Efficient Training of Deep Neural Networks

Richa Ranjan
ID: 29757282
MSc Data Science, University of Southampton, UK
Email: rr2n17@soton.ac.uk

***Abstract*—In the last decade, Deep Learning has achieved major breakthroughs, however, "What makes Deep Neural Networks hard to train?" still remains a complex question. Some of the most recent works in Deep Learning and unsupervised feature learning have provided practical recommendations for training deeper networks to improve the network's performance. This paper is a general review based on several literatures related to different techniques for efficiently training Deep Neural Networks.**

I. Introduction

Deep Neural Networks (DNNs) are extremely powerful machine learning models that have showcased promising outcomes in a number of practical applications like Text processing, Computer Vision, Speech Recognition, and other similar areas. Ever since the inception of Neural Networks in the 1950s, there have been numerous attempts to simulate the working of a human brain. It was not until the major breakthrough in 2006 (Hinton *et al.*), that *Deep Learning* gradually moved to the layer-wise feature learning approach.

Machine Learning algorithms are built upon a large number of training examples. Better performances have been observed with more number of labeled training instances. In order to handle such enormous amounts of data, computational power is the need of the hour. Computation is a big step in training deep networks. DNNs have shown massive amounts of computational capabilities over wide range of applications. In light of the state-of-the-art performances achieved across various domains, there can be two categories of approaches to efficiently train these networks, namely *Hardware Acceleration* and *Algorithmic Advances*.

A. Hardware Acceleration

The accuracy of results tends to increase with an increase in the number of training examples and network parameters. In order to deal with massive computations, the usual approach is to scale the hardware. The advent of Multi-core machines with better computational capabilities and GPUs have significantly contributed towards successful training approaches. GPUs have brought a revolution in Deep Learning researches, due to their large-scale Matrix multiplication capabilities and other parallelization techniques, thereby making it a more preferred choice in research laboratories. In addition to its massive computational capabilities, GPUs have a limitation of slow learning rate in cases where the model does not fit in the GPU memory. In this light, congestions can be avoided by reducing the data size.[1] However, this approach is not a common practice. Some of the algorithmic changes can be applied to the model, which is the crux of this paper.

B. Algorithmic Advances

In addition to scaling up machines, researchers have looked at algorithmic enhancements for training DNNs. The standard learning process comprises of randomly initializing weights and further using gradient descent using back propagation to learn. Ruder [2] throws light on some of the gradient descent optimization algorithms such as *Gradient Descent with Momentum*, *AdaGrad*, *AdaDelta*, *RMSProp* and some others. Some of these algorithms are discussed further in section 2A.

In light of the aforementioned techniques, another algorithm called *Adam* (Adaptive Moment Estimation) was introduced. It is "an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments."[6] Adam combines the advantages of *AdaGrad* (Duchi et al., 2011) and the *RMSProp* (Hinton et al., 2012) algorithms, thereby working well with sparse gradients and moving averages. Further explanation of this approach can be found in section 2A.

The standard learning process for DNNs involves gradient-based updates of weights and these gradients are derived using the chain rule. With the networks going deeper, due to the chain of multiplications, the magnitude of the derivatives starts diminishing during backpropagation, giving rise to issues like *vanishing gradient problem*. One of the ways to address this problem can be *cascading* [3]. In this approach, the training is done in a bottom-up, incremental fashion, in a network of *convolutional* layers, thus ensuring that the layers being trained are close to the output layer throughout the training process.

Platt's work [4] on *Resource Allocating Network* (RAN) describes an approach where the network allocates a new computational unit whenever a poor performance is experienced on a captured pattern. These representations are then learned and the current responses are corrected. Further, if the network performance is good, then the LMS gradient descent was used to update the parameters. This network was later enhanced by including the *Extended Kalman Filter* (EKF), thereby improving the performance.[5] These algorithms are further elaborated in section 2C.

II. Overview of Efficient Learning Techniques

A. Gradient Decent Optimization

Gradient descent is one of the most widely used algorithms for optimized training of DNNs. Implementations of various algorithms for optimizing gradient descent are available as

part of Deep Learning libraries such as `Theano` [1], `Caffe` [2], `keras` [3] and `lasagne` [4]. Gradient descent is defined as an approach to minimize the objective function $J(\theta)$ by updating the parameters in backward direction (opposite to the forward propagation direction) of the gradient of $J(\theta)$ w.r.t the parameters $\theta \in \mathbb{R}^d$. The update sizes are governed by the learning rate η. Depending upon the data size, *Batch*, *Stochastic* and *Mini-batch gradient decent* variants can be used.

In light of the aforementioned process, various algorithms have been utilized by the Deep Learning community, and have shown better performances. Stochastic Gradient Decent (SGD) with **Momentum** computes an exponentially weighted average of the gradients and uses these gradients to update the weights. The Momentum accelerates the SGD, which causes less oscillations, thereby converging faster to the minima.[2] **AdaGrad** is another optimization algorithm that adjusts learning rate according to the parameters. With its ability to deal better with sparse data, AdaGrad have been instrumental in training large NNs owing to the increased robustness of SGD. The **RMSProp** or the *Root Mean Square* propagation algorithm is another adaptive learning technique, proposed by Geoff Hinton. With Adagrad having different learning rates for each parameter, there was a higher probability of learning rate dimishing. To address this, RMSProp and *Adadelta* (an extension of AdaGrad) were developed. The idea was to divide the learning rate η by "exponentially decaying average of squared gradients". [2]

Another stochastic optimization algorithm that proved to be efficient for first-order gradient-based optimization is **Adam**. Adam is an adaptive moment estimate algorithm, suitable for optimizing the cost functions with high-dimensional parameter spaces. In cases where the objective functions have dropout regularization noises, this algorithm combines the advantages of RMSProp (Hinton et al., 2012) and AdaGrad (Duchi et al., 2011) approaches. The Adam algorithm pseudocode (taken from [6]) can be found in Algorithm 1 section. With default choice of α = 0.001, β_1 = 0.9, β_2 = 0.999 and $\epsilon = 10^{-8}$, Adam has projected a more efficient order of computation for scaling high-dimensional problems.[6]

Algorithm 1 Adam Algorithm, taken from *Jimmy Lei Ba* et al.[6]

Require: α : Stepsize
Require: β_1,β_2: Exponential decay rates for moment estimates
Require: $f(\theta)$: Stochastic cost function with parameter θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Gradients w.r.t objective at *t*)
    $m_t \leftarrow \beta_1.m_{t-1} + (1 - \beta_1).g_t$ (first moment update)
    $v_t \leftarrow \beta_2.v_{t-1} + (1 - \beta_2).g_t^2$ (second moment update)
    $\hat{m}_t \leftarrow m_t/(1 - \beta_1^t)$ (First Bias-corrected moment)
    $\hat{v}_t \leftarrow v_t/(1 - \beta_2^t)$ (Second Bias-corrected moment)
    $\theta_t \leftarrow \theta_{t-1} - \alpha.\hat{m}_t/(\sqrt{\hat{v}_t} + \epsilon)$ (Update parameters)
  end while
    return θ_t (Resulting parameters)

B. Hyperparameter tuning

According to some of the practical recommendations by Bengio [7] on commonly used hyperparameters and ways to tune them, layer-wise unsupervised training has shown more interesting results. The much talked about breakthrough in Deep Learning (Hinton et al., 2006; Bengio et al., 2007; Ranzato et al., 2007) revolved around the idea of greedy unsupervised learning at each hierarchical level of the features. [7] Additionally, a supervised fine-tuning stage is included in some of the Deep Learning algorithms like Boltzmann machine and auto-encoders. The greedy layer-wise pre-training process works by learning a new representation, which can further be used to make predictions. *Automatic Differentiation* is another approach recommended by Bengio [7] where the gradients are calculated recursively in a *flow graph*.

Hyperparameters play a vital role in training DNNs. These are variables whose values are to be set prior to applying any learning algorithms. The initial learning rate (typically defaulted to 0.01), learning rate schedule (chosen by adaptive learning rate heuristic), mini-batch size (typically, larger size yields faster computations), number of training iterations (optimized by *early stopping*) and Momentum are some of the NN-specific hyperparameters. With respect to models, some of the hyperparameters are number of hidden units (larger value is advisable because of early stopping and other regularizers), Preprocessing and Weight Decay regularization coefficient λ. In order to avoid overfitting, a *regularization term* is added to the objective function. Typically, the L2 regularization ($\lambda \sum_i {\theta_i}^2$) is used as the L1 regularization ($\lambda \sum_i |\theta_i|$) might produce sparse results.

C. Resource Allocating Networks

Platt's work [4] brought about the hypothesis of Resource Allocating Networks (RANs) that learns by allocating new computational units based on the captured pattern, and adjusting the existing parameters. If the network performance is poor for some captured pattern, a new unit is allocated and the current response is corrected. If the network performs well, the parameters are adjusted using the LMS gradient descent. The RAN learning algorithm uses a two-part "novelty" condition. If the new input is $\vec{I}$ and the desired output is $\vec{T}$, then the input-output pair $(\vec{I}, \vec{T})$ is considered novel if the distance between this input and the existing centers is big:

$$||\vec{I} - \vec{c_{nearest}}|| > \delta(t) \quad (1)$$

[1] http://deeplearning.net/software/theano/
[2] http://caffe.berkeleyvision.org/tutorial/solver.html
[3] https://keras.io/optimizers/
[4] http://lasagne.readthedocs.io/en/latest/modules/updates.html

and if the actual and the desired output difference is large:

$$||\vec{T} - \vec{y}(\vec{I})|| > \epsilon \tag{2}$$

where ϵ is the desired output accuracy and $\delta(t)$ is the resolution scale of the network. If a new unit is not allocated, the Wildrow-Hoff LMS algorithm, (Wildrow & Hoff, 1960) is used to reduce the error. [4] RANs have proved to be advantageous in a number of ways such as, fewer weights requirement, adjusting the centers of Gaussian units based on output error etc.. This approach has shown good results in one of the well-known applications, the *Mackey Glass chaotic time series prediction* [5] problem.

A further enhancement to the RANs includes the *Extended Kalman filter* (EKF) instead of the LMS algorithm. [5] As mentioned in Platt's work [4], the vanilla RAN approach of sequentially estimating the objective function is essentially a Gaussian Radial Basis Function (GaRBF). "An extension to this F-projection principle would be the *Recursive nonlinear least-squares* (RNLS) algorithm" [5] in which the previous input distribution is also recursively estimated. The use of EKF in minimizing the RNLS cost function estimation has shown improved rate of convergence of the RAN and less complex network. For a parameter vector w, the EKF algorithm obtains the posterior estimate w^n from its prior estimate $w^{(n-1)}$:

$$w^n = w^{(n-1)} + e_n k_n \tag{3}$$

where k_n is the Kalman gain vector. Further, the Recursive Least Squares (RLS) algorithm was used to sequentially grow Multi-layer Perceptrons (MLPs), and the learning complexity was significantly reduced (Azimi-Sadjadi, Sheedvash, 1991).

D. Deep Cascade Learning

With the development of **LeNet-5**, the idea of Deep Networks was generalized to *Convolutional Neural Networks* (CNNs), but the major breakthrough of Deep CNNs was **AlexNet**, the ImageNet competition winner in 2012. [8] This ImageNet classifier used techniques like *Data Augmentation* and *Dropout regularization*, thereby showcasing results upto 78.1% and 60.9% on *LabelMe* dataset. Inspired by the Cascaded Correlation algorithm idea proposed by Fahlman et. al of sequentially training perceptrons and connecting their outputs to perform only one classification, cascaded layer-wise learning approach was designed.[3] In this bottom-up incremental learning, the network is split into layers and each layer is trained one at a time, connecting it to the output block. Once the current layer weights converge, the subsequent layer is learned, connected to the output block, and then trained with the pseudo-inputs created by forward propagation of initial inputs through the fixed layer. In this way, the vanishing gradient problem is handled, because the layers being trained are always close to the output layer. The pseudocode of Cascade learning approach (taken from [3]) can be observed below:

[5]The Mackey-Glass model is a popular chaotic time series, obtained by integrating nonlinear differential equation. The Mackey-Glass delayed differential equation is solved using the 4th order Runge Kutta method.

Algorithm 2 Cascade Learning, taken from *Enrique* et al. [3]

procedure CASCADE LEARNING(LAYERS,η,EPOCHS,
EPOCHSUPDATE,OUT)
Inputs: *layers* : model layers parameters
η : Learning rate
epochs : starting number of epochs
k : epochs update constant
out : output block specifications
Output W : L layers with w_l trained weights per layer
    for layer index = 1 : L **do**
Init new layer and connect output block
        $i_l \leftarrow epochs + k * layer\_index$
        for $i = 0; i++; i < i_l$ **do**
            $w^{new} \leftarrow w^{old} - \eta \nabla J(w)$
            if Validation error plateaus **then**
                $\eta \leftarrow \eta/10$
            end if
        end for
Disconnect output block and get new inputs
    end for
end procedure

III. Conclusion

Obtaining better results in Deep Learning is an empirical process. Choice of learning rate, network size, mini-batch size, regularization and early stopping are some of the hyperparameter-selection techniques. Also, RAN with EKF and cascading are some of the algorithmic techniques which have performed significantly well on Deep Networks. With the growing need for making machines more 'human-like', enhancements are imperative, to make the network learn quickly and accurately, with a reasonable amount of computation. Evidently, there is no perfect result in machine learning, and every outcome can further be improved using these techniques, which is going to be the basis for the upcoming research.

References

[1] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V.Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang and Andrew Y. Ng, *"Large Scale Distributed Deep Networks"*. NIPS'12 Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1, URL- https://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf

[2] Sebastian Ruder (2016). *"An overview of gradient descent optimization algorithms"*. URL- https://arxiv.org/abs/1609.04747

[3] E. S. Marquez, J. S. Hare and M. Niranjan, *"Deep Cascade Learning"*, in IEEE Transactions on Neural Networks and Learning Systems, 2018.

[4] J. Platt, *"A resource-allocating network for function interpolation"* Neural computation, vol. 3, no. 2, pp. 213225, 1991.

[5] V. Kadirkamanathan and M. Niranjan, *"A function estimation approach to sequential learning with neural networks"*, Neural Computation, vol. 5, pp. 954975, 1993.

[6] Kingma, Diederik P. and Jimmy Ba. *"Adam: A Method for Stochastic Optimization"*. CoRR abs/1412.6980 (2014)

[7] Yoshua Bengio. *"Practical recommendations for gradient-based training of deep architectures"*. Version 2, Sept 2012. CoRR abs/1206.5533.

[8] Alex Krizhevsky and Sutskever, Ilya and Hinton, Geoffrey E. *"ImageNet Classification with Deep Convolutional Neural Networks"*, 2012. URL - http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf