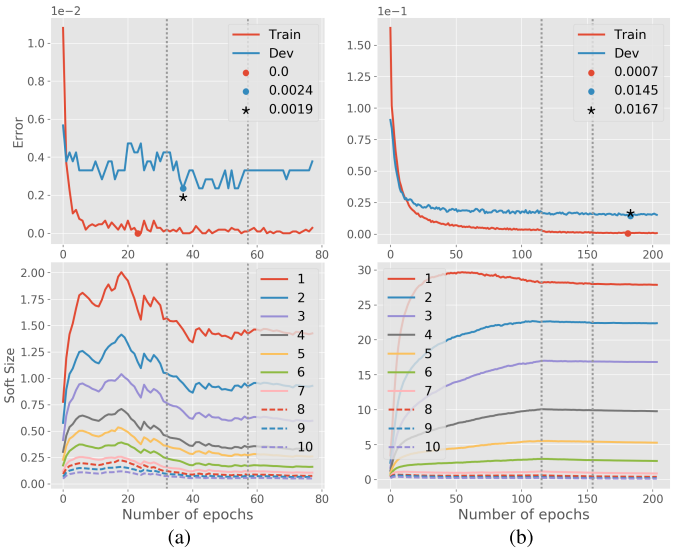

Fig. 7. Results on MNIST using tunnel networks. Top and bottom rows: error rates and layerwise complexities for each layer. (a) Two class. (b) Ten class.

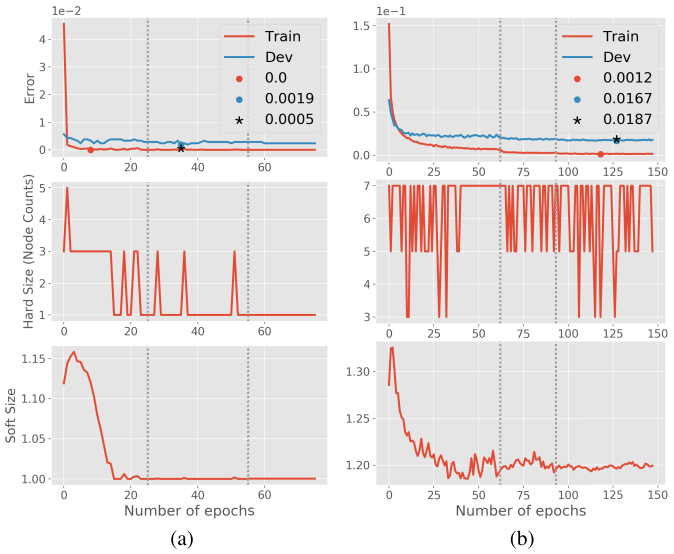

Fig. 8. Results on MNIST using budding perceptrons. Rows: error rates, hard tree size, and soft tree size, respectively. (a) Two class. (b) Ten class.

task (left) and the ten-class task (right), we see that the model grows as it trains more until the learning rate shrinkage happens (shown with vertical dashed lines). However, on neither of the tasks do we observe any pruning back of the model. Since there is no regularization term, highway networks do not have any incentive to reduce the model size.

Fig. 7 shows similar statistics for the tunnel networks. The top row shows the training, validation, and test errors on the two-class (left) and ten-class (right) tasks. The optimal performance is reached in about 40 epochs on the easy case, whereas for the difficult case, it takes about 175 epochs. On the ten-class task, the tunnel network can reach as low as 1.67% error rate on the test set.

In the bottom row, we see layerwise soft sizes for the tunnel network. On the two-class task, layers grow and shrink together very consistently (curves seem to be mere translations/rescalings of one another). This is likely because the two-class case is very close to being, if not exactly, linearly

separable. Hence, there is no need for the network to divide the functionality among different layers differently. We also see that the maximum size ever reached is 2. On the other hand, in the ten-class case (right), the behavior is different, and the peak size can reach 30 for the first layer. In both cases, earlier layers are used more actively.

Fig. 8 shows the results using budding perceptrons. As before, the top row shows error rates: The budding perceptron can reach near-zero test error on the two-class task (left) and 1.84% error rate on the ten-class task (right). Again, the optimum is reached much more quickly for the easy task, in about 40 epochs compared to 125. In terms of hard sizes, which are shown in the middle row, the budding perceptron seems to grow initially and, then, shrinks to the minimum size of 1 for the two-class task (left), whereas it reaches bigger sizes and stays almost flat for the ten-class task (right).

In the bottom row, we observe that for both the easy task (left) and the ten-class task (right), soft sizes grow initially, and then, the model is pruned back. Soft sizes for the ten classes are slightly bigger: 1.2 versus 1. We also observe that the soft and hard sizes for the ten-class task do not look correlated, verifying the need to have the soft measure in addition to hard counts. Both hard and soft sizes reaching the minimum possible value of 1 for the two-class task provide another evidence for it to be very close to linearly separable.

We also observe that the learning rate is important for growth/shrinkage dynamics in all three models. When we shrink the learning rate (shown in vertical dashed bars), we see stabilization over the sizes of the models even though performance continues to improve, and after we shrink it a second time, sizes remain roughly constant.

C. Results on Image Tagging

1) Data: We evaluate our models also on MIRFLICKR data set that has 25 000 Flickr raw images with associated topic labels [34]. We use the partitioning used in [35] with ratios 2:1:2 into training, validation, and test sets. Output labels are the tags of images that are represented as a 38-D vector of binary values (not necessarily one-hot); therefore, the task is posed as a multilabel classification problem, where each instance might have multiple (or no) labels. Images are represented as a 3857-D feature vector, using the preprocessed and extracted image features, as in [35].

To vary the difficulty of the data set and create an easier version of the problem, we use the label hierarchy to collapse the secondary labels [34]. For instance, all of the tags, *people*, *baby*, *female*, *male*, and *portrait*, are collapsed into a single *people* tag. This reduces the label space from 38 to 10.

2) Training: Learning rate scheduling is done similarly as done previously. We use a multilabel binary cross entropy as the objective since labels are not mutually exclusive. This is simply a sum over individual binary cross-entropy losses over all labels. Our output layer, therefore, uses a multidimensional (elementwise) sigmoid function. We use a minibatch with a size of 32. We use 300 hidden units in each layer. We fix the learning rate to 0.0003, which behaves well without oscillatory behavior for any of the models. For budding tree networks and tunnel networks, we use an L1 regularizer penalty of 0.1. We have an additional dropout regularization at the input layer with a probability of 0.25 [33].

3) Performance Measures: In addition to the total average error rate (summed over all labels), we use the Macro-F1-score that is computed by averaging F1-scores for each individual label over all labels. Model selection is performed using Macro-F1 scores over the validation set.

4) Results: Our plots for highway networks, tunnel networks, and budding perceptrons are shown in Figs. 9–11, respectively. We show the Macro-F1 score on the test set (achieved by the best performing model on the validation set) with an asterisk.

In the top row of Fig. 9, we see the error rates (left), Macro-F1 scores (middle), and layer complexities (right) of highway networks on the easy task. We see that the model can reach a training error rate of 0.0448 per instance, whereas the validation error rate is 1.685 (note that one instance can have more than one error in the multilabel case). We also see that the validation curve stays mostly flat during training. This suggests that the initial features might be too expressive, making it easy to overfit. We see a similar phenomenon for the Macro-F1 scores (top middle), where training Macro-F1 score is 98.95%, whereas it is 55.56% on the validation set.

On the difficult task, the behavior is very similar (bottom left and bottom middle). The error rates are 0.2233 for training versus 3.687 for the validation set, and Macro-F1 scores are of 97.16% for training and 41.04% for the validation set. The model sizes also show similar behavior (top right and bottom right) in terms of how each layer monotonically grows. Counterintuitively, the easy task reaches higher complexity values (for instance, about 175 for layer 1 for the easy task versus around 150 for the difficult task), which might be due to the aggressive early learning dynamics of the highway network.

We see similar statistics for tunnel networks in Fig. 10. In terms of error rates (left column) and Macro-F1 scores (middle column), a large gap between training and validation scores remains. Tunnel networks reach 97.15% and 56.86% training and validation Macro-F1 scores on the easy task and 95.7% and 43.91% on the difficult task. For layer complexities (right column), tunnel networks show a different behavior than highway networks. Instead of using many layers, the tunnel network utilizes mostly only the first two on the easy task (top right) and the first four on the difficult task (bottom right). We see that other layers see a small bump initially, which is pruned afterward. We also see that individual layer sizes (and, hence, the total size) are bigger on the difficult task.

Results using budding perceptrons are shown in Fig. 11. For error rates (first column) and Macro-F1 scores (second column), there is still a large difference between training and validation scores as with the previous two models. Budding perceptrons reach 99.75% and 55.45% training and validation Macro-F1 scores on the easy task and 99.56% and 43.39% on the difficult task. In terms of the number of nodes (third column), the easy task seems to consistently require between seven and three nodes (full layers), mostly spending

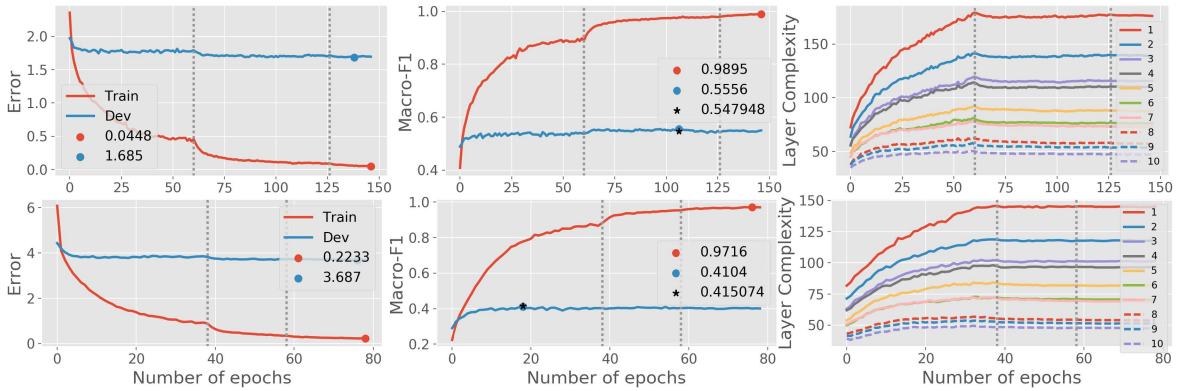

Fig. 9. Results on MIRFLICKR using highway networks. Top and bottom rows use easy task (ten labels) and original task (38 labels), respectively. Columns (left to right): error rates, Macro-F1, and layerwise complexities, respectively.

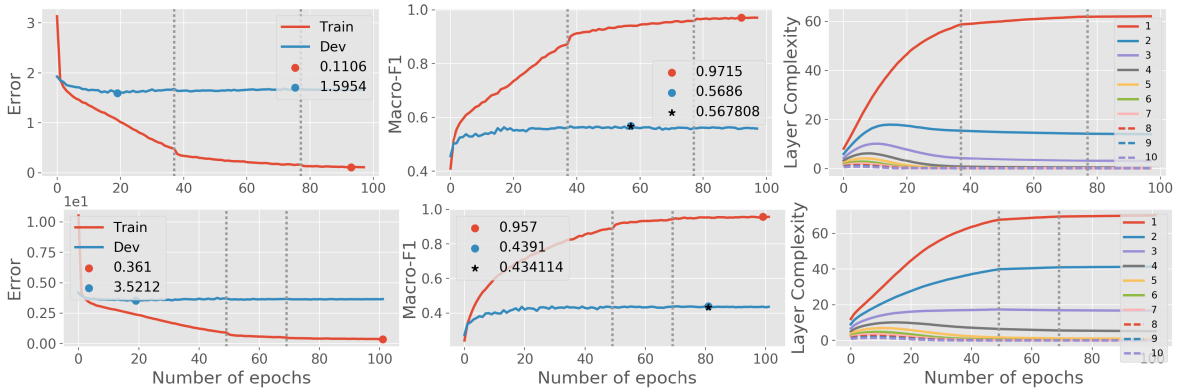

Fig. 10. Results on MIRFLICKR using tunnel networks. Top and bottom rows use easy task (ten labels) and original task (38 labels), respectively. Columns (left to right): error rates, Macro-F1, and layerwise complexities, respectively.

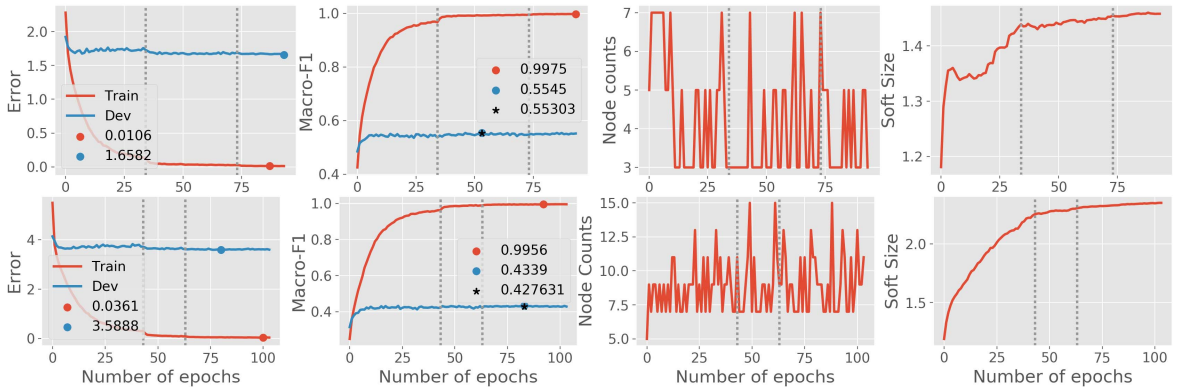

Fig. 11. Results on MIRFLICKR using budding perceptrons. Top and bottom rows use easy task (ten labels) and original task (38 labels), respectively. Columns (left to right): error rates, Macro-F1, hard tree size, and soft tree size, respectively.

time on three and five (top). On the hard task, however, we see both bigger values (always above or equal to 7 after the very first epoch) as well as a tendency for growth. For soft sizes (last column), the easy task shows an initial quick growth and, then, a pruning stage, which is followed by a more stable growth curve (top). On the hard task, we see a more consistent growth that is slowed by learning rate shrinkage (bottom).

D. Comparisons With Baseline Algorithms

We compare our tunnel networks and budding perceptrons with two previously proposed architecture search methods, namely, AdaNet [10] and NAS using reinforcement learning [9] as well as highway networks.

AdaNet works by incrementally proposing new candidate architectures (deeper or at the same depth), which takes the existing layers (with the input vector) as its input, alongside the architecture learned so far. Candidates are evaluated according to an objective function, and the best one is picked. This process is recursively applied until a stopping criterion is met, which finalizes the architecture. We take our base architecture as a ReLu perceptron layer of 50 units, which is used to incrementally grow the architecture. The maximum depth is 10. Since the formulation of AdaNet is in the binary classification setting, we use the one-versus-all approach to extend it to multiclass, and we report the maximum width (number of hidden units) $\times$ maximum depth (number of layers), averaged over these multiple networks. With NAS, a specialized recurrent

TABLE I
COMPARISON WITH BASELINE MODELS ON MNIST (TEN CLASS)

Model	% Error	Complexities	Time
AdaNet	3.75	245 × 5 (avg. max width × depth)	23.7m
NAS	2.23	237.2 × 5 (avg. width × depth)	118.0m
Highway	2.83	154.8 (total soft unit count)	32.9m
Tunnel	1.67	86.9 (total soft unit count)	17.3m
Budding	1.87	1.20 × 100 (soft size × width)	33.0m

TABLE II
COMPARISON WITH BASELINE MODELS ON FLICKR (38 MULTILABELS)

Model	Macro F1	Complexities	Time
AdaNet	20.3	72.3 × 2.3 (avg. max width × depth)	3.8m
NAS	43.1	114 × 3 (average width × depth)	212.0m
Highway	41.5	781 (total soft unit count)	11.5m
Tunnel	43.4	134.4 (total soft unit count)	16.2m
Budding	42.8	2.33 × 300 (soft size × width)	14.6m

neural network is trained using reinforcement learning to propose architectures that are then trained and evaluated. The evaluation performance determines the reward for the recurrent network. In the original work, filter sizes of a convolution network are learned; in our case of fully connected layers, we learn the optimal number of hidden units (layer width) per five layers, and we report the average width × five layers.

Results over MNIST are given in Table I. We observe that the tunnel network outperforms all other models with a test error of 1.67%, and the next best is the budding perceptron. NAS is slightly better than the highway network (2.23% versus 2.83). AdaNet has the poorest performance (3.75%), which may be due to the one-versus-all approach that uses ten binary classifiers instead of a single multiclass classifier. Tunnel and budding perceptrons also learn the smallest networks in terms of the number of nodes.

Table I also shows the total wall clock time for the training of all methods on eight CPU cores in minutes. Note that the total time depends on the time to reach to the best model since all methods use some notion of patience or an early stopping criterion. All, except NAS, seem to perform similarly (for AdaNet, time is averaged per class); NAS is slower due to many runs at each iteration of the controller network.

Results over FLICKR data are presented in Table II. We use a similar setting for AdaNet and NAS. Since this is a multilabel classification setting, we use AdaNet to train 38 binary classifiers (the labels are not mutually exclusive). The top two performers are the tunnel network and NAS (43.4% and 43.1%), followed by the budding perceptron and highway network. Tunnel network again learns the smallest network. In terms of time complexity, we see similar results to MNIST, except, here, AdaNet is very fast but performs poorly; many of the networks seem to converge very quickly to always predicting a negative response.

We also use the CIFAR data set that has 60 000 images of 32 × 32 images with three color channels, each tagged with a class label out of 100 classes [36]. We use 20 coarse-grained labels to pose it as a 20-class classification task. To represent each image, we use the version 3 of the Inception network [37], pretrained on ImageNet [38]. After running the network on each image, we extract the 2048-D response from the third pooling layer, which we give as input to our different learners.

TABLE III
COMPARISON WITH BASELINE MODELS ON CIFAR (20 CLASSES)

Model	% Error	Complexities	Time
AdaNet	19.90	162.5 × 3.75 (avg. max width × depth)	82.0m
NAS	21.67	215.3 × 3 (avg. width × depth)	114.0m
Highway	19.93	271.4 (total soft unit count)	89.3.0m
Tunnel	19.19	350 (total soft unit count)	56.2m
Budding	19.51	1.42 × 300 (soft size × width)	45.9m

Our results are given in Table III, where we observe that tunnel network, budding tree network, and AdaNet perform well (19.19%, 19.51%, and 19.90%) with highway network and NAS following closely (19.93% and 21.67%). In terms of training time, tunnel and budding tree networks seem to complete the soonest.

Note that with both AdaNet and NAS, architecture learning and parameter learning happen in separate stages, and for each architecture, parameters need to be retrained, which requires an entire run of SGD. Since tunnel networks and budding perceptrons parameterize the architecture space continuously, a single run of SGD is used to determine both the architecture as well as the parameters.

V. CONCLUSION

We propose two methods for learning the structure of a deep neural network, where network complexity at the level of hidden unit or layer is coded by continuous parameters. These parameters are adjusted together with the network weights during gradient descent, which implies softly modifying the network structure together with the network weights. Our contributions in this paper are twofold.

1) We propose tunnel networks as a simplified version of highway networks and interpret unitwise g parameters as a soft notion of adding (or pruning) a unit to the network. We show that tunnel networks perform as effectively as highway networks while having much fewer parameters.
2) We propose budding perceptrons that utilize a tree structure to represent a continuous and complete space of feedforward networks of arbitrary depth and optimize over this space using gradient descent. To the best our knowledge, this is the first constructive method that operates continuously and fully jointly, rather than making individual decisions to add or prune layers in stages.

Our experiments on the synthetic two-spiral data illustrate how tunnel networks and budding perceptrons can adapt to tasks of different complexities using the same set of hyperparameters, by adapting the number of units for tunnel networks and the number of layers for budding perceptrons.

On real-world tasks of MNIST, MIRFLICKR, and CIFAR, we have observed that tunnel networks achieve better performance by providing a better-regularized model and using fewer numbers of parameters, compared to highway networks—AdaNet and NAS. We also observe that in all three tasks, tunnel networks start by a growing exploratory phase and then shift into a pruning phase, where the size is reduced.

By setting the learning rate in a decreasing manner, we ensure that different layers grow at different paces and are utilized differently. Combined with regularization, this allows

for tunnel networks to keep some unused upper layers linear, allowing the possibility to actually prune them at the end.

Similarly, budding perceptrons have shown comparable or better performance on all three tasks. Compared to tunnel networks, budding perceptrons grow more and prune less. They are able to grow to arbitrary depths instead of having to specify a maximum size. For instance, throughout this paper, we have used a maximum of ten layers for highway and tunnel networks; however, on MIRFLICKR, the budding perceptron sometimes reaches a layer count of 15.

REFERENCES

[1] Y. Bengio, "Learning deep architectures for AI," *Found. Trends Mach. Learn.*, vol. 2, no. 1, pp. 1–127, 2009.

[2] X. Shu, G.-J. Qi, J. Tang, and J. Wang, "Weakly-shared deep transfer networks for heterogeneous-domain knowledge propagation," in *Proc. 23rd ACM Int. Conf. Multimedia*, Oct. 2015, pp. 35–44.

[3] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Trans. Neural Netw.*, vol. 5, no. 2, pp. 157–166, Mar. 1994.

[4] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[5] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," 2014, *arXiv:1412.3555*. [Online]. Available: https://arxiv.org/abs/1412.3555

[6] T. Ash, "Dynamic node creation in backpropagation networks," *Connection Sci.*, vol. 1, no. 4, pp. 365–375, Jan. 1989.

[7] R. Reed, "Pruning algorithms—A survey," *IEEE Trans. Neural Netw.*, vol. 4, no. 5, pp. 740–747, Sep. 1993.

[8] R. Józefowicz, W. Zaremba, and I. Sutskever, "An empirical exploration of recurrent network architectures," in *Proc. 32nd Int. Conf. Mach. Learn. (ICML)*, Jun. 2015, pp. 2342–2350.

[9] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," 2016, *arXiv:1611.01578*. [Online]. Available: https://arxiv.org/abs/1611.01578

[10] C. Cortes, X. Gonzalvo, V. Kuznetsov, M. Mohri, and S. Yang, "AdaNet: Adaptive structural learning of artificial neural networks," 2016, *arXiv:1607.01097*. [Online]. Available: https://arxiv.org/abs/1607.01097

[11] R. K. Srivastava, K. Greff, and J. Schmidhuber, "Highway networks," 2015, *arXiv:1505.00387*. [Online]. Available: https://arxiv.org/abs/1505.00387

[12] O. Irsoy, O. T. Yildiz, and E. Alpaydin, "Budding trees," in *Proc. Int. Conf. Pattern Recognit.*, Aug. 2014, pp. 3582–3587.

[13] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *J. Mach. Learn. Res.*, vol. 13, pp. 281–305, Feb. 2012.

[14] J. Snoek *et al.*, "Scalable Bayesian optimization using deep neural networks," in *Proc. Int. Conf. Mach. Learn.*, Jun. 2015, pp. 2171–2180.

[15] S. E. Fahlman and C. Lebiere, "The cascade-correlation learning architecture," in *Proc. Adv. Neural Inf. Process. Syst.*, 1990, pp. 524–532.

[16] R. Setiono, "Feedforward neural network construction using cross validation," *Neural Comput.*, vol. 13, no. 12, pp. 2865–2877, Dec. 2001.

[17] G. Castellano, A. M. Fanelli, and M. Pelillo, "An iterative pruning algorithm for feedforward neural networks," *IEEE Trans. Neural Netw.*, vol. 8, no. 3, pp. 519–531, May 1997.

[18] P. V. S. Ponnapalli, K. C. Ho, and M. Thomson, "A formal selection and pruning algorithm for feedforward artificial neural network optimization," *IEEE Trans. Neural Netw.*, vol. 10, no. 4, pp. 964–968, Jul. 1999.

[19] J. Ma, T. Wang, and L. Xu, "A gradient BYY harmony learning rule on Gaussian mixture with automated model selection," *Neurocomputing*, vol. 56, pp. 481–487, Jan. 2004.

[20] J. P. Vila, V. Wagner, and P. Neveu, "Bayesian nonlinear model selection and neural networks: A conjugate prior approach," *IEEE Trans. Neural Netw.*, vol. 11, no. 2, pp. 265–278, Mar. 2000.

[21] T. M. Nabhan and A. Y. Zomaya, "Toward generating neural network structures for function approximation," *Neural Netw.*, vol. 7, no. 1, pp. 89–99, 1994.

[22] G.-B. Huang, P. Saratchandran, and N. Sundararajan, "A generalized growing and pruning RBF (GGAP-RBF) neural network for function approximation," *IEEE Trans. Neural Netw.*, vol. 16, no. 1, pp. 57–67, Jan. 2005.

[23] O. Aran, O. T. Yildiz, and E. Alpaydin, "An incremental framework based on cross-validation for estimating the architecture of a multilayer perceptron," *Int. J. Pattern Recognit. Artif. Intell.*, vol. 23, no. 2, pp. 159–190, Mar. 2009.

[24] F. H. F. Leung, H. K. Lam, S. H. Ling, and P. K. S. Tam, "Tuning of the structure and parameters of a neural network using an improved genetic algorithm," *IEEE Trans. Neural Netw.*, vol. 14, no. 1, pp. 79–88, Jan. 2003.

[25] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," in *Proc. Int. Conf. Comput. Vis. (ICCV)*, Oct. 2017, pp. 2736–2744.

[26] Z. Huang and N. Wang, "Data-driven sparse structure selection for deep neural networks," 2017, *arXiv:1707.01213*. [Online]. Available:https://arxiv.org/abs/1707.01213

[27] B. McDanel, S. Teerapittayanon, and H. Kung, "Incomplete dot products for dynamic computation scaling in neural network inference," in *Proc. 16th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, Dec. 2017, pp. 186–193.

[28] A. Veit and S. Belongie, "Convolutional networks with adaptive inference graphs," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, Munich, Germany, 2018, pp. 3–18.

[29] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2016, pp. 770–778.

[30] K. J. Lang and M. Witbrock, "Learning to tell two spirals apart," in *Proc. Connectionist Models Summer School*, Jan. 1988, pp. 52–59.

[31] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*. [Online]. Available: https://arxiv.org/abs/1412.6980

[32] X. Glorot, B. Antoine, and Y. Bengio, "Deep sparse rectifier networks," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 315–323, Jan. 2011.

[33] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," 2012, *arXiv:1207.0580*. [Online]. Available: https://arxiv.org/abs/1207.0580

[34] M. J. Huiskes and M. S. Lew, "The MIR Flickr retrieval evaluation," in *Proc. Int. Conf. Multimedia Inf. Retr.*. Jan. 2008, pp. 39–43.

[35] N. Srivastava and R. R. Salakhutdinov, "Multimodal learning with deep Boltzmann machines," *J. Mach. Learn. Res.*, vol. 15, pp. 2949–2980, Oct. 2014.

[36] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," Univ. Toronto, Toronto, ON, Canada, Tech. Rep., 2009, p. 7, vol. 1, no. 4.

[37] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2016, pp. 2818–2826.

[38] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.

Ozan İrsoy received the B.Sc. degree in computer engineering and the B.A. degree in mathematics from Boğaziçi University, Istanbul, Turkey, in 2012, and the Ph.D. degree from Cornell University in 2017.

He is currently an AI Research Scientist at Bloomberg LP, New York, NY, USA.

Ethem Alpaydın received the B.Sc. degree from Boğaziçi University, Istanbul, Turkey, in 1987, and the Ph.D. degree from the École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, in 1990.

From 1991 to 2019, he was with the Department of Computer Engineering, Boğaziçi University, as an Assistant Professor, an Associate Professor, and then a Full Professor. He is currently a Full Professor with the Department of Computer Science, Özyeğin University, Istanbul. His book *Introduction to Machine Learning* (The MIT Press, Third Edition, in 2014), and has been translated into Chinese, German, and Turkish. His other book, *Machine Learning: The New AI* (The MIT Press, in 2016) as part of the Essential Knowledge Series, and since then has been translated into Japanese, Russian, and Korean.