Deep Cascade Learning

Enrique Marquez, Jonathon Hare, and Mahesan Niranjan

***Abstract*—In this paper, we propose a novel approach for efficient training of deep neural networks in a bottom-up fashion using a layered structure. Our algorithm, which we refer to as Deep Cascade Learning, is motivated by the Cascade Correlation approach of Fahlman [1] who introduced it in the context of perceptrons. We demonstrate our algorithm on networks of convolutional layers, though its applicability is more general. Such training of deep networks in a cascade, directly circumvents the well-known vanishing gradient problem by ensuring that the output is always adjacent to the layer being trained. We present empirical evaluations comparing our deep cascade training with standard End-End training using back propagation of two convolutional neural network architectures on benchmark image classification tasks (CIFAR-10 and CIFAR-100). We then investigate the features learned by the approach and find that better, domain-specific, representations are learned in early layers when compared to what is learned in End-End training. This is partially attributable to the vanishing gradient problem which inhibits early layer filters to change significantly from their initial settings. While both networks perform similarly overall, recognition accuracy increases progressively with each added layer, with discriminative features learnt in every stage of the network, whereas in End-End training, no such systematic feature representation was observed. We also show that such cascade training has significant computational and memory advantages over End-End training, and can be used as a pre-training algorithm to obtain a better performance.**

***Index Terms*—Deep Learning, Convolutional Neural Networks, Cascade Correlation, Image Classification, Adaptive Learning**

I. Introduction

DEEP Convolutional Networks have recently shown impressive results in a range of hard problems in AI, such as computer vision. However there still is not a clear understanding regarding how, and what, they learn. These models are typically trained End-End to capture low and high level features on every convolutional layer. There are still a number of problems with these networks that have yet to be overcome in order to obtain even better performance in computer vision tasks. In particular, one current community-wide trend is to build deeper and deeper networks; during training, these networks fall foul of an issue known as the vanishing gradient problem. The vanishing gradient problem manifests itself in these networks because the gradient-based weight updates derived through the chain rule for differentiation are the products of n small numbers, where n is the number of layers being backward propagated through. In this paper we aim to directly tackle the vanishing gradient problem by proposing a training algorithm that trains the network from the bottom to top layer incrementally, and ensures that the layers being trained are always *close* to the output layer. This algorithm has advantages in terms of complexity by reducing training time and can potentially also use less memory. The algorithm also has prospective use in building architectures without static depth that adapt their complexity to the data.

Several attempts have been proposed to circumvent complexity in learning. Platt [2] developed the Resource Allocating Network (RAN) that allocates the memory based on the number of captured patterns, and learns these representations quickly. This network was then further enhanced by changing the LMS algorithm to include the extended kalman filter (EKF), and by pruning and replacing it improved in both terms of memory and performance [3], [4]. Further, Shadafan *et. al* present a sequential construction of multi-layer perceptron classifiers trained locally by Recursive Least Squares (RLS) algorithm. Compressing, pruning and binarization of the weights in a deep model have also been developed to diminish the learning complexity of convolutional neural networks [5], [6].

In the late 1980s, Fahlman *et. al* [1] proposed the cascade correlation algorithm/architecture as an approach to sequentially train perceptrons and connect their outputs to perform a single classification. Inspired by this idea, we have developed an approach to cascaded layer-wise learning that can be applied to modern deep neural network architectures that we term Deep Cascade Learning. Our algorithm reduces the memory and time requirements of the training compared to traditional End-End backpropagation, and circumvents the vanishing gradient problem by learning feature representations that have increased correlation with the output on every layer.

Many of the core ideas behind Convolutional Neural Networks (CNNs) occurred in the late seventies with the Neocognitron model [7] but failed to fully catch on for computational reasons. It was not until the development of LeNet-5 that CNNs took shape [8]. A great contribution to convolutional networks and an upgrade on LeNet style architectures came from generalizing the Deep Belief Network idea [9] to a Convolutional Network [10]. However, with recent community-wide shift towards the use of very large models [11] (e.g. 19.4M parameters) trained on very large datasets [12] (e.g. ImageNet with 1.4M images) using extensive computational resources we see a revolution in achievable performances as well as our thinking about such inference problems. The breakthrough of deep convolutional neural networks (CNNs) arrived with the ImageNet competition winner 2012 AlexNet [13]. Since then, deep learning has constantly been pushing the state-of-the-art accuracy in image classification. The community has now been using these extensive convolutional networks architecture not only on classification problems, but in other computer vision and signal processing settings, such as Object Localization [14], Semantic Segmentation [15], Face Recognition and Identification [16], Speech Recognition [17], and Text Detection [18]. Convolutional networks are very

E. Marquez, J. Hare and M. Niranjan are with the Department of Electronics and Computer Science, University of Southampton, UK. Correspondence to E. Marquez at esm1g14@soton.ac.uk.

flexible because they are often trained as feature extractors and not only as classification devices. Furthermore, these nets can not only learn robust feature, but can also learn discriminative binary hash codes [19]. In any case, deep learning still has a long way to go in order to substantially outperform human level knowledge [20], [12].

Recently, networks have increased depth in order to capture low and high level features at different stages of the network. A few years back, the deepest network was AlexNet with five convolutional layers and two dense layers, but now techniques such as the stochastic depth procedure [21] have used more than 1200 layers to increase the performance of the network. The rationale for these deeper networks is that more layers should capture better high level features. However, when performing backpropagation on deep networks, because of the multiplicative effect of the chain rule the magnitude of the gradient is greater on layers that are closer to the output, making the weight updates of the initial layers significantly smaller (layers that are closer to the input then learn at a slower rate). This issue is called the vanishing gradient problem, and it affects every network that is trained with any kind of backpropagation algorithm that has multiple weight layers.

Multiple algorithms have been proposed to overcome the vanishing gradient problem. Residual Networks [11] are non-feedforward networks made of residual blocks, which are composed of convolutional layers, batch normalization [22], and a bypass connection that helps to alleviate the vanishing gradient problem. However, ResNets are equivalent to ensembles of shallow networks and do not fully overcome the vanishing gradient [23]. More recently, Deep Stochastic Depth networks [21] combine the Residual Networks architecture with an extended version of dropout to again further solve the vanishing gradient problem, obtaining improvements of $\sim$ 1% over ResNets.

The reminder of the paper is organized as follows. Section I-A explains the Cascade Learning algorithm and analyses its advantages. Section I-B shows the results and discussion of two experiments performed on two architectures. Finally, Section II summarizes the findings, contributions, and potential further work of this paper.

A. The Deep Cascade Learning Algorithm

In this section we describe the proposed Deep Cascade Learning algorithm and discuss the computational advantages of training in a layer-wise manner. All the code used to generate the results in this manuscript can be found in the GitHub repository available at *http://github.com/EnriqueSMarquez/CascadeLearning*.

1) Algorithm description: As opposed to the cascade correlation algorithm, which sequentially trains perceptrons, we cascade layers of units. The proposed algorithm allows us to train deep networks in a cascade-like, or bottom up layer-by-layer, manner. For the purposes of this paper, we focus on convolutional neural networks architectures. The deep cascade learning algorithm splits the network into its layers and trains each layer one by one until all the layers in the input architecture have been trained, however, if no architecture is given, one can use the cascade learning to train as many layers as desired (e.g. until the validation error stabilizes). This training procedure allows us to counter the vanishing gradient problem by forcing the network to learn features correlated with the output on each and every layer. The training procedure can be generalized as "several" single layer convolutional neural networks (sub-networks) that interconnect and can be trained one at a time from the bottom up (see Figure 1).

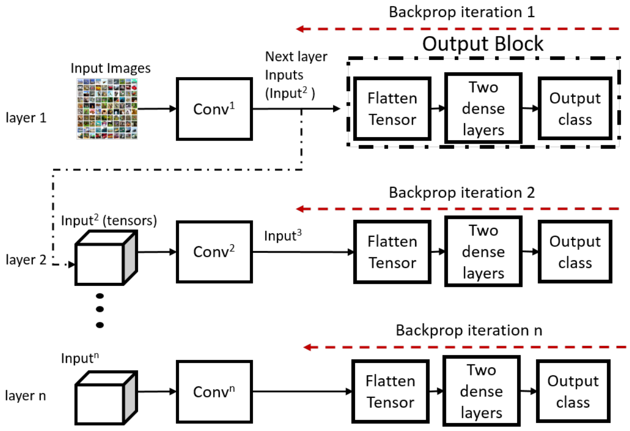


Fig. 1: Overview of Deep Cascade Learning on a convolutional network with n layers. Input^i is the tensor generated by propagating the images through the layers up to and including Conv^{i-1}. Training proceeds layer by layer; at each stage using convolutional layer outputs as inputs to train the next layer. The features are flattened before feeding them to the classification stage. In contrast with the cascade correlation algorithm, the output block is discarded at the end of the iteration (see Algorithm I-A1), and typically it contains a set of fully connected layers with non-linearities and dropout.

The algorithm takes as inputs the hyper-parameters of the training algorithm (e.g. optimizer, loss, epochs) and the model to train. Pseudocode of the Cascade Learning procedure can be found in Algorithm I-A1, and will be referred in further explanations of the algorithm. Learning starts by taking the first layer of the model and connecting it to the output with an 'output block' (line 9), which might be several dense layers connected to the output [13], [25], or, as it is sometimes shown in the literature, an average pooling layer and the output layer with an activation function [26]. Training using standard backpropagation then commences (using the pre-supplied parameters, loop in line 11) to learn weights for the first model layer (and the weights for the output block). Once the weights of the first layer have converged, the second layer can be learned by taking the second layer from the input model, connecting it to an output block (with the same form as for the first layer, but potentially different dimensionality), and training it against the outputs with pseudo-inputs created by forward propagating the actual inputs through the (fixed) first layer. This process can then repeat until all layers have been learned. At each stage the pseudo-inputs are generated by forward propagating the actual inputs through all the previously trained layers. It should be noted that once a layers'

Algorithm 1 Pseudocode of Cascade Learning adapted from the Cascade Correlation algorithm [24]. Training is performed in batches, hence every epoch is performed by doing backpropagation through all the batches of the data.

```
procedure CASCADE LEARNING($layers, \eta, epochs, epochsUpdate, out$)
    Inputs  $layers$ : model layers parameters (loss function, activation, regularization, number of filters, size of filters, stride)
            $\eta$ : Learning rate
            $epochs$ : starting number of epochs
            $k$ : epochs update constant
            $out$ : output block specifications
    Output W : $L$ layers with $\mathbf{w}_l$ trained weights per layer
    for $layer_index = 1 : L$ do                                  ▷ Cascading through trainable layers
        $Init\ new\ layer\ and\ connect\ output\ block$
        $i_l \leftarrow epochs + k \times layer\_index$
        for $i = 0; i{+}{+}; i < i_l$ do                          ▷ Loop through data $i_l$ times
            $\mathbf{w}^{new} \leftarrow \mathbf{w}^{old} - \eta \nabla J(\mathbf{w})$       ▷ Update weights by gradient descent
            if $Validation\ error\ plateaus$ then
                $\eta \leftarrow \eta/10$                          ▷ Change learning rate if update criteria is satisfied
            end if
        end for
        $Disconnect\ output\ block\ and\ get\ new\ inputs$
    end for
end procedure
```

weights have been learned that they are fixed for all subsequent layers. Figure 1 gives a graphical overview of the entire process.

Most hyper-parameters in the algorithm remain the same across each layer, however we have found it beneficial to dynamically increase the number of learning epochs as we get deeper into the network. Additionally, we start training the initial layers with orders of magnitude fewer epochs than we would if training End-End. The rationale for this is that each sub-network fits the data faster than the End-End model and we do not want to overfit the data, specially at in the lower layers. Overfitting in the lower layers would severely hamper the generalisation ability of later layers. In our experiments we have found that the number of epochs required to fit the data is dependable on the layer index, if a layer requires $i_{(epochs)}$, the subsequent layer should require $i_{(epochs)} + k$, where k is a constant whose value is set dependent on the dataset.

A particular advantage of such cascaded training is that the backward propagated gradient is not diminished by hidden layers as happens in the End-End training. This is because every trainable layer is immediately adjacent to the output block. In essence, this should help the network obtain more robust representations at every layer. In Section I-B we demonstrate this by comparing confusion matrices at different layers of networks trained using Deep Cascade Learning and standard End-End backpropagation. The other advantages, as we demonstrate in the following subsections, is that the complexity of learning is reduced over End-End learning, both in terms of training time and memory.

2) Cascade Learning as supervised pre-training algorithm: A particular appeal of deep neural networks is pre-training the weights to obtain a better initialization, and further achieve a better minima. Starting from the work of Hinton *et. al* on Deep Belief Networks [9], unsupervised learning has been considered in the past as effective pre-training, initializing the weights which are then improved in a supervised learning setting. While this was a great motivation, recent architectures [20] [11] [27], however, have ignored this and focused on pure supervised learning with random initialization.

The cascade learning can be used to initialize the filters in a CNN and diminish the impact of the vanishing gradient problem. After the weights have been pre-trained using cascade learning, the network is tuned using traditional End-End training (both stages are supervised). When applying this procedure it is imperative to re-initialize the output block after pre-training the network, otherwise the network would rapidly reach the sub-optimal minimum obtained by the cascade learning. This does not provide better performance in terms of accuracy. In later sections we discuss how this technique may lead the network to better generalization.

3) Time Complexity: In a convolutional neural network the time complexity of the convolutional layers is:

$$O\left(\sum_{l=1}^{d} n_{l-1}\ s_l^2\ n_l\ m_l^2\ i\right), \tag{1}$$

where i is the number of training iterations, l is the layer index, d is the last layer index, n is the number of filters, s and m is the size of the input and output (spatial size) respectively[1] [28].

Training a convolutional neural network using the Deep Cascade Learning algorithm changes the time complexity as follows:

$$O\left(\sum_{l=1}^{d} n_{l-1}\ s_l^2\ n_l\ m_l^2\ i_l\right), \tag{2}$$

[1] Note that this is the time complexity of a single forward pass; training increases this by a constant factor of about 3.

where i_l represents the number of training iterations for the l-th layer. The main difference between both equations is the number of epochs for every layer, in Equation 1 i is constant while in Equation 2 depends on the layer index. Note in this analysis, we have purposefully ignored the cost of performing the forward passes to compute the pseudo-inputs as this is essentially 'free' if the algorithm is implemented in two threads (see below). The number of iterations in the cascade algorithm depends on the dataset and the model architecture. The algorithm proportionally increases the number of epochs on every iteration since the early layers must not be overfit, while later layers should be trained to more closely fit the data. In practice, as shown in the simulations (Section I-B), one can choose each i_l such that $i_1 << i$ and $i_L \leq i$, and obtain almost equivalent performance to the End-End trained network in a much shorter period of time. If $\sum_{l=1}^{d} i_l = i$, the time complexity of both training algorithms is the same, noting that improvements coming from caching the pseudo-inputs are not considered.

There are two main ways of implementing the algorithm. The best and most efficient approach is by saving the pseudo-inputs on disk once they have been computed; in order to compute the pseudo-inputs for the next layer one only has to forward propagate the cached pseudo-inputs through a single layer. An alternate, naive, approach would be implementing the algorithm using two threads (or two GPUs), with one thread using the already trained layers to generate the pseudo-inputs on demand, and the other thread training the current layer. The disadvantage of this is that it would require the input to be forward propagated on each iteration. The first approach can further drastically decrease the runtime of the algorithm and the memory required to train the model at the expense of disk space used for storing cached pseudo-inputs.

4) Space Complexity: When considering the space complexity and memory usage of a network, we have to both consider the number of parameters of the model, but also the amount of data that needs to be in memory in order to perform training of those parameters. In standard End-End backpropagation, intermediary results (e.g. response maps from convolutional layers and vectors from dense layers) need to be stored for an iteration of backpropagation. With modern hardware and optimisers (based on variants of mini-batch stochastic gradient descent) we usually consider batches of data being used for the training, so the amount of intermediary data at each layer is multiplied by the batch size.

Aside from offline storage for caching pseudo-inputs and storing trained weights, the cascade algorithm only requires that the weights of a single model layer, the output block weights, and the pseudo-inputs of the current training batch are stored in RAM (on the CPU or GPU) at any one time. This *potentially* allows memory to be used much more effectively and allows models to be trained whose weights exceed the amount of available memory, however this is drastically affected by the choice of output block architecture, and also the depth and overall architecture of the network in question.

model	parameters	data storage	total
VGG-16	13.9M	311178	14.2M
VGG-19	19.1M	337802	19.5M
VGG-22	24.5M	364426	24.8M
VGG-25	29.8M	391050	30.2M
VGG-28	35.1M	417674	35.5M

TABLE I: Space complexity of End-End training of various depths of VGG style networks. The number of parameters increases with depth. The data storage units of the training depends on the computational precision.

trainable layer #	parameters	data storage	total
1	33.6M	69386	33.7M
2	8.6M	131850	8.6M
Pooling			
3	16.9M	49930	17.0M
4	4.5M	66314	4.6M
Pooling			
5	8.8M	25354	8.8M
6	2.8M	25354	2.9M
...			

TABLE II: Space complexity of cascade training of various layers of a VGG style network. The number of parameters decreases with depth. The data storage units of the training depends on the computational precision.

To explore this further, consider the parameter and data complexity of a VGG-style (Visual Geometry Group Net) network of different depths. Assume that we can grow the depth in the same way as going between the VGG-16 and VGG-19 models in the original paper by [25] (note we are considering Model D in the original paper to be VGG-16 and Model E to be VGG-19), whereby to generate the next deeper architecture we add an additional convolutional layer to the last three blocks of similarly sized convolutions. This process allows us to define models VGG-22, VGG-25, VGG-28, etc. The number of parameters and training memory complexity of these models is shown in Table I. The numbers in this table were computed on the assumption of a batch size of 1, input size of 32×32, and the output block (last 3 fully-connected/dense layers) consisting of 512, 256 and 10 units respectively. The remainder of the model matches the description in the original paper ([25]), with blocks of 64, 128, 256, and 512 convolutional filters with a spatial size of 3×3 and the relevant max-pooling between blocks. For simplicity we assume the convolutional filters are zero-padded so the size of the input does not diminish.

The key point to note from Table I is that as the model gets bigger, the amount of memory required for both parameters and for data storage of End-End training increases. With our proposed cascade learning approach, this is not the case; the total memory complexity is purely a function of the most complex cascaded sub-network (network trained in one iteration of the cascade learning). In the case of all the above VGG-style networks, this happens very early on in the cascading process. More specifically this happens when cascading the second layer, as can be seen in Table II, which illustrates that after the second layer (or more concretely after the first max-pooling) the complexity of subsequent iterations of cascading reduces. The assumption in computing the numbers in this table is that the output blocks mirrored those of the end to end training and had 512, 256 and 10 units respectively.

If we consider Tables I and II together, we can see with the architectures in question that for smaller networks the end to end training will use less memory (although it is slower), whilst for deeper networks the cascading algorithm will require less peak memory whilst bringing time complexity reductions. Given that the bulk of the space complexity for cascading comes as a result of the potentially massive number of trainable parameters in connecting the feature maps from the early convolutional layers to the first layer of output block, an obvious question is could we change the output block specification to reduce the space complexity for these layers? Whilst not the key focus of this manuscript, initial experiments described in Section I-B1a start to explore the effect of reduced complexity output blocks on overall network classification performance.

B. Experiments

The first experiment was performed on a less complex backpropagation problem and not on a CNN as explained in Section I-A. We decided to execute this experiment to quickly determine the efficiency of cascade learning. In this case we have chosen a small three hidden layers Multi-Layer Perceptron (MLP) applied on the flattened MNIST dataset. The results show that this algorithm is feasible and can obtain better generalization in early stages of the network with small improvements (∼0.5%). This was a preliminar experiment, details can be found in the github repository.

To demonstrate the effectiveness of the Deep Cascade Learning algorithm, we apply it to two widely known architectures: a 'VGG-style' network [25], and the 'All CNN' [26]. We have chosen these architectures for several reasons. Firstly, they are still extensively used in the computer vision community, and secondly, they inspired state of the art architectures, such as ResNets and FractalNets. Explicitly, the VGG net shows how small filters (3x3) can capture patterns at different scales by just performing enough subsampling. The All CNN gave the idea of performing the subsampling with an extra convolutional layer rather than a pooling layer, and performs the classification using global average pooling and a dense layer to diminish the complexity of the network. The representations learned in each layer through End-End training are compared to the ones generated by Deep Cascade Learning. In order to make a layer-wise comparison we first train an End-End model, and then use the already trained filters to train classifiers by attaching and training output blocks (the model layer weights are fixed at this point however in contrast to cascade learning). The training parameters of the models remain as similar as possible to make a fair comparison.

The learning rate in both experiments is diminished when the validation error plateaus. Taking into account that the dataset is noisy and the error does not necessarily decrease after every epoch, we evaluate the performance after each epoch to determine whether the learning rate should be changed. More specifically, we use a mean-window approach that computes the average of the last five epochs and the last ten epochs, and if the difference is negative then the learning rate is decreased by a factor of ten. The size of the window was tuned for the cascade learning only; if this approach is used in other training procedures it might be necessary to increase the size of the window.

The increase in the epochs in the cascade algorithm varies depending on the dataset. We performed experiments with an initial number of epochs ranging from ten to one hundred without any real change in the overall result, hence ten epochs as starting point is the most convenient. In all the experiments presented here, every layer iteration initialises a new output block, which in this case consists of two dense layers with ReLu activation [29]. The number of neurons in the first layer will depend on the dimensionality of the input vector, it may vary between 64 to 512 units, the second layer contains half as many units as in the first layer. The final layer uses Softmax activation and 10 or 100 units depending on the dataset.

Datasets: We have performed experiments using the CIFAR-10 and CIFAR-100 [30] image classification datasets, which have 10 and 100 target labels respectively. Both datasets contain 60k RGB 32x32 images split in three sets: 45k images for training, 5k images for validation, and 10k images for testing. In our experiments the datasets were normalized and whitened, however we performed no further data augmentation, similar to the Stochastic Depth procedure [21].

1) CIFAR-10:

VGG-style networks: The VGG network uses a weight decay of 0.001, and stochastic gradient descent with a starting learning rate of 0.01. Our VGG model contains six convolutional layers, starting with 128 3x3 filters and duplicating them after a MaxPooling layer. The initial weights remained the same in the networks trained by the two approaches to make the convergence comparable.

a) Space complexity and output block specifications.: In order to test the memory complexity of this network we must take into account the output block specifications. Specifically, we must consider the first fully connected layer, which in most networks contains the biggest number of trainable parameters, particularly when connected to an early convolutional layer (see Section I-A4). On the first iteration of cascade learning, the output is 128x32x32, hence, the number of neurons (n) in the first fully connected layer must be small enough to avoid running out of memory but without jeopardising robustness in terms of predictive accuracy. We have performed an evaluation by cascading this architecture with output blocks with a range of different parameter complexities. Table III shows the number of parameters of every layer as well as the performance for output blocks with first fully connected layer sizes of $n = \{64, 128, 256, 512\}$. In terms of parameters, cascade learning for early iterations can require more space than the entire End-End network unless the overall model is deep. The impact of this disadvantage can be overcome by choosing a smaller n, and as shown in Table III, the hit on accuracy need not be particularly high when compared to the reduction in parameters and saving of memory.

Reducing the number of units can efficiently diminish the parameters of the network. However, in cases where the input image is massive, more advanced algorithms to counter the exploding number of parameters are applicable, such as Tensorizing Neural Networks and Hashed Nets[31], [32]. Based

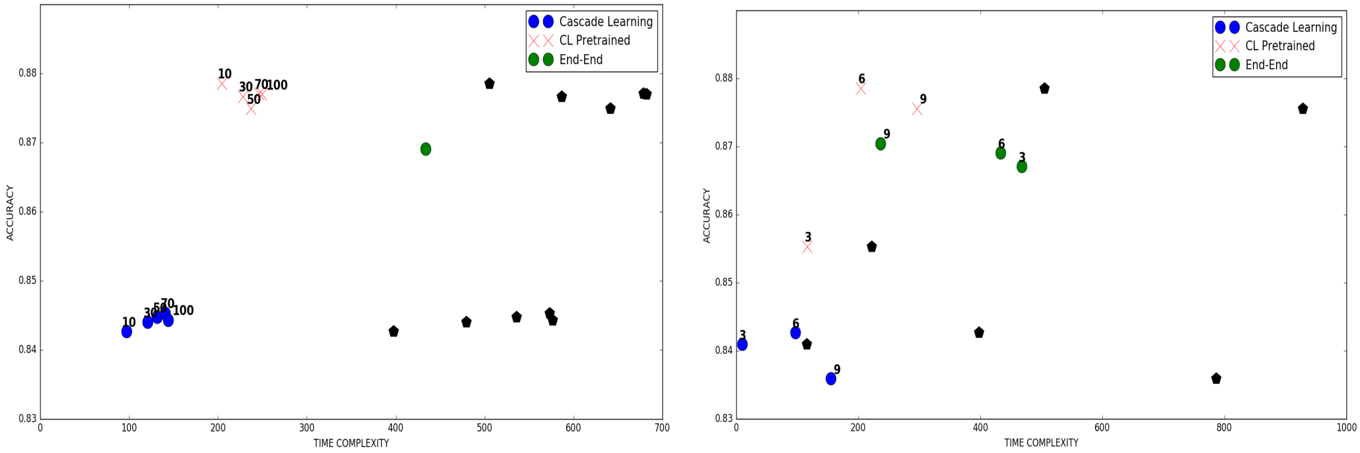


Fig. 2: Time complexity comparison between cascade learning, End-End, and pretrained using cascade learning (see section I-B3 for details and results on pretraining using cascade learning). Multiple VGG networks were executed within a range of, (left) starting number of epochs (10-100) , (right) depth (3,6,9). Black pentagons represent runs executing the naive approach for both Cascade Learning and the pretraining stage. Solid blue dots represent optimal run, which caches the pseudo-inputs after every iteration.

Training Regime.	Iter.	First output block unit count							
		64		**128**		**256**		**512**	
		param.	acc.	param.	acc.	param.	acc.	param.	acc.
CL	1	8.4e6	0.63	1.7e7	0.64	3.4e7	0.66	6.7e7	0.66
	2	8.5e6	0.69	1.7e7	0.72	3.4e7	0.72	6.7e7	0.73
	3	2.5e6	0.77	4.4e6	0.78	8.6e6	0.79	1.7e7	0.80
	4	4.5e6	0.80	8.7e6	0.81	1.7e7	0.81	3.4e7	0.81
	5	4.8e6	0.82	9.0e6	0.83	1.7e7	0.83	3.4e7	0.83
	6	1.6e6	0.83	2.7e6	0.84	4.8e6	0.84	9.1e6	0.84
End-End		2.8e6	0.86	3.9e6	0.87	6.0e6	0.87	1.0e7	0.87

TABLE III: Parameter complexity comparison using different output block specifications. Shows the effect of using between 64 and 512 units in the first fully connected layer (which is most correlated with the complexity). (Left) number of parameters, (Right) accuracy. Bottom row shows the parameters complexity of the End-End model. The increase in memory complexity on early stages can be naively reducen by decreasing n. Potentially, memory reduction techniques on the first fully connected layer are applicable at early stages of the network. Later layers are less complex.

on their findings, applying those types of transformations to the first fully connected layer should not affect the results.

b) Training time complexity and relationship with depth and starting number of epochs.: Equation 2 is dependant on the starting number of epochs i_l and its proportionality with depth. In Figure 2 We explored the affect of the time complexity by these two variables. To reproduce the left figure, several networks were cascaded with $i_l = [10, 30, 50, 70, 100]$, the overall required time is not drastically affected by i_l. For this particular experiment if $i_l > 50$, each iteration is more likely to be stopped early due to overfitting. The right figure shows the results on a similar experiment with varying network depth ($d = [3, 6, 9]$). Cascading shallow networks outperforms End-End training in terms of time. The epochs update constant (k in I-A1) should be minimized on deeper networks to avoid an excessive overall number of epochs. Both figures show the importance of caching the pseudo-inputs, the black pentagons (naive run) are shifted to the right in relation to solid blue dots (enhanced run).

Figure 3 shows confusion matrices from both algorithms across the classifiers trained on each layer. In this experiment we found that the features learnt using the cascade algorithm are less noisy, and more correlated with the output in most stages of the network. The results of the experiment shows that the features learnt using the End-End training in the first and second layer are not correlated with the output; in this case the trained output block classifier always makes the same

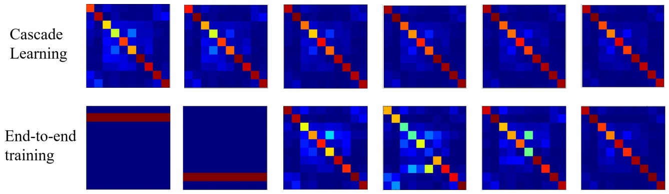


Fig. 3: Comparison of confusion matrices in a VGG network trained using the cascade algorithm and the End-End training on CIFAR-10. First two layers of the End-End training do not show correlation with the output. While accuracy increases proportionally with the number of layer using the cascade learning, showing more stable features at every stage of the network.

prediction, and hence the feature vector is not sparse at all. The third layer starts building the robustness of the features with an accuracy of 67.2%, and the peak is reached in the last layer with 85.6%. In contrast, with the cascade learning, discriminative features are learnt on every layer of the network. At the third layer, classes such as airplane and ship are strongly correlated with the features generated in both cases. The End-End training mostly fails to generalize correctly in classes related to animals.

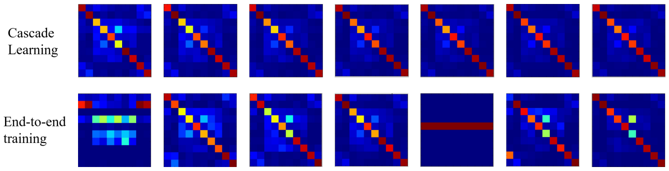


Fig. 4: Comparison of confusion matrices in a The All CNN network trained using the cascade algorithm and the End-End training on CIFAR-10. Features learnt by cascading the layers are less noisy, and more stable.

On every iteration of the cascade algorithm, the sub-networks have a tendency to overfit the data. However, this is not entirely a problem as we have found that overfitting mostly occurs in the dense layers connected in the output block, and those are not part of the resulting model. In this way, we avoid generating overfitted pseudo-inputs for the next iteration, hence disconnecting the dense layers works as a matter of regularisation.

One of the ways of determining if the vanishing gradient problem has been somehow diminished is by observing the filters/weights on the first layer (the most affected one by this issue). If the magnitude of the gradient in the first layer is small, then the filters do not change much from the initialized one. Figure 5 shows the filters learnt using both algorithms. The cascade algorithm learnt a range of different filters with different orientation and frequency responses, while using an End-End training the filters learnt are less representative. Some filters in the End-End training are overlapping, this generates a problem since the information that is being captured is redundant.

It is naive to assume the problem is alleviated because the filters on the cascade learning are further apart from the initial filters. Hence, to complement the visualization of the filters, we calculated the magnitude of the gradient after every mini-batch forward pass on both cascade learning and End-End and plotted the results on Figure 6. For the End-End training, the gradient was computed at every convolutional layer for all the epochs. For the Cascade Learning, the gradients were calculated on every iteration on the core correspondent convolutional layer. The curves are generated by averaging the mini-batch gradients in each epoch.

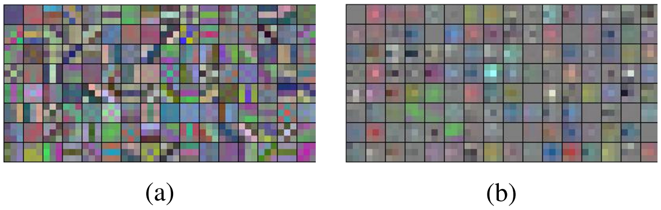


Fig. 5: Visualization of the filters learnt in the first layer in both algorithms, (a) cascade learning, (b) End-End. Each patch corresponds to one 3x3 filter. Filters learnt using the Cascade Learning show more clear representations with a wide number of rotations, while in the End-End most filters are redundant.

In contrast with Cascade Learning, the magnitude of the gradients of End-End training, on early layers, is significantly smaller than those on deeper layers. Overall, the gradients are higher for the Cascade Learning. Cascade Learning requires fewer epochs with high updates on the weights to quickly fit the data on every iteration. With End-End training the opposite occurs; it requires more epochs (because of the small updates) to fit the data.

The All CNN: This architecture contains only convolutional layers, the downsampling is performed by using a convolutional layer with stride of 2 rather than a pooling operation. It also performs the classification by downsampling the image until the dimensionality of the output matches the targets. The All CNN paper [26] describes three model architectures. We have performed our experiments using model C which contains seven core convolutional layers, and four 1x1 convolutional layers to perform the classification with an average pooling and softmax layers as the output block. In the case where the output block contains an average pooling and a softmax activation, each layer would learn the filters required to classify the data and not to generate robust filters. Hence, to make a fair comparison of the filters we have changed the output block of The All CNN to three dense layers with softmax activation at the end. In the All CNN report it is stated that changing the output block may results in a decrease of the performance, however in this study we aim to fairly compare both algorithms in every stage rather than final classification result. The parameters used when cascading this architecture varies between $2.7*10^6$ and $0.33*10^6$; on the other hand the end to end training requires us to store $1.3*10^6$ parameters.

The All CNN, using an End-End training, learns better representations on early layers than the VGG style net. The first convolutional layer achieves a performance in the orders of 20% by learning three classes at the most, this can be observed in the confusion matrix in Figure 4. In contrast with the End-End training, the accuracy when cascading this architecture progressively increases with the iterations, learning discriminative representations at every stage of the network going from 65% to 83.4%.

Figure 7 compares the performance of both algorithms on each layer. The accuracy in the cascade learning increases with the number of layers. In addition, the variance of the performance is very low in comparison with the End-End, because it forces the network to learn similar filters in every run, decreasing the impact of a poor initialization.

We have found that for a given iteration more than 50 epochs are not necessary to achieve a reasonable level of accuracy without overfitting the data. We also tested the time complexity of this model within a range of starting epochs

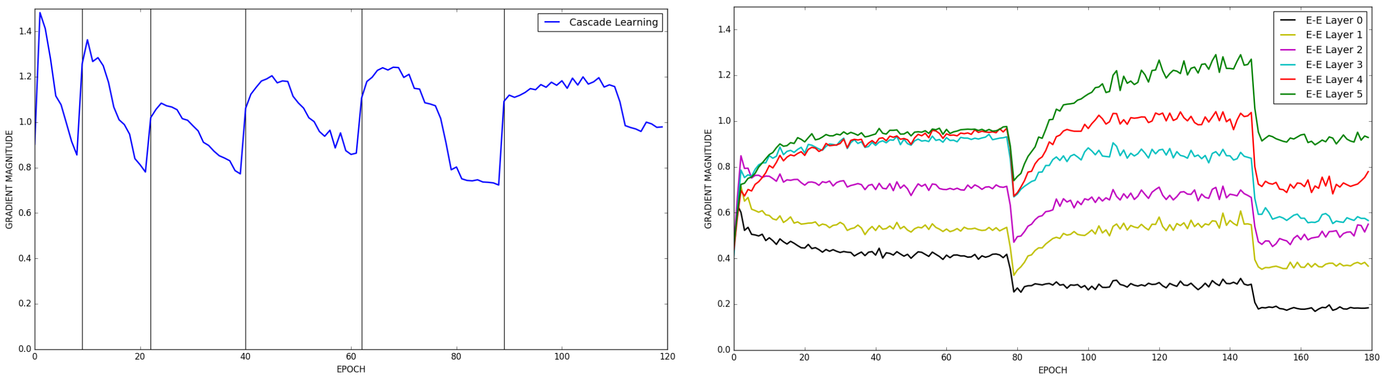


Fig. 6: Magnitude of the gradient after every mini-batch forward pass on the convolutional layers of the End-End training (right), and the concatenated gradients of every Cascade Learning (left) iteration. Vertical lines represent the start of a new iteration. Curves were smoothed by averaging the gradients (of every batch) on every epoch.

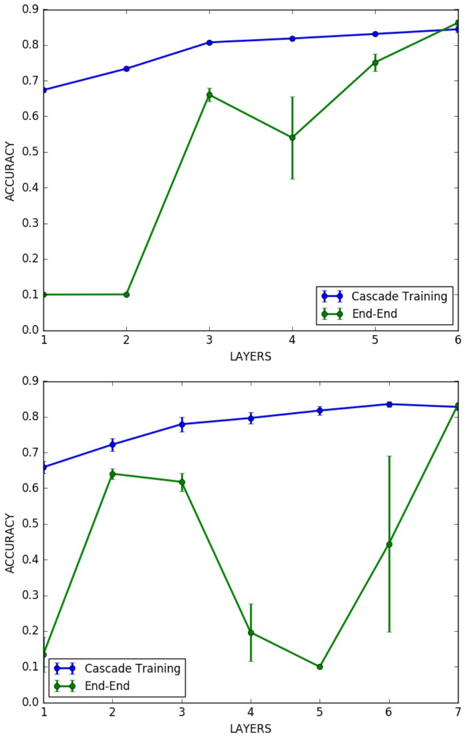


Fig. 7: Performance on every layer on both architectures, (top) VGG, (bottom) The All CNN. Cascade learning has a lower variance making the initialization less relevant to the classification at each layer. It also shows a progressive increase in the performance without the fluctuations presented in the End-End training.

(similar experiment in previous section). We tested the time complexity from 10 starting epochs to 50 (epochs increase by ten on every iteration with a ceiling on 50). The time complexity for The All CNN model C is reduced by ~ 2.5 regardless of the starting number of epochs.

2) CIFAR-100: Similarly to the previous experiments, we have tested how the cascade algorithm behaves with a one-hundred class problem using the CIFAR-100 data set [30].

		1	**2**	**3**	**4**	**5**	**6**	**7**	**Pre**
VGG	CL	0.35	0.39	0.50	0.50	0.53	0.59	-	**0.63**
	E-E	0.01	0.03	0.22	0.14	0.35	0.60	-	
The All CNN	CL	0.31	0.39	0.47	0.46	0.49	0.54	0.52	**0.67**
	E-E	0.03	0.05	0.03	0.41	0.54	0.61	0.62	

TABLE IV: Comparison of accuracy per layer using the cascade algorithm and End-End training on CIFAR-100 in both architectures. Using the cascade learning outperforms almost all the layers in a VGG network, and almost achieves the same accuracy in the final stage. The All CNN with an End-End training outperforms in the final classification, however the first three layers do not learn strong correlations like when using the cascade learning.

The experimental settings remain the same as the previous section, and the main change to the model is that the output layer now has 100 units to match the number of classes.

In a VGG-style network, the comparison between both algorithms is similar to a ten class problem. In End-End training, the first two layers do not learn meaningful representations, and each layer learns better features using the cascade algorithm. However, the End-End training performs better by 1% on the final classification.

In The All CNN Network, the features learnt in the End-End model remained more stable than in CIFAR-10. Similarly than in the previous experiment, the first four layers were outperformed by the cascaded model. The cascade network in overall had better performance in the End-End model by 6% and 10% on the last layers.

The results on a 100 class problem are arguably the same as in a ten class one. It is noted that The All CNN Network, when trained End-End, can outperform the cascade algorithm in the final classification but not in the early layers. In the VGG style network, Deep Cascade Learning build more robust features in all the layers, except for the last layer which had a difference of 1%. Table IV shows a summary of the results on every layer for both algorithms.

3) Pre-training with cascade learning: In the experimental work described so far, the main advantages of cascade learning come from: (a) reduced computation, albeit at the loss of some

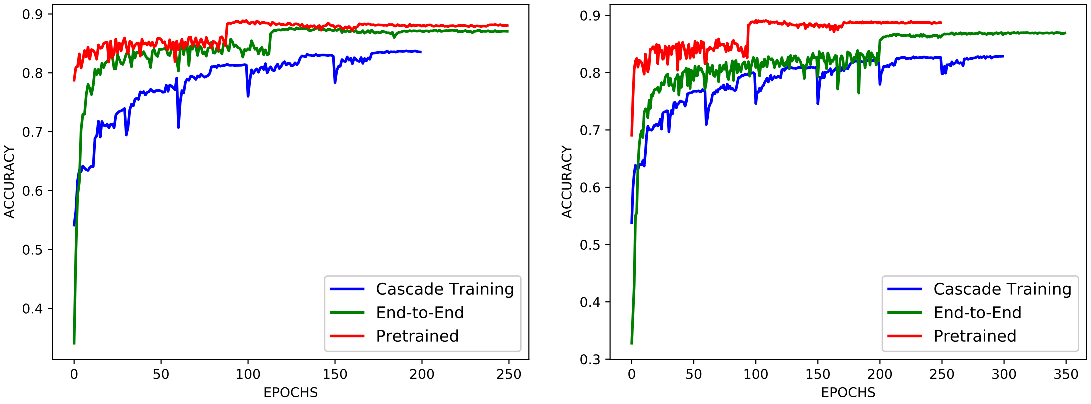


Fig. 8: Performance comparison on CIFAR-10 between pre-trained network and random initialization, (left) VGG, (right) The All CNN. The step bumps in the Cascade learning are generated due to the start of a new iteration or changes in the learning rate.

performance in comparison to End-End training; and (b) a better representation at intermediate layers. We next sought to explore if the representations learned by the computationally efficient cascading approach could form good initialisations of End-End trained networks and achieve performance improvements.

The weights are initialised randomly. Then the procedure is divided into two stages: firstly, we cascade the network with the minimum number of epochs to diminish its time complexity. Finally, the network is fine tuned using a back-propagation and stochastic gradient descent, similarly to the End-End training. We applied this technique using a VGG style network and The All CNN. For more details on the architectures refer to Section I-B1.

Figure 8 shows the difference in performance given random and cascade learning initialisation. The learning curves in the figures are for the VGG and The All CNN architectures trained on CIFAR-10. The improvements in testing accuracy varies between $\sim$2 to $\sim 3\%$ for the experiments developed in this section. However, the most interesting property comes as a consequence of the variation of the resulting weights after executing the cascade learning. As shown in the previous section this variation is significantly smaller in contrast with its End-End counterpart. Hence, the results obtained after pre-initializing the network are more stable and less affected by poor initialization. Results on Figure 2 show that even including the time of the tuning training stage, the time complexity can be reduced if the correct parameters for the cascade learning are chosen. It is important to mention that, the End-End training typically requires up to 250 epochs, while the tuning stage may only require a small fraction since the training is stopped when the training accuracy reaches ~ 0.999.

The filters generated by the cascade learning filters are slightly overfitted (first layer typically achieves $\sim 60\%$ on unseen data and $\sim 95\%$ on training data) as opposed to the End-End training, on which the filters are more likely to be close to its initialization. By pre-training with cascade learning, the network learns filters that are in between both scenarios (under and overfitness), this behaviour can be spotted on Figure 9.

Figure 10 shows the test accuracy during training of a cascaded pre-trained VGG model on CIFAR-100. Improvements of $\sim 2.5\%$ were achieved in the final classification. More details on this experiment are available in the GitHub repository accompanying this paper.

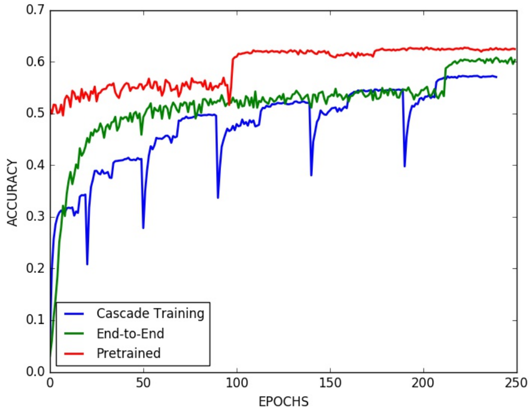
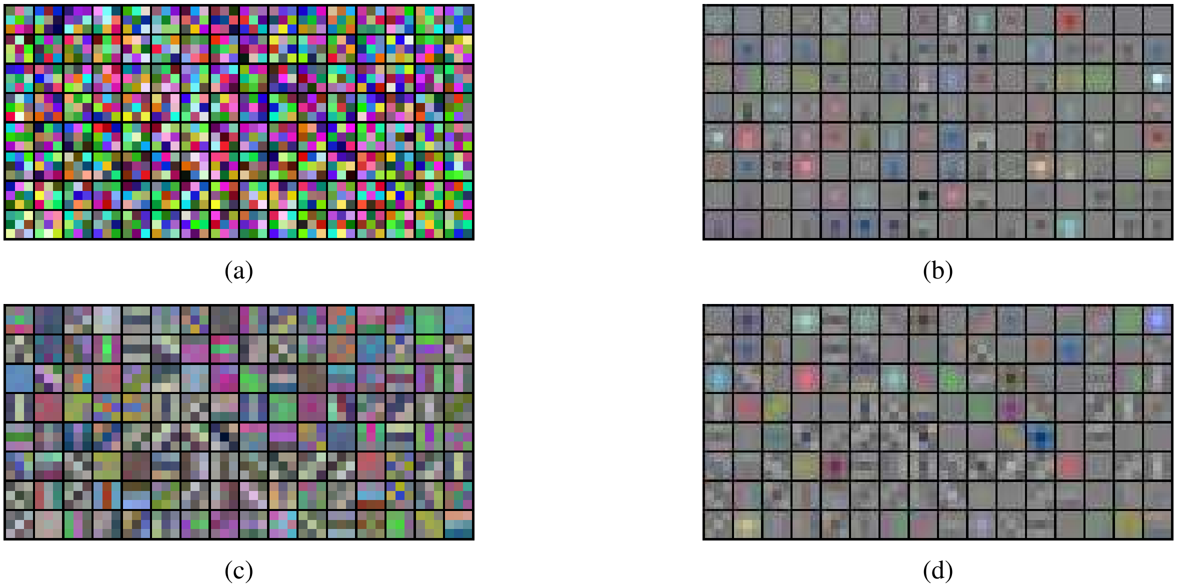


Fig. 10: Performance comparison between pretrained network and random initialization on CIFAR-100 using a VGG network.

II. Conclusion

In this paper we have proposed a new supervised learning algorithm to train deep neural networks. We validate our technique by studying an image classification problem on two widely used network architectures. The vanishing gradient problem is diminished by our Deep Cascade Learning, because it is focused on learning more robust features and filters in early layers of the network. In addition, the time complexity is reduced because it no longer needs to forward propagate the data through the already trained layers on every epoch. In our experiments the memory complexity is decreased more than three times for the VGG style network and four times for



(a) (b) (c) (d)

Fig. 9: Filters on first layer for at different stages of the procedure on the VGG network defined in the previous section. (a) initial random weights, (b) End-End, (c) cascaded, (d) End-End trained network initialized by cascade learning

The All CNN. Standard End-End training has a high variance in the performance, meaning that the initialization plays an important role in ensuring a good minimum is reached by each layer. Deep Cascade Learning generates a more stable output on every stage by learning similar representations at every run. In addition, the cascade learning algorithm has demonstrated to scale in 10 and 100 class problems, and shows improvements in the features that are learnt across the stages of the network. Using this algorithm allows us to train deeper networks without the need to store the entire network in memory. It should be noted that our algorithm is not aimed at obtaining better performance than standard approaches, but with significant reduction in the memory and time requirements. We have shown that if improvements in generalization are expected, this algorithm has to be used as a pre-training algorithm technique.

There are many questions that are still yet to be answered. How deep can this algorithm go without losing robustness? We believe that if the performance cannot be improved by appending a new convolutional layer, l, it should at least be as good as in the previous layer, $l-1$, by learning filters that directly map the input to the output (filters with 1 in the centre, and zero in the borders). This might not happen because the layer might quickly find a local minimum. This could be avoided with a different type of initialization; most probably one specialised for this algorithm. Our immediate next steps include observing how deep can the cascading algorithm can go without losing performance, similar to the experiment performed with Deep Residual Network [11] and Fractal Networks [27], in order to measure to what extent the vanishing gradient problem is solved. In [11], the accuracy diminished when they went beyond 1200 layers, and hence the vanishing gradient problem was not entirely circumvented. We believe this algorithm might be able to go deeper without losing performance by partially overcoming the vanishing gradient problem, learning "mapping" filters to maintain the features sparseness, and learn a bigger set of high level features. In addition, the Deep Cascade Learning has the potential to find the number of layers required to fit a certain problem (adaptive architecture), similarly to the Cascade Correlation [1], Infinite Restricted Boltzmann Machine [33], and AdaNet [34].

Acknowledgment

The authors would like to acknowledge the Department of Electronics and Computer Science at the University of Southampton for funding the work presented in this manuscript. We also gratefully acknowledge the support of the NVIDIA Corporation with the donation of the Titan X GPU used for this research. We would like to thank reviewers for their insightful comments and suggestions, as these led us to an improvement of the manuscript.

References

[1] S. E. Fahlman and C. Lebiere, "The cascade-correlation learning architecture," D. S. Touretzky, Ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990, ch. Advances in neural information processing systems 2, pp. 524–532. [Online]. Available: http://dl.acm.org/citation.cfm?id=109230.107380

[2] J. Platt, "A resource-allocating network for function interpolation," *Neural computation*, vol. 3, no. 2, pp. 213–225, 1991.

[3] V. Kadirkamanathan and M. Niranjan, "A function estimation approach to sequential learning with neural networks," *Neural Computation*, vol. 5, pp. 954–975, 1993.

[4] C. Molina and M. Niranjan, "Pruning with replacement on limited resource allocating networks by F-projections," *Neural Computation*, vol. 8, no. 4, pp. 855–868, 1996.

[5] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR, abs/1510.00149*, vol. 2, 2015.

[6] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," *arXiv preprint arXiv:1603.05279*, 2016.

[7] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological Cybernetics*, vol. 36, no. 4, pp. 193–202, 1980. [Online]. Available: http://dx.doi.org/10.1007/BF00344251

[8] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[9] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural Computation*, vol. 18, no. 7, pp. 1527–1554, 2006, pMID: 16764513.

[10] H. Lee, R. Grosse, R. Ranganath, and A. Y. Ng, "Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations," in *Proceedings of the 26th annual international conference on machine learning*. ACM, 2009, pp. 609–616.

[11] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[12] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, 2012, pp. 1097–1105.

[14] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014, pp. 580–587.

[15] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015, pp. 3431–3440.

[16] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, "Deepface: Closing the gap to human-level performance in face verification," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2014.

[17] O. Abdel-Hamid, A. R. Mohamed, H. Jiang, and G. Penn, "Applying convolutional neural networks concepts to hybrid NN-HMM model for speech recognition," in *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, March 2012, pp. 4277–4280.

[18] C. Yan, H. Xie, S. Liu, J. Yin, Y. Zhang, and Q. Dai, "Effective uyghur language text detection in complex background images for traffic prompt identification," *IEEE transactions on intelligent transportation systems*, 2017.

[19] C. Yan, H. Xie, D. Yang, J. Yin, Y. Zhang, and Q. Dai, "Supervised hash coding with deep neural network for environment perception of intelligent vehicles," *IEEE Transactions on Intelligent Transportation Systems*, vol. PP, no. 99, pp. 1–12, 2017.

[20] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 1026–1034.

[21] G. Huang, Y. Sun, Z. Liu, D. Sedra, and K. Q. Weinberger, "Deep networks with stochastic depth," *CoRR*, vol. abs/1603.09382, 2016. [Online]. Available: http://arxiv.org/abs/1603.09382

[22] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift." in *ICML*, ser. JMLR Workshop and Conference Proceedings, F. R. Bach and D. M. Blei, Eds., vol. 37. JMLR.org, 2015, pp. 448–456.

[23] A. Veit, M. J. Wilber, and S. Belongie, "Residual networks behave like ensembles of relatively shallow networks," in *Advances in Neural Information Processing Systems*, 2016, pp. 550–558.

[24] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern classification 2nd Edition*. John Wiley & Sons, 2012.

[25] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014. [Online]. Available: http://arxiv.org/abs/1409.1556

[26] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. A. Riedmiller, "Striving for simplicity: The all convolutional net," *CoRR*, vol. abs/1412.6806, 2014. [Online]. Available: http://arxiv.org/abs/1412.6806

[27] G. Larsson, M. Maire, and G. Shakhnarovich, "Fractalnet: Ultra-deep neural networks without residuals," *arXiv preprint arXiv:1605.07648*, 2016.

[28] K. He and J. Sun, "Convolutional neural networks at constrained time cost," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015, pp. 5353–5360.

[29] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, J. Fürnkranz and T. Joachims, Eds. Omnipress, 2010, pp. 807–814. [Online]. Available: http://www.icml2010.org/papers/432.pdf

[30] A. Krizhevsky, "Learning multiple layers of features from tiny images," Tech. Rep., 2009.

[31] A. Novikov, D. Podoprikhin, A. Osokin, and D. P. Vetrov, "Tensorizing neural networks," in *Advances in Neural Information Processing Systems*, 2015, pp. 442–450.

[32] W. Chen, J. Wilson, S. Tyree, K. Weinberger, and Y. Chen, "Compressing neural networks with the hashing trick," in *International Conference on Machine Learning*, 2015, pp. 2285–2294.

[33] M.-A. Côté and H. Larochelle, "An infinite restricted boltzmann machine," *Neural computation*, vol. 28, pp. 1265–1288, 2016.

[34] C. Cortes, X. Gonzalvo, V. Kuznetsov, M. Mohri, and S. Yang, "Adanet: Adaptive structural learning of artificial neural networks," *arXiv preprint arXiv:1607.01097*, 2016.

Enrique Marquez received his B.S. (2013) in electrical engineering from Universidad Rafael Urdaneta in Venezuela, and the MSc. (2015) in Artificial Intelligence at University of Southampton, Southampton, UK. He is currently working on his Ph.D. in Computer Science at University of Southampton. His research interests include machine learning, computer vision, and image processing.

Jonathon S. Hare is a lecturer in Computer Science at the University of Southampton. He holds a BEng degree in Aerospace Engineering and Ph.D. in Computer Science. His research interests lie in the area of multimedia data mining, analysis and retrieval. This research area is at the convergence of machine learning and computer vision, but also encompasses other modalities of data. The long-term goal of his research is to innovate techniques that can allow machines to learn to understand the information conveyed by multimedia data and use that information to fulfil the information needs of humans.

Mahesan Niranjan is Professor of Electronics and Computer Science at the University of Southampton. Prior to this appointment in February 2008, he has held academic positions at the Universities of Sheffield (1998-2007) and Cambridge (1990-1997). At the University of Sheffield he has served as Head of the Department of Computer Science and Dean of the Faculty of Engineering. He has a long track record of research in the subject of Machine Learning and has made contributions to the algorithmic and applied aspects of the subject.