A way to reduce the time consumption effect of for-loops for training neural networks: optimized propagation

Dr. Amir Valizadeh (thisisamirv@gmail.com)
Tehran University of Medical Scences

Method Article

Keywords: Artificial Neural Networks

Posted Date: August 5th, 2021

DOI: https://doi.org/10.21203/rs.3.rs-776504/v3

Abstract

In this paper, an alternative approach to the conventional backpropagation algorithm is presented that results in faster convergence of the loss of neural network models. The new algorithm (called optimized propagation) was used to model a neural network that was trained on some data and results were compared with the same data being modeled using the conventional backpropagation algorithm. These results indicate the superiority of optimized propagation to conventional backpropagation in the terms of reducing the loss of the model faster.

Backpropagation And The Vectorization Problem

While training a neural network (NN), the developer defines a loss function (cost function) (Bishop, 1995) that quantifies how poorly the network is currently achieving its goals. The main purpose of a training algorithm is to reduce this error as much as possible. In the era of deep learning, backpropagation (backprop) is the most widely used algorithm for training feedforward NNs. First discovered in 1986 (Rumelhart et al., 1986), the main idea behind the backprop is quite simple: It works by computing the gradient of the loss function with respect to each weight by the chain rule, computing the gradient one layer at a time, iterating backward from the last layer to avoid redundant calculations of intermediate terms in the chain rule. Then, using the computed gradients, weights of the network will be updated using a gradient method (such as batch gradient descent). After updating the weights, the same forward propagation and backpropagation will be iterated until the loss function is minimized to a reasonable loss. In order to iterate, a so-called “for-loop” will be used in the computer code of the network. Unfortunately, for-loops are known to speed down computation speed, as they do not use “code vectorization”.

Many central processing units (CPUs) have "vector" or "SIMD" (Single instruction, multiple data) instruction sets that apply the same operation simultaneously to two, four, or more pieces of data. "Code vectorization" is the process of rewriting a loop so that instead of processing a single element of array N times, it processes X elements ($X \geq 1$) of the array simultaneously N/X times. This process is known to significantly speed up the processing time for an iterative task (Van Der Walt et al., 2011). The process is presented in **Figure 1**.

The fact that the iterative task in training an NN using a backpropagation algorithm can not be vectorized, results in sometimes huge time consumption to train a model. Unfortunately, up to this date, there are no robust methods to completely overcome this problem. In this paper, I aim to present a way to bypass this problem (to some considerable extent) by modifying the backpropagation algorithm.

Optimized [back] Progression

Complex calculus and mathematical equations aside, to put it very simply, a backpropagation algorithm works as presented in **Figure 2**. First, the model takes a forward pass. After that, the gradient of the loss

function with respect to each weight by the chain rule will be calculated. Using the calculated gradients, weights are then updated by a gradient method. This process iterates over and over again until the updated weights result in a model outcome with a reasonably low loss.

As mentioned before, this process can not be vectorized, which means each iteration must take place one at a time in the CPU. In my proposed method, first, some alterations are made to the backpropagation algorithm.

For initiation of training an NN, the first step is to initialize some random weights. As these weights are random, they may be or may not be close to the optimal weights one desires. The closer the weights, the fewer iterations are needed to train the model to obtain optimal weights. Considering that in a standard backpropagation, it is needed to compute the gradients all the way back to the gradient of the first weights of the model in order to update the parameters, if the random initial weights are far from the optimal weights desired, this could result in a time-consuming procedure that updates the weights to their optimal value very slowly for the first layers. It is because due to the chain rule, updating the first weights requires the updating of the weights after them and so on, and thus, calculating all the weights to update the current weights and then forward passing the model to obtain some new loss to calculate new gradients, can be very time-consuming. In my proposed method, weights are updated from the last layer up to the first layer one by one. In other words, after implementing a forward pass, gradients for the last layer are calculated and its weights are updated. Then, another forward pass only using the last layer will be performed to obtain a new loss. Using the new loss, this time, we implement a backpropagation step to compute the gradients of the last layer and the layer before it. Using the new calculated gradients, the weights of these two layers are updated, and then another forward pass using only these two layers will be implemented. The same process repeats, each time by including another layer backward until we reach the first (input) layer and update all the weights once. Then another forward pass, this time using the whole model is implemented. Using the new loss calculated after this pass, we will repeat the same process as above to update all parameters only by one again. A simple illustration of this technique is presented in **Figure 3**.

This process has two advantages: first, we update parameters one by one which reduces the number of iterations needed to obtain the optimal weights, which in turn results in less time consumption. Second, each “vector” given to the CPU includes more than one iteration of the model, which also reduces the time needed to obtain our desired optimal weights. Although this process is not equal to vectorizing the code at all, to some significant extent, it speeds up the training process. In the following section, I implement this technique in a four layers NN built for the cat classification task and will report the results for both standard backpropagation and this new propagation technique, so the reader can compare the results and gain a better intuition of the proposed method.

Training An Nn Model To Classify Cats, Using The Optimized Propagation Algorithm

Data fed to this model came from Dr. Andrew Ng's dataset in one of his training courses (Ng, 2017). All the codes used to build the model are presented in the **Supplementary file**. The model was implemented in Python 3.9. Three packages were used to build this model: NumPy, which is a package that helps with vectorizing some sections of the code; h5py, which is used to import the data into Python; and MatPlotLib, which is used to present plots to compare the models. The data consisted of 209 images of cats in the training dataset. Each image is of size: (64, 64, 3). First, I imported the data into Python using the h5py package. All data of images were reshaped into one vector, used for the training process. I decided my model to have 3 hidden layers of the dimensions 20, 7, and 5 respectively. The input layer was a matrix of the dimension of (12288, 209). Weights were initialized using the following formula:

See formula 1 in the supplementary files section.

The bias term (b) for each layer was simply set to zero. After initiating the weights, each weight was copied into two new variables, each used to train a model (standard backprop and optimized backprop). The model had 4 activation units. Starting from the activation of the first layer to the last, it included: ReLU, ReLU, ReLU, and Sigmoid activation functions. At the end of each iteration, the loss for the model was calculated and saved in a list. Two models, one standard backprop, and another, optimized propagation were built and trained on the data for 1000 epochs, using a constant learning rate (α) of 0.0075. After completing the training process for the models, the calculated costs were plotted against the epoch number for each model, using the matplotlib package. The y-axis of the plots was limited to a range of 0.1-0.7 to enable the readers to compare the models more conveniently. In **Figure 4**, the output plots for both models are presented. The blue-line plot resembles the model which used the standard backpropagation algorithm, while the red-line plot resembles the model which used the optimized propagation algorithm. As it can be seen, the optimized model reached a loss of 0.1 after 1000 iterations, while the standard model only managed to reach a loss of around 0.3.

The time needed to train each model was measured. The standard model took one minute and 5 seconds to train. While the optimized model took roughly a little higher (1 minute and 7 seconds). Given the small difference in time used, the optimized model managed to reach a significantly lower loss than the standard model.

Conclusions

In this paper, I have presented some optimization to the standard backpropagation algorithm which results in faster training of an NN model to achieve desired optimal weights in a faster manner than the standard conventional backpropagation algorithm.

Declarations

The author declares no conflicts of interest.

References

Bishop, C.M. (1995). *Neural networks for pattern recognition*. Oxford university press.

Ng, A. (2017). *Deep learning specialization*. Coursera. https://www. coursera. org/specializations/deep-learning

Rumelhart, D.E., Hinton, G.E., & Williams, R.J. (1986). Learning representations by back-propagating errors. *nature*, *323*(6088), 533-536.

Van Der Walt, S., Colbert, S.C., & Varoquaux, G. (2011). The NumPy array: a structure for efficient numerical computation. *Computing in science & engineering*, *13*(2), 22-30.

Figures

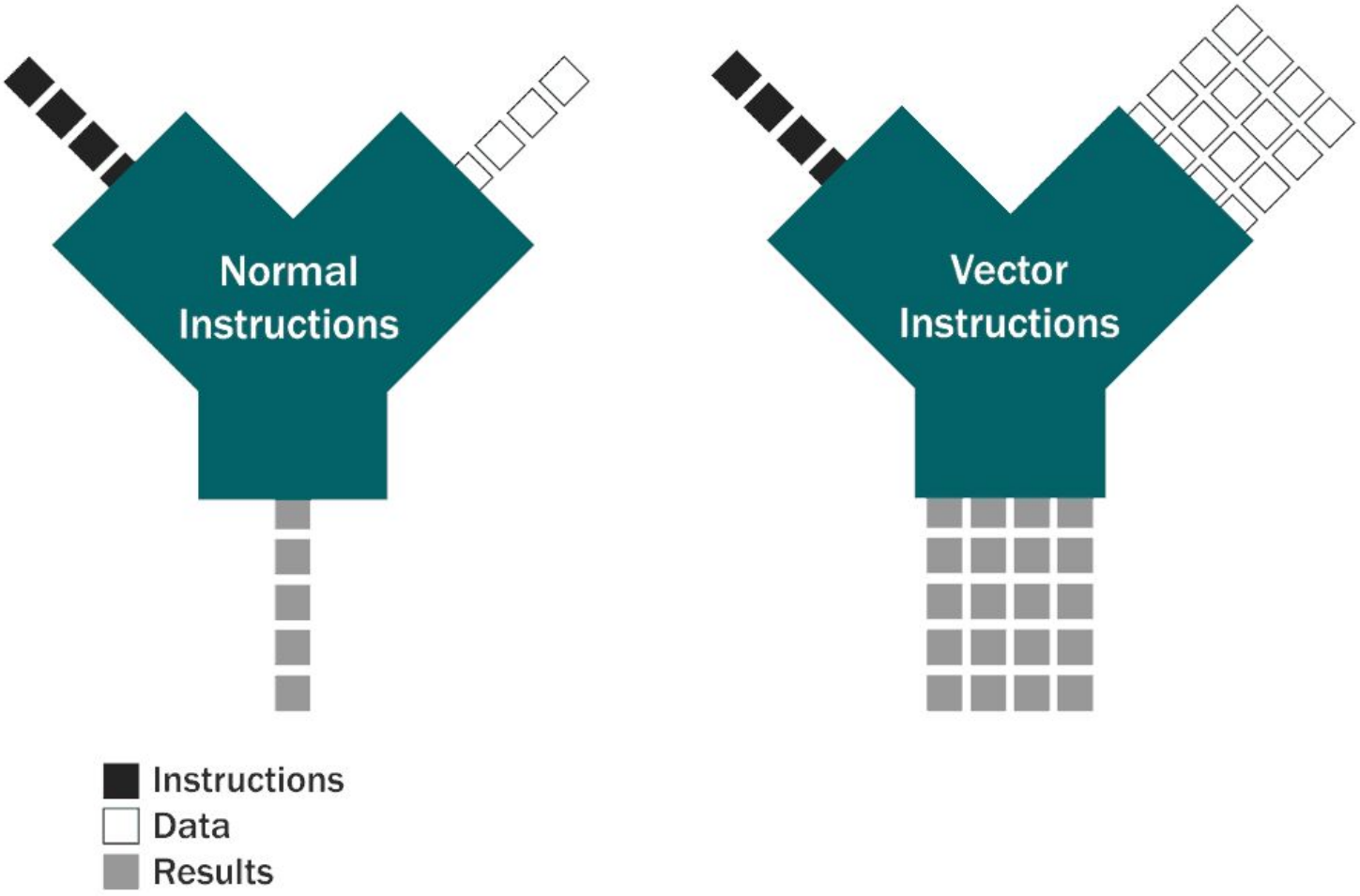

Figure 1

Normal vs vectorized code processing.

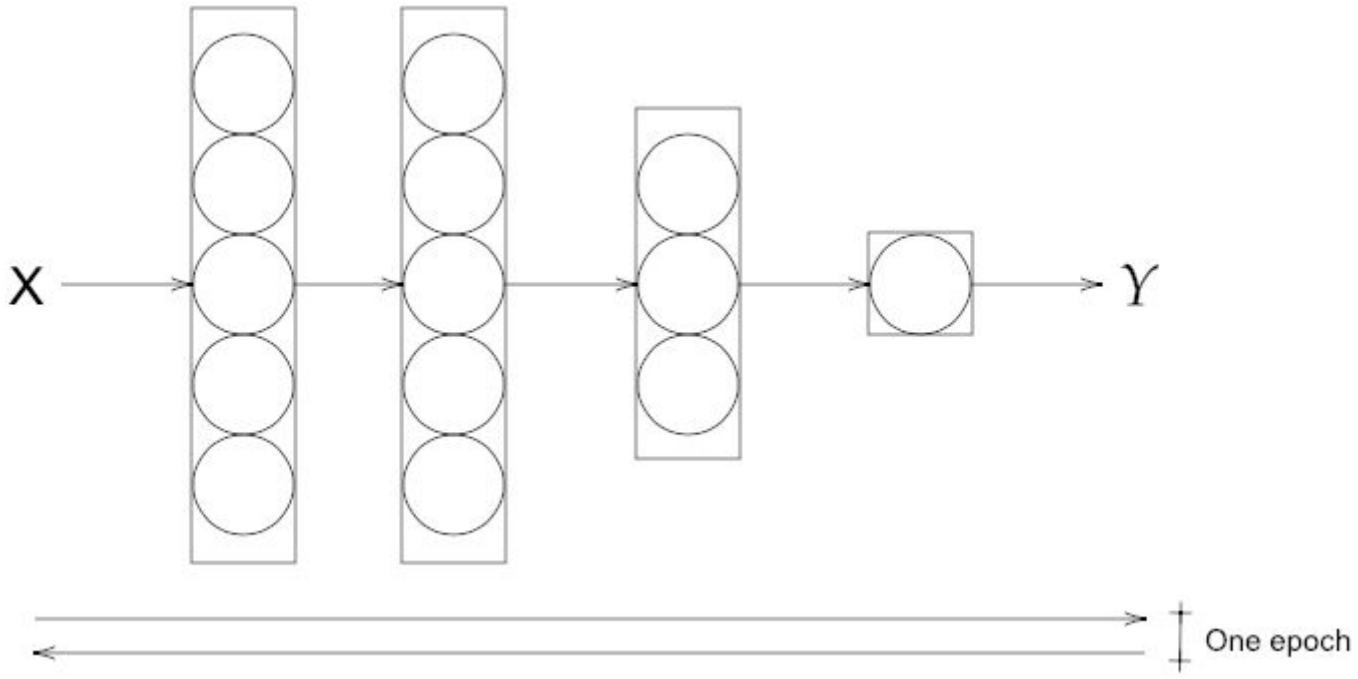

Figure 2

One epoch of a conventional backpropagation model.

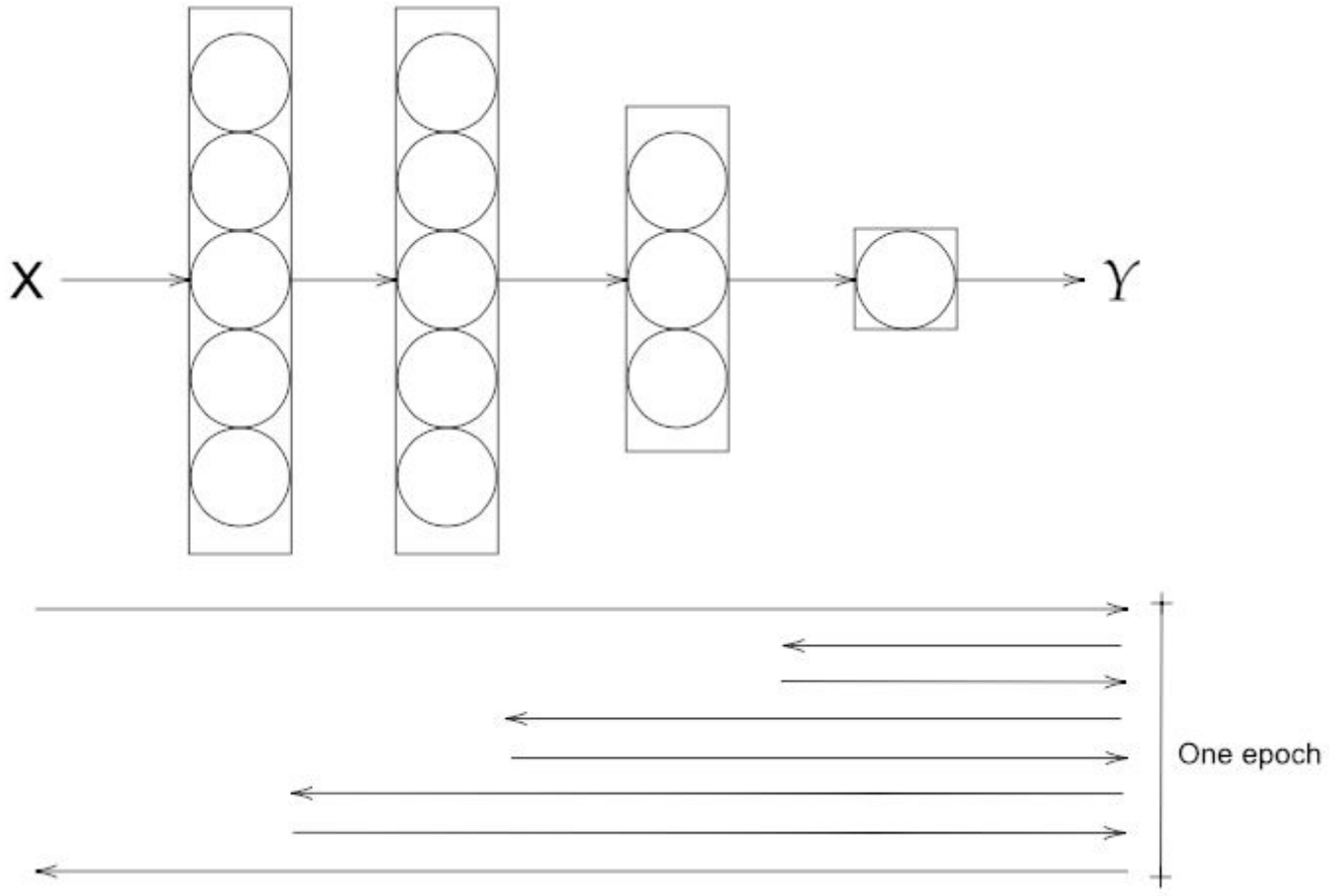

Figure 3

One epoch of the optimized backpropagation model.

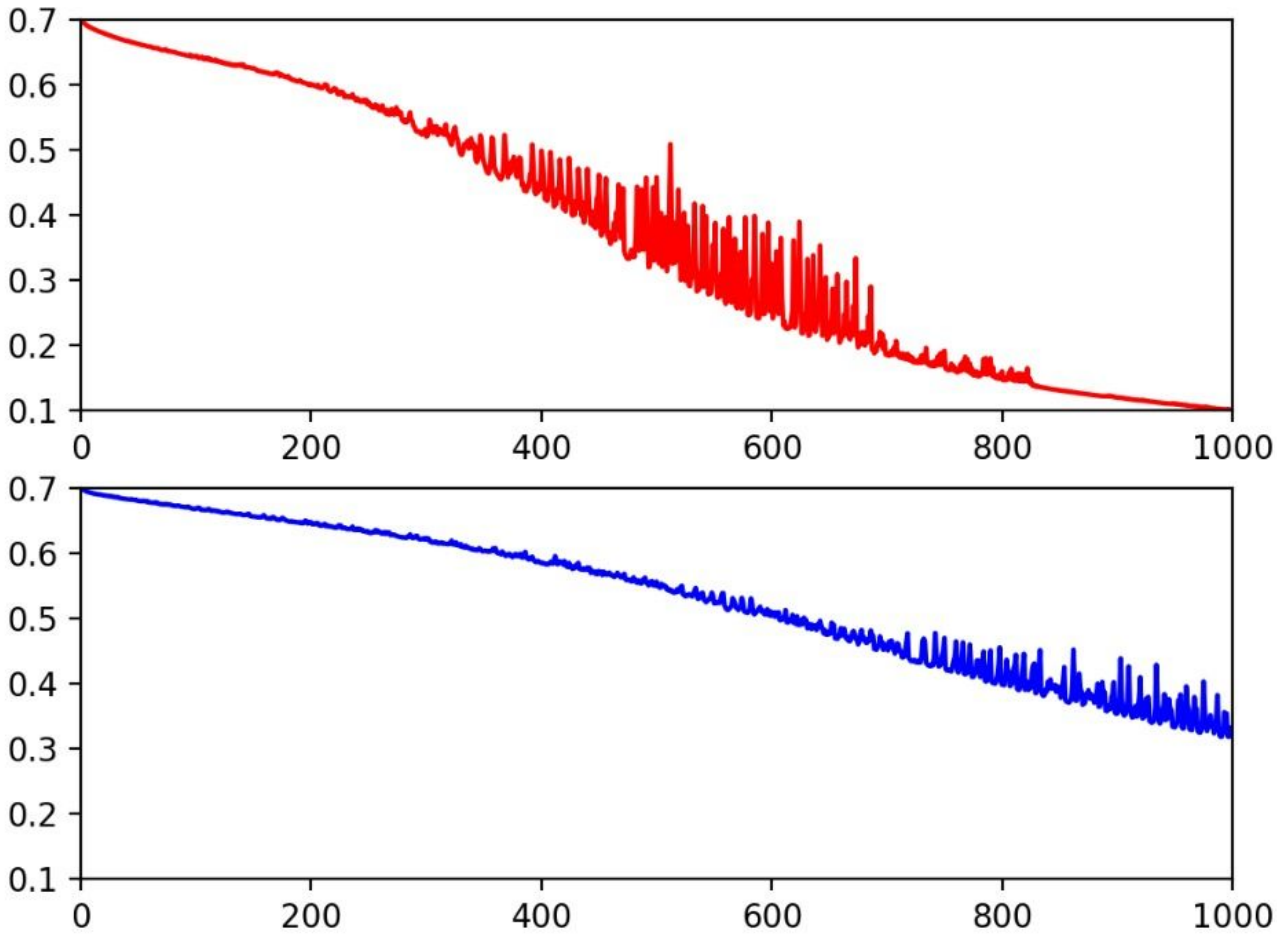

Figure 4

The loss of the model plotted against the epoch number for each model. Red = optimized model; blue = conventional backpropagation model.

Supplementary Files

This is a list of supplementary files associated with this preprint. Click to download.

- Supplementaryfile.zip
- formula.docx