HOW FAR CAN WE GO WITHOUT CONVOLUTION: IMPROVING FULLY-CONNECTED NETWORKS

Zhouhan Lin & Roland Memisevic
Université de Montréal
Canada
zhouhan.lin@umontreal.ca, roland.umontreal@gmail.com

Kishore Konda
Goethe University Frankfurt
Germany
konda.kishorereddy@gmail.com

ABSTRACT

We propose ways to improve the performance of fully connected networks. We found that two approaches in particular have a strong effect on performance: linear bottleneck layers and unsupervised pre-training using autoencoders without hidden unit biases. We show how both approaches can be related to improving gradient flow and reducing sparsity in the network. We show that a fully connected network can yield approximately 70% classification accuracy on the permutation-invariant CIFAR-10 task, which is much higher than the current state-of-the-art. By adding deformations to the training data, the fully connected network achieves 78% accuracy, which is close to the performance of a decent convolutional network.

1 INTRODUCTION

Convolutional neural networks (CNN) have had huge successes since Krizhevsky et al. (2012), especially in computer vision applications. The main computational idea behind these, *weight sharing*, is unfortunately not biologically plausible, and it does not map nicely to simple, densely parallel hardware designs, which may one day yield lower-energy, and more efficient ways to run neural networks. The reason is that weight sharing requires long-range communication, for example, when distributing derivatives, during learning.

In this paper, we explore the performance that one can possibly achieve with a neural network without convolutions, in other words, a neural network learning on "permutation invariant" classification tasks. We use these as a test-bed for studying alternative architectural design choices beside convolution, which are biologically plausible and potentially more hardware friendly. Studying these design choices is relevant also because many tasks and sub-tasks do not come with any local translation invariance, making them not amenable to convolution (an example being the hidden layer in a recurrent neural network).

Two architectural choices we found to have a strong impact on performance are (i) linear bottleneck layers and (ii) pre-training using autoencoders whose hidden units have no biases. Taken together, these two approaches allow us a network to achieve state-of-the-art performance on the permutation-invariant CIFAR-10 task [1], and by adding deformations (thus removing the permutation invariance requirement) it achieves performance close to the range of that achieved by CNNs.

[1] An example implementation that generates the state-of-the-art on this task is available at https://github.com/hantek/zlinnet

1.1 SPARSITY IN NEURAL NETWORKS

Both approaches can be viewed from the perspective of sparsity in a neural network. Sparsity is commonly considered as a desirable property, as it can provide an optimal balance between high-capacity, but "entangled", distributed representations on the one hand, and low-capacity, but easily-decodable, one-hot representations on the other (eg., Foldiak (2003)). Because of its benefits, sparsity is often encouraged explicitly as a regularizer in statistical machine learning models. While sparse codes could also provide energy-efficiency when implemented in the right type of hardware, the floating-point hardware underlying most common machine learning models does not exploit this.

Unfortunately, in deep, multilayer neural networks sparsity comes at a price: in common activation functions, such as sigmoid or ReLU, zero (or almost zero) activations are attained at values where derivatives are zero, too. This prevents derivatives from flowing through inactive hidden units, and makes the optimization difficult. Stated in terms of the vanishing gradients problem (e.g., Hochreiter et al. (2001)) this means that for a ReLU activation many Jacobians are diagonal matrices containing many zeros along the diagonal.

To alleviate this problem, recently several activation functions were proposed, where zero derivatives do not, or are less likely to, occur. In the PReLU activation function (He et al., 2015), for example, the zero-derivative regime is replaced by a learned, and typically small-slope, linear activation. Another approach is the Maxout activation (Goodfellow et al., 2013), which is defined as the maximum of several linear activations. Accordingly, preventing zero-derivatives this way was shown to improve the optimization and the performance of multilayer networks. Unfortunately, these methods solve the problem by giving up on sparsity altogether, which raises the question whether sparsity is simply not as desirable as widely assumed or whether the benefit of the optimization outweigh any detrimental effect on sparsity. Sparsity also plays a crucial role in unsupervised learning (which can also be viewed as a way to help the optimization (Saxe et al., 2013)), where these activations have accordingly never been successfully applied.

This view of sparsity motivates us to revisit a simple, but as we shall show surprisingly effective, approach to retaining the benefits of sparsity without preventing gradient-flow. The idea is to interleave linear, low-dimensional layers with the sparse, high-dimensional layers containing ReLU or sigmoid activations. We show that this approach outperforms equivalent PReLU and Maxout networks on the fully supervised, permutation invariant CIFAR-10 task.

A second detrimental side-effect of sparsity, discussed in more detail in Konda et al. (2014), is that for a ReLU or sigmoid unit to become sparse it typically learns strongly negative bias-terms. In other words, while sparse activations can be useful in terms of the learning objective (for example, by allowing the network to carve up space into small regions) it forces the network to utilize bias terms that are not optimal for the *representation* encoded in hidden layer activations. Konda et al. (2014), for example, suggest bias-terms equal to zero to be optimal and propose a way to train an autoencoder with zero-bias hidden units. We suggest in Section 3 that pre-training a network with these autoencoders may be viewed as a way to orthogonalize subsets of weights in the network, and show that this yields an additional strong performance improvement.

We shall discuss the motivation for linear bottleneck layers in the next section and pre-training using autoencoders in Section 3, followed by experimental results in Section 4.

2 LINEAR BOTTLENECK LAYERS

One drawback of sparsity in a deep network is that it amounts to data scarcity: a weight whose post-synaptic unit is off 80% of the time will effectively get to see only 20% of the training data. In lower layers of a convolutional network, this problem is compensated by weight-sharing, which increases the effective number of training examples (patches in that case) per weight by a factor of several thousand. In a fully connected layer (more precisely, a layer without weight sharing) such compensation is not possible and the only way to counter this effect would be by increasing the training set size.

Sparsity is a common and stable effect in neural networks containing ReLU or sigmoid units, and it can be related to the fact the biases are driven to zero in response to regularization (eg., Konda

et al. (2014)). Figure 1 (left) shows the sparsity level attained after training ReLU MLPs. [2] The plots show that the sparsity of the hidden presentation increases with the depth of the network and increase further in the presence of dropout.

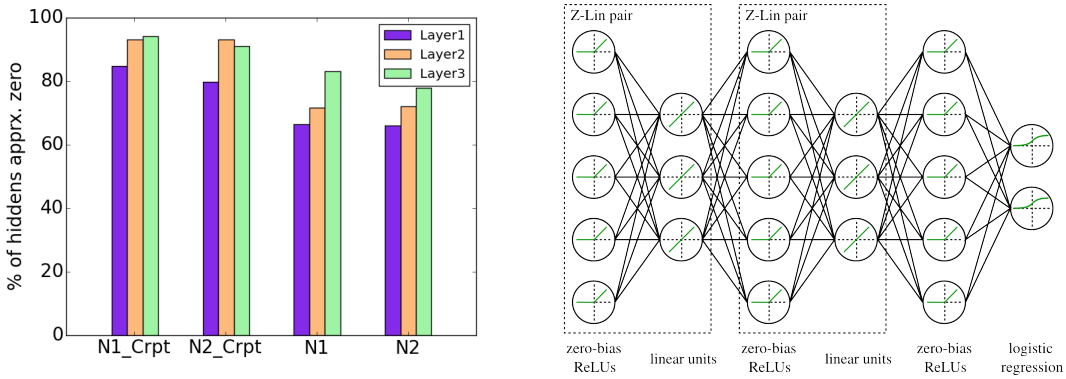


Figure 1: (left) The sparsity in different layers of two MLPs N1(1000-2000-3000 units) and N2(2000-2000-2000 units) trained with and without dropout on CIFAR-10 dataset. N1_Crpt, N2_Crpt: Experiments with dropout. (right) Illustration of a network with linear bottleneck layers.

2.1 Linear Layers as Distribution Reshaping

It is well-known that different activation functions work best when the input is in a reasonable scale, which is the motivation for using pre-processing by mean subtraction and scaling, for example. Furthermore, incorporating them into hidden unit activation, can yield further performance improvements (Ioffe & Szegedy, 2015).

Here, we focus on the distribution over the outputs of a unit. The output of a linear neuron Y with inputs $\vec{X} = (X_1, X_2, ..., X_i, X_N)$ is given by $Y = \sum_i^N w_i X_i + b$, where w_i is the entry in the weight vector corresponding to input node i, b is the bias of the linear neuron, and N denotes the input dimension. Assume the data fed into each input node X_i is independent and has a finite mean μ_i and variance σ_i^2. In the net input to a neuron, the mean and variance of each $w_i X_i$ term are tuned by its corresponding weight values:

$$\hat{\mu}_i = w_i \mu_i; \qquad \hat{\sigma_i}^2 = w_i^2 \sigma_i^2 \tag{1}$$

Thus, Y can be viewed as bias b plus a sum of N different random variables, whose means and variances are $\hat{\mu}_i$ and $\hat{\sigma_i}^2$ correspondingly. Note that here we cannot apply central limit theorem directly to draw the conclusion that Y subjects to Gaussian distribution as N goes infinity, because those $w_i X_i$ terms are not i.i.d. However, there is an important extension to the central limit theorem called Lyapunov theorem (DeGroot et al., 1986). It basically describes, if a sequence of random variables with finite mean and variance are independent but not necessarily identically distributed, the distribution of the sum:

$$S = \Big(\sum_{i=1}^{N} w_i X_i - \sum_{i=1}^{N} \hat{\mu}_i \Big) \Big(\sum_{i=1}^{N} \hat{\sigma_i}^2 \Big)^{-\frac{1}{2}} \tag{2}$$

tends to standard Gaussian distribution for $N \to \infty$. If we write Y in terms of S, that is, $Y = \Big(\sum_{i=1}^{N} \hat{\sigma_i}^2 \Big)^{\frac{1}{2}} S + b + \sum_{i=1}^{N} \hat{\mu}_i$ we see that Y approximates a Gaussian distribution whose mean is $b + \sum_{i=1}^{N} \hat{\mu}_i$ and whose variance equals to $\sum_{i=1}^{N} \hat{\sigma_i}^2$, when the input dimension $N \to \infty$. For

[2] Two MLPs (1000-2000-3000 units and 2000-2000-2000 units, respectively), trained for 501 epochs on CIFAR-10. Each network was trained once with dropout and once without. For the experiments with dropout, an input noise level of 0.2 and hidden noise level of 0.5 was used.

mean-centered data, we have $\hat{\mu}_i \approx 0$. Thus the actual mean value of Y is merely dominated by b in which case the p.d.f. of Y is:

$$p_{Lin}(y) \approx \frac{1}{\sqrt{2\pi \sum_{i=1}^{N} \hat{\sigma_i}^2}} e^{-\frac{(y-b)^2}{2\sum_{i=1}^{N} \hat{\sigma}_i^2}} \tag{3}$$

This form of asymptotic distribution holds regardless of the weight w_i. That means, no matter how the network is trained, or even not trained, the asymptotic distribution of output of a linear unit tends to be Gaussian.

Since the pre-hidden activation of ReLU is linear it tends to be Gaussian as well. The ReLU activation then simply sets all negative values to zero which yields:

$$p_{ReLU}(y) = \begin{cases} \int_{-\infty}^{0} p_{Lin}(y)\,\mathrm{d}y \cdot \delta(0), y \leq 0 \\ p_{Lin}(y), y > 0 \end{cases} \tag{4}$$

where $\delta(0)$ is the Dirac delta. The distribution has a delta spike at zero and a Gaussian tail at its positive end (Figure 2).

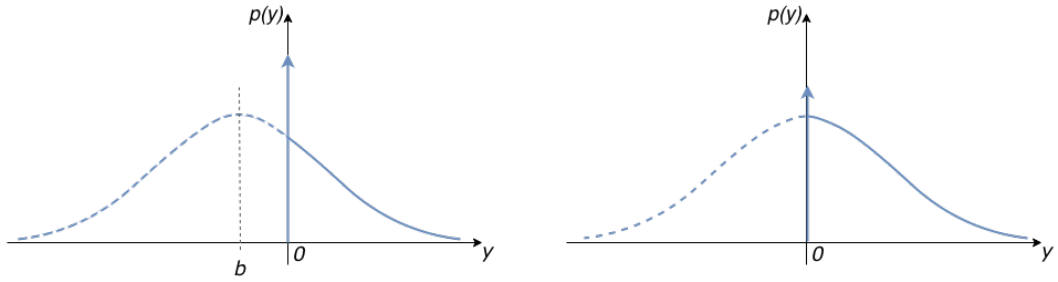
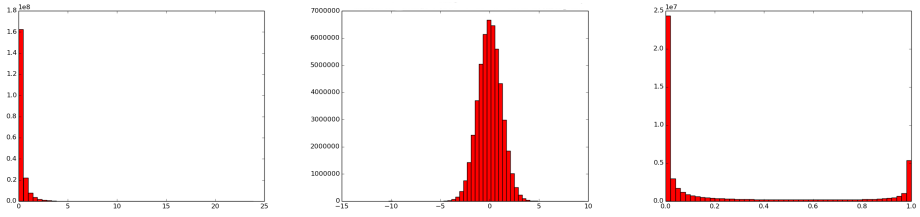


Figure 2: P.d.f. of ReLU output (left) and zero-bias ReLU output (right). The arrow at zero indicates a delta spike, and the dashed part stands for the probability mass absorbed into that delta spike.

Since the bias controls the intensity of the $\delta(0)$ spike, it controls the sparsity of output representation (left plot). The observation that biases tend to zero motivated Konda et al. (2014) to introduce a "zero-bias" activation function which uses a fixed threshold followed by linear activation. Typically while using zero-bias ReLU, a threshold of 1 is introduced during pre-training stage, and set back to zero while training its subsequent layers and fine-tuning. The distribution of pre-hidden activity of a zero-bias ReLU stretches equally on both sides of zero. As a result, for zero-bias ReLU activation half of the probability mass concentrates on a delta spike located at zero, as illustrated it Figure 2 (right). While batch normalization alone will push the mean and variance into the optimal range for a subsequent ReLU unit, it will not resolve the issue that the distribution is peaked at the negative bias. In fact, the estimate of mean and variance will suffer from the presence of a highly non-Gaussian distribution. Typical histograms over hidden unit activations for several activation functions are shown in Figure 3a.



(a) zero bias ReLU activation (b) linear layer activation (c) Sigmoid activation

Figure 3: Histogram of output activation with different types of activation function.

2.2 DERIVATIVES IN THE PRESENCE OF LINEAR LAYERS

We now consider the derivatives of a ReLU network. The activation of layer $i+1$ is given by:

$$\vec{H_{i+1}} = R(w_i \vec{H_i} + \vec{b_i}), \tag{5}$$

where $R()$ is the element-wise activation function. The back-propagated updates on w_i will be:

$$\Delta w_i = \delta \circ R^{'} \left(\vec{H_{i+1}} \right) \cdot \vec{H_i} \tag{6}$$

where δ stands for the down flowing error signal coming from the upper layer, $\circ$ stands for element-wise multiply, and $\cdot$ is the product of a vertical vector and a horizontal vector. As discussed in Section 2.1, at least 50% (usually much more in practice due to negative biases) of the values in the representation H_i and H_{i+1} are typically equal to zero (cf., Figure 1, left) making a large fraction of the entries of Δw_i per training case zero. If we introduce a linear layer between two ReLU layers,

$$\vec{H_l} = w_l \vec{H_i} + \vec{b_l} \quad \vec{H_{i+1}} = R\left(w_i \vec{H_l} + \vec{b_i} \right) \tag{7}$$

where $\vec{H_l}$ stands for the output of linear layer, and w_l, b_l are the weights and biases in the linear layer, we obtain updates of the form:

$$\Delta w_l = \delta_l \cdot \vec{H_i}\ , \quad \Delta w_i = \delta \circ R^{'} \left(\vec{H_{i+1}} \right) \cdot \vec{H_l} \tag{8}$$

where δ_l is the error signal passed to the linear layer, $\delta_l = \delta w_i$. Since the linear layer representation $\vec{H_l}$ is dense, both Δw_l and Δw_i become denser: only half of the values in these two update matrices are zeros. More importantly, there always exists a simple ReLU layer that is equivalent to a ReLU/linear combination because any linear layer can be absorbed into the weights of the ReLU layer:

$$w = w_i w_l, \quad b = w_i b_l + b_i \tag{9}$$

Then, the equivalent update on w becomes:

$$\Delta w = \Delta w_i \Delta w_l + w_i \Delta w_l + \Delta w_i w_l \tag{10}$$

Even if half of the values in Δw_i and Δw_l are zero, their product is a dense matrix. The second and third term in Equation 10 are also dense, so with a linear bottleneck layer, we actually obtain a dense update in an equivalent ReLU layer.

2.3 REDUCING PARAMETERS

Besides helping with back propagation, it is important to note that linear layers also reduce the total number of parameters. Suppose a linear layer with L units is inserted between two nonlinear layers with N units each. The total number of parameters would become $2NL + L + N$. This is much less than the number of parameters that would result from directly connecting the two nonlinear layer, which would amount to $N^2 + N$ parameters. Convolutional network also reduces the number of parameters by convolution kernels. Note that for any trained convolutional network, we can always find a fully connected network with the same accuracy by expanding the convolutional kernels.

3 PRE-TRAINING AND ZERO-BIAS ACTIVATIONS

It was suggested by Saxe et al. (2013) that the benefit of unsupervised pre-training of a network using RBMs or autoencoders may result in weight matrices that are closer to orthogonal and thus less affected by vanishing gradients problems. It is interesting to note, however, that the sparsity-inducing negative biases yield reconstructions that are affine not linear and accordingly may not orthogonalize weights after all (Konda et al., 2014). This may be one of the reasons why the practical success of these pre-training schemes has been quite limited by comparison to fully supervised learning using back propagation.

It is also important to note that due to sparsity, the number of active units in a layer is often smaller than that the same number in the layer below, so the "effective" weight matrix for a given input

example is not a square matrix. Rather than simply orthogonal weight matrices, we should be looking for networks where hidden units which tend to be active on the same inputs have weights that tend to be orthogonal to one another. In other words, we should be looking for "orthogonal active paths" through the network rather than overall orthogonal weight matrices.

In order to obtain hidden units with linear not affine encodings Konda et al. (2014) introduce "zero-bias autoencoders" (ZAE) whose activations are thresholded when pre-trained by minimizing the autoencoder reconstruction error, and whose thresholds are removed when the weights are used for classification or for initializing a feed forward network. In other words, the activations for hidden units in layer $i+1$ are given by (Konda et al., 2014):

$$\vec{H_{i+1}} = \left(w_i \vec{H}_i > \theta\right) w_i \vec{H}_i \tag{11}$$

where we set $\theta = 1$ during pre-training and $\theta = 0$ otherwise.

As discussed in Konda et al. (2014) minimizing squared error under the linear encoding should encourage weights corresponding to hidden units that tend to be "on" together to orthogonalize (because the orthogonal projection is what minimizes reconstruction error). Konda et al. (2014) reported decent classification performance in various supervised tasks, but found only a weak improvement over standard autoencoders when used to initialize a single-hidden-layer MLP.

The view of a zero-bias activation function as a way to orthogonalize weights suggests especially deep networks to profit from ZAE pre-training, and so we investigated the performance of ZAE-pretrained networks with many hidden layers, as well as in conjunction with interleaved bottleneck layers (as discussed in Section 2). We found them to yield a separate, and significant, performance improvement in fully connected networks.

4 Experiments

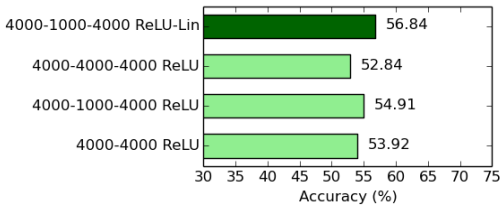
4.1 CIFAR-10

The CIFAR-10 dataset is a subset of the 80 Million Tiny Images Torralba et al. (2008), and contains 10 balanced classes. It provides a training set with 50000 samples and a test set of 10000 samples. Each sample is a colour image with 32×32 RGB pixels. We first compare mixed models with linear bottleneck layers and ordinary networks. We compare two different activation functions: ReLU and zero-bias ReLU (Konda et al., 2014). Comparison are based on classification accuracy. All the experiments in this subsection share the same pre-processing pipeline and the same type of classifier. For pre-processing, the raw data is contrast normalized and centred to have zero mean, followed by PCA whitening, retaining 99% of the variance.

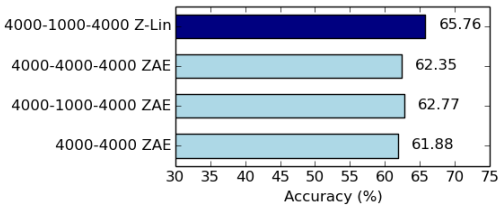
We trained 3 ReLU networks with different configurations, and compare their results with the ReLU-Lin network $4000\mathrm{ReLU} - 1000\mathrm{Linear} - 4000\mathrm{ReLU}$, that interleaves a linear layer between two ReLU layers. All networks in Figure 4a are trained by supervised back propagation using stochastic gradient descent with 0.9 momentum. The ReLU-Lin network outperforms all the pure ReLU networks and reaches an accuracy of 56.84%. We also found that the ReLU-Lin network tends to be more stable than the pure ReLU networks: if we pre-train these networks layer-wise, then for the ReLU networks the majority of the units in higher layers become "dead", whereas the ReLU-Lin network is still stable for unsupervised pre-training without dead units at the second ReLU layer.

We repeat the same experiment by substituting ReLU with ZAE (Figure 4b). The Z-Lin network is configured as $4000\mathrm{Z} - 1000\mathrm{Linear} - 4000\mathrm{Z}$ (same as in Figure 1, but with only one Z-Lin pair). Since ZAE does not suffer from the problem of dead units that much as ReLU, all the networks are first layer-wise pre-trained in an autoencoder, and then fine-tuned with stochastic gradient descent. During pre-training the linear layer, we have 1.0 weight decay added to the cost. Similar to the case of ReLU, it is also observed that introducing linear bottleneck layers makes the stacked deeper model outperform its shallow counterpart. Having a linear inserted in the middle makes the Z-Lin model outperform all the other models, yielding an accuracy of 65.76%.

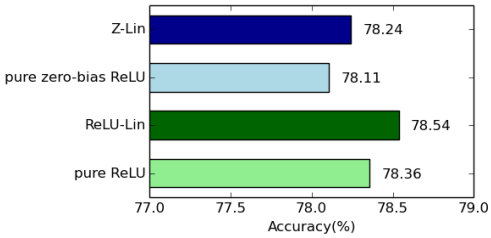
PReLU He et al. (2015) and Maxout Goodfellow et al. (2013) are two alternative approaches to fixing the sparsity incurred by the ReLU activation. We compare the ReLU-Lin and Z-Lin networks with PReLU and Maxout in Table 1. All models are trained by supervised learning, and share the same preprocessing steps as described in Section 4.1. For ReLU-Lin, Z-Lin and PReLU, we use



(a) ReLU-Lin network and various networks with pure ReLU activation.



(b) Z-Lin network and various networks with pure Zero-bias ReLU activation.



(c) Z-Lin, zero-bias ReLU, ReLU-Lin and ReLU network trained on HIGGS dataset.

Figure 4: Comparing bottlenecked network with its various counterparts. Models compared in (a) and (b) are trained on CIFAR-10, while those in (c) are trained on HIGGS dataset.

SGD with 0.9 momentum. For Maxout, we add 0.01 weight decay, and use 0.1 dropout (droping 10% of hidden activations randomly) for each layer. The table shows that the bottleneck layers outperform PReLU and Maxout on this task by a large margin.

Table 1: Comparing with PReLU and Maxout

Method	test set accuracy(%)
ReLU-LIN network 4000ReLU-1000Lin-4000ReLU-10	56.84
Z-LIN network 4000Z-1000Lin-4000Z-10	56.43
PReLU, 4000-4000-10	51.65
PReLU, 4000-1000-4000-10	51.94
Maxout, 4000-1000-4000-10	52.80

While doing pure supervised learning using bottleneck layers, the linear layer remaps the non-linear representation, which results in subtracting the mean values, and rescales the representation onto a proper standard deviation. In this sense, our way of remapping the representation is simillar to batch normalization Ioffe & Szegedy (2015). While batch normalization is applied before the non-linearity, and normalizes each dimension individually, linear bottleneck layers are applied after the non-linearity, and use a whole fully connected layer to modify the representation. To be fair We have to compare the two methods in supervised learning, as originally batch normalization is proposed for, though linear bottleneck layer can also be applied for unsupervised learning. In Table 2, two of the most frequently used activation functions, ReLU and sigmoid, are used for comparison between batch normalization and linear bottleneck layers. We use a $4000-1000-4000-10$ network trained fully supervised. The table shows that the linear bottleneck layers yield an improvement comparable to that of batch normalization. However, we should note that batch normalization also accelerates the convergence, which is not observed when using the bottleneck layers. It is important to note, however, that the linear bottleneck layers are amenable also to unsupervised learning which on this task yields a large performance improvement as we shall show.

Table 2: Comparing with Batch Normalization

models	ReLU	sigmoid
network without normalization	55.23	44.60
network with linear bottleneck layer	56.84	46.44
network with batch normalization	56.29	46.63

4.2 THE HIGGS DATASET

The HIGGS dataset has 11 million samples with 28 dimensions. The first 21 features are kinematic properties measured by particle detectors in a particle accelerator, and the remaining 7 features are functions of the first 21 features. Thus the dataset itself is permutation invariant. The task is to decide if or not a given sample corresponds to a Higgs Boson.

We tried both ReLU and ZAE with/without linear bottleneck layers on this dataset. Similar to before, for each model PCA whitening retaining 99% of the variance is used for pre-processing. (corresponding to 27 principle components). We use the same model size for four different models that we compare (zero-bias ReLU, Z-Lin, ReLU, and ReLU-Lin). The structure is $27 - 800 - 100 - 800 - 100 - 2$ for all, so the models differ by using different activation functions. We train all the models using SGD with momentum, and tune learning rates individually. We do not use any pre-training. The results are shown in Figure 4c, which also confirm the effectiveness of the bottleneck layers, albeit not as pronounced as on the CIFAR-10 data. Also, zero-bias units do not yield an improvement here.

The reason why the Z-LIN network does not strongly improve performance on HIGGS dataset might be related to its low dimensionality. Consider a Z-LIN pair which has only one input node, one linear layer output node, but with many zero-bias ReLUs in the hidden layer. The functions that such a Z-LIN pair can model are piece-wise linear functions that have only one inflection point at the origin. This extends to multi-dimensional cases. Since the model response can only change linearly by scaling the input by a positive linear factor, the nonlinearity of the model response could be observed only by changing the direction of the input feature vector. In that way, the model is concentrating all its non-linear learning capacity on the *direction* of feature vectors, and cares less about the *magnitude* of feature vectors. Since the direction becomes more and more important as the dimension increases, the advantage of the Z-LIN network would be more pronounced in high dimensional tasks.

4.3 REDUCING PARAMETERS

In this subsection we explore how the network's performance is affected by increasingly reducing the number of parameters, using bottlenecks of different sizes.

We define a network by stacking 3 Z-LIN pairs, plus a ZAE layer and a logistic regression classifier. Each ZAE layer has 4000 hidden units. We reduce the linear layer size from 1000 down to 100 units. Training involves dropout, pre-training and fine-tuning. The results are shown in Figure 5. We observe that even with a hidden layer of size 100, which has only $1/20$ of the parameters, the network still works reasonably well and does not loose too much accuracy.

A 7-layer fully connected network with 4000 units each layer has around 112 million parameters. With linear layers, the smallest model in Figure 5 has only around 2.44 million parameters. By comparison, a typical convolutional network yielding an accuracy higher than 80% on CIFAR10 would have around 3.5 million parameters.

4.4 ACHIEVING STATE-OF-THE-ART ON PERMUTATION INVARIANT CIFAR-10

We compare the performance of the Z-Lin model with other published methods on the CIFAR-10. For our method, pre-processing steps are the same to Section 4.1. For the Z-Lin network, each ZAE layer 4000 hidden units, and each linear autoencoder has 1000 hidden units. We use logistic regression on top of the last ZAE layer for classification. Following Konda et al. (2014), the threshold of

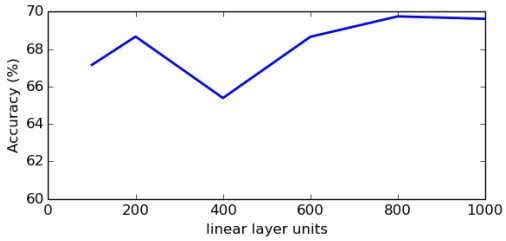


Figure 5: Classification accuracy w.r.t different linear layer size.

all ZAEs are fixed at 1.0 during pre-training, and set to 0 while training the subsequent layers and performing fine-tuning. As before, we subtract the mean value and normalize the activations to have a standard deviation of 1.0 in all layers.

We train the networks using stochastic gradient descent with 0.9 momentum and decreasing learning rate. The learning rate is set to 0.001 for the ZAEs and to 0.0001 for the linear autoencoder. Weight decay is used for the linear autoencoder and for the logistic regression layer. The latter is trained using nonlinear conjugate gradients. After pre-training, we use a tiny learning rate of 5×10^{-6} to fine-tune the whole network, which yields an overall accuracy of 65.7%. This already outperforms all previous published permutation invariant CIFAR-10 results, the next best-performing of which are 63.1% (Le et al., 2013), and 63.9% (Konda et al., 2014).

By adding dropout (Srivastava et al., 2014) during pre-training and fine-tuning, these performances can be further improved. A same model in the last paragraph but trained with dropout would yield 69.1% accuracy, and a very deep Z-LIN network (3 Z-LIN pairs, plus a ZAE layer and logistic regression classifier, i.e., $4000\text{Z} - 1000\text{Lin} - 4000\text{Z} - 1000\text{Lin} - 4000\text{Z} - 1000\text{Lin} - 4000\text{Z} - 10$) yields 69.62%, which exceeds the current state-of-the-art on permutation invariant CIFAR10 by a very large margin.

If we give up on permutation-invariance by using data augmentation (eg., Krizhevsky et al. (2012)) but retain the use of a fully connected network, the performance improves much further. Here, we add flipping, rotation, and shifting to the original data during training of a $4000\text{Z} - 1000\text{Lin} - 4000\text{Z} - 10$ Z-LIN network pushing the performance to 78.62%. This is a much higher accuracy on CIFAR-10 achieved by any fully connected network we are aware of, and it is not far behind the performance of a convolutional network.

5 Related Work

The idea of a linear bottleneck layer is very old and has been used as early as the 1980's in the context of autoencoders (eg., Baldi & Hornik (1989)). More recently, Ba & Caruana (2014) used a linear bottleneck layer to factorize a single-layer network and showed that it helped speed up learning. An application of a linear bottleneck layer in the last layer of a neural network for dealing with high-dimensional outputs is described in Sainath et al. (2013) and Xue et al. (2013). In contrast to our work, in none of these methods is the goal to alleviate vanishing gradients and deal with sparsity, and (accordingly) they use just a single bottleneck layer in the network. Recently, Srivastava et al. (2015) have introduced linear skip connections in order to train very deep networks. These can also be viewed as a way to prevent gradients from exploding or vanishing.

Reshape distributions over activations using linear layers is also related to the recently introduced batch-normalization trick (Ioffe & Szegedy, 2015), in that it is also a way to adjust the distribution of inputs to a subsequent layer. In contrast to that work, linear bottleneck layers not only adjust the mean and variance of the inputs to the subsequent layer, but reshape the whole *distribution*.

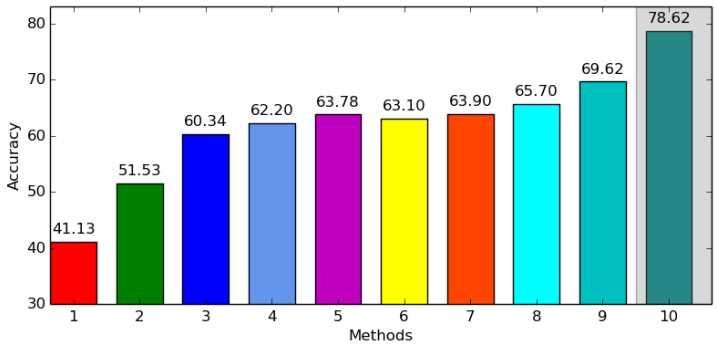


Figure 6: Test set accuracy of various methods. They are (from left to right): 1) Logistic Regression on whitened data; 2) Pure backprop on a 782-10000-10 network; 3) Pure backprop on a 782-10000-10000-10 network; 4) RBM with 2 hidden layers of 10000 hidden units each, plus a logistic regression; 5) RBM with 10000 hiddens plus logistic regression; 6) "Fastfood FFT" model (Le et al., 2013); 7) Zero-bias autoencoder of 4000 hidden units with logistic regression (Konda et al., 2014); 8) 782-4000-1000-4000-10 Z-Lin network trained without dropout; 9) 782-4000-1000-4000-1000-4000-1000-4000-10 Z-Lin network, trained with dropout 10) Z-Lin network the same as (8) but trained with dropout and data augmentation; Results (1)-(5) are from Krizhevsky & Hinton (2009). The final one is distinguished with a grey background because it uses data augmentation.

6 Discussion

It is well known that a single-hidden-layer neural network can model any non-linear function under mild conditions (Funahashi, 1989; Cybenko, 1989). The intuition behind this observation is that the hidden layer carves up space into half-spaces, or "tiles", and the subsequent linear layer composes the non-linear function by combining different linear regions to produce the output. It is interesting to note that this view may suggest using a *pair* of layers (a non-linear followed by a linear layer) to define the non-linear function, leading thus to interleaved linear/non-linear layers.

The practical usefulness of this result is limited, however, because to approximate any given function it would require an exponentially large number of hidden units. In practice, this is one motivation for using multilayer networks which compute a sequence of consequently more restricted, but tractable non-linear functions. Arguably, in the presence of enough training data and computational resources, wide hidden layers would still be preferable to narrow ones. However, in practice, wider hidden layers also entail more sparsity, which prevents the flow of derivatives. Also, as sparse activations propagate upwards through the network, they tend to proliferate, aggravating the problem in higher layers.

Acknowledgments

The authors would like to thank the developers of Theano (Bastien et al., 2012). We acknowledge the support of the following agencies for research funding and computing support: Samsung, NSERC, Calcul Québec, Compute Canada, the Canada Research Chairs and CIFAR.

References

Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? In *Advances in Neural Information Processing Systems*, pp. 2654–2662, 2014.

P. Baldi and K. Hornik. Neural networks and principal component analysis: Learning from examples without local minima. *Neural networks*, 2(1), 1989.

Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

Morris H DeGroot, Mark J Schervish, Xiangzhong Fang, Ligang Lu, and Dongfeng Li. *Probability and statistics*, volume 2. Addison-Wesley Reading, MA, 1986.

Peter Foldiak. Sparse coding in the primate cortex. *The Handbook of Brain Theory and Neural Networks*, 2003.

Ken-Ichi Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural networks*, 2(3):183–192, 1989.

Ian J Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. *arXiv preprint arXiv:1302.4389*, 2013.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *arXiv preprint arXiv:1502.01852*, 2015.

Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, and Jürgen Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Kishore Konda, Roland Memisevic, and David Krueger. Zero-bias autoencoders and the benefits of co-adapting features. *stat*, 1050:20, 2014.

Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. *Computer Science Department, University of Toronto, Tech. Rep*, 1(4):7, 2009.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

Quoc Le, Tamas Sarlos, and Alex Smola. Fastfood - approximating kernel expansions in loglinear time. In *30th International Conference on Machine Learning (ICML)*, 2013.

Tara N Sainath, Brian Kingsbury, Vikas Sindhwani, Ebru Arisoy, and Bhuvana Ramabhadran. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 6655–6659. IEEE, 2013.

Andrew M Saxe, James L McClelland, and Surya Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv preprint arXiv:1312.6120*, 2013.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.

Antonio Torralba, Robert Fergus, and William T Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(11):1958–1970, 2008.

Jian Xue, Jinyu Li, and Yifan Gong. Restructuring of deep neural network acoustic models with singular value decomposition. 2013.