Training Deep Neural Networks

20337025 Cui Canming

2023·1·1

Abstract

Deep neural network(DNN) is very popular, but it also faces many training difficulties. This paper aims to analyze the training depth neural network. We will introduces back-propagation algorithm, stochastic gradient descent algorithm and other commonly used optimization algorithms in this article, and studies how to select appropriate training methods.

This is the final paper of the optimization course of Sun Yat-sen University. The writing content is determined by the course requirements. Due to the limited level of the author, there may be many problems. Please correct and criticize the shortcomings.

1 Introduction

A deep neural network (DNN)[GBC16] is a type of artificial neural network that is composed of multiple layers of interconnected nodes. These layers are called hidden layers because they are not visible to the input or output layers, which are called the input layer and the output layer, respectively. DNNs are used for a variety of tasks such as image classification, language translation, and speech recognition.

DNNs[LBH15] are inspired by the structure and function of the human brain, and they are able to learn and adapt to new data by adjusting the strengths of the connections between the nodes. They are trained using large datasets and an optimization algorithm, such as stochastic gradient descent, to find the optimal values for the weights and biases of the connections between the nodes.

There are several challenges and difficulties that can arise when training deep neural networks (DNNs). Some of the common ones include: Overfitting: DNN is too complex for the training data and fits noise rather than the underlying pattern, leading to poor generalization. Underfitting: DNN is too simple and fails to learn the underlying pattern. Vanishing gradients: Gradients of the parameters with respect to the loss function become very small, slowing down training. Exploding gradients: Gradients of the parameters with respect to the loss function become very large, causing unstable training and parameter divergence. Local minima: Optimization algorithm gets stuck in a suboptimal point in the loss function. Slow convergence: DNN has many parameters and requires a large amount of data, making training slow and computationally intensive. Data imbalance: DNN learns to predict the majority class more accurately, while performing poorly on the minority class. Poor quality data: Training data is of poor quality, negatively impacting DNN performance.

Accordingly, there are several methods that can be used to train deep neural networks (DNNs). Some of the common ones include:

- Stochastic Gradient Descent (SGD): This is a popular optimization algorithm that is used to update the weights of the DNN. It makes updates to the weights after processing each training example, rather than waiting to process the entire dataset. SGD can be used with mini-batch learning, where the weights are updated after processing a small batch of training examples, rather than a single example.

- Back-propagation: This is an algorithm used to train DNNs by propagating the error from the output layer back through the network and updating the weights of the connections between the nodes. It uses the gradient of the loss function with respect to the weights to compute the updates.

- Mini-batch learning: This is a variant of SGD that updates the weights of the DNN after processing a small batch of training examples, rather than a single example. This can be more computationally efficient and can lead to faster convergence.

- Momentum: This is a technique that can be used to accelerate SGD by adding a momentum term to the updates. It helps the optimization algorithm escape from local minima and speeds up convergence.
- Adaptive learning rates: These methods adjust the learning rate of the optimization algorithm based on the training data and the optimization progress. This can help the algorithm converge faster and avoid getting stuck in suboptimal points. Examples include AdaGrad and Adam.
- Weight decay: This is a regularization technique that reduces the complexity of the model by adding a penalty on the weights of the connections in the DNN. It helps prevent overfitting and improve the generalization of the model.
- Dropout: This is a regularization technique that randomly sets a fraction of the activations of the hidden units to zero during training. It helps prevent overfitting and improve the generalization of the model.

In this article, we will focus on the back-propagation algorithm in the second section, and then introduce several optimization algorithms that can be used to train neural networks, such as SGD, SGD with momentum, Adagrad, RMSProp, and Adam in the third section. In the fourth section, we will introduce specific application scenarios, and finally we will have a summary.

2 Back-propagation Algorithm

Back-propagation was proposed by Rumelhart[RHW86] et al. It is a widely used algorithm for training deep neural networks (DNNs). It is based on the gradient of the loss function with respect to the weights of the connections between the nodes in the network. The goal of back-propagation is to adjust the weights in such a way as to minimize the loss function and improve the prediction accuracy of the DNN.

To derive the Back-propagation algorithm, We define a neural network with any number of layers, with the input layer at the bottom, any number of intermediate layers (that is, hidden layers) in the middle, and the output layer at the top.

For units at any level in the network, the total input x_j to unit j is a linear function of the output of units at the level above unit j and the weights on these connections:

$$x_j = \sum_i y_i w_{ij}$$

Of course, We can add bias to the network unit by introducing a constant 1 as input. The weight on the additional input is called a bias and can be treated like any other weight.

The output of unit j is the result of input x_j passing through a nonlinear function. We have chosen sigmoid function here, but it could have been any other function:

$$y_j = \frac{1}{1 + e^{-x_j}}$$

Our goal is to be able to find a set of weights that will ensure that for each input vector, the neural network will produce an actual output vector that is the same or very close to the desired output vector. We measure network performance by calculating the sum of squares of the difference between the real desired output and the actual output. The loss function L is defined by the following equation:

$$L = \frac{1}{2} \sum_c \sum_j (y_{j,c} - d_{j,c})^2$$

Where, c is the sample index, j is the index of the output unit, y is the actual output value, and d is the expected output value. In order to minimize the loss function through gradient descent, it is necessary to calculate the partial derivative of the loss function L to each weight.

The above explains the process of forward propagation, that is, the state vector of each layer is determined by the input value they receive from the lower layer using first equation and second equation.

Next is the process of back propagation, which is realized by calculating the partial derivatives of each output unit. We starts by computing $\frac{\partial L}{\partial y}$ for each of the output units:

$$\frac{\partial L}{\partial y_j} = y_j - d_j$$

Then, according to the chain rule and the derivation of sigmoid function, we can compute $\frac{\partial L}{\partial x}$:

$$\frac{\partial L}{\partial x_j} = \frac{\partial L}{\partial y_j} \cdot \frac{dy_j}{dx_j} = \frac{\partial L}{\partial y_j} \cdot y_i(1 - y_i)$$

Input x_j is a linear function of the weights on the connections, so we can compute how the loss function will be affected by changing weights. For a weight w_{ji}, from i to j , the derivative is:

$$\frac{\partial L}{\partial w_{ji}} = \frac{\partial L}{\partial x_j} \frac{\partial x_j}{\partial w_{ji}} = \frac{\partial L}{\partial x_j} \cdot y_i$$

For unit i of the layer before unit j, the derivative $\frac{\partial L}{\partial y_i}$ of loss function to its output through unit j can be obtained by the following equation:

$$\frac{\partial L}{\partial x_j} \cdot \frac{\partial x_j}{\partial y_i} = \frac{\partial L}{\partial x_j} \cdot w_{ji}$$

Taking into account all connections sent from i, we can get:

$$\frac{\partial L}{\partial y_i} = \sum_j \frac{\partial L}{\partial x_j} \cdot w_{ji}$$

So far, when the $\frac{\partial L}{\partial y}$ of all units in the last layer are given, we can calculate the partial derivative $\frac{\partial L}{\partial y}$ of any element in the penultimate layer according to the above formula. We can repeat this process to calculate forward, continuously calculate the partial derivative of the previous layer, and then use the gradient descent method to update the weight of each layer. This is the process of back-propagation.

In summary, the back-propagation algorithm is an effective method for training DNNs by adjusting the weights in such a way as to minimize the loss function and improve the prediction accuracy of the DNN. It is based on the gradient of the loss function with respect to the weights and uses an optimization algorithm such as SGD to compute the updates.

3 Optimization Algorithms

3.1 Stochastic gradient descent(SGD)

Stochastic gradient descent (SGD)[RM51] is an optimization algorithm widely used to train deep neural networks (DNNs). It is based on the idea of using the gradient of the loss function with respect to the model parameters to iteratively adjust the parameters in the direction that minimizes the loss.

Before we introduce stochastic gradient descent, let's first understand gradient descent. As mentioned in the previous section, we can obtain $\frac{\partial L}{\partial w_i}$ for any weight i of the DNN through back-propagation.

We want to find a group of weights to minimize the value of the loss function. Obviously, it is a good choice to use the loss function to derive the weight value. The meaning of the derivative is to provide a direction. Changing the weight value in this direction will make the loss function larger. More vividly, it is called gradient. Since the gradient determines the steepest rising direction of the loss function, the training rule for gradient descent is:

$$w_i = w_i + \Delta w_i$$
$$\Delta w_i = -\alpha \frac{\partial L}{\partial w_i}$$

We call α learning rate, that is, the step size of each weight update. The learning rate is very important, because if it is too small, it will be slow to find the minimum value of the function. If it is too large, it will diverge on both sides of the extreme point.

The ordinary gradient descent algorithm needs to traverse the entire data set when updating the regression coefficients, which is a batch processing method. In this way, when the training data is extremely busy, the following problems may occur:

1. The convergence process may be very slow;
2. If there are multiple local minima on the loss function, there is no guarantee that the process will find the global minimum.

In order to solve the above problem, in practice, we apply a variant of gradient descent called stochastic gradient descent. The loss function in the above formula is obtained for all training samples, that is, updating w_i once requires traversing the entire sample set, while the idea of random gradient descent is to update the weight value according to each individual training sample, that is, updating w_i only requires one sample. So the SGD loss function should be:

$$L = \frac{1}{2}\sum_j (y_{j,ci} - d_{j,ci})^2$$

Where ci is the subscript of randomly selected samples.

In general, gradient descent converges to the optimal solution more slowly than SGD, especially when working with large datasets. This is because SGD is more computationally efficient and can make progress on the optimization problem faster. However, SGD is also more sensitive to the learning rate, and it may be necessary to experiment with different learning rates to find one that works well for a particular problem.

Another limitations of SGD is that it can be prone to getting stuck at local minima, which are points in the optimization landscape where the loss is relatively low but not the lowest possible.

There are a few reasons why SGD can be prone to getting stuck at local minima:

- Noise: One reason for getting stuck at local minima is the presence of noise in the training data, which can cause the optimization process to oscillate around a local minimum rather than converging to the global minimum.
- Learning rate: Another reason for getting stuck at local minima is the choice of the learning rate, which determines the step size of the optimization process. If the learning rate is too large, the optimization process may overshoot the global minimum and get stuck at a local minimum. On the other hand, if the learning rate is too small, the convergence may be slow, and the optimization process may get stuck at a local minimum.
- Non-convex loss: Another reason for getting stuck at local minima is the presence of non-convex loss functions, which have multiple local minima rather than a single global minimum. In these cases, the optimization process may get stuck at a local minimum and fail to find the global minimum.

To avoid getting stuck at local minima, it is important to use appropriate initialization techniques and choose a learning rate that is neither too large nor too small. It is also helpful to use optimization algorithms that are less sensitive to local minima, such as SGD with momentum or Adam.

3.2 SGD+Momentum

Stochastic gradient descent (SGD) with momentum[SMDH13] is an extension of the SGD algorithm.

One limitation of SGD is that it has trouble navigating ravines, that is, areas where the surface curves much more steeply in one dimension than in another[Sut86], which are common around local optima. In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum as in Figure 1a.

Momentum is a technique that helps the optimization process proceed more smoothly and avoid falling into local minima. It does this by adding a portion of the previous update to the current update, as shown in Figure 1b, which helps to suppress oscillations and improve convergence speed.

It does this by adding a portion of the past weight to the new weight:

$$v_t = \gamma v_{t-1} + \alpha \frac{\partial L}{\partial w_i}$$

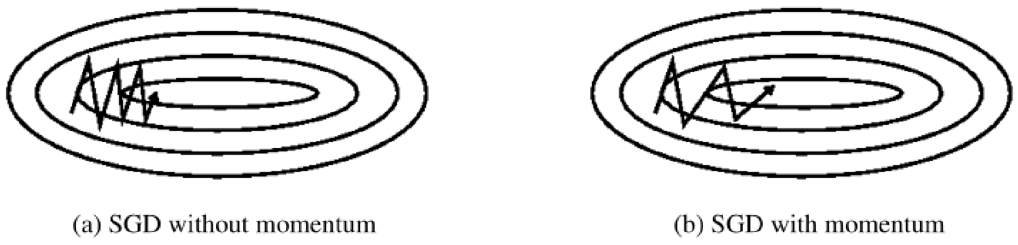


Figure 1: Convergence Process of SGD with and without Momentum, This picture is from Sebastian Ruder's paper[Rud16]

$$w_i = w_i - v_t$$

γ is the momentum term, which is usually set to 0.9 or a similar value. The momentum term makes SGD increase in the dimension where the gradient points in the same direction, and reduce the update in the dimension where the gradient changes direction.

Sutskever et al.[SMDH13] show that using momentum can significantly improve the convergence of the SGD algorithm and lead to better performance on a variety of tasks, including training deep autoencoders and training deep belief networks.

3.3 Adagrad

AdaGrad (Adaptive Gradient)[DHS11] is an extension of Gradient descent algorithm. It is a gradient descent optimization method with adaptive learning rate. It makes the learning rate of parameters adaptive, and performs large updates for infrequent parameters and small updates for frequent parameters.

In order to express it more clearly, we record the gradient of the loss function in SGD to w_i at time t as $g_{t,i}$:

$$g_{t,i} = (\frac{\partial L}{\partial w_i})_t$$

On the basis of the SGD method, AdaGrad adjusts the step size in different component directions by recording the cumulative gradient of each component:

$$w_{t+1,i} = w_{t+1,i} - \frac{\alpha}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

Here, G_t is a diagonal matrix, where $G_{t,ii}$ is the square sum of all gradients as from unit i to time step t. ϵ is a smoothing term that avoids division by zero, which is usually set to a small value.

Adagrad is particularly well-suited for problems with sparse gradients, where some parameters are updated more frequently than others. The per-parameter learning rate allows Adagrad to automatically adjust the learning rate for each parameter and improve the stability of the optimization process. Apart from this, the adaptive learning rate of Adagrad helps to stabilize the convergence of the optimization process, especially when working with large datasets.

The main disadvantage of Adagrad is that it accumulates the square of the gradient on the denominator: because each increased item is positive, the total number accumulated in the training process is increasing. In turn, this will lead to a reduction in the learning rate and eventually become infinitesimal. At this time, the algorithm can no longer obtain additional knowledge. However, the Adadelta[Zei12] algorithm derived from the extension of Adagrad can effectively solve this problem, which will not be described here.

3.4 RMSPro

RMSProp extends Adagrad to solve the problem that AdaGrad's learning rate drops sharply. I can't find the paper of RMSProp, it was developed by Geoff Hinton in a lecture at the University of Toronto in 2012. In order to further optimize the problem that the loss function has too large swing amplitude in updating, and further accelerate the convergence speed of the function, RMSProp algorithm uses differential square weighted average for the gradient of weight :

$$E[g^2]_t = \lambda E[g^2]_{t-1} + (1-\lambda) g_t^2$$
$$w_{t+1,i} = w_{t,i} - \frac{\alpha}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_{t,i}$$

We can see that Adadelta does not accumulate the square of all past gradients before t, but adopts an $E[g^2]$ that only depends on the previous average value and the current gradient. It restricts the window of accumulated past gradients to some fixed size w, which inhibits the rapid decline of learning rate of Adagrad.

3.5 Adam

Adam(Adaptive Moment Estimation)[KB14] is also a method to calculate the adaptive learning rate of each parameter, which combines the advantages of AdaGrade and RMSProp optimization algorithms. The first moment estimation $_t$(i.e. the mean value of the gradient) and the second moment estimation v_t(i.e. the non centralized variance of the gradient) of the gradient are comprehensively considered to calculate the update step:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

When m_t and v_t are initialized to vectors of zero, they are biased towards zero, especially in the initial time step and when the decay rate is small (i.e 1 and 2 is close to 1). So we need to counteract these biases by computing bias-corrected first and second moment estimates:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

The equation for updating the weight is as follows:

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{\hat{v}_t + \epsilon}} \cdot \hat{m}_t$$

Although Adam is currently the dominant optimization algorithm, in many areas (such as object recognition in computer vision, machine translation in NLP), the best results are still obtained using SGD with momentum. The results of the paper by Wilson et al[WRS+17]. show that in object recognition, character-level modeling, and syntax parsing, adaptive learning rate methods (including AdaGrad, AdaDelta, RMSProp, Adam, etc.) generally perform worse than the SGD with momentum.

4 Selection

In addition to the above optimization algorithms, there are many other optimization algorithms in training DNN. You may be confused: Which Optimizer Should I Use in training my Machine Learning models? Philipp Wirth, a machine learning engineer at Lightly ai, wrote a guide that summarized popular optimizers commonly used in computer vision, natural language processing and machine learning, and gave suggestions on how to select appropriate optimizers. You can find this article in here.

Specifically, this article proposes to select an optimizer based on the following three questions:

1. Find a related research paper and start with using the same optimizer.

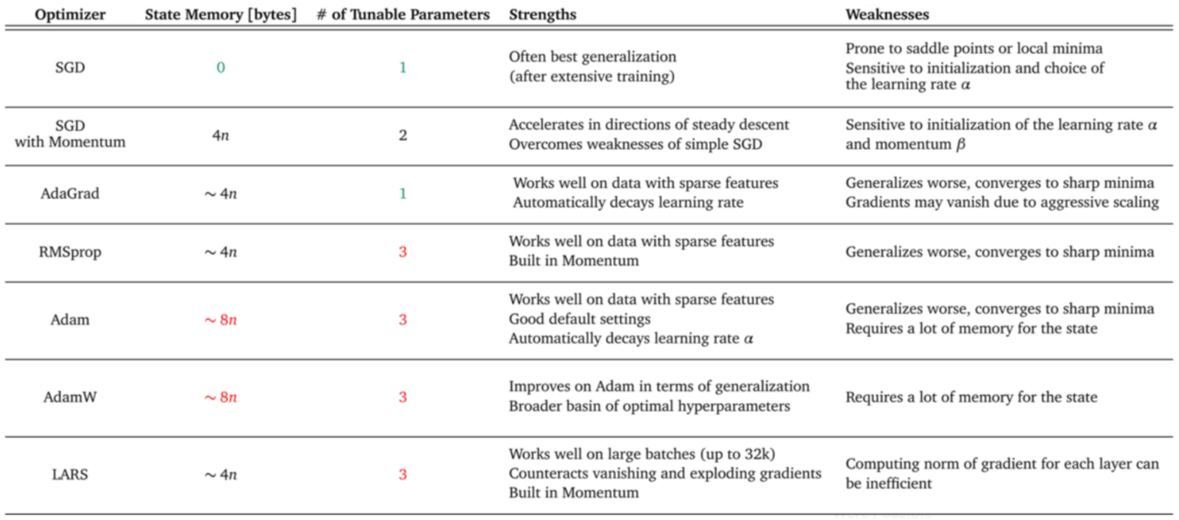
2. Consult the table in the article and compare properties of your dataset to the strengths and weaknesses of the different optimizers.

3. Adapt your choice to the available resources.

In addition, before choosing to optimize it, you should ask yourself the following three questions:

1. What are the SOTA results of similar data sets and tasks?
2. What optimizers are used?
3. Why use these optimizers?

The table in Philipp Wirth's article is shown in Figure 2.



Optimizer	State Memory [bytes]	# of Tunable Parameters	Strengths	Weaknesses
SGD	0	1	Often best generalization (after extensive training)	Prone to saddle points or local minima Sensitive to initialization and choice of the learning rate α
SGD with Momentum	$4n$	2	Accelerates in directions of steady descent Overcomes weaknesses of simple SGD	Sensitive to initialization of the learning rate α and momentum β
AdaGrad	$\sim 4n$	1	Works well on data with sparse features Automatically decays learning rate	Generalizes worse, converges to sharp minima Gradients may vanish due to aggressive scaling
RMSprop	$\sim 4n$	3	Works well on data with sparse features Built in Momentum	Generalizes worse, converges to sharp minima
Adam	$\sim 8n$	3	Works well on data with sparse features Good default settings Automatically decays learning rate α	Generalizes worse, converges to sharp minima Requires a lot of memory for the state
AdamW	$\sim 8n$	3	Improves on Adam in terms of generalization Broader basin of optimal hyperparameters	Requires a lot of memory for the state
LARS	$\sim 4n$	3	Works well on large batches (up to 32k) Counteracts vanishing and exploding gradients Built in Momentum	Computing norm of gradient for each layer can be inefficient

Figure 2: The table in Philipp Wirth's article

5 Conclusion

In this article, we first introduce the deep neural network and analyze its training difficulties. Secondly, we review the back-propagation algorithm. Then we analyze stochastic gradient descent(SGD) and several algorithms used to optimize SGD: Momentum, Adagrad, RMSprop, Adam. Finally, based on the work of Philipp Wirth's, some suggestions on the selection of optimization algorithms are given.

References

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[KB14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[RM51] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[Rud16] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[SMDH13] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147. PMLR, 2013.

[Sut86] Richard Sutton. Two problems with back propagation and other steepest descent learning procedures for networks. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society, 1986*, pages 823–832, 1986.

[WRS$^+$17] Ashia C Wilson, Rebecca Roelofs, Mitchell Stern, Nati Srebro, and Benjamin Recht. The marginal value of adaptive gradient methods in machine learning. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[Zei12] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.