Modern Approaches to Reducing the Complexity of Neural Networks

Samuel Jackson, University of Aberystwyth

I. Introduction

Neural networks (NN) are a collection of perceptrons (figure 1) organised into multiple layers [17]. The input to the network, usually represented as a vector, is passed (forward propagated) from one layer to another through the activation function of each unit. The activation of each unit is dependant upon the weighted input of the activation from the previous layer. The activation of the final layer l_n is interpreted as the output. The matrix equation governing the forward propagation between layers is given by:

$$\boldsymbol{a}^l = g(\boldsymbol{W}^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l) \tag{1}$$

Where $\boldsymbol{a}^l$ is the activation of layer l as a vector, $g(x)$ is the activation function, $\boldsymbol{W}^l$ is the weight matrix associated with layer l, $\boldsymbol{a}^{l-1}$ is the activation of previous layer, and $\boldsymbol{b}^l$ is the bias of layer l. For computational simplicity the bias is usually represented as an additional weight in $\boldsymbol{W}$ and the input vector has an additional element fixed to 1 for all examples. This is known as the bias trick. The activation function $g(x)$ is traditionally a sigmoid function (often the logistic or tanh functions) [16] although more recent research has shown that rectified linear units (ReLU) [7] can often be used to avoid the "vanishing gradient" problem associated with sigmoid perceptrons.

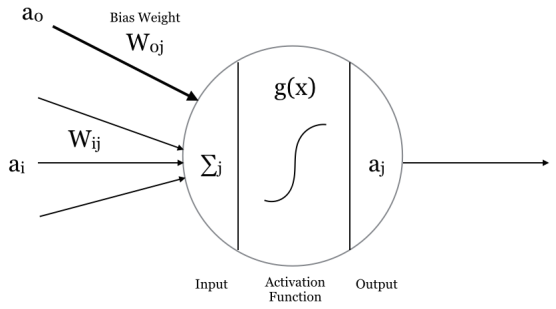

Fig. 1. Diagram of a basic perceptron units of a neural network. The weighted input of the training data is passed through an activation function to produce an output vector. Based on the diagram by Russell and Norvig [19].

Training neural networks is typically performed using the back-propagation algorithm [16]. Back-propagation first computes the error between the actual and expected output for a given training example in the final layer. The error in previous layers can be computed by "back-propagating" the error backwards from the final layer towards the input [2]. More formally, the error in the final layer L is given in matrix form by:

$$\boldsymbol{\delta}^L = \nabla_a C \odot g'(\boldsymbol{z}^L) \tag{2}$$

Where $\boldsymbol{\delta}^L$ is the error in the final layer, $\nabla_a C$ is the vector of partial derivatives of the cost C function (typically quadratic cost or cross-entropy), $\boldsymbol{z}^L$ is the weighted vector of inputs to the final layer ($\boldsymbol{W}^L \boldsymbol{a}^{L-1}$), and $\odot$ is the Hadamard product. The vector $\boldsymbol{\delta}^L$ is then used to compute the error in previous layers as follows:

$$\boldsymbol{\delta}^l = ((\boldsymbol{W}^{l+1})^T \delta^{l+1} \odot g'(\boldsymbol{z}^l) \tag{3}$$

Once the gradient vectors $\boldsymbol{\delta}^l$ have been computed for every layer in the network the weights can be updated using an optimisation algorithm. The simplest optimisation approach is gradient descent [16]. In gradient descent the weights for each layer are updated according to:

$$\boldsymbol{W}^l \leftarrow \boldsymbol{W}^l - \alpha \boldsymbol{\delta}^l \tag{4}$$

Where α is parameter controlling the step size and in the simplest implementation is α is a constant. Alternative approaches can make use of Newton or Quasi-Newton methods [16]. Gradient descent training can either preformed in "batches" where the update is the sum of the error over all training examples, or stochastically after each example.

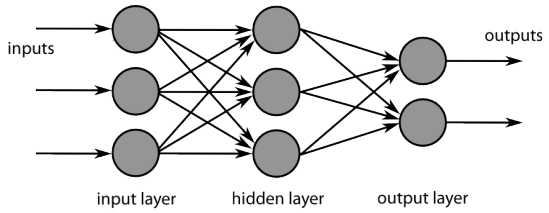

Fig. 2. Diagram of a very simple fully connected neural network architecture. The connections between the nodes are represented using the weight matrix $\boldsymbol{W}$. Each node is an activation function for a single element in the weighted input vector $\boldsymbol{z}^l$. Image source: [1]

The basic architecture of a classical neural network is structured so that every node in one layer is connected to every node in the next via the weight matrix. This architecture is known as a "fully connected" network. Fully connected networks are feasible on a small scale. However, the complexity of a network rapidly grows in proportion on the size of the training examples and the number of layers used. This particularly becomes an issue when attempting to process very high dimensional data such as images.

Subsequently, much of the current research in neural networks is focused on the reducing the complexity of the architecture and the number of weight parameters used. Denil et al. [6] demonstrated that there is a significant amount of redundancy in the parameterisation of neural networks. By reducing the number of parameters used or simplifying the architecture of the neural networks it is hoped that significant gains in time and memory complexity can be achieved. A reduction in network parameters also leads to a model that is potentially easier to train as there are less "knobs" to turn in order to fine tune the network.

There have been several notable prior methods to compress neural networks. In the simplest sense, the driving intuition behind convolutional neural networks [12] is that weights are effectively tied for a receptive field, forcing the network to only learn locally connected features. This combined with pooling layers [22] forms an architectural approach to parameter sharing. Early techniques such as "Optimal Brain Damage" by LeCun et al. [15] looked at the effect of removing parameters after training from a network based on the second order derivative. Nowlan & Hinton [18] experimented with soft weight sharing with Gaussian mixtures as part of weight regularisation. More recently Srivastava et al. [20] proposed Dropout for probabilistically dropping weights.

In the remainder of this paper we will examine three recent papers (<3 years old) each taking a different approach to the problem of reducing the complexity of neural networks. Specifically we shall critique the work of Chen et al. on *HashedNets* [3], Han et al. on "*Learning both Weights and Connections for Effcient Neural Networks*" [8], and Denil et al. on *Low-Rank Decomposition* [6].

II. HashedNets

A. Research Undertaken

HashedNets by Chen et al. [3] uses random weight sharing between layers in order to reduce the number of connections and utilises a hash function in order to avoid additional memory overheads and to automatically pool weights.

More formally, in their design, the connections between each layer is represented by a "virtual" weight matrix $\boldsymbol{V}^l$ in which the elements of each matrix are mapped to a vector of real weights $\boldsymbol{w}^l$ containing K elements where $K \ll N$ and where N is the total number of connections between layers. The connections are mapped from $\boldsymbol{V}^l$ to $\boldsymbol{w}^l$ using a hash function $\phi : \mathbf{R}^m \rightarrow \mathbf{R}^n$. Weights are pooled by taking advantage of the fact that hash collisions will happen when $K<N$. This has the effect of allowing a virtual number of parameters to be represented using only a K sized vector. Theoretically the size of $\boldsymbol{V}^l$ may be increased or decreased while the memory footprint of the network remains the same.

The mapping from the virtual matrix to the actual weight vector causes the feed forward equation (equation 1 in element wise form for just two layers) to be modified as follows:

$$a_i = g(\sum_{j=1}^{m} V_{ij} a_j) = g(\sum_{j=1}^{m} w_{\phi(i,j)} a_j) \tag{5}$$

Instead of using a hash function to transform $\boldsymbol{V}^l$ to $\boldsymbol{w}^l$ they hash the activation from the previous layer into a K dimensional vector. This avoids the need for a matrix explicitly mapping virtual weights to real weights, saving memory. Therefore the final feed-forward equation becomes:

$$a_i = g(\boldsymbol{w}^T \phi_i(\boldsymbol{a})) \tag{6}$$

Where the hash function ϕ is defined as the sum of activations from the previous layer hashed into the k^{th} bucket. The authors provide a proof that the two formulations are equivalent in [3].

Additionally, and perhaps most importantly, the authors provide a derivation of a modified version of the back-propagation equations (equations 2 and 3) which we have omitted here for brevity.

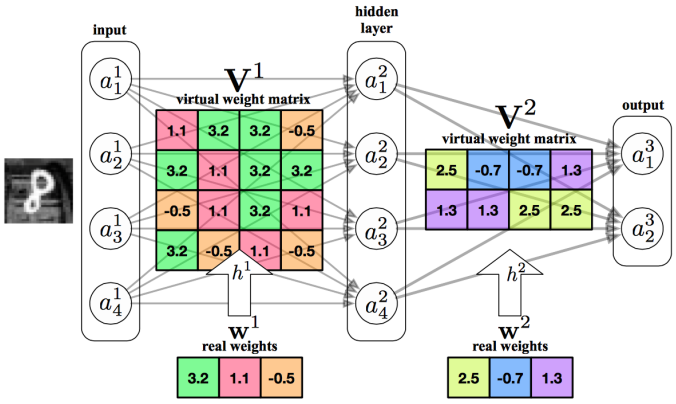

Fig. 3. Conceptual view of the relationship between virtual and weights in *HashedNets*. In practice it is the inputs which get hashed. Image source: [3]

B. Critique of Research

Chen et al. clearly show that they have formulated a neural network architecture that can reduce the actual number of parameters used by a neural network by exploiting the hashing trick. While the method has its limitations, some of which are listed by the authors themselves, none are significant enough to render the study irrelevant. The results of the study are quite promising, providing good results in contrast to baseline methods with a design that is both easy to understand and implement.

They begin by clearly stating the motivation for their research by relating it to one of the major challenges facing modern neural network: the complexity of the models needed to work with modern datasets. Their justification for their specific direction of research is based on the work of Denil et al. [6] who showed that there is a great deal of redundancy in neural network weight matrices. They argue that they can use the well known feature hashing trick to reduce this redundancy. Based on reading around this subject we would agree that this is both a fully justified motivation and sensible direction of research provided that compression gains are well balanced with the performance deterioration incurred by pooling weights.

The authors clearly state the method they used to implement feature hashing along with the relevant mathematical mechanics. Specifically, they provide two key derivations: firstly the show how the hashing function can be conceptually flipped from the hashing weights to hashing the activation output. Secondly they provide a derivation of the modified back propagation algorithm with the hash function included, necessarily proving that the network still can be trained conventionally. The derivations and their accompanying explanations are clear and accessible to the reader.

There are several limitations mentioned by the authors about their method which should be noted but are not significant enough for the method to not be useful in some dimension. The first is that feature hashing works best on sparse input where many features will be zero. The authors therefore encourage this by using ReLUs in activation. While these units are frequently used in modern NN approaches it would be interesting to see the effect of using a sigmoid function or very dense data and seeing how much this reduces the benefit of compression. Although this may be limited more by the ability to successfully train the network with different activations than by the compression technique.

Another limitation mentioned by the authors is that they have not managed to optimise their approach for GPU architectures where random memory access imposed by hash functions becomes an issue. It is noted that one of the authors is a NVIDIA employee which perhaps lowers the bias of the authors in relation to their own opinion as to how limiting this criticism is. Lack of an efficient GPU implementation is a major limiting factor for this methodology. However, if the parameter reduction achieved is significant enough a drop in GPU efficiency might be offset.

In the results of the paper the authors clearly demonstrate that their method out performs or is comparable to *Low-Rank Decomposition* [6] and *Random Edge Removal* [4] over all chosen datasets and compression factors. It would have been better if they had provided some additional results regarding the training or convergence time compared to a standard network. This would give a sense of if it is more difficult to train compared to a standard approach or even if the methodology actually accelerates learning.

Importantly the datasets used are all variations well known neural network community. Specifically they use MNIST [14] and a couple of variations [13] and two artificial datasets CONVEX and RECT [13]. However, this study could of perhaps gone further in this area and experimented with a more challenging real world natural image dataset such as CIFAR-10 [11] or ImageNet [5]. This would of given some results for a more realistic real world yard stick in comparison with the fairly dated MNIST. They could have also combined complex image datasets with different types of layers (i.e. convolution) with hashed layers for more practical performance results. Additionally, it would have been nice to see some results on larger architectures or commonly used models such as those that are available in Caffe [10].

Their choice to produce results combined with the "Dark Knowledge" approach of Hinton et al. [9] is quite interesting. The results of their experiments show that best result were achieved using the combination of feature hashing and a distilled model. The difference between the vanilla HashNet and a joined model is not hugely significant. However, this does show a good example of how the author's methodology lends itself to being combined with alternative approaches.

III. Pruning Connections and Neurons

A. Research Undertaken

In their recent paper Han et al. [8] propose a simple technique to enable a neural network to learn sparse connectivity between neurons. The high level view of their methodology is to initially train a large network, then prune weights which only contribute a small amount using a threshold parameter. Neurons which are left with no incoming or outgoing connections are removed from the network. The pruning and retraining phases are repeated many times in order to find the minimum number of connections required by the network. In the experiments described the pruning threshold was chosen to be a parameter multiplied by the standard deviation of the layer's weights.

To prevent over-fitting during the retraining stages the authors made use of the Dropout [20] technique. The authors note that they must make an adjustment to the dropout ratio due to the "hard pruning" effect in the connection pruning phase. The authors suggest a derivation for the ratio, suggesting the correct dropout rate is proportional to the square root of the quotient of connections to a layer before and after pruning.

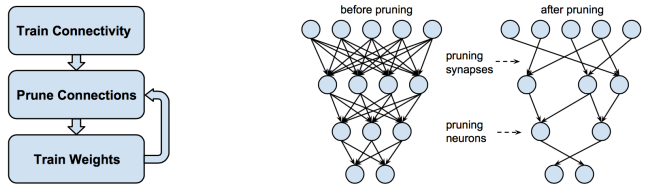

Fig. 4. Left: Overview of the iterative pruning approach. Right: How connectivity is leant, first weights are dropped, then neurons. Image source: [8]

The authors also experimented, based on prior literature and their own research, with a couple of implementation specifics to give optimum results in their own tests. L2 regularisation was shown to be the best choice overall despite L1 regularisation giving better accuracy after the pruning phase (as it forces more weights towards zero). The authors also make the case that training the network from the surviving parameters rather than reinitialising trained layers produces better results. This also has the advantage that only the pruned layers need to be retrained, which can be done using regular back propagation.

B. Critique of Research

Han et al. [8] show that they have developed an effective method of training a NN to produce a sparse architecture with solid and verifiable results to back their conclusions.

Their proposed method is based on the prior work of Srivastava et al. [20] but modified so that weights cannot

be brought back into existence once dropped. Some of the subsections outlining the methodology they used appears to be a little terse in places. In particular subsection 3.3 could use expansion and possibly a diagram to more clearly express their method and their justification for the retraining procedure.

The authors clearly show that they have a method of achieving excellent compression vs. test accuracy rates compared to recent competing methods in the field. They carried out all of their experiments using the well known NN framework Caffe [10] and used reference models making their methodology easier to validate in comparison with *HashedNets*. The authors show strong compression results across several different well known network implementations across two datasets. Beginning with MNIST [14] the authors show compression rates of up to 12x for LeNet-300-100 and LeNet-5 with only a small (< 0.06) drop in accuracy. As well as testing on MNIST, the authors show good compression results on the more complex ImageNet database [5].

A higher compression vs. test error rate seems likely compared with a effectively random technique such as *HashedNets* because they are training the network to lose only irrelevant parameters. In other words they can achieve a high compression rate and high test accuracy because they are making a choice about which weights to keep. This is better than choosing at random which is the case with *HashedNets*. This is reflected in the deterioration in accuracy seen in the 5 layer versions of HashNets where a larger network could lead to randomly pooling weights having a greater attenuation effect across layers.

A strong advantage to this methodology over approaches such as *HashedNets* is that the level of compression is automatically chosen by the algorithm. Only the hyper-parameter controlling the pruning threshold need be selected, either globally or layer by layer. This means that the technique is perhaps easier to tune that other approaches. This could also be interpreted as a weakness because the rate of compression is effectively heavily dependant on the sparsity of the data which will change between datasets.

A major limitation of this approach in comparison with that of *HashedNets* in section II is a full, "uncompressed" neural network must first be trained, then pruned and retrained instead of starting at a smaller size initially. If the network needs to be constrained with restricted memory then this approach would not work. This is a big issue for extremely large networks where a distributed implementation is required in order to train them. If the purpose of reducing the number of parameters so that the network may be trained on a single machine then this method may not be appropriate. Using this approach also requires the network to be retrained after the pruning stage, while in contrast *HashedNets* are simply trained in a compressed state.

One element of the paper which was particularly insightful was the visualisation of a layer's sparsity pattern. NNs are notorious black boxes and this paper contributed an interesting visualisation of a layer's sparsity and linked it with the characteristics of the MNIST dataset. Such techniques may be helpful in understanding of a dataset and tweaking the design of a NN accordingly.

IV. Low-Rank Decomposition

A. Research Undertaken

Denil et al. [6] illustrate an approach to compression known in literature as *Low-Rank Decomposition*. The basic principle behind their methodology is to take the weight matrix $\boldsymbol{W}$ and decompose the matrix into two smaller matrices:

$$\boldsymbol{W} = \boldsymbol{U}\boldsymbol{V} \tag{7}$$

Where $\boldsymbol{U}$ has the same number of rows as the input vector $\boldsymbol{a}^{l-1}$ and $\boldsymbol{V}$ has the same number of columns as the size of the output vector $\boldsymbol{a}^{l}$. The remaining dimension n_α (the columns of $\boldsymbol{U}$ and the rows of $\boldsymbol{V}$) determines the level of compression. The smaller n_α becomes the less effective parameters a layer has.

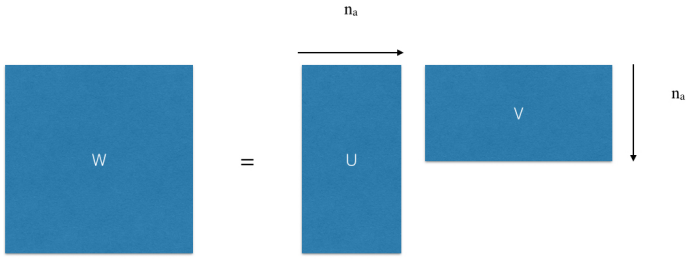

Fig. 5. Visualisation of the decomposition of the weight matrix $\boldsymbol{W}$ into two smaller matrices $\boldsymbol{U}$ and $\boldsymbol{V}$

In their method they propose to fix the value of $\boldsymbol{U}$ and let only $\boldsymbol{V}$ be learned. This intuitively makes sense because the two matrices will still have redundancies as they are both linear combinations of each other. Subsequently the matrix $\boldsymbol{U}$ can be chosen to be a matrix where the columns are bases for $\boldsymbol{V}$. The rest of their methodology concerns finding good choices for $\boldsymbol{U}$.

The authors present a number of different methods for choosing the columns of $\boldsymbol{U}$. One choice is to randomly choose columns for the identity function which equates to just picking random connections. Another is to choose the columns to be random unit vectors orthogonal to each other, as in random projections. They also present more advanced choices that encode prior knowledge about the feature space through the use of a kernel. Complex features are shown to be predicted using the kernel ridge predictor [21]:

$$\boldsymbol{w} = \boldsymbol{k}_{\boldsymbol{\alpha}}^{T}(\boldsymbol{K}_{\boldsymbol{\alpha}} + \lambda I)\boldsymbol{w}_{\boldsymbol{\alpha}} \tag{8}$$

The preceding equation is predicting a single column $\boldsymbol{w}$ of the final weight matrix $\boldsymbol{W}$ given a subset of the original weight vector $\boldsymbol{w}_{\boldsymbol{\alpha}}$. Note that this can be parallelised by giving multiple example inputs as a matrix $\boldsymbol{W}_{\boldsymbol{\alpha}}$. Equation 8 can be also be intuitively broken apart and equated to $\boldsymbol{U}$ and $\boldsymbol{V}$:

$$\boldsymbol{U} = \boldsymbol{k}_{\boldsymbol{\alpha}}^{T}(\boldsymbol{K}_{\boldsymbol{\alpha}} + \lambda I) \tag{9}$$

$$\boldsymbol{V} = \boldsymbol{w}_{\boldsymbol{\alpha}} \tag{10}$$

The choice of kernel function used for $\boldsymbol{k_\alpha}$ and $\boldsymbol{K_\alpha}$ are experimented with as part of the author's research. For example, in the case of an image this could be the Gaussian radial basis kernel. They also show that construction of the kernel also be inferred by the data by training a layer as an autoencoder.

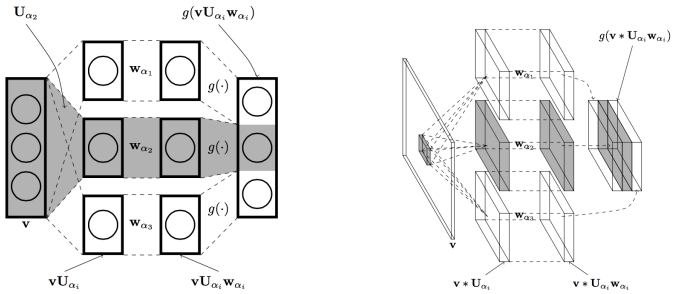

Fig. 6. Columnar architecture of a fully connected (left) and convolutional (right) network. Each column corresponds to a different set of weight indices (α). Image source: [6]

The authors also propose an architectural structure (figure 6) in which multiple basis dictionaries can be be used in the case that one such dictionary is too restrictive. They propose that a sub matrix $\boldsymbol{U}_{\alpha j}$ can be created and used to predict the a corresponding feature matrix $\boldsymbol{W}_j = \boldsymbol{U}_{\alpha j}\boldsymbol{W}_{\alpha j}$ where $j \in J$ is the index of the number of sub matrices. Each sub matrix can then be concatenated to produce the final matrix $\boldsymbol{W}$.

B. Critique of Research

Denil et al. [6] show a flexible solution for reducing the parameters of a neural network using matrix decomposition in an influential paper that also showed that NN weights have a great deal of redundancy.

One of the biggest strengths of this technique, outlined by the authors, is that the formulation is independent of many of the peripheral choices when designing a neural network architecture. The performance of their approach is not dependant on the activation function, choice of regularisation, architecture of layers etc. This is very useful both because it means that the technique is likely to be applicable to a wide variety of practical uses. This also makes it readily applicable to being combined with other compression approaches.

A major advantage in contrast with techniques like the one presented in section III is that, like *HashedNets*, a the network is trained with the compressed representation. A full neural network is never grown at any point. This means it is suitable for making a very large network small enough to be trained on a single physical machine and could avoid the need for a distributed architecture.

The choice of the basis dictionary $\boldsymbol{U}$ in their formulation is highly flexible. The authors propose a number of different approaches for choosing $\boldsymbol{U}$. This means that given a specific neural network $\boldsymbol{U}$ can be tailored to its requirements using methods such as kernel ridge regression and pre-training layers as autoencoders. However, this is also a weakness of the method because it adds an additional number of choices we must make about our system. The question: "what basis dictionary is best for my input?" is not trivial to answer and highly dependant on the dataset. Additionally, the choice of indices (α) used suffers from the same problems. The authors propose just choosing randomly, but if we wish to do better than this we must carefully formulate a selection criteria, which is again likely to be data specific.

While the proposed method is perhaps not the easiest choice to work with in practice, their paper did show a high level of redundancy in the parameters of both fully connected and convolutional neural networks. Based on their results they provide a strong and enjoyable write up of future directions of research.

The results of the paper are detailed and interesting. The authors provide several different implementations of their design using different basis dictionaries and showed how almost all of them can be trained to give good performance compared to a conventional network. The fact that the authors used two common datasets (MNIST and CIFAR-10) makes their results easily verifiable. It was also encouraging to see results of their approach combined with an alternative architecture (i.e. convolutional), showing that the approach will work with other architectures.

In terms of the papers discussed here, the authors of *HashedNets* directly compared their approach with the low rank approach presented in this section. It is worth noting that low-rank decomposition was shown to be inferior to the plain *HashedNets* implementation. The authors of *HashedNets* used a zero-mean Gaussian distribution as their fixed matrix. However, it may be the case that a better choice for the basis dictionary would of yielded better results over a relatively naive Gaussian distribution implementation.

REFERENCES

[1] Multi-Layer Neural Network. https://commons.wikimedia.org/wiki/File:MultiLayerNeuralNetworkBigger_english.png. Accessed: 2015-10-10.

[2] Neural Networks and Deep Learning. http://neuralnetworksanddeeplearning.com/chap2.html. Accessed: 2015-10-10.

[3] Wenlin Chen, James T Wilson, Stephen Tyree, Kilian Q Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. *arXiv preprint arXiv:1504.04788*, 2015.

[4] Dan C Cireşan, Ueli Meier, Jonathan Masci, Luca M Gambardella, and Jürgen Schmidhuber. High-performance neural networks for visual object classification. *arXiv preprint arXiv:1102.0183*, 2011.

[5] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[6] Misha Denil, Babak Shakibi, Laurent Dinh, Nando de Freitas, et al. Predicting parameters in deep learning. In *Advances in Neural Information Processing Systems*, pages 2148–2156, 2013.

[7] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 315–323, 2011.

[8] Song Han, Jeff Pool, John Tran, and William J Dally. Learning both weights and connections for efficient neural networks. *arXiv preprint arXiv:1506.02626*, 2015.

[9] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[10] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*, pages 675–678. ACM, 2014.

[11] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images, 2009.

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

[13] Hugo Larochelle, Dumitru Erhan, Aaron Courville, James Bergstra, and Yoshua Bengio. An empirical evaluation of deep architectures on problems with many factors of variation. In *Proceedings of the 24th international conference on Machine learning*, pages 473–480. ACM, 2007.

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

[15] Yann LeCun, John S Denker, Sara A Solla, Richard E Howard, and Lawrence D Jackel. Optimal brain damage. In *NIPs*, volume 89, 1989.

[16] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[17] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[18] Steven J Nowlan and Geoffrey E Hinton. Simplifying neural networks by soft weight-sharing. *Neural Computation*, 4(4):473–493, 1992.

[19] Stuart Russell and Peter Norvig. Artificial intelligence: a modern approach. 1995.

[20] 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.

[21] Max Welling. Kernel ridge regression. *Max Wellings Classnotes in Machine Learning (http://www. ics. uci. edu/welling/classnotes/classnotes. html)*, pages 1–3, 2013.

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