Learning by Stretching Deep Networks

Gaurav Pandey GP88@CSA.IISC.ERNET.IN
Ambedkar Dukkipati AD@CSA.IISC.ERNET.IN
Department of Computer Science and Automation
Indian Institute of Science, Bangalore-560012, India

Abstract

In recent years, deep architectures have gained a lot of prominence for learning complex AI tasks because of their capability to incorporate complex variations in data within the model. However, these models often need to be trained for a long time in order to obtain good results. In this paper, we propose a technique, called 'stretching', that allows the same models to perform considerably better with very little training. We show that learning can be done tractably, even when the weight matrix is stretched to infinity, for some specific models. We also study tractable algorithms for implementing stretching in deep convolutional architectures in an iterative manner and derive bounds for its convergence. Our experimental results suggest that the proposed stretched deep convolutional networks are capable of achieving good performance for many object recognition tasks. More importantly, for a fixed network architecture, one can achieve much better accuracy using stretching rather than learning the weights using backpropagation.

1. Introduction

In recent years, there has been a growing amount of research to map the raw representation of input (primarily, images and speech signals) to a feature representation suitable for classification. This has led many researchers to wonder about whether or not the representation learning stage should be kept independent of the classification stage. Most feature extraction algorithms, such as SIFT (Lowe, 2004) and SURF (Bay et al., 2006) keep the two stages separate and have been shown to obtain good performance for object recognition tasks. Furthermore, many feature learning algorithms introduced in the past decade, such as sparse coding (Lee et al., 2006), restricted Boltzmann machines (Hinton, 2002), deep Boltzmann machines (Salakhutdinov & Hinton, 2009), autoencoders (Bengio et al., 2007), $k-$means (Coates et al., 2011), also learn features independently of the classification stage. Recent results have shown that using one or more layers of these feature learning algorithms is sufficient to obtain state of the art results for many object recognition tasks (Bo et al., 2012; Coates et al., 2011).

On the other end of the spectrum, there are learning models that support the coupling of the two stages. Neural networks form the primary examples of such models. While neural networks were introduced three decades back, the lack of a proper way to initialize their weights and the extensive time and memory required for training these models, have kept these models dormant for a long time. In recent years, the development of unsupervised feature learning algorithms, new weight-initialization methods and inexpensive GPUs have brought back neural networks into the main stream of machine learning research. In fact, many state of the art results for object recognition tasks have been obtained using deep convolutional neural networks (Ciresan et al., 2012; Zeiler & Fergus, 2013b).

The reason behind the exceptional performance of neural networks as compared to other classifiers, is that, unlike other classifiers, representational learning is already embedded into the neural network architecture. Hence, if we can provide a proper feature representation for other classifiers, they may be able to perform as well as neural networks, as has been shown in many recent results (Bo et al., 2012; Coates et al., 2011; Pandey & Dukkipati, 2014). Towards this end, we introduce a method for computing kernel (equivalent to representational learning), which can then be fed to a kernel-based classifier. The method for computing the kernel involves 'stretching' the weight matrix learnt by an arbitrary feature learning algorithm.

Contributions: We introduce a technique called 'stretching', that uses minimal amount of learning to achieve reasonable performance for classification tasks. The technique takes a feature learning algorithm as the input and stretches

Proceedings of the 31st International Conference on Machine Learning, Beijing, China, 2014. JMLR: W&CP volume 32. Copyright 2014 by the author(s).

the weight matrix learnt by the algorithm to generate new weight vectors.

1. We show that when the weight matrix has been stretched to infinity, learning can be done tractably with the help of kernels for a special case. The result utilizes the derivation of arc-cosine kernels given by Cho & Saul (2010). We refer to the corresponding architecture as stretched deep network (SDN). In our experimental results, we show that for character recognition tasks, SDNs can achieve reasonable performance with minimal training.

2. In order to deal with realistic images of moderate size, we need to resort to a convolution + pooling based architecture (LeCun et al., 1998). We show the limitations of implementing infinite stretching in such an architecture.

3. We give an iterative method to compute an approximation to infinite stretching. Such an architecture is referred to as convolutional stretched deep network or convolutional SDN.

4. We show that the approximate kernel matrix indeed converges to the kernel matrix corresponding to infinite stretching, when the convolutional SDN has a single convolution + pooling layer.

5. In our experimental results, we show that convolutional SDNs are capable of achieving good performance for many well-known object recognition datasets.

2. Randomly Weighted Neural Networks and the Arc-cosine Kernel

It has long been known that the shallow architectures with random features are capable of achieving reasonable performance for many machine learning tasks (Amit & Geman, 1997). In (Rahimi & Recht, 2008), it was shown that training a shallow neural network by randomly choosing the non-linearities, results in a network that performs no worse than a network where the weights are optimally chosen. This was the motivation for considering randomly weighted neural networks with an infinite number of neurons by Cho & Saul (2010), where they show that learning can be performed tractably for such models by using the kernel trick for many thresholding based non-linearities.

In particular, let the non-linearity under consideration be $\max(0, \mathbf{x})$, which is denoted by $(\mathbf{x})_+$. Let $\mathbf{x}, \mathbf{y} \in \mathbb{R}^D$ be two input instances, and let $\mathbf{h_x}$ and $\mathbf{h_y}$ be their corresponding feature representations in a single-layer neural network with random weights, i.e., $\mathbf{h_x} = \frac{1}{\sqrt{L}}(W^{\mathrm{T}}\mathbf{x})_+$ and $\mathbf{h_y} = \frac{1}{\sqrt{L}}(W^{\mathrm{T}}\mathbf{y})_+$, where $W \in \mathbb{R}^{D \times L}$ is a Gaussian random matrix with independent entries. As $L \to \infty$, the inner product between the feature representations converges, and is given by

$$k(\mathbf{x}, \mathbf{y}) = \frac{1}{2\pi}\|\mathbf{x}\|\|\mathbf{y}\|(\sin\theta + (\pi - \theta)\cos\theta) \ ,$$

where θ is the angle between $\mathbf{x}$ and $\mathbf{y}$ (Cho & Saul, 2010). This kernel is known as the arc-cosine kernel. Feature learning can be incorporated into this kernel, by sampling the columns of W from a multivariate Gaussian distribution with 0 mean and covariance matrix Σ, where Σ has been learnt from the data. Such an approach has been considered by Pandey & Dukkipati (2014). The corresponding kernel is also known as covariance arc-cosine kernel.

3. The Notion of Stretching

In representation learning, one often fixes the desired number of features to a fixed finite quantity. Then, one of the possibly many unsupervised learning algorithms (for instance, RBMs, autoencoders, $k-$means) are used to learn the features from the data. The most commonly used feature learning algorithms include restricted Boltzmann machine and autoencoders. In recent years, triangular $k-$means (Coates et al., 2011), has also been shown to be effective for representation learning from data. Irrespective of the algorithm used, the motive of feature learning is to find a set of weight vectors (centroids in $k-$means) that optimize some objective. The weight vectors have the same dimension as the input data.

The idea behind the introduced concept of stretching is to take the weight vectors learnt by a representation learning model, and generate new weight vectors from the original weight vectors. One possibility is to look at linear combination of the learnt weight vectors. In our model, we assume that the scalars corresponding to these linear combinations have been sampled independently from Gaussian distribution $\mathcal{N}(0, 1)$. Hence, formally we define stretching for our model as follows:

Definition 3.1 (Matrix stretching). *Let $A \in \mathbb{R}^{D \times M}$ be a matrix whose columns denote the weight vectors learnt by some feature learning algorithm. If no feature learning algorithm is used, A can be assumed to be an identity matrix. Let $W \in \mathbb{R}^{M \times L}, L > M$ be a matrix whose entries have been sampled from the standard normal distribution. Then, the stretched matrix A_{cst} is defined as*

$$A_{cst} = \frac{1}{\sqrt{L}}(A \times W)$$

We illustrate the concept of column stretching in two dimensions in Figure 1. As can be observed readily from the

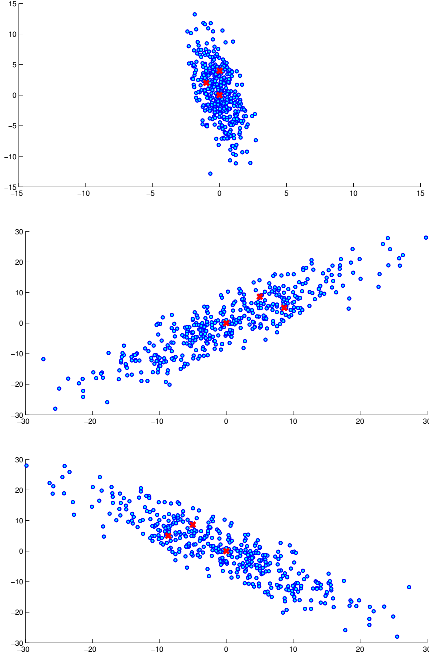

Figure 1. Original weight vectors (in red) and the weight vectors obtained after stretching (in blue)

figure, the new weight vectors appear to be generated from a multivariate Gaussian distribution, whose covariance is controlled by the original points. We will make this result more formal as follows:

Theorem 3.2. *Let $A \in \mathbb{R}^{D \times M}$ be a fixed real valued matrix and $A_{cst} \in \mathbb{R}^{D \times L}$ be the matrix obtained after applying column stretching to A. Then the columns of A_{cst} are distributed according to a multivariate Gaussian distribution with mean 0 and covariance matrix Σ, where Σ is given by*

$$\Sigma = \frac{1}{M} \sum_{m=1}^{M} a_m a_m^{\mathrm{T}} = \frac{1}{M} A A^{\mathrm{T}} \ . \tag{1}$$

Here, a_m denote the columns of A.

4. Single Layer Stretched Network

A single layer stretched architecture comprises of a visible layer, a stretched feature layer and an output layer. Let $\mathbf{x}$ be an instance and A be a matrix whose columns indicate the weight vectors learnt by a feature learning algorithm. Let $W \in \mathbb{R}^{K \times L}$ be a random matrix whose entries have been sampled from the standard normal distribution. The encoding of a node in the stretched feature layer is then given by

$$h_j = \frac{1}{\sqrt{L}} f(\mathbf{x}, A w_j) \, , 1 \leq j \leq L \ , \tag{2}$$

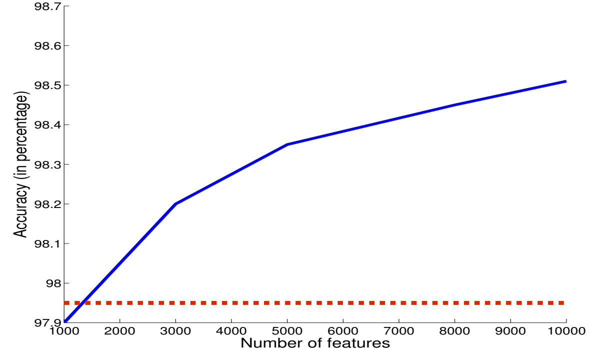

Figure 2. The plot depicts the improvement in classification performance for MNIST data, as the weight matrix learnt by an RBM for the dataset is stretched to increase the number of features. The dotted red line indicates the classification performance achieved by an RBM without stretching.

where f is an activation function that depends upon the feature learning algorithm used. For instance, in case of ReLu RBM, $f(\mathbf{x}, w) = (\mathbf{x}^{\mathrm{T}} w)_+$ and $f(\mathbf{x}, w) = \sigma(\mathbf{x}^{\mathrm{T}} w)$ in case of sigmoid RBM. As is obvious from the above expression, we have just replaced the original weight vectors by their linear combinations.

In order to evaluate the effect of stretching on the performance of the subsequent classifier, we compare the unstretched model with the stretched model for ReLu RBM for increasing values of L for MNIST dataset. The classifier used at the output layer is a linear SVM. No fine-tuning is performed for either of the models. In order to account for the randomness introduced by the matrix W, we average the results over 10 iterations. The weight matrix used for the stretched and the unstretched model is the same and is obtained after training the model for 5 epochs ($\sim 150s$). The results are plotted in Figure 2.

It can be observed from the figure that stretching affects the performance of the subsequent classifier. Furthermore, the performance of the classifier improves as the number of units in the stretched layer increase. Hence, it is particularly interesting to study the case when the number of weight vectors in the stretched matrix A_{cst}, tend to infinity. This means that A_{cst} consists of infinitely many linear combinations of the original weight vectors, where the scalars for these linear combinations are sampled independently from the standard normal distribution. In matrix notation, one can rewrite the above statement as $A_{cst} = A \times W$, where $A \in \mathbb{R}^{D \times M}$ is the original weight matrix, and $W \in \mathbb{R}^{\mathbb{M} \times \infty}$ is the random matrix whose entries have been sampled independently from the standard normal distribution.

Learning can be made feasible for infinitely stretched net-

works by using the kernel trick for the special case, when the activation function is of the form $f(\mathbf{x}) = (\mathbf{x}^{\mathrm{T}} w)_+$. In particular, we note that learning a linear classifier is equivalent to learning a kernel machine with the linear kernel. Here, the linear kernel corresponds to an inner product in the feature space, where the feature representation of an instance $\mathbf{x}$ is obtained by using equation (2). The next theorem allows us to compute the inner product in the feature space as $L \to +\infty$.

Theorem 4.1. *Let A be the weight matrix learnt by ReLu RBM. Let $W \in \mathbb{R}^{K \times L}$ be a random matrix whose entries have been sampled indepependently from standard normal distribution. Let $\mathbf{x}$ and $\mathbf{y}$ be two instances whose feature representations are given by*

$$h_\ell(\mathbf{x}) = \frac{1}{\sqrt{L}} (\mathbf{x}^{\mathrm{T}} A w_\ell)_+, \ \ 1 \le \ell \le L \ \text{ and}$$
$$h_\ell(\mathbf{y}) = \frac{1}{\sqrt{L}} (\mathbf{y}^{\mathrm{T}} A w_\ell)_+, \ \ 1 \le \ell \le L \ .$$

Let $\mathbf{h_x} = (h_\ell(\mathbf{x}))^{\mathrm{T}}_{1 \le \ell \le L}$ and $\mathbf{h_y} = (h_\ell(\mathbf{x}))^{\mathrm{T}}_{1 \le \ell \le L}$. Then the inner product between the feature representation of $\mathbf{x}$ and $\mathbf{y}$ as $L \to \infty$ is given by

$$\lim_{L \to \infty} \mathbf{h}_\mathbf{x}^{\mathrm{T}} \mathbf{h_y} = \frac{1}{2\pi} \|A^{\mathrm{T}} \mathbf{x}\| \|A^{\mathrm{T}} \mathbf{y}\| (\sin\theta_A + (\pi - \theta_A)\cos\theta_A), \tag{3}$$

where $\theta_A = \cos^{-1} \frac{\mathbf{x}^{\mathrm{T}} A A^{\mathrm{T}} \mathbf{y}}{\|A\mathbf{x}\| \|A\mathbf{y}\|}$.

The proof of the above theorem is given in the supplementary material. If we replace $\Sigma = \frac{1}{M} A A^{\mathrm{T}}$ in the equation (3), we get

$$k_\Sigma(\mathbf{x}, \mathbf{y}) = \lim_{L \to \infty} \mathbf{h}_\mathbf{x}^{\mathrm{T}} \mathbf{h_y}$$
$$= \frac{M}{2\pi} \|\mathbf{x}\|_\Sigma \|\mathbf{y}\|_\Sigma (\sin\theta_\Sigma + (\pi - \theta_\Sigma)\cos\theta_\Sigma) \ ,$$

where $\|\mathbf{x}\|_\Sigma = \sqrt{\mathbf{x}^{\mathrm{T}} \Sigma \mathbf{x}}$ and $\theta_\Sigma = \cos^{-1} \frac{\mathbf{x}^{\mathrm{T}} \Sigma \mathbf{y}}{\sqrt{\mathbf{x}^{\mathrm{T}} \Sigma \mathbf{x}} \sqrt{\mathbf{y}^{\mathrm{T}} \boldsymbol{\Sigma} \mathbf{y}}}$. This is a scaled version of the covariance arc-cosine kernel proposed by Pandey & Dukkipati (2014).

5. Incorporating Translational Invariance in SDNs

As we show in our experimental results, infinitely stretched networks allow us to obtain good performance for object recognition tasks when the images are all centred and have few pose variations, for instance, MNIST data set. However, for complex real world tasks such as scene labelling, the above technique can not be expected to perform equally well, since no prior information about the data (for instance, the information that objects can be present at any location in the model) is incorporated in the model. The usual way to incorporate translational invariance in neural networks is to divide the image into overlapping patches, where each patch is a $d \times d$ square of pixels in an image. The patches are flattened from matrices to vectors and the same weight matrix is then learnt using each flattened patch in either supervised or unsupervised mode. The features are then pooled over a region in the image, where the region may be fixed *a priori* (LeCun et al., 1998), or chosen stochastically (Zeiler & Fergus, 2013a). The above process is then repeated by treating the pooled feature maps from the previous layer as input. Such an architecture is called a convolutional neural network.

5.1. Convolutional Stretched Deep Networks

Assuming that one prefers to use average pooling and a single pooling layer, there is a straightforward method to incorporate translational invariance in infinitely stretched networks. Let $\{p_1, .., p_I\}$ be the flattened patches in a region of an image $\mathbf{x}$, and $\{q_1, ..., q_I\}$ be the flattened patches of the corresponding region in $\mathbf{y}$. Let A be the weight matrix learnt by some feature learning algorithm. If no feature learning algorithm is used, A can be assumed to be an identity matrix. Then, the inner product between the stretched and pooled feature representation of $\mathbf{x}$ and $\mathbf{y}$ for the chosen region is given by

$$\mathbf{h}_\mathbf{x}^{\mathrm{T}} \mathbf{h_y} = \frac{1}{LI^2} \left(\sum_{i=1}^{I} (p_i^{\mathrm{T}} A W)_+ \right) \left(\sum_{i=1}^{I} (q_i^{\mathrm{T}} A W)_+ \right) \ .$$

As $L \to \infty$, the above inner product can be rewritten as

$$k(\mathbf{x}, \mathbf{y}) = \lim_{L \to \infty} \mathbf{h}_\mathbf{x}^{\mathrm{T}} \mathbf{h_y}$$
$$= \lim_{L \to \infty} \frac{1}{LI^2} \sum_{i=1}^{I} \sum_{j=1}^{I} \sum_{\ell=1}^{L} (p_i^{\mathrm{T}} A w_\ell)_+ (q_j^{\mathrm{T}} A w_\ell)_+$$
$$= \frac{1}{I^2} \sum_{i=1}^{I} \sum_{j=1}^{I} k_\Sigma(p_i, q_j) \ .$$

Several important points can be noted about the manner in which the above kernel has been defined. Firstly, the above approach can be applied only if the pooling mechanism used is sum pooling or average pooling. Currently, we are not aware of any other approach to incorporate more general pooling mechanisms exactly in infinitely stretched networks. Secondly, the computation of a single entry of the pooled kernel matrix involves $\mathcal{O}(I^2)$ kernel computations, where I is the number of patches in the region. In general, the number of patches can be as high as the number of pixels in the region. Assuming a region of size 100×100, this means that computation of one entry of the pooled kernel matrix may require 10^8 kernel computations, clearly a prohibitively large number.

However, the main drawback of the above architecture is not its practical infeasibility, but the inherent limitation

of the model itself. In a general convolutional architecture (LeCun et al., 1998), the first pooling layer pools the features over a smaller region, with successive layers pooling over larger and larger regions. The possible exception is the architecture in (Coates et al., 2011), where the features are pooled over a large region in a single pooling layer. However, in the approach described above, once we obtain the kernel matrix after first layer pooling, all information about location of pixels in the original image is lost. Hence, there is no way to apply a second layer pooling to the model. Hence, we are limited to using single layer pooling only.

5.2. Approximating Convolutional SDNs

Because of the limitations of using pooling in infinitely stretched networks, we need to resort to an approximation of the model. In Algorithm 1, we describe the architecture of an approximately stretched network with a single convolution layer and a pooling layer. The architecture addressed the first two issues associated with a convolutional SDN mentioned in Section 5.1. The issue of multiple pooling layers is addressed in Section 5.3. In order to convey the most important points of the algorithm, we have assumed that each image has a single pooling region in the first pooling layer. It is quite straightforward to extend the model for the case where each image has multiple pooling regions in the first pooling layer.

The proposed algorithm gives an iterative, memory-efficient method of computing the pooled kernel matrix. We show below that if the algorithm is run long enough, the resulting pooled kernel matrix indeed converges to the true pooled kernel matrix, that we obtained in the previous section. Here, we have shown the result for average pooling. For max-pooling, we need to use the inequality $\mathrm{var}(\max_{i=1}^{I} X_i) \leq \sum_{i=1}^{I} \mathrm{var}(X_i)$, where $\mathrm{var}(X)$ denotes the variance of the random variable X. Note that, one can not obtain the mean ($k(\mathbf{x}, \mathbf{y})$ in equation (6)) in closed forms for max-pooling. However, the result will still remain valid.

Theorem 5.1. *The pooled kernel matrix of approximately stretched network converges in probability to the pooled kernel matrix of infinitely stretched network. Furthermore, let T be the number of iterations for which Algorithm 1 has been run. Then, for fixed instances $\mathbf{x}$ and $\mathbf{y}$, with probability at least 1-δ,*

$$|k_{ap}(\mathbf{x}, \mathbf{y}) - k(\mathbf{x}, \mathbf{y})| \leq \sqrt{\frac{2}{TL\delta}} \sqrt{\frac{1}{I^2} \sum_{i=1}^{I} \sum_{j=1}^{I} \|Ap_i\|^2 \|Aq_j\|^2} \ , \quad (5)$$

where $k(\mathbf{x}, \mathbf{y})$ is the exact value of the kernel entry and $k_{ap}(\mathbf{x}, \mathbf{y})$ is the approximate kernel value after T iterations.

Algorithm 1 Iterative computation of the convolved kernel matrix

---

Input: A set of training instances
Output: The pooled kernel matrix

- Initialize the kernel matrix k_{ap} to all zeros.
- Learn the weight matrix A from patches of images using an unsupervised feature learning algorithm.
- Repeat the following steps till convergence
  1. Sample the entries of the random matrix $W \in \mathbb{R}^{D \times L}$ independently from standard normal distribution
  2. Compute the pooled feature representation of each instance using the random matrix W. In particular, the pooled feature representation of $\mathbf{x}$ is given by
  $$\mathbf{h}_{\mathbf{x}} = \frac{1}{I\sqrt{L}} \sum_{i=1}^{I} (WA^{\mathrm{T}} p_i)_+ \ ,$$
  where p_i are the patches in $\mathbf{x}$.
  3. Update the approximate kernel matrix by using the following equation
  $$k_{ap}^{(t+1)}(\mathbf{x}, \mathbf{y}) = \frac{1}{t+1}(t \times k_{ap}^{(t)}(\mathbf{x}, \mathbf{y}) + \mathbf{h}_{\mathbf{x}}^{\mathrm{T}} \mathbf{h}_{\mathbf{y}}) \quad (4)$$
- Return the kernel matrix k_{ap}.

The proof of the above theorem is give in the supplementary material. The above result implies that, if the difference between a kernel entry in approximate and exact kernel matrix after 1 iteration is ϵ with probability at least $1-\delta$, one can say that with the same probability, we obtain a 10 fold reduction in estimation error by scaling L ten times and repeating the experiment for ten iterations.

By using the above approximation, one can incorporate any non-linearity in the model architecture. One disadvantage of the model is that one may need to run the algorithm multiple number of times to obtain the pooled kernel matrix. However, since the entries of the random matrix W are sampled independently, it is trivial to parallelize the model on multiple nodes or adapt it to any distributed system. Furthermore, in our experiments, we found that 20 iterations were often enough to obtain good results for many object recognition tasks.

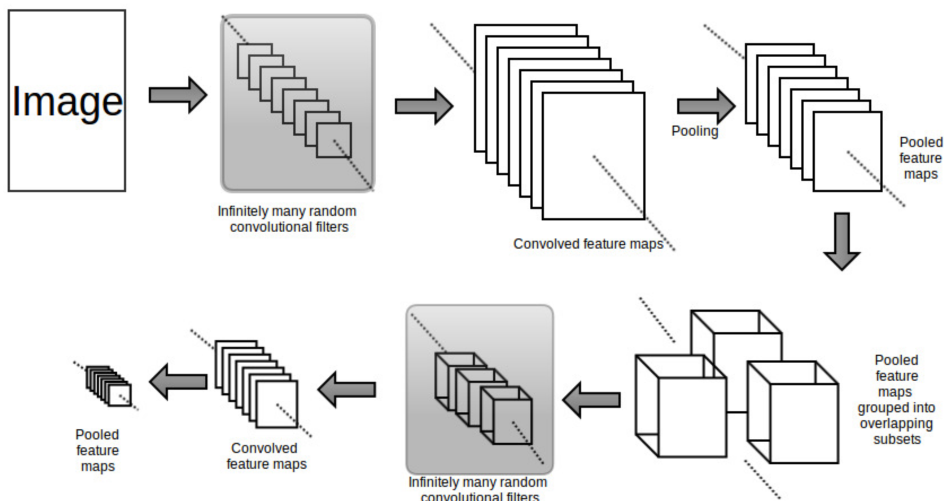

Figure 3. A convolutional SDN with two pooling layers. A convolutional SDN is equivalent to a convolutional neural network with infinitely many random filters in each convolutional layer. The dotted lines indicate infinitely many feature maps/filters.

5.3. Extension to multiple layers

Instead of using the output of the first pooling layer to compute the kernel matrix, one can feed it to another layer of convolution + pooling. Specifically, let $\mathbf{x}$ be an image of size $m \times m$ and U be a tensor of size $d \times d \times L_1$ whose entries are sampled from $\mathcal{N}(0,1)$. Let $u_1, ..., u_{L_1} \in \mathbb{R}^{d \times d}$ be the sub-tensors of U obtained by fixing the last dimension of U. For the sake of clarity, we assume that the matrix A learnt from flattened patches of $\mathbf{x}$, is an identity matrix. It is straightforward to merge the matrix A with tensor U, when it is not an identity matrix.

In the first convolutional layer, $\mathbf{x}$ is convolved with $u_1, ..., u_{L_1}$ (followed by rectification) to obtain L_1 feature maps of size $(m-d+1) \times (m-d+1)$, which are then pooled using any of the possible pooling methods. Let the resultant size of each feature map be $m_1 \times m_1$. Let us denote the feature maps at the output of the first pooling layer as $f_1, ..., f_{L_1}$. Hence, $f_i = \text{pool}(x * u_i)$, where * denotes the 2D convolution operator and $\text{pool}(.)$ denotes the pooling operation.

Next, the L_1 feature maps at the output of the first pooling layer are divided into L_2 overlapping subsets, where each subset consists of Q feature maps. One can represent each subset as a tensor of size $m_1 \times m_1 \times Q$. Let us denote them as $\hat{x}_1, ..., \hat{x}_{L_2}$. Let $V \in \mathbb{R}^{d \times d \times Q \times L_2}$ be a tensor whose entries have been sampled from $\mathcal{N}(0,1)$. Let $v_1, ..., v_{L_2} \in \mathbb{R}^{d \times d \times Q}$ denote the sub-tensors of V obtained by fixing the last dimension. Each subset $\hat{x}_1, ..., \hat{x}_{L_2}$ is convolved (3-D convolution) with its corresponding subtensor to obtain L_2 feature maps, which are then pooled. We denote the resultant feature maps as $g_1, ...g_{L_2}$, where $g_i = \text{pool}\,(\hat{x}_i * v_i)$.

One can add further layers in a similar manner. The output of the last pooling layer is then fed to a linear kernel. This entire procedure corresponds to one single iteration of kernel computation. A pictorial representation of SDN is given in Figure 3. The entire algorithm for computing the kernel is informally described in Algorithm 2.

Algorithm 2 Iterative computation of the convolved kernel matrix for multiple stages of convolution + pooling

Input: A set of training instances
Output: The pooled kernel matrix

- Initialize the kernel matrix k_{ap} to be all zeros.
- Repeat until the kernel matrix k_{ap} converges
  - Sample the entries of tensors U, V etc., from $\mathcal{N}(0,1)$.
  - Use the steps described in Section 5.3 to compute the feature maps at the output of the last pooling layer for each instance. Let they be denoted as $g_1, ...g_{L_2}$. Flatten it to get a single vector $\mathbf{g}$.
  - Update the kernel matrix as

$$k_{ap}^{(t+1)}(\mathbf{x},\mathbf{y}) = \frac{1}{t+1}(t \times k_{ap}^{(t)}(\mathbf{x},\mathbf{y}) + \mathbf{g}_\mathbf{x}^\mathrm{T}\mathbf{g}_\mathbf{y})$$

- Return the kernel matrix k_{ap}.

Finally, we would like to mention that in all our exper-

iments using convolution SDNs, we assume an infinite fully-connected random layer as the final layer, while computing the final kernel matrix. Let k_{ap} be the kernel matrix obtained after multiple stages of convolution + pooling and k_{full} be the kernel matrix after the fully connected random layer. Furthermore, let $\mathbf{g_x}$ and $\mathbf{g_y}$ be the feature representation of two instances $\mathbf{x}$ and $\mathbf{y}$ after the the last pooling stage.

$$k_{full}(\mathbf{x},\mathbf{y}) = \frac{1}{2\pi}\|\mathbf{g_x}\|\|\mathbf{g_y}\|(\sin\theta + (\pi-\theta)\cos\theta)$$
$$= \frac{1}{2\pi}\sqrt{k_{ap}(\mathbf{x},\mathbf{x})}\sqrt{k_{ap}(\mathbf{y},\mathbf{y})}(\sin\theta + (\pi-\theta)\cos\theta) \ ,$$

where θ is the angle between $\mathbf{g_x}$ and $\mathbf{g_y}$ and is given by

$$\theta = \cos^{-1}\left(\frac{k_{ap}(\mathbf{x},\mathbf{y})}{\sqrt{k_{ap}(\mathbf{x},\mathbf{x})}\sqrt{k_{ap}(\mathbf{y},\mathbf{y})}}\right) \ .$$

Hence, the kernel matrix after the fully connected stage only depends upon the kernel matrix after the last pooling stage and hence, it can be computed after Algorithm 2 stops.

6. Experimental results

We tested the performance of stretched networks for classification on various data sets. We have separated the results for non-convolutional stretched networks from those of convolutional SDNs. Furthermore, whenever we refer to convolutional SDNs, we mean the approximated convolutional SDN derived using Algorithm 1. The classifier used in all the experiments for SDN, is kernel SVM.

6.1. Experiments results for SDNs

6.1.1. MNIST

MNIST (LeCun et al., 1998) is a standard dataset for character recognition with 50000 training and 10000 test samples of digits ranging from 0 to 9. It is a relatively simple dataset for the task of object recognition, and gives good performance with non-pooling architectures as well. Hence, we use this dataset to test the performance of SDN against other well-known classifiers.

Among architectures that don't incorporate translational invariance, the best results for MNIST have been obtained by using deep Boltzmann machines (DBM). However, unsupervised pre-training of deep Boltzmann machines is very costly. Even for the MNIST dataset, pretraining a DBM with 500 units in the first hidden layer and 1000 units in the second hidden layer, takes nearly 2 days to achieve an accuracy of 99.05% (after fine-tuning with backpropagation and no dropout). In contrast, we obtained the same result by infinitely stretching the weight matrix learnt by a ReLu RBM with 1000 hidden units after unsupervised training for nearly 200 seconds (6 iterations). No fine-tuning is used for this model.

Table 1. Classification accuracies on MNIST dataset using various algorithms. Here, SDN-RBM refer to a single layer infinitely stretched network, whose weight matrix has been learnt using RBM. Unsupervised training of RBM took 200 seconds while learning the parameters of kernel SVM took about 30 mins. DBN stands for a deep belief network as described by Hinton et al. (2006). The numbers in parenthesis indicate the number of hidden units in each layer.

ALGORITHM	ACCURACY	TIME TAKEN (APPROX.)
SDN-RBM (1000)	**99.05%**	$<$ 1 HR
DBN-RBM (1000)	98.35%	$<$ 1 HR
DBN-RBM (1000-500-2000)	98.8%	1.5 HRS
DBM (NO DROPOUT)	99.05%	2 DAYS
DBM (WITH DROPOUT)	**99.21%**	2 DAYS

Table 2. Classification accuracies on Caltech-101 dataset using various algorithms. The numbers in parenthesis indicate the number of hidden units in each layer. The state of the art is achieved by an 8 layered convolutional DBN with 5 convolution layers and 3 pooling layers (Zeiler & Fergus, 2013b).

ALGORITHM	ACCURACY
CONV. SDN-RBM (64-256)	**74.3%**
CONV. DBN (64-256) (NOT FINETUNED)	64.1%
CONV. DBN (64-256) (FINETUNED)	65.5%
CONV. DBN (8 LAYERED)	**86.5%**

6.2. Experimental results for Convolutional SDNs

Next, we present the results obtained by using convolutional SDNs for standard image recognition tasks for images of moderate size.

6.2.1. CALTECH-101 DATASET

The Caltech-101 dataset (Fei-Fei et al., 2007) consists of pictures of objects belonging to 101 categories with about 40 to 800 objects per categories. For each object class, we used 30 samples for training and a maximum of 30 samples for testing. We use the same preprocessing as done by Jarrett et al. (2009). Furthermore, we average the result over 5 draws from the training set.

Effectively, we use a similar architecture as used by Jarrett et al. (2009). Three stages of feature extraction are used. The first two stages are convolution + pooling stages,

while the last stage is a fully connected layer. We randomly extract 9×9 patches from the images and learn a weight matrix with 64 weight vectors from these patches using an ReLu RBM. The weight matrix is then stretched by multiplication with a random matrix of size 64×64. We use average pooling with a 10×10 boxcar filter and 5×5 down-sampling. The output of first stage of feature extraction stage is 64 feature maps of size 26×26. In the second stage, we randomly combine 30 feature maps from the previous layers using 256 9×9 kernels . Again, we use average pooling in this layer with a 6×6 boxcar filter with a 4×4 down-sampling step. The output of the second layer of feature extraction is 256 feature maps of size 4×4. In both stages, we use the $\max(0, x)$ non-linearity and local contrast normalization. In the third stage, we map the output of the second stage (256 feature maps of size 4×4) to infinitely many feature maps using an arc-cosine kernel. We repeat the entire process for 20 iterations to construct the final kernel matrix as discussed in Algorithm 2.

By iterating over the above architecture for 20 iterations, we obtain an accuracy of 74.3%. We plot the change in classification performance with the no. of iterations for Caltech-101 dataset in Figure 4 for the first 10 iterations. The result should be compared with accuracy achieved by an unstretched architecture of the same size, which is 64.1%. The accuracy achieved by finetuning the same architecture is 65.5% as mentioned in (Jarrett et al., 2009).

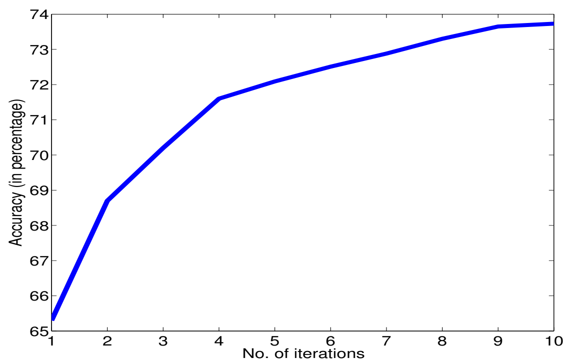

Figure 4. The plot depicts the improvement in classification performance for Caltech-101 dataset with the respect to no. of iterations of stretching.

6.2.2. STL-10 DATASET

STL-10 dataset (Coates et al., 2011) is an image recognition dataset with 10 classes and 500 training and 800 test images per class. We use the same preprocessing as in the previous case. Again, three layers of feature extraction are used with the final layer being a fully connected layer. As in the previous case, we randomly extract 9×9 patches from the images and learn a weight matrix with 64 weight vectors from these patches using an ReLu RBM. The first pooling layer uses an 8×8 boxcar filter with 3×3 average sampling. The second pooling layer uses 5×5 boxcar filter with 4×4 average sampling. The rest of the architecture remains the same as in the previous case. The output of the second stage is mapped to infinitely many feature maps by using the arc-cosine kernel. After iterating over the above architecture for 20 iterations, we achieve an accuracy of 65.7%. The change in accuracy of an SDN with the number of iterations is given in Figure 5. The result should be compared with accuracy achieved by an unstretched architecture of the same size, which is 53.9%. The accuracy achieved by finetuning the same architecture is 55.3%.

Table 3. Classification accuracies on STL-10 dataset using various algorithms. The best result is obtained using the algorithm proposed in (Swersky et al., 2013)

Algorithm	Accuracy
Conv. SDN-RBM (64-256)	**65.7%**
Conv. DBN (64-256) (not finetuned)	53.9%
Conv. DBN (64-256) (finetuned)	55.3%
Multi-task BayesOpt	**70.1%**

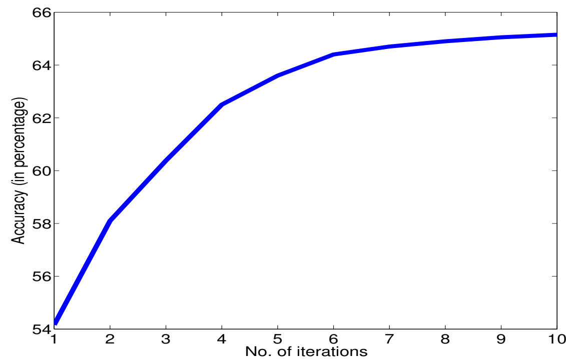

Figure 5. The plot depicts the improvement in classification performance for STL-10 dataset with the respect to no. of iterations of stretching.

7. Conclusion

In this paper, we proposed the technique called stretching for deep networks, and gave exact as well as approximate algorithms to compute the kernel matrix for stretched deep networks. We showed that for convolutional SDNs with a single pooling layer, the approximate kernel entry converges to the true kernel entry. Results on benchmark datasets suggests that stretching a convolutional neural network can sometimes give much better performance than fine-tuning it using backpropagation.

References

Amit, Yali and Geman, Donald. Shape quantization and recognition with randomized trees. *Neural computation*, 9(7):1545–1588, 1997.

Bay, Herbert, Tuytelaars, Tinne, and Gool, Luc Van. Surf: Speeded up robust features. In *Proceedings of the Ninth European Conference on Computer Vision*, May 2006.

Bengio, Yoshua, Lamblin, Pascal, Popovici, Dan, and Larochelle, Hugo. Greedy layer-wise training of deep networks. *Advances in Neural Information Processing Systems*, 19:153, 2007.

Bo, Liefeng, Ren, Xiaofeng, and Fox, Dieter. Multipath sparse coding using hierarchical matching pursuit. In *NIPS Workshop on Deep Learning*, 2012.

Cho, Youngmin and Saul, Lawrence K. Large-margin classification in infinite neural networks. *Neural Computation*, 22(10):2678–2697, 2010.

Ciresan, Dan, Meier, Ueli, and Schmidhuber, Jürgen. Multi-column deep neural networks for image classification. In *Computer Vision and Pattern Recognition (CVPR)*, pp. 3642–3649. IEEE, 2012.

Coates, Adam, Ng, Andrew Y, and Lee, Honglak. An analysis of single-layer networks in unsupervised feature learning. In *International Conference on Artificial Intelligence and Statistics*, pp. 215–223, 2011.

Fei-Fei, Li, Fergus, Rob, and Perona, Pietro. Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories. *Computer Vision and Image Understanding*, 106 (1):59–70, 2007.

Hinton, Geoffrey E. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14 (8):1771–1800, 2002.

Hinton, Geoffrey E, Osindero, Simon, and Teh, Yee-Whye. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006.

Jarrett, Kevin, Kavukcuoglu, Koray, Ranzato, MarcAurelio, and LeCun, Yann. What is the best multi-stage architecture for object recognition? In *International Conference on Computer Vision*, pp. 2146–2153. IEEE, 2009.

LeCun, Yann, Bottou, Léon, Bengio, Yoshua, and Haffner, Patrick. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Lee, Honglak, Battle, Alexis, Raina, Rajat, and Ng, Andrew. Efficient sparse coding algorithms. In *Advances in Neural Information Processing Systems*, pp. 801–808, 2006.

Lowe, David G. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

Pandey, Gaurav and Dukkipati, Ambedkar. To go deep or wide in learning? In *Seventeenth International Conference on Artificial Intelligence and Statistics, JMLR W& CP*, volume 33, pp. 724–732, 2014.

Rahimi, Ali and Recht, Benjamin. Weighted sums of random kitchen sinks: Replacing minimization with randomization in learning. In *Advances in neural information processing systems*, pp. 1313–1320, 2008.

Salakhutdinov, Ruslan and Hinton, Geoffrey E. Deep boltzmann machines. In *International Conference on Artificial Intelligence and Statistics*, pp. 448–455, 2009.

Swersky, Kevin, Snoek, Jasper, and Adams, Ryan P. Multi-task bayesian optimization. In *Advances in Neural Information Processing Systems*, pp. 2004–2012, 2013.

Zeiler, Matthew D and Fergus, Rob. Stochastic pooling for regularization of deep convolutional neural networks. *arXiv preprint arXiv:1301.3557*, 2013a.

Zeiler, Matthew D and Fergus, Rob. Visualizing and understanding convolutional neural networks. *arXiv preprint arXiv:1311.2901*, 2013b.