Engineering Applications of Artificial Intelligence 67 (2018) 14–23

Contents lists available at ScienceDirect

Engineering Applications of Artificial Intelligence

journal homepage: www.elsevier.com/locate/engappai

Performance improvement of deep neural network classifiers by a simple training strategy

Abdullah Caliskan[a], Mehmet Emin Yuksel[a,*], Hasan Badem[b,c], Alper Basturk[b]

[a] Department of Biomedical Engineering, Erciyes University, Kayseri, Turkey
[b] Department of Computer Engineering, Erciyes University, Kayseri, Turkey
[c] Department of Computer Engineering, Kahramanmaras Sutcu Imam University, Kahramanmaras, Turkey

ARTICLE INFO

Keywords:
Autoencoder
Deep neural network
Deep learning
Limited memory BFGS
Softmax classifier
Stacked autoencoder

ABSTRACT

Improving the classification performance of Deep Neural Networks (DNN) is of primary interest in many different areas of science and technology involving the use of DNN classifiers. In this study, we present a simple training strategy to improve the classification performance of a DNN. In order to attain our goal, we propose to divide the internal parameter space of the DNN into partitions and optimize these partitions individually. We apply our proposed strategy with the popular L-BFGS optimization algorithm even though it can be applied with any optimization algorithm. We evaluate the performance improvement obtained by using our proposed method by testing it on a number of well-known classification benchmark data sets and by performing statistical analysis procedures on classification results. The DNN classifier trained with the proposed strategy is also compared with the state-of-the-art classifiers to demonstrate its effectiveness. Our classification experiments show that the proposed method significantly enhances the training process of the DNN classifier and yields considerable improvements in the accuracy of the classification results.

1. Introduction

In the last few years, DNNs have extensively been utilized in classification problems because of their excellent classification performance in challenging classification tasks (Xu et al., 2016b; LeCun et al., 2015; Xu et al., 2016a; Krizhevsky et al., 2012; Luo et al., 2016; Yin and Zhao, 2016; Grozdic et al., 2017; Badem et al., 2016a, b; Caliskan et al., 2017). A typical DNN is constructed by combining a stacked autoencoder (SAE), which comprises a desired number of cascaded autoencoder (AE) layers, with a softmax classifier (Ng, 2011). Hence, a DNN offers the joint advantages of efficient generation of new features from raw data, which is a key property of SAE, and accurate classification of feature vectors, which is a distinctive property of softmax classifiers. These two properties, which are highly desirable in classification problems, complement each other and allow DNN classifiers to exhibit superior classification capability over conventional classification methods available in the literature.

One of the most important factors affecting the classification performance of DNNs is their training. Training of a DNN corresponds to a large scale optimization problem, which requires the minimization of a complicated objective function in a multidimensional search space. This is a difficult problem in any classification task because the dimension of the optimization search space is usually very high due to the high number of optimization parameters, which are network weights connecting the nodes in SAE and softmax layers of the DNN under training. Consequently, the performance of a DNN classifier is strictly dependent on the choice of the optimization algorithm employed for training. It is therefore of key importance in training of a deep neural network classifier to choose a suitable optimization algorithm, which can handle high numbers of optimization parameters and find acceptable optimal solutions in high dimensional search spaces without getting trapped in non-optimal solutions.

There are many different optimization methods available in the literature, but majority of these methods are not suitable for use with the training of DNNs due to the highly demanding nature of the problem, as discussed above. However, there are a number of optimization algorithms proposed for large scale optimization problems and some of these algorithms have been utilized to deal with complex machine learning problems, including the training of DNNs. These algorithms include the gradient descent (GD) (Hinton and Salakhutdinov, 2006; Lecun et al., 1998; Nesterov, 2012), the stochastic gradient descent

[*] Corresponding author.
E-mail addresses: acaliskan@erciyes.edu.tr (A. Caliskan), yuksel@erciyes.edu.tr (M.E. Yuksel), hbadem@erciyes.edu.tr (H. Badem), ab@erciyes.edu.tr (A. Basturk).

https://doi.org/10.1016/j.engappai.2017.09.002
Received 16 February 2017; Received in revised form 12 July 2017; Accepted 3 September 2017
0952-1976/© 2017 Elsevier Ltd. All rights reserved.

(SGD) (Sohl-Dickstein et al., 2014; Ngiam et al., 2011), the conjugate gradient (CG) (Ngiam et al., 2011) and the limited memory BFGS (L-BFGS) (Sohl-Dickstein et al., 2014; Ngiam et al., 2011) algorithms.

The GD algorithm is a large scale optimization algorithm that is easy to implement and can conveniently be used for linear systems and smooth objective functions (Nesterov, 2012; Duda et al., 1973). However, it is usually not preferred for challenging optimization tasks where the dimension of the search space is very high and there are many local minima in the objective function (Lecun et al., 1998; Ackley et al., 1985; Hinton and Sejnowski, 1986), which is usually the case in complex machine learning applications. The GD algorithm can be recruited for the training of DNNs, but this produces acceptable results only if the initial values of the optimization parameters (i.e. DNN weights) are close to an optimum solution (Hinton and Salakhutdinov, 2006). This requirement is difficult to satisfy and the GD algorithm easily gets trapped at local minima in most problems. In addition, the convergence speed of the GD algorithm is usually slow especially for large training data sets.

The SGD algorithm has been frequently used in high dimensional optimization problems including machine learning problems (Lecun et al., 1998; Nesterov, 2012; Bottou, 1991; Shalev-Shwartz et al., 2011; Bousquet and Bottou, 2008). One of the major advantages of this algorithm is its simplicity in implementation. It also has a relatively faster convergence rate compared to similar algorithms such as the GD algorithm even for problems consisting of large training data sets. However, the performance of the SGD algorithm is sensitive to a number of user-supplied external control parameters such as learning rate, convergence criteria, etc. Unfortunately, there is no analytical method for determining the optimal values of these control parameters for a given problem. The best values of these control parameters are intuitively determined and experimentally verified for each particular optimization problem. Moreover, the SGD algorithm also has some stability issues (Ngiam et al., 2011).

The CG algorithm is generally more stable and easier to check whether it has converged to an optimum solution. It utilizes conjugacy information during the optimization process which results in considerable convergence speed benefits. However, it is relatively complex to implement and too sensitive to noise (Nesterov, 2012; Ngiam et al., 2011). Especially for large scale problems, such as machine learning, where the dimension of the search space and the size of the training data set are usually high, the CG algorithm suffers from the problem of getting trapped at local minima of the objective function, which leads to non-optimal solutions.

The L-BFGS algorithm is another popular algorithm for large scale optimization problems. Its primary advantage is that it needs much less memory than the other algorithms by utilizing the history of gradient evaluations to build up an approximation to the inverse hessian of the objective function (Nocedal, 1980). It is relatively more stable, has fewer control parameters and its converge rate is considerably faster since it uses the approximated inverse hessian matrix. On the other hand, the L-BFGS algorithm has relatively higher implementation complexity and sometimes converges to local minima, generating non-optimal solutions especially when the dimension of the search space is high. In spite of these drawbacks, its attractive properties discussed above usually makes this algorithm a useful choice for complex machine learning problems, including the training of DNNs.

In a typical application, training of a DNN by the L-BFGS algorithm is usually accomplished by tuning the internal parameters of the DNN by using the L-BFGS algorithm so as to minimize the energy of the error signal between the training data set and the output of the DNN. It can obviously be seen that the performance of the training, and hence the performance of the DNN classifier, depends on the performance of the L-BFGS algorithm (Badem et al., 2017). As discussed before, the most prominent factor degrading the performance of the L-BFGS algorithm is the dimension of the search space, i.e. the total number of optimization parameters. Unfortunately, the number of parameters in a DNN training session is very high and this frequently causes the L-BFGS algorithm to converge to local minima and generate non-optimal DNN weights.

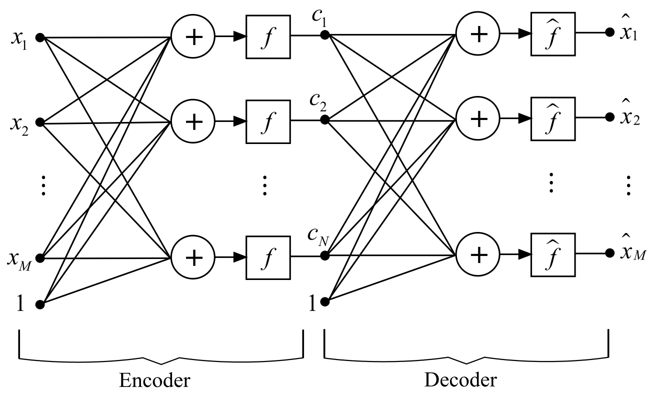

Fig. 1. An autoencoder network with a hidden layer.

In this paper, we propose a simple but very effective strategy, termed as partial limited memory BFGS strategy (pL-BFGS), for the application of the L-BFGS algorithm. We demonstrate that the proposed strategy significantly improves the performance of the L-BFGS algorithm especially when it is employed for the training of a DNN classifier. We also demonstrate that a very efficient classifier can be constructed by combining a two layer SAE with a softmax layer and then training them by the L-BFGS algorithm applied under the proposed training strategy. We test the classification performance of this classifier on a number of popular benchmark data sets from the literature and show that this well-tuned custom combination of different machine learning methodologies leads to considerable improvements in classification results. The DNN classifier trained with strategy is also compared with state-of-the-art classifiers including, support vector machine (SVM), k-nearest neighbor (KNN), decision three (DT) and naive Bayes (NB) to show the efficiency of the strategy.

2. Method

2.1. The autoencoder

An AE is a feed forward artificial neural network that comprises an input layer, a hidden layer and an output layer. The AE is trained to generate its own input at the output, hence the number of neurons in the output layer is always the same as the number of inputs (Ng, 2011; Ngiam et al., 2011; Bengio, 2012; Ranzato et al., 2006). The output of the hidden layer in an AE network represents a different encoding of the input vector and is termed a *code* (Hinton and Salakhutdinov, 2006). The AE is trained to establish a mapping from its input space to the code (feature) space, which usually has a lower dimension than the input space when the number of neurons in the hidden layer is fewer than the number of inputs. In order to improve classification efficiency in some situations, however, the dimension of the code space may be chosen higher than that of the input space. In both cases, the AE attempts to provide a better representation of the input vector by replacing it with an appropriate code.

Fig. 1 shows an AE network with a hidden layer. Here the number of neurons in the output layer is equal to the number of inputs, M, which is the dimension of the input space, and the number of the neurons in the hidden layer is N, which is the dimension of the code space, where M and N are positive integer numbers. The left half of the AE is called the *encoder*, whose input is the input of the AE and output is the output of the hidden layer of the AE. The encoder converts a given input vector into a *code*, which is actually a more efficient representation of the input vector.