Gradient Descent based Optimization Algorithms for Deep Learning Models Training

Jiawei Zhang jiawei@ifmlab.org
Founder and Director
Information Fusion and Mining Laboratory
(First Version: February 2019; Revision: March 2019.)

Abstract

In this paper, we aim at providing an introduction to the gradient descent based optimization algorithms for learning deep neural network models. Deep learning models involving multiple nonlinear projection layers are very challenging to train. Nowadays, most of the deep learning model training still relies on the back propagation algorithm actually. In back propagation, the model variables will be updated iteratively until convergence with gradient descent based optimization algorithms. Besides the conventional vanilla gradient descent algorithm, many gradient descent variants have also been proposed in recent years to improve the learning performance, including Momentum, Adagrad, Adam, Gadam, etc., which will all be introduced in this paper respectively.

Keywords: Gradient Descent; Optimization Algorithm; Deep Learning

1. Introduction

In the real-world research and application works about deep learning, training the deep models effectively still remains one of the most challenging work for both researchers and practitioners. By this context so far, most of the deep model training is still based on the back propagation algorithm, which propagates the errors from the output layer backward and updates the variables layer by layer with the gradient descent based optimization algorithms. Gradient descent plays an important role in training the deep learning models, and lots of new variant algorithms have been proposed in recent years to further improve its performance. Compared with the high-order derivative based algorithms, e.g., Newton's method, etc., gradient descent together with its various variant algorithms based on first-order derivative are much more efficient and practical for deep models. In this paper, we will provide a comprehensive introduction to the deep learning model training algorithms.

Formally, given the training data $\mathcal{T} = \{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \cdots, (\mathbf{x}_n, \mathbf{y}_n)\}$ involving n pairs of feature-label vectors, where the feature vector $\mathbf{x}_i \in \mathbb{R}^{d_x}$ and label vector $\mathbf{y}_i \in \mathbb{R}^{d_y}$ are of dimensions d_x and d_y respectively. We can represent the deep learning model used to classify the data as a mapping: $F(\cdot; \boldsymbol{\theta}) : \mathcal{X} \to \mathcal{Y}$, where $\mathcal{X}$ and $\mathcal{Y}$ denotes the feature and label space respectively and $\boldsymbol{\theta}$ denotes the variable vector involved in the mapping. Formally, given an input data instance featured by vector $\mathbf{x}_i \in \mathcal{X}$, we can denotes the output by the deep learning model as $\hat{\mathbf{y}}_i = F(\mathbf{x}_i; \boldsymbol{\theta})$. Compared against the true label vector $\mathbf{y}_i$ of the input instance, we can represent the introduced error by the model with the loss term $\ell(\hat{\mathbf{y}}_i, \mathbf{y}_i)$. In the remaining part of this paper, we may use error and loss interchangeably without any differentiations.

Many different loss functions, i.e., $\ell(\cdot,\cdot)$, have been proposed to measure the introduced model errors. Some representative examples include

- *Mean Square Error (MSE)*:

$$\ell_{MSE}(\hat{\mathbf{y}}_i, \mathbf{y}_i) = \frac{1}{d_y}\sum_{j=1}^{d_y}\left(\mathbf{y}_i(j) - \hat{\mathbf{y}}_i(j)\right)^2. \tag{1}$$

- *Mean Absolute Error (MAE)*:

$$\ell_{MAE}(\hat{\mathbf{y}}_i, \mathbf{y}_i) = \frac{1}{d_y}\sum_{j=1}^{d_y}\left|\mathbf{y}_i(j) - \hat{\mathbf{y}}_i(j)\right|. \tag{2}$$

- *Hinge Loss*:

$$\ell_{Hinge}(\hat{\mathbf{y}}_i, \mathbf{y}_i) = \sum_{j\neq l_{\mathbf{y}_i}}\max(0, \hat{\mathbf{y}}_i(j) - \hat{\mathbf{y}}_i(l_{\mathbf{y}_i}) + 1), \tag{3}$$

  where $l_{\mathbf{y}_i}$ denotes the true class label index for the instance according to vector $\mathbf{y}_i$.

- *Cross Entropy Loss*:

$$\ell_{CE}(\hat{\mathbf{y}}_i, \mathbf{y}_i) = -\sum_{j=1}^{d_y}\mathbf{y}_i(j)\log\hat{\mathbf{y}}_i(j). \tag{4}$$

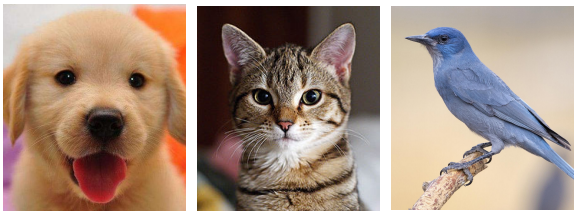
Depending on the number of classes involved as well as the activation functions being used for the model, the *cross entropy* function can be further specified into the *sigmoid cross entropy* (for the binary classification tasks with *sigmiod* function as the activation function for the output layer) and *softmax cross entropy* (for the multi-class classification tasks with *softmax* function as the activation function for the output layer).

The *mean square error* and *mean absolute error* are usually used for the regression tasks, while the *hinge loss* and *cross entropy* are usually used for the classification tasks instead. Besides these loss functions introduced above, many other loss functions can be used for the specific learning problems as well, e.g., *huber loss*, *cosine distance*, and the readers may refer to the referred articles for more information.

To provide more information about the *hinge loss* and the *cross entropy loss* functions, we will use an example to illustrate their usages in real-world problems.

Example 1 *As show in Figure 1, based on the three input images (i.e., the data instances), we can achieve their prediction labels by a deep learning model as shown in the bottom table. For the dog input image (i.e., instance 1), we know its true label should be* $\mathbf{y}_1 = [1,0,0]^\top$ *and the prediction label is* $\hat{\mathbf{y}}_1 = [0.49, 0.43, 0.08]^\top$*, where these three entries in the vector correspond to the dog, cat and bird class labels respectively. Similarly, we can represent the true labels and prediction labels of the cat image and bird image (i.e., instance 2 and instance 3) as vectors* $\mathbf{y}_2 = [0,1,0]^\top$*,* $\hat{\mathbf{y}}_2 = [0.45, 0.53, 0.02]^\top$*,* $\mathbf{y}_3 = [0,0,1]^\top$*, and* $\hat{\mathbf{y}}_3 = [0.21, 0.15, 0.64]^\top$ *respectively.*

Based on the true labels and the prediction labels, we can represent the introduced loss terms by the deep learning model as follows



Class	Instance 1	Instance 2	Instance 3
Dog	0.49	0.45	0.21
Cat	0.43	0.53	0.15
Bird	0.08	0.02	0.64

Figure 1: An Example to Illustrate the Hinge Loss and Cross Entropy Loss Functions.

- *Instance 1: For instance* 1*, we know its true label is dog (i.e., entry* 1 *in the label vector), and its true label index is* $l_{\mathbf{y}_1} = 1$*. Therefore, we can represent the introduced hinge loss for instance* 1 *as follows:*

$$\begin{aligned} \ell_{Hinge}(\hat{\mathbf{y}}_1, \mathbf{y}_1) &= \sum_{j \neq 1} \max(0, \hat{\mathbf{y}}_1(j) - \hat{\mathbf{y}}_1(1) + 1) \\ &= \max(0, \hat{\mathbf{y}}_1(2) - \hat{\mathbf{y}}_1(1) + 1) + \max(0, \hat{\mathbf{y}}_1(3) - \hat{\mathbf{y}}_1(1) + 1) \\ &= \max(0, 0.43 - 0.49 + 1) + \max(0, 0.08 - 0.49 + 1) \\ &= 1.53. \end{aligned} \tag{5}$$

- *Instance 2: For instance* 2*, we know its true label is cat (i.e., entry* 2 *in the label vector), and its true label index will be* $l_{\mathbf{y}_2} = 2$*. Therefore, the introduced hinge loss for instance* 2 *can be represented as*

$$\begin{aligned} \ell_{Hinge}(\hat{\mathbf{y}}_2, \mathbf{y}_2) &= \sum_{j \neq 2} \max(0, \hat{\mathbf{y}}_2(j) - \hat{\mathbf{y}}_2(2) + 1) \\ &= \max(0, \hat{\mathbf{y}}_2(1) - \hat{\mathbf{y}}_2(2) + 1) + \max(0, \hat{\mathbf{y}}_2(3) - \hat{\mathbf{y}}_2(2) + 1) \\ &= \max(0, 0.45 - 0.53 + 1) + \max(0, 0.02 - 0.53 + 1) \\ &= 1.41. \end{aligned} \tag{6}$$

- *Instance 3: Similarly, for the input instance* 3*, we know its true label index is* $l_{\mathbf{y}_3} = 3$*, and the introduced hinge loss for the instance can be denoted as*

$$\begin{aligned}\ell_{Hinge}(\hat{\mathbf{y}}_3, \mathbf{y}_3) &= \sum_{j \neq 3} \max(0, \hat{\mathbf{y}}_3(j) - \hat{\mathbf{y}}_3(3) + 1) \\ &= \max(0, \hat{\mathbf{y}}_3(1) - \hat{\mathbf{y}}_3(3) + 1) + \max(0, \hat{\mathbf{y}}_2(2) - \hat{\mathbf{y}}_3(3) + 1) \\ &= \max(0, 0.21 - 0.64 + 1) + \max(0, 0.15 - 0.64 + 1) \\ &= 1.08.\end{aligned} \tag{7}$$

Meanwhile, based on the true label and the prediction label vectors of these 3 *input data instances, we can represent their introduced cross entropy loss as follows:*

- *Instance 1:*

$$\begin{aligned}\ell_{CE}(\hat{\mathbf{y}}_1, \mathbf{y}_1) &= -\sum_{j=1}^{d_y} \mathbf{y}_1(j) \log \hat{\mathbf{y}}_1(j) \\ &= -1 \times \log 0.49 - 0 \times \log 0.43 - 0 \times \log 0.08 \\ &= 0.713.\end{aligned} \tag{8}$$

- *Instance 2:*

$$\begin{aligned}\ell_{CE}(\hat{\mathbf{y}}_2, \mathbf{y}_2) &= -\sum_{j=1}^{d_y} \mathbf{y}_2(j) \log \hat{\mathbf{y}}_2(j) \\ &= -0 \times \log 0.45 - 1 \times \log 0.53 - 0 \times \log 0.02 \\ &= 0.635.\end{aligned} \tag{9}$$

- *Instance 3:*

$$\begin{aligned}\ell_{CE}(\hat{\mathbf{y}}_3, \mathbf{y}_3) &= -\sum_{j=1}^{d_y} \mathbf{y}_3(j) \log \hat{\mathbf{y}}_3(j) \\ &= -0 \times \log 0.21 - 0 \times \log 0.15 - 1 \times \log 0.64 \\ &= 0.446.\end{aligned} \tag{10}$$

Furthermore, based on the introduced loss for all the data instances in the training set, we can represent the total introduced loss term by the deep learning model as follows

$$\mathcal{L}(\boldsymbol{\theta}; \mathcal{T}) = \sum_{(\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{T}} \ell(\hat{\mathbf{y}}_i, \mathbf{y}_i) = \sum_{(\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{T}} \ell(F(\mathbf{x}_i; \boldsymbol{\theta}), \mathbf{y}_i). \tag{11}$$

By fitting the training data, the training of the deep learning model aims at minimizing the loss introduced on the training set so as to achieve the optimal variables, i.e., the optimal $\boldsymbol{\theta}$. Formally, the objective function of deep learning model training can be represented as follows:

$$\min_{\boldsymbol{\theta} \in \Theta} \mathcal{L}(\boldsymbol{\theta}; \mathcal{T}), \tag{12}$$

where Θ denotes the variable domain and we can have $\Theta = \mathbb{R}^{d_\theta}$ (d_θ denotes the dimension of variable vector $\boldsymbol{\theta}$) if there is no other constraints on the variables to be learned. To address the objective function, in the following sections, we will introduce a group of gradient descent based optimization algorithms, which can learn the globally optimal or locally optimal variables for the deep learning models in the case if the loss function is convex or non-convex respectively. A more comprehensive experimental analysis about their performance will be provided at the end of this paper with some hands-on experiments on some real-world datasets.

To provide a roadmap for the readers, we list the algorithms introduced in the following sections as follows and the readers may pick the algorithms to read according to their own interests.

- Section 2: Conventional Gradient Decent based Learning Algorithms
  - Section 2.1: Vanilla Gradient Descent
  - Section 2.2: Stochastic Gradient Descent
  - Section 2.3: Mini-batch Gradient Descent
- Section 3: Momentum based Learning Algorithms
  - Section 3.1: Momentum
  - Section 3.2: Nesterov Accelerated Gradient
- Section 4: Adaptive Gradient based Learning Algorithms
  - Section 4.1: Adagrad
  - Section 4.2: RMSprop
  - Section 4.3: Adadelta
- Section 5: Momentum & Adaptive Gradient based Learning Algorithms
  - Section 5.1: Adam
  - Section 5.2: Nadam
- Section 6: Hybrid Evolutionary Gradient Descent Learning Algorithms
  - Section 6.1: Gadam

2. Conventional Gradient Descent based Learning Algorithms

In this section, we will introduce the conventional *vanilla gradient descent*, the *stochastic gradient descent* and the *mini-batch gradient descent* algorithms. The main differences among them lie in the amount of training data used to update the model variables in each iteration. They will also serve as the base algorithms for the variant algorithms to be introduced in the following sections.

2.1 Vanilla Gradient Descent

Vanilla gradient descent is also well-known as the the *batch gradient descent*. Given the training set $\mathcal{T}$ as introduced in the previous section, the *vanilla gradient descent* optimization algorithm optimizes the model variables iteratively with the following equation:

$$\boldsymbol{\theta}^{(\tau)} = \boldsymbol{\theta}^{(\tau-1)} - \eta \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^{(\tau-1)}; \mathcal{T}), \tag{13}$$

where $\tau \geq 1$ denotes the updating iteration.

In the above updating equation, the physical meanings of notations $\boldsymbol{\theta}^{(\tau)}$, $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^{(\tau-1)}; \mathcal{T})$ and η are illustrated as follows:

1. Term $\boldsymbol{\theta}^{(\tau)}$ denotes the model variable vector updated via iteration τ. At $\tau = 0$, vector $\boldsymbol{\theta}^{(0)}$ denotes the initial model variable vector, which is usually randomly initialized subject to certain distributions, e.g., uniform distribution $\mathcal{U}(a, b)$ or normal distribution $\mathcal{N}(\mu, \sigma^2)$. To achieve better performance, the hyper-parameters a, b in the uniform distribution, or μ, σ in the normal distribution can be fine-tuned.

2. Term $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^{(\tau-1)}; \mathcal{T})$ denotes the derivative of the loss function $\mathcal{L}(\boldsymbol{\theta}; \mathcal{T})$ regarding the variable $\boldsymbol{\theta}$ based on the complete training set $\mathcal{T}$ and the variable vector value $\boldsymbol{\theta}^{(\tau-1)}$ achieved in iteration $\tau - 1$.

3. Term η denotes the *learning rate* in the gradient descent algorithm, which is usually a hyper-parameter with a small value (e.g., 10^{-3}). Fine-tuning of the parameter is necessary to achieve a fast convergence in practical applications of the *vanilla gradient descent* algorithm.

Such an iterative updating process continues until convergence, and the variable vector $\boldsymbol{\theta}$ achieved at convergence will be outputted as the (globally or locally) optimal variable for the deep learning models. The pseudo-code of the *vanilla gradient descent* algorithm is available in Algorithm 1. In Algorithm 1, the model variables are initialized with the normal distribution. The normal distribution standard deviation σ is the input parameters, and the mean μ is set as 0. As to the the convergence condition, it can be (1) the loss term changes in two sequential iterations is less than a pre-specified threshold, (2) the model variable θ changes is within a pre-specified range, or (3) the pre-specified iteration round number has been reached.

Algorithm 1 Vanilla Gradient Descent

Require: Training Set: $\mathcal{T}$; Learning Rate η; Normal Distribution Std: σ.
Ensure: Model Parameter $\boldsymbol{\theta}$

```
Initialize parameter with Normal distribution θ ~ N(0, σ^2)
Initialize convergence tag = False
while tag == False do
    Compute gradient ∇_θ L(θ; T) on the training set T
    Update variable θ = θ − η · ∇_θ L(θ; T)
    if convergence condition holds then
        tag = True
    end if
end while
Return model variable θ
```

According to the descriptions, in the *vanilla gradient descent* algorithm, to update the model variables, we need to compute the gradient of the loss function regarding the variable, i.e., $\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}^{(\tau-1)};\mathcal{T})$, in each iteration, which is usually very time consuming for a large training set. To improve the learning efficiency, we will introduce the *stochastic gradient descent* and *mini-batch gradient descent* learning algorithms in the following two subsections.

2.2 Stochastic Gradient Descent

Vanilla gradient descent computes loss function gradient with the complete training set, which will be very inefficient if the training set contains lots of data instances. Instead of using the complete training set, the *stochastic gradient descent* (SGD) algorithm updates the model variables by computing the loss function gradient instances by instances. Therefore, *stochastic gradient descent* can be much faster than *vanilla gradient descent*, but it may also cause heavy fluctuations of the loss function in the updating process.

Formally, given the training set $\mathcal{T}$ and the initialized model variable vector $\boldsymbol{\theta}^{(0)}$, for each instance $(\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{T}$, *stochastic gradient descent* algorithm will update the model variable with the following equation iteratively:

$$\boldsymbol{\theta}^{(\tau)} = \boldsymbol{\theta}^{(\tau-1)} - \eta \cdot \nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}^{(\tau-1)}; (\mathbf{x}_i, \mathbf{y}_i)), \tag{14}$$

where the loss term $\mathcal{L}(\boldsymbol{\theta}; (\mathbf{x}_i, \mathbf{y}_i)) = \ell(F(\mathbf{x}_i; \boldsymbol{\theta}), \mathbf{y}_i)$ denotes the loss introduced by the model on data instance $(\mathbf{x}_i, \mathbf{y}_i)$. The pseudo-code of the *stochastic gradient descent* learning algorithm is available in Algorithm 2

Algorithm 2 Stochastic Gradient Descent

Require: Training Set $\mathcal{T}$; Learning Rate η; Normal Distribution Std: σ.
Ensure: Model Parameter $\boldsymbol{\theta}$

```
Initialize parameter with Normal distribution θ ~ N(0, σ²)
Initialize convergence tag = False
while tag == False do
    Shuffle the training set T
    for each data instance (x_i, y_i) ∈ T do
        Compute gradient ∇_θ L(θ; (x_i, y_i)) on the training instance (x_i, y_i)
        Update variable θ = θ − η · ∇_θ L(θ; (x_i, y_i))
    end for
    if convergence condition holds then
        tag = True
    end if
end while
Return model variable θ
```

According to Algorithm 2, the whole training set will be shuffled before the updating process. Even though the learning process of *stochastic gradient descent* may fluctuate a lot, which actually also provides *stochastic gradient descent* with the ability to jump out of local optimum. Meanwhile, by selecting a small learning rate η and decreasing it in the learning process, *stochastic gradient descent* can almost certainly converge to the local or global optimum.

2.3 Mini-batch Gradient Descent

To balance between the *vanilla gradient descent* and *stochastic gradient descent* learning algorithms, *mini-batch gradient descent* proposes to update the model variables with a mini-batch of training instances instead. Formally, let $\mathcal{B} \subset \mathcal{T}$ denote the mini-batch of training instances sampled from the training set $\mathcal{T}$. We can represent the variable updating equation of the deep learning model with the mini-batch as follows:

$$\boldsymbol{\theta}^{(\tau)} = \boldsymbol{\theta}^{(\tau-1)} - \eta \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^{(\tau-1)}; \mathcal{B}), \tag{15}$$

where $\mathcal{L}(\boldsymbol{\theta}; \mathcal{B})$ denotes the loss term introduced by the model on the mini-batch $\mathcal{B}$. The pseudo-code of the *mini-batch gradient descent* algorithm can be illustrated in Algorithm 3.

Algorithm 3 Mini-batch Gradient Descent

Require: Training Set $\mathcal{T}$; Learning Rate η; Normal Distribution Std σ; Mini-batch Size b.
Ensure: Model Parameter $\boldsymbol{\theta}$

```
Initialize parameter with Normal distribution θ ~ N(0, σ^2)
Initialize convergence tag = False
while tag == False do
    Shuffle the training set T
    for each mini-batch B ⊂ T do
        Compute gradient ∇_θ L(θ; B) on the mini-batch B
        Update variable θ = θ − η · ∇_θ L(θ; B)
    end for
    if convergence condition holds then
        tag = True
    end if
end while
Return model variable θ
```

In the *mini-batch gradient descent* algorithm, the batch size b is provided as a parameter, which can take values, like 64, 128 or 256. In most of the cases, b should not be very large and its specific value depends on the size of the training set a lot. The mini-batches are usually sampled sequentially from the training set $\mathcal{T}$. In other words, the training set $\mathcal{T}$ can be divided into multiple batches of size b, and these batches can be picked one by one for updating the model variables sequentially. In some versions of the *mini-batch gradient descent*, instead of sequential batch selection, they propose to randomly sample the mini-batches from the training set, which is also referred to as the random mini-batch generation process.

Compared with *vanilla gradient descent*, the *mini-batch gradient descent* algorithm is much more efficient especially for the training set of an extremely large size. Meanwhile, compared with the *stochastic gradient descent*, the *mini-batch gradient descent* algorithm greatly reduces the variance in the model variable updating process and can achieve much more stable convergence.

2.4 Performance Analysis of Vanilla GD, SGD and Mini-Batch GD

These three *gradient descent* algorithms introduced here work well for many optimization problems, and can all converge to a promising (local or global) optimum. However, these algorithms also suffer from several problems listed as follows:

- *Learning Rate Selection*: The learning rate η may affect the convergence of the gradient descent algorithms a lot. A large learning rate may diverge the learning process, while a small learning rate renders the convergence too slow. Therefore, selecting a good learning rate will be very important for the gradient descent algorithms.
- *Learning Rate Adjustment*: In most of the cases, a fixed learning rate in the whole updating process cannot work well for the gradient descent algorithms. In the initial stages, the algorithm may need a larger learning rate to reach a good (local or global) optimum fast. However, in the later stages, the algorithm may need to adjust the learning rate with a smaller value to fine-tune the performance instead.
- *Variable Individual Learning Rate*: For different variables, their update may actually require a different learning rates instead in the updating process. Therefore, how to use an individual learning rate for different variables is required and necessary.
- *Saddle Point Avoid*: Formally, the saddle point denotes the points with zero gradient in all the dimensions. However, for some of the dimensions, the saddle point is the local minimum, while for some other dimensions, the point is the local maximum. How to get out from such saddle points is a very challenging problem.

In the following part of this paper, we will introduce several other gradient descent based learning algorithm variants, which are mainly proposed to resolve one or several of the above problems.

3. Momentum based Learning Algorithms

In this section, we will introduce a group of gradient descent variant algorithms, which propose to update the model variables with both the current gradient as well as the historical gradients simultaneously, which are named as the *momentum based gradient descent algorithm*. Here, these algorithms will be talked about by using the mini-batch GD as the base learning algorithm.

3.1 Momentum

To smooth the fluctuation encountered in the learning process for the gradient descent algorithms (e.g., mini-batch gradient descent), Momentum Qian (1999) is proposed to accelerate the updating convergence of the variables. Formally, Momentum updates the variables with the following equation:

$$\boldsymbol{\theta}^{(\tau)} = \boldsymbol{\theta}^{(\tau-1)} - \eta \cdot \Delta\mathbf{v}^{(\tau)} \text{ where } \Delta\mathbf{v}^{(\tau)} = \rho \cdot \Delta\mathbf{v}^{(\tau-1)} + (1-\rho) \cdot \nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}^{(\tau-1)}), \tag{16}$$

In the above equation, $\mathbf{v}^{(\tau)}$ denotes the momentum term introduced for keeping record of the historical gradients till iteration τ and function $\mathcal{L}(\boldsymbol{\theta}) = \mathcal{L}(\boldsymbol{\theta}; \mathcal{B})$ denotes the loss function

on a mini-batch $\mathcal{B}$. Parameter $\rho \in [0, 1]$ denotes the weight of the momentum term. The pseudo-code of the Momentum based gradient descent learning algorithm is available in Algorithm 4.

Algorithm 4 Momentum based Mini-batch Gradient Descent

Require: Training Set $\mathcal{T}$; Learning Rate η; Normal Distribution Std σ; Mini-batch Size b; Momentum Term Weight: ρ.
Ensure: Model Parameter $\boldsymbol{\theta}$

```
Initialize parameter with Normal distribution θ ~ N(0, σ^2)
Initialize Momentum term Δv = 0
Initialize convergence tag = False
while tag == False do
    Shuffle the training set T
    for each mini-batch B ⊂ T do
        Compute gradient ∇_θ L(θ; B) on the mini-batch B
        Update term Δv = ρ · Δv + (1 − ρ) · ∇_θ L(θ; B)
        Update variable θ = θ − η · Δv
    end for
    if convergence condition holds then
        tag = True
    end if
end while
Return model variable θ
```

Based on such a recursive updating equation of vector $\boldsymbol{\theta}^{(\tau)}$, we can actually achieve an equivalent representation of $\boldsymbol{\theta}^{(\tau)}$ merely with the gradient terms of the loss function according to the following Lemma.

Lemma 1 *Vector $\Delta\mathbf{v}^{(\tau)}$ can be formally represented with the following equation:*

$$\Delta\mathbf{v}^{(\tau)} = \sum_{t=0}^{\tau-1} \rho^t \cdot (1-\rho) \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^{(\tau-1-t)}). \tag{17}$$

Proof The Lemma can be proved by induction on τ.

1. Base Case: In the case that $\tau = 1$, according to the recursive representation of $\boldsymbol{\theta}$, we have

$$\begin{aligned} \Delta\mathbf{v}^{(1)} &= \rho \cdot \Delta\mathbf{v}^{(0)} + (1-\rho) \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^{(0)}) \\ &= (1-\rho) \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^{(0)}), \end{aligned} \tag{18}$$

   where $\Delta\mathbf{v}^{(0)}$ is initialized as a zero vector.

2. Assumption: We assume the equation holds for $\tau = k$, i.e.,

$$\Delta\mathbf{v}^{(k)} = \sum_{t=0}^{k-1} \rho^t \cdot (1-\rho) \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^{(k-1-t)}). \tag{19}$$

3. Induction: For $\tau = k+1$, based on Equation (16), we can represent the vector $\Delta \mathbf{v}^{(k+1)}$ to be

$$\begin{aligned}
\Delta \mathbf{v}^{(k+1)} &= \rho \cdot \Delta \mathbf{v}^{(k)} + (1-\rho) \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^{(k)}) \\
&= \rho \cdot \sum_{t=0}^{k-1} \rho^t \cdot (1-\rho) \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^{(k-1-t)}) + (1-\rho) \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^{(k)}) \\
&= \sum_{t=1}^{k} \rho^t \cdot (1-\rho) \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^{(k-t)}) + (1-\rho) \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^{(k)}) \\
&= \sum_{t=0}^{k} \rho^t \cdot (1-\rho) \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^{(k-t)}),
\end{aligned} \tag{20}$$

which can conclude this proof.

■

Considering the parameter $\rho \in [0, 1]$, the gradients of the iterations which are far away from the current iteration will decay exponentially. Compared with the conventional gradient descent algorithm, the Momentum can achieve a faster convergence rate, which can be explained based on the gradient updated in the above equation. For the gradient descent algorithm, the gradient updated in iteration τ can be denoted as $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^{(\tau-1)})$, while the updated gradient for Momentum can be denoted as $\Delta \mathbf{v}^{(\tau)} = (1-\rho) \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^{(\tau-1)}) + \rho \cdot \Delta \mathbf{v}^{(\tau-1)}$. Term $\Delta \mathbf{v}^{(\tau-1)}$ keeps records of the historical gradients. As the gradient descent algorithms updates the variables from the steep zone to the relatively flat zone, the large historical gradient terms stored in $\Delta \mathbf{v}^{(\tau-1)}$ allows will actually accelerate the algorithm to reach the (local or global) optimum.

3.2 Nesterov Accelerated Gradient

For the gradient descent algorithms introduced so far, they all update the variables based on the gradients at both the historical or current points without knowledge about the future points they will reach in the learning process. It makes the learning process to be blind and the learning performance highly unpredictable.

The *Nesterov Accelerated Gradient* (i.e., NAG) Nesterov (1983) method propose to resolve this problem by updating the variables with the gradient at an approximated future point instead. The variable $\boldsymbol{\theta}$ to be achieved at step τ can be denoted as $\boldsymbol{\theta}^{(\tau)}$, and it can be estimated as $\hat{\boldsymbol{\theta}}^{(\tau)} = \boldsymbol{\theta}^{(\tau-1)} - \eta \cdot \rho \cdot \Delta \mathbf{v}^{(\tau-1)}$, where $\Delta \mathbf{v}^{(\tau-1)}$ denotes the momentum term at iteration $\tau - 1$. Formally, assisted with the *look-ahead gradient* and *momentum*, the variable updating equations in NAG can be formally represented as follows:

$$\boldsymbol{\theta}^{(\tau)} = \boldsymbol{\theta}^{(\tau-1)} - \eta \cdot \Delta \mathbf{v}^{(\tau)}, \text{ where } \begin{cases} \hat{\boldsymbol{\theta}}^{(\tau)} & = \boldsymbol{\theta}^{(\tau-1)} - \eta \cdot \rho \cdot \Delta \mathbf{v}^{(\tau-1)}, \\ \Delta \mathbf{v}^{(\tau)} & = \rho \cdot \Delta \mathbf{v}^{(\tau-1)} + (1-\rho) \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L}(\hat{\boldsymbol{\theta}}^{(\tau)}). \end{cases} \tag{21}$$

By computing the gradient one step forward, the NAG algorithm is able to update the model variables much more effectively. The pseudo-code of the NAG learning algorithm is available in Algorithm 5.

Algorithm 5 NAG based Mini-batch Gradient Descent

Require: Training Set $\mathcal{T}$; Learning Rate η; Normal Distribution Std σ; Mini-batch Size b; Momentum Term Weight: ρ.
Ensure: Model Parameter $\boldsymbol{\theta}$

```
Initialize parameter with Normal distribution θ ~ N(0, σ^2)
Initialize Momentum term Δv = 0
Initialize convergence tag = False
while tag == False do
    Shuffle the training set T
    for each mini-batch B ⊂ T do
        Compute θ̂ = θ − η · ρ · Δv
        Compute gradient ∇_θ L(θ̂; B) on the mini-batch B
        Update term Δv = ρ · Δv + (1 − ρ) · ∇_θ L(θ̂; B)
        Update variable θ = θ − η · Δv
    end for
    if convergence condition holds then
        tag = True
    end if
end while
Return model variable θ
```

4. Adaptive Gradient based Learning Algorithms

In this section, we will introduce a group of learning algorithms which updates the variables with adaptive learning rates. The algorithms introduced here include Adagrad, RMSprop and Adadelta respectively.

4.1 Adagrad

For the learning algorithms introduced before, the learning rate in these methods are mostly fixed and identical for all the variables in vector $\boldsymbol{\theta}$. However, in the variable updating process, for different variables, their required learning rates can be different. For the variables reaching the optimum, a smaller learning rate is needed; while for the variables far away from the optimum, we may need to use a relatively larger learning rate so as to reach the optimum faster. To resolve these two problems, in this part, we will introduce one *adaptive gradient method*, namely Adagrad Duchi et al. (2011).

Formally, the learning equation for Adagrad can be represented as follows:

$$\boldsymbol{\theta}^{(\tau)} = \boldsymbol{\theta}^{(\tau-1)} - \frac{\eta}{\sqrt{diag(\mathbf{G}^{(\tau)}) + \epsilon \cdot \mathbf{I}}}\mathbf{g}^{(\tau-1)} \text{ where } \begin{cases} \mathbf{g}^{(\tau-1)} & = \nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}^{(\tau-1)}), \\ \mathbf{G}^{(\tau)} & = \sum_{t=0}^{\tau-1} \mathbf{g}^{(t)}(\mathbf{g}^{(t)})^{\top}. \end{cases} \tag{22}$$

In the above updating equation, operator $diag(\mathbf{G})$ defines a diagonal matrix of the same dimensions as $\mathbf{G}$ but only with the elements on the diagonal of $\mathbf{G}$, and ϵ is a smoothing term to avoid the division of zero values on the diagonal of the matrix. Matrix $\mathbf{G}^{(\tau)}$ keeps records of the computed historical gradients from the beginning until the current iteration. Considering the values on the diagonal of matrix $\mathbf{G}^{(\tau)}$ will be different, the learning rate of variables in vector $\boldsymbol{\theta}$ will be different, e.g., $\eta_i^{(\tau)} = \frac{\eta}{\sqrt{\mathbf{G}^{(\tau)}(i,i)+\epsilon}}$ for variable $\boldsymbol{\theta}(i)$ in iteration

τ. In addition, since the matrix $\mathbf{G}^{(\tau)}$ will be updated in each iteration, the learning rate for the same variable in different iterations will be different as well, which is the reason why the algorithm is called the adaptive learning algorithms. The pseudo-code of Adagrad is provided in Algorithm 6.

Algorithm 6 Adagrad based Mini-batch Gradient Descent

Require: Training Set $\mathcal{T}$; Learning Rate η; Normal Distribution Std σ; Mini-batch Size b.
Ensure: Model Parameter $\boldsymbol{\theta}$

```
Initialize parameter with Normal distribution θ ~ N(0, σ^2)
Initialize Matrix G = 0
Initialize convergence tag = False
while tag == False do
    Shuffle the training set T
    for each mini-batch B ⊂ T do
        Compute gradient vector g = ∇_θ L(θ; B) on the mini-batch B
        Update matrix G = G + g g^⊤
        Update variable θ = θ − η / sqrt(diag(G) + ε·I) g
    end for
    if convergence condition holds then
        tag = True
    end if
end while
Return model variable θ
```

Such a learning mechanism in Adagrad allows both adaptive learning rate in the learning process without the need of manual tuning, as well as different learning rate for different variables. Furthermore, as the iteration continues, the values on the diagonal of matrix $\mathbf{G}$ will be no decreasing, i.e., the learning rates of the variables will keep decreasing in the learning process. It will also create problems for the learning in later iterations, since the variables can no longer be effectively updated with information from the training data. In addition, Adagrad still needs an initial learning rate parameter η to start the learning process, which can also be treated as one of the shortcomings of Adagrad.

4.2 RMSprop

To resolve the problem with monotonically decreasing learning rate in Adagrad (i.e., entries in matrix $\frac{\eta}{\sqrt{diag(\mathbf{G}^{(\tau)})+\epsilon\cdot\mathbf{I}}}$), in this part, we will introduce another learning algorithm, namely RMSprop Tieleman and Hinton (2012). RMSprop properly decays the weight of the historical accumulated gradient in the matrix $\mathbf{G}$ defined in Adagrad, and also allows the adjustment of learning rate in the updating process. Formally, the variable updating equations in RMSprop can be represented with the following equations:

$$\boldsymbol{\theta}^{(\tau)} = \boldsymbol{\theta}^{(\tau-1)} - \frac{\eta}{\sqrt{diag(\mathbf{G}^{(\tau)}) + \epsilon \cdot \mathbf{I}}}\mathbf{g}^{(\tau-1)}, \tag{23}$$

where

$$\begin{cases} \mathbf{g}^{(\tau-1)} & = \nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}^{(\tau-1)}), \\ \mathbf{G}^{(\tau)} & = \rho \cdot \mathbf{G}^{(\tau-1)} + (1-\rho)\cdot \mathbf{g}^{(\tau-1)}(\mathbf{g}^{(\tau-1)})^\top. \end{cases} \tag{24}$$

In the above equation, the denominator term is also usually denoted as $RMS(\mathbf{g}^{(\tau-1)}) = \sqrt{diag(\mathbf{G}^{(\tau)}) + \epsilon \cdot \mathbf{I}}$ (RMS denotes *root mean square* metric on vector $\mathbf{g}^{(\tau-1)}$). Therefore, the updating equation is also usually written as follows:

$$\boldsymbol{\theta}^{(\tau)} = \boldsymbol{\theta}^{(\tau-1)} - \frac{\eta}{RMS(\mathbf{g}^{(\tau-1)})}\mathbf{g}^{(\tau-1)}. \tag{25}$$

In the representation of matrix $\mathbf{G}$, parameter ρ denotes the weight of the historically accumulated gradients, and ρ is usually set as 0.9 according to Geoff Hinton in his Lecture at Coursera. Based on the representation of $\mathbf{G}$, for the gradient computed in t iterations ahead of the current iteration, they will be assigned with a very small weight, i.e., $\rho^{(t)} \cdot (1-\rho)$, which decays exponentially with t. The pseudo-code of RMSprop is provided in Algorithm 7, where the learning rate η is still needed and provided as a parameter in the algorithm.

Algorithm 7 RMSprop based Mini-batch Gradient Descent

Require: Training Set $\mathcal{T}$; Learning Rate η; Normal Distribution Std σ; Mini-batch Size b; Parameter ρ.
Ensure: Model Parameter $\boldsymbol{\theta}$

```
Initialize parameter with Normal distribution θ ~ N(0, σ^2)
Initialize Matrix G = 0
Initialize convergence tag = False
while tag == False do
    Shuffle the training set T
    for each mini-batch B ⊂ T do
        Compute gradient vector g = ∇_θ L(θ; B) on the mini-batch B
        Update matrix G = ρ · G + (1 − ρ) · g g^⊤
        Update variable θ = θ − η / sqrt(diag(G) + ε · I) g
    end for
    if convergence condition holds then
        tag = True
    end if
end while
Return model variable θ
```

4.3 Adadelta

Adadelta Zeiler (2012) and RMSprop are proposed separately by different people almost at the same time, which address the monotonically decreasing learning rate in Adagrad with identical methods. Meanwhile, compared with RMSprop further improves Adagrad by introducing a mechanism to eliminate the learning rate from the updating equations of the variables. Based on the following updating equation we introduce before in RMSprop:

$$\boldsymbol{\theta}^{(\tau)} = \boldsymbol{\theta}^{(\tau-1)} - \frac{\eta}{RMS(\mathbf{g}^{(\tau-1)})}\mathbf{g}^{(\tau-1)}. \tag{26}$$

However, as introduced in Zeiler (2012), the learning algorithms introduced so far fail to consider the units of the learning variables in the updating process. Here, the units effectively indicate the physical meanings of the variables, which can be "km", "s" or "kg". If the parameter $\boldsymbol{\theta}$ has hypothetical units, then the updates in parameter, i.e., $\Delta\boldsymbol{\theta} =$

$\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})$, should have the same units as well. However, for the learning algorithms we have introduced before, e.g., SGD, Momentum, NAD, and Adagrad, such an assumption cannot hold. For instance, for the case of SGD, the units of update term is actually proportional to the inverse of the units of $\boldsymbol{\theta}$:

$$\text{units of } \Delta\boldsymbol{\theta} \propto \text{units of } \mathbf{g} \propto \text{units of } \frac{\partial\mathcal{L}(\cdot)}{\partial\boldsymbol{\theta}} \propto \frac{1}{\text{units of } \boldsymbol{\theta}}. \tag{27}$$

To handle such a problem, Adadelta proposes to look at the second-order methods, e.g., Newton's method that uses Hessian approximation. In Newton's method, the units of variables updated can be denoted as

$$\text{units of } \Delta\boldsymbol{\theta} \propto \text{units of } \mathbf{H}^{-1}\mathbf{g} \propto \text{units of } \frac{\frac{\partial\mathcal{L}(\cdot)}{\partial\boldsymbol{\theta}}}{\frac{\partial^2\mathcal{L}(\cdot)}{\partial\boldsymbol{\theta}^2}} \propto \text{units of } \boldsymbol{\theta}, \tag{28}$$

where $\mathbf{H} = \frac{\partial^2\mathcal{L}(\boldsymbol{\theta})}{\partial\boldsymbol{\theta}^2}$ denotes the Hessian matrix computed with the derivatives of the loss function regarding the variables.

To match the units, Adadelta rearranges the Newton's method updating term as

$$\Delta\boldsymbol{\theta} = -\frac{\frac{\partial\mathcal{L}(\cdot)}{\partial\boldsymbol{\theta}}}{\frac{\partial^2\mathcal{L}(\cdot)}{\partial\boldsymbol{\theta}^2}}, \tag{29}$$

from which we can derive

$$\mathbf{H}^{-1} = -\frac{1}{\frac{\partial^2\mathcal{L}(\boldsymbol{\theta})}{\partial\boldsymbol{\theta}^2}} = -\frac{\Delta\boldsymbol{\theta}}{\frac{\partial\mathcal{L}(\boldsymbol{\theta})}{\partial\boldsymbol{\theta}}}. \tag{30}$$

Therefore, we can rewrite the updating equation for the variables based on Newton's method as follows:

$$\begin{aligned}\boldsymbol{\theta}^{(\tau)} &= \boldsymbol{\theta}^{(\tau-1)} - (\mathbf{H}^{(\tau)})^{-1}\mathbf{g}^{(\tau-1)} \\ &= \boldsymbol{\theta}^{(\tau-1)} - \frac{\Delta\boldsymbol{\theta}^{(\tau)}}{\frac{\partial\mathcal{L}(\boldsymbol{\theta}^{(\tau)})}{\partial\boldsymbol{\theta}}}\mathbf{g}^{(\tau-1)}.\end{aligned} \tag{31}$$

Similar to RMSprop, Adadelta approximates the denominator in the above equation with the RMS of the previous gradients. Meanwhile, the $\Delta\boldsymbol{\theta}^{(\tau)}$ term in the current iteration is unknown yet. Adadelta proposes to approximate the numerator in the above equation, and adopts a similar way to use $RMS(\Delta\boldsymbol{\theta})$ to replace $\Delta\boldsymbol{\theta}$ instead. Formally, the variable updating equation in Adadelta can be formally written as follows:

$$\boldsymbol{\theta}^{(\tau)} = \boldsymbol{\theta}^{(\tau-1)} - \frac{RMS(\Delta\boldsymbol{\theta}^{(\tau-1)})}{RMS(\mathbf{g}^{(\tau-1)})}\mathbf{g}^{(\tau-1)}, \tag{32}$$

where $RMS(\Delta\boldsymbol{\theta}^{(\tau-1)})$ keep records of the $\Delta\boldsymbol{\theta}$ in the prior iterations until the previous iteration $\tau-1$. The pseudo-code of the Adadelta algorithm is provided in Algorithm 8. According to the algorithm description, Adadelta doesn't use any learning rate in the updating equation, which effectively resolves the two weakness of Adagrad introduced before.

Algorithm 8 Adadelta based Mini-batch Gradient Descent

Require: Training Set $\mathcal{T}$; Learning Rate η; Normal Distribution Std σ; Mini-batch Size b; Parameter ρ.
Ensure: Model Parameter $\boldsymbol{\theta}$

Initialize parameter with Normal distribution $\boldsymbol{\theta} \sim N(0, \sigma^2)$
Initialize Matrix $\mathbf{G} = \mathbf{0}$
Initialize Matrix $\boldsymbol{\Theta} = \mathbf{0}$
Initialize convergence $tag = False$
while $tag == False$ **do**
  Shuffle the training set $\mathcal{T}$
  for each mini-batch $\mathcal{B} \subset \mathcal{T}$ **do**
    Compute gradient vector $\mathbf{g} = \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}; \mathcal{B})$ on the mini-batch $\mathcal{B}$
    Update matrix $\mathbf{G} = \rho \cdot \mathbf{G} + (1-\rho) \cdot \mathbf{g}\mathbf{g}^\top$
    Computer updating vector $\Delta\boldsymbol{\theta} = -\frac{\sqrt{diag(\boldsymbol{\Theta}) + \epsilon \cdot \mathbf{I}}}{\sqrt{diag(\mathbf{G}) + \epsilon \cdot \mathbf{I}}} \mathbf{g}$
    Update matrix $\boldsymbol{\Theta} = \rho \cdot \boldsymbol{\Theta} + (1-\rho) \cdot \Delta\boldsymbol{\theta}(\Delta\boldsymbol{\theta})^\top$
    Update variable $\boldsymbol{\theta} = \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end for
  if convergence condition holds **then**
    $tag = True$
  end if
end while
Return model variable $\boldsymbol{\theta}$

5. Momentum & Adaptive Gradient based Learning Algorithms

In this section, we will introduce the learning algorithms which combine the advantages of both the Momentum algorithm and the algorithm with adaptive learning rate, including Adam and Nadam respectively.

5.1 Adam

In recent years, a new learning algorithm, namely *Adaptive Moment Estimation* (Adam) Kingma and Ba (2014), has been introduced, which only computes the first-order gradients with little memory requirement. Similar to RMSprop and Adadelta, Adam keeps records of the past squared first-order gradients; while Adam also keeps records of the past first-order gradients as well, both of which will decay exponentially in the learning process. Formally, we can use vectors $\mathbf{m}^{(\tau)}$ and $\mathbf{v}^{(\tau)}$ to denote the terms storing the first-order gradients and the squared first-order gradients respectively, whose concrete representations are provided as follows:

$$\begin{aligned} \mathbf{m}^{(\tau)} &= \beta_1 \cdot \mathbf{m}^{(\tau-1)} + (1-\beta_1) \cdot \mathbf{g}^{(\tau-1)}, \\ \mathbf{v}^{(\tau)} &= \beta_2 \cdot \mathbf{v}^{(\tau-1)} + (1-\beta_2) \cdot \mathbf{g}^{(\tau-1)} \odot \mathbf{g}^{(\tau-1)}, \end{aligned} \tag{33}$$

where vector $\mathbf{g}^{(\tau-1)} = \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^{(\tau-1)})$ and $\mathbf{g}^{(\tau-1)} \odot \mathbf{g}^{(\tau-1)}$ denote the element-wise product of the vectors. In the above equation, vector $\mathbf{v}^{(\tau)}$ actually stores the identical information as $diag(\mathbf{G}^{(\tau)})$ used in Equation (24). However, instead of storing the whole matrix, Adam tends to use less space by keeping such a vector record.

As introduced in Kingma and Ba (2014), vectors $\mathbf{m}^{(\tau)}$ and $\mathbf{v}^{(\tau)}$ are biased toward zero, especially when β_1 and β_2 are close to 1, since $\mathbf{m}^{(0)}$ and $\mathbf{v}^{(0)}$ are initialized to be the zero vectors respectively. To resolve such a problem, Adam introduces to rescale the terms as follows:

$$\begin{aligned} \hat{\mathbf{m}}^{(\tau)} &= \frac{\mathbf{m}^{(\tau)}}{1-\beta_1^{\tau}}, \\ \hat{\mathbf{v}}^{(\tau)} &= \frac{\mathbf{v}^{(\tau)}}{1-\beta_2^{\tau}}, \end{aligned} \tag{34}$$

where the superscripts τ of terms β_1^{τ} and β_2^{τ} denotes the power instead of the iteration count index (τ) used before.

Based on the rescaled vector $\hat{\mathbf{m}}^{(\tau)}$ and matrix $\hat{\mathbf{v}}^{(\tau)}$, Adam will update the model variable with the following equation:

$$\boldsymbol{\theta}^{(\tau)} = \boldsymbol{\theta}^{(\tau-1)} - \frac{\eta}{\sqrt{\hat{\mathbf{v}}^{(\tau)}} + \epsilon} \odot \hat{\mathbf{m}}^{(\tau)} \tag{35}$$

According to the above descriptions, Adam can be viewed as an integration of the RMSprop algorithm with the Momentum algorithm, which allows both faster convergence and adaptive learning rate simultaneously. Formally, the pseudo-code of the Adam learning algorithm is provided in Algorithm 9, where β_1 and β_2 are inputted as the parameters for the algorithm. According to Kingma and Ba (2014), the parameters in Adam can be initialized as follows: $\epsilon = 10^{-8}$, $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

Algorithm 9 Adam

Require: Training Set $\mathcal{T}$; Learning Rate η; Normal Distribution Std σ; Mini-batch Size b; Decay Parameters β_1, β_2.
Ensure: Model Parameter $\boldsymbol{\theta}$

Initialize parameter with Normal distribution $\boldsymbol{\theta} \sim N(0, \sigma^2)$
Initialize vector $\mathbf{m} = \mathbf{0}$
Initialize vector $\mathbf{v} = \mathbf{0}$
Initialize step $\tau = 0$
Initialize convergence $tag = False$
while $tag == False$ **do**
  Shuffle the training set $\mathcal{T}$
  for each mini-batch $\mathcal{B} \subset \mathcal{T}$ **do**
    Update step $\tau = \tau + 1$
    Compute gradient vector $\mathbf{g} = \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}; \mathcal{B})$ on the mini-batch $\mathcal{B}$
    Update vector $\mathbf{m} = \beta_1 \cdot \mathbf{m} + (1-\beta_1) \cdot \mathbf{g}$
    Update vector $\mathbf{v} = \beta_2 \cdot \mathbf{v} + (1-\beta_2) \cdot \mathbf{g} \odot \mathbf{g}$
    Rescale vector $\hat{\mathbf{m}} = \mathbf{m}/(1-\beta_1^{\tau})$
    Rescale vector $\hat{\mathbf{v}} = \mathbf{v}/(1-\beta_2^{\tau})$
    Update variable $\boldsymbol{\theta} = \boldsymbol{\theta} - \frac{\eta}{\sqrt{\hat{\mathbf{v}}}+\epsilon} \odot \hat{\mathbf{m}}$
  end for
  if convergence condition holds **then**
    $tag = True$
  end if
end while
Return model variable $\boldsymbol{\theta}$

5.2 Nadam

Adam introduced in the previous section can be viewed as an integration of Momentum based learning algorithm with the adaptive gradient based learning algorithm, where the vanilla momentum is adopted. In Dozat, a learning algorithm is introduced to replace the vanilla momentum with the *Nesterov's accelerated gradient* (NAG) instead, and the new learning algorithm is called the *Nesterov-accelerated Adam* (Nadam). Before providing the updating equation of Nadam, we would like to illustrate the updating equation of NAG as follows again (Equation (21)):

$$\boldsymbol{\theta}^{(\tau)} = \boldsymbol{\theta}^{(\tau-1)} - \eta \cdot \Delta \mathbf{v}^{(\tau)}, \text{ where } \begin{cases} \Delta \mathbf{v}^{(\tau)} & = \rho \cdot \Delta \mathbf{v}^{(\tau-1)} + (1-\rho) \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L}(\hat{\boldsymbol{\theta}}^{(\tau)}), \\ \hat{\boldsymbol{\theta}}^{(\tau)} & = \boldsymbol{\theta}^{(\tau-1)} - \eta \cdot \rho \cdot \Delta \mathbf{v}^{(\tau-1)}. \end{cases} \tag{36}$$

According to the above updating equation, the momentum term $\Delta \mathbf{v}^{(\tau-1)}$ is used twice in the process: (1) $\Delta \mathbf{v}^{(\tau-1)}$ is used to compute $\hat{\boldsymbol{\theta}}^{(\tau)}$; and (2) $\Delta \mathbf{v}^{(\tau-1)}$ is used to compute $\Delta \mathbf{v}^{(\tau)}$. Nadam proposes to change the above method with the following updating equations instead:

$$\boldsymbol{\theta}^{(\tau)} = \boldsymbol{\theta}^{(\tau-1)} - \eta \cdot \left(\rho \cdot \Delta \mathbf{v}^{(\tau)} + (1-\rho) \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^{(\tau-1)}) \right), \tag{37}$$

where

$$\Delta \mathbf{v}^{(\tau)} = \rho \cdot \Delta \mathbf{v}^{(\tau-1)} + (1-\rho) \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^{(\tau-1)}). \tag{38}$$

Compared with NAG, the above equation doesn't look-ahead in computing the gradient term; while compared with vanilla momentum, the above equation uses both the momentum term and the gradient term in the current iteration. Term $\rho \cdot \Delta \mathbf{v}^{(\tau)} + (1-\rho) \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^{(\tau-1)})$ used in the updating equation actually is an approximation to $\Delta \mathbf{v}^{(\tau+1)}$ in the next iteration, which achieve the objective of looking ahead in updating the variables.

Meanwhile, according to Equation (35), we can rewrite the updating equation of Adam as follows:

$$\boldsymbol{\theta}^{(\tau)} = \boldsymbol{\theta}^{(\tau-1)} - \frac{\eta}{\sqrt{\hat{\mathbf{v}}^{(\tau)}} + \epsilon} \odot \hat{\mathbf{m}}^{(\tau)}, \tag{39}$$

where

$$\hat{\mathbf{m}}^{(\tau)} = \frac{\mathbf{m}^{(\tau)}}{1-\beta_1^{\tau}}, \text{ and } \mathbf{m}^{(\tau)} = \beta_1 \cdot \mathbf{m}^{(\tau-1)} + (1-\beta_1) \cdot \mathbf{g}^{(\tau)}. \tag{40}$$

By replacing the $\hat{\mathbf{m}}^{(\tau)}$ term into Equation (39), we can rewrite it as follows:

$$\begin{aligned} \boldsymbol{\theta}^{(\tau)} &= \boldsymbol{\theta}^{(\tau-1)} - \frac{\eta}{\sqrt{\hat{\mathbf{v}}^{(\tau)}} + \epsilon} \odot \left(\frac{\beta_1 \cdot \mathbf{m}^{(\tau-1)}}{1-\beta_1^{\tau}} + \frac{(1-\beta_1) \cdot \mathbf{g}^{(\tau)}}{1-\beta_1^{\tau}} \right), \\ &= \boldsymbol{\theta}^{(\tau-1)} - \frac{\eta}{\sqrt{\hat{\mathbf{v}}^{(\tau)}} + \epsilon} \odot \left(\beta_1 \cdot \hat{\mathbf{m}}^{(\tau-1)} + \frac{(1-\beta_1) \cdot \mathbf{g}^{(\tau)}}{1-\beta_1^{\tau}} \right). \end{aligned} \tag{41}$$

Similar to the analysis provided in Equation (37), Nadam proposes to look-ahead by replacing the $\hat{\mathbf{m}}^{(\tau-1)}$ term used in the parentheses with $\hat{\mathbf{m}}^{(\tau)}$ instead, which can bring about the following updating equation:

$$\boldsymbol{\theta}^{(\tau)} = \boldsymbol{\theta}^{(\tau-1)} - \frac{\eta}{\sqrt{\hat{\mathbf{v}}^{(\tau)}} + \epsilon} \odot \left(\beta_1 \cdot \hat{\mathbf{m}}^{(\tau)} + \frac{(1-\beta_1) \cdot \mathbf{g}^{(\tau)}}{1-\beta_1^{\tau}} \right). \tag{42}$$

The pseudo-code of the Nadam algorithm is provided in Algorithm 10, where most of the code are identical to those in Algorithm 9, except the last line in updating the variable $\boldsymbol{\theta}$.

Algorithm 10 Nadam

Require: Training Set $\mathcal{T}$; Learning Rate η; Normal Distribution Std σ; Mini-batch Size b; Decay Parameters β_1, β_2.
Ensure: Model Parameter $\boldsymbol{\theta}$

Initialize parameter with Normal distribution $\boldsymbol{\theta} \sim N(0, \sigma^2)$
Initialize vector $\mathbf{m} = \mathbf{0}$
Initialize vector $\mathbf{v} = \mathbf{0}$
Initialize step $\tau = 0$
Initialize convergence $tag = False$
while $tag == False$ **do**
  Shuffle the training set $\mathcal{T}$
  for each mini-batch $\mathcal{B} \subset \mathcal{T}$ **do**
    Update step $\tau = \tau + 1$
    Compute gradient vector $\mathbf{g} = \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}; \mathcal{B})$ on the mini-batch $\mathcal{B}$
    Update vector $\mathbf{m} = \beta_1 \cdot \mathbf{m} + (1 - \beta_1) \cdot \mathbf{g}$
    Update vector $\mathbf{v} = \beta_2 \cdot \mathbf{v} + (1 - \beta_2) \cdot \mathbf{g} \odot \mathbf{g}$
    Rescale vector $\hat{\mathbf{m}} = \mathbf{m}/(1 - \beta_1^\tau)$
    Rescale vector $\hat{\mathbf{v}} = \mathbf{v}/(1 - \beta_2^\tau)$
    Update variable $\boldsymbol{\theta} = \boldsymbol{\theta} - \frac{\eta}{\sqrt{\hat{\mathbf{v}}} + \epsilon} \odot \left(\beta_1 \cdot \hat{\mathbf{m}} + \frac{(1-\beta_1) \cdot \mathbf{g}}{1 - \beta_1^\tau} \right)$
  end for
  if convergence condition holds **then**
    $tag = True$
  end if
end while
Return model variable $\boldsymbol{\theta}$

6. Hybrid Learning Algorithms

Adam together with its many variants have been shown to be effective for optimizing a large group of problems. However, for the non-convex objective functions of deep learning models, Adam cannot guarantee to identify the globally optimal solutions, whose iterative updating process may inevitably get stuck in local optima. The performance of Adam is not very robust, which will be greatly degraded for the objective function with non-smooth shape or learning scenarios polluted by noisy data. Furthermore, the distributed computation process of Adam requires heavy synchronization, which may hinder its adoption in large-cluster based distributed computational platforms.

On the other hand, genetic algorithm (GA), a metaheuristic algorithm inspired by the process of natural selection in evolutionary algorithms, has also been widely used for learning the solutions of many optimization problems. In GA, a population of candidate solutions will be initialized and evolved towards better ones. Several attempts have also been made to use GA for training deep neural network models Zhang and Gouza (2018a); Such et al. (2017); David and Greental (2017) instead of the gradient descent based methods. GA has demonstrated its outstanding performance in many learning scenarios, like non-convex

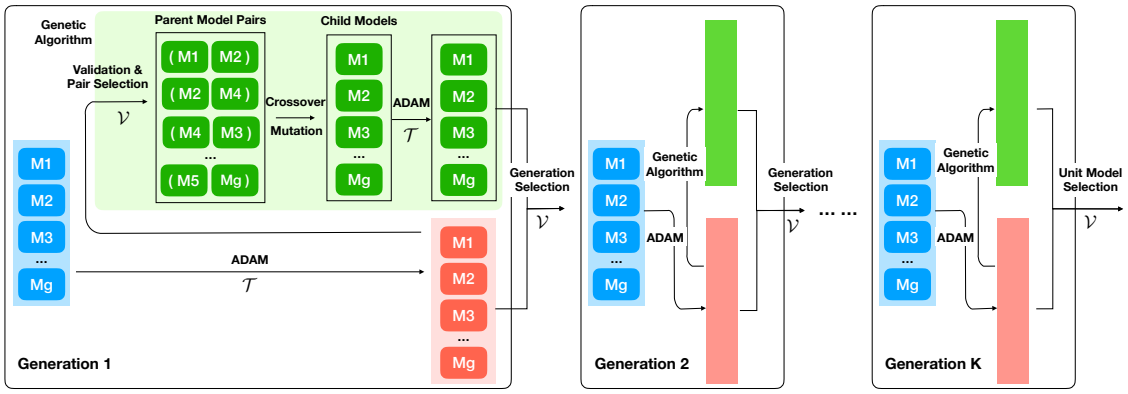


Figure 2: Overall Architecture of Gadam Model.

objective function containing multiple local optima, objective function with non-smooth shape, as well as a large number of parameters and noisy environments. GA also fits the parallel/distributed computing setting very well, whose learning process can be easily deployed on parallel/distributed computing platforms. Meanwhile, compared with Adam, GA may take more rounds to converge in addressing optimization objective functions.

In this section, we will introduce a new optimization algorithm, namely Gadam (Genetic Adaptive Momentum Estimation) Zhang and Gouza (2018b), which incorporates Adam and GA into a unified learning scheme. Instead of learning one single model solution, Gadam works with a group of potential unit model solutions. In the learning process, Gadam learns the unit models with Adam and evolves them to the new generations with genetic algorithm. In addition, Algorithm Gadam can work in both standalone and parallel/distributed modes, which will also be studied in this section.

6.1 Gadam

The overall framework of the model learning process in Gadam is illustrated in Figure 2, which involves multiple learning generations. In each generation, a group of unit model variable will be learned with Adam from the data, which will also get evolved effectively via the genetic algorithm. Good candidate variables will be selected to form the next generation. Such an iterative model learning process continues until convergence, and the optimal unit model variable at the final generation will be selected as the output model solution. For simplicity, we will refer to unit models and their variables interchangeably without distinguishing their differences.

6.1.1 Model Population Initialization

Gadam learns the optimal model variables based on a set of unit models (i.e., variables of these unit models by default), namely the unit model population, where the initial unit model generation can be denoted as set $\mathcal{G}^{(0)} = \{M_1^{(0)}, M_2^{(0)}, \cdots, M_g^{(0)}\}$ (g is the population size and the superscript represents the generation index). Based on the initial generation, Gadam will evolve the unit models to new generations, which can be represented as $\mathcal{G}^{(1)}, \mathcal{G}^{(2)}, \cdots, \mathcal{G}^{(K)}$ respectively. Here, parameter K denotes the total generation number.

For each unit model in the initial generation $\mathcal{G}^{(0)}$, e.g., $M_i^{(0)}$, its variables $\boldsymbol{\theta}_i^{(0)}$ is initialized in Gadam with random values sampled from certain distributions (e.g., the standard normal distribution). These initial generation serve as the starting search points, from which Gadam will expand to other regions to identify the optimal solutions. Meanwhile, for the unit models in the following generations, their variable values will be generated via GA from their parent models respectively.

6.1.2 Model Learning with Adam

In the learning process, given any model generation $\mathcal{G}^{(k)}$ ($k \in \{1, 2, \cdots, K\}$), Gadam will learn the (locally) optimal variables for each unit model with Adam. Formally, Gadam trains the unit models with several epochs. In each epoch, for each unit model $M_i^{(k)} \in \mathcal{G}^{(k)}$, a separated training batch will be randomly sampled from the training dataset, which can be denoted as $\mathcal{B} = \{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \cdots, (\mathbf{x}_b, \mathbf{y}_b)\} \subset \mathcal{T}$ (here, b denotes the batch size and $\mathcal{T}$ represents the complete training set). Let the loss function introduced by unit model $M_i^{(k)}$ for training mini-batch $\mathcal{B}$ be $\ell(\boldsymbol{\theta}_i^{(k)})$, and the learned model variable vector by the training instance can be represented as

$$\bar{\boldsymbol{\theta}}_i^{(k)} = Adam\left(\ell(\boldsymbol{\theta}_i^{(k)})\right). \tag{43}$$

Depending on the specific unit models and application settings, the loss function will have different representations, e.g., mean square loss, hinge loss or cross entropy loss. Such a learning process continues until convergence, and the updated model generation $\mathcal{G}^{(k)}$ with (locally) optimal variables can be represented as $\bar{\mathcal{G}}^{(k)} = \{\bar{M}_1^{(k)}, \bar{M}_2^{(k)}, \cdots, \bar{M}_g^{(k)}\}$, whose corresponding variable vectors can be denoted as $\bar{\boldsymbol{\theta}}_1^{(k)}, \bar{\boldsymbol{\theta}}_2^{(k)}, \cdots, \bar{\boldsymbol{\theta}}_g^{(k)}\}$ respectively.

6.1.3 Model Evolution with Genetic Algorithm

In this part, based on the learned unit models, i.e., $\bar{\mathcal{G}}^{(k)}$, Gadam will further search for better solutions for the unit models effectively via the genetic algorithm.

Model Fitness Evaluation

Unit models with good performance may fit the learning task better. Instead of evolving models randomly, Gadam proposes to pick good unit models from the current generation to evolve. Based on a sampled validation batch $\mathcal{V} \subset \mathcal{T}$, the fitness score of unit models, e.g., $\bar{M}_i^{(k)}$, can be effectively computed based on its loss terms introduced on $\mathcal{V}$ as follows

$$\mathcal{L}_i^{(k)} = \mathcal{L}(\bar{M}_i^{(k)}; \mathcal{V}) = \sum_{(\mathbf{x}_j, \mathbf{y}_j) \in \mathcal{V}} \ell(\mathbf{x}_j, \mathbf{y}_j; \bar{\boldsymbol{\theta}}_i^{(k)}). \tag{44}$$

Based on the computed loss values, the selection probability of unit model $\bar{M}_i^{(k)}$ can be defined with the following softmax equation

$$P(\bar{M}_i^{(k)}) = \frac{\exp^{(-\hat{\mathcal{L}}_i^{(k)})}}{\sum_{j=1}^{g} \exp^{(-\hat{\mathcal{L}}_j^{(k)})}}. \tag{45}$$

Necessary normalization of the loss values is usually required in real-world applications, as $\exp^{(-\mathcal{L}_i^{(k)})}$ may approach 0 or ∞ for extremely large positive or small negative loss values

$\mathcal{L}_i^{(k)}$. As indicated in the probability equation, the normalized loss terms of all unit models can be formally represented as $[\hat{\mathcal{L}}_1^{(k)}, \hat{\mathcal{L}}_2^{(k)}, \cdots, \hat{\mathcal{L}}_g^{(k)}]^\top$, where $\hat{\mathcal{L}}_i^{(k)} \in [0,1], \forall i \in \{1,2,\cdots,g\}$. According to the computed probabilities, from the unit model set $\bar{\mathcal{G}}^{(k)}$, g pairs of unit models will be selected with replacement as the parent models for evolution, which can be denoted as $\mathcal{P} = \{(\bar{M}_{i_1}^{(k)}, \bar{M}_{j_1}^{(k)}), (\bar{M}_{i_2}^{(k)}, \bar{M}_{j_2}^{(k)}), \cdots, (\bar{M}_{i_g}^{(k)}, \bar{M}_{j_g}^{(k)})\}$.

Unit Model Crossover

Given a unit model pair, e.g., $(\bar{M}_{i_p}^{(k)}, \bar{M}_{j_p}^{(k)}) \in \mathcal{P}$, Gadam inherits their variables to the child model via the *crossover* operation. In crossover, the parent models, i.e., $\bar{M}_{i_p}^{(k)}$ and $\bar{M}_{j_p}^{(k)}$, will also compete with each other, where the parent model with better performance tend to have more advantages. We can represent the child model generated from $(\bar{M}_{i_p}^{(k)}, \bar{M}_{j_p}^{(k)})$ as $\tilde{M}_p^{(k)}$. For each entry in the weight variable $\tilde{\boldsymbol{\theta}}_p^{(k)}$ of the child model $\tilde{M}_p^{(k)}$, Gadam initializes its values as follows:

$$\tilde{\boldsymbol{\theta}}_p^{(k)}(m) = \mathbb{1}(\text{rand} \leq p_{i_p,j_p}^{(k)}) \cdot \bar{\boldsymbol{\theta}}_{i_p}^{(k)}(m) + \mathbb{1}(\text{rand} > p_{i_p,j_p}^{(k)}) \cdot \bar{\boldsymbol{\theta}}_{j_p}^{(k)}(m). \tag{46}$$

In the equation, binary function $\mathbb{1}(\cdot)$ returns 1 iff the condition holds. Term "rand" denotes a random number in $[0,1]$. The probability threshold $p_{i_p,j_p}^{(k)}$ is defined based on the parent models' performance:

$$p_{i_p,j_p}^{(k)} = \frac{\exp^{(-\hat{\mathcal{L}}_{i_p}^{(k)})}}{\exp^{(-\hat{\mathcal{L}}_{i_p}^{(k)})} + \exp^{(-\hat{\mathcal{L}}_{j_p}^{(k)})}}. \tag{47}$$

If $\hat{\mathcal{L}}_{i_p}^{(k)} > \hat{\mathcal{L}}_{j_p}^{(k)}$, i.e., model $\bar{M}_{i_p}^{(k)}$ introduces a larger loss than $\bar{M}_{j_p}^{(k)}$, we will have $0 < p_{i_p,j_p}^{(k)} < \frac{1}{2}$.

With such a process, based on the whole parent model pairs in set $\mathcal{P}$, Gadam will be able to generate the children model set as set $\tilde{\mathcal{G}}^{(k)} = \{\tilde{M}_1^{(k)}, \tilde{M}_2^{(k)}, \cdots, \tilde{M}_g^{(k)}\}$.

Unit Model Mutation

To avoid the unit models getting stuck in local optimal points, Gadam adopts an operation called *mutation* to adjust variable values of the generated children models in set $\{\tilde{M}_1^{(k)}, \tilde{M}_2^{(k)}, \cdots, \tilde{M}_g^{(k)}\}$. Formally, for each child model $\tilde{M}_q^{(k)}$ parameterized with vector $\tilde{\boldsymbol{\theta}}_q^{(k)}$, Gadam will mutate the variable vector according to the following equation, where its m_{th} entry can be updated as

$$\tilde{\boldsymbol{\theta}}_q^{(k)}(m) = \mathbb{1}(\text{rand} \leq p_q) \cdot \text{rand}(0,1) + \mathbb{1}(\text{rand} > p) \cdot \tilde{\boldsymbol{\theta}}_q^{(k)}(m), \tag{48}$$

In the equation, term p_q denotes the *mutation rate*, which is strongly correlated with the parent models' performance:

$$p_q = p \cdot \left(1 - P(\bar{M}_{i_q}^{(k)}) - P(\bar{M}_{j_q}^{(k)})\right). \tag{49}$$

For the child models with good parent models, they will have lower mutation rates. Term p denotes the base mutation rate which is usually a small value, e.g., 0.01, and probabilities $P(\bar{M}_{i_q}^{(k)})$ and $P(\bar{M}_{j_q}^{(k)})$ are defined in Equation (45). These unit models will be further trained with Adam until convergence, which will lead to the k_{th} children model generation $\tilde{\mathcal{G}}^{(k)} = \{\tilde{M}_1^{(k)}, \tilde{M}_2^{(k)}, \cdots, \tilde{M}_g^{(k)}\}$.

6.1.4 New Generation Selection and Evolution Stop Criteria

Among these learned unit models in the learned parent model set $\bar{\mathcal{G}}^{(k)} = \{\bar{M}_1^{(k)}, \bar{M}_2^{(k)}, \cdots, \bar{M}_g^{(k)}\}$ and children model set $\tilde{\mathcal{G}}^{(k)} = \{\tilde{M}_1^{(k)}, \tilde{M}_2^{(k)}, \cdots, \tilde{M}_g^{(k)}\}$, Gadam will re-evaluate their fitness scores based on a shared new validation batch. Among all the unit models in $\bar{\mathcal{G}}^{(k)} \cup \tilde{\mathcal{G}}^{(k)}$, the top g unit models will be selected to form the $(k+1)_{th}$ generation, which can be formally represented as set $\mathcal{G}^{(k+1)} = \{M_1^{(k+1)}, M_2^{(k+1)}, \cdots, M_g^{(k+1)}\}$. Such an evolutionary learning process will stop if the maximum generation number has reached or there is no significant improvement between consequential generations, e.g., $\mathcal{G}^{(k)}$ and $\mathcal{G}^{(k+1)}$:

$$\left| \sum_{M_i^{(k)} \in \mathcal{G}^{(k)}} \mathcal{L}_i^{(k)} - \sum_{M_i^{(k+1)} \in \mathcal{G}^{(k+1)}} \mathcal{L}_i^{(k+1)} \right| \leq \lambda, \tag{50}$$

The above equation defines the stop criterion of Gadam, where λ is the evolution stop threshold.

The optimization algorithm Gadam actually incorporate the advantages of both Adam and genetic algorithm. With Adam, the unit models can effectively achieve the (locally/globally) optimal solutions very fast with a few training epochs. Meanwhile, via genetic algorithm based on a number of unit models, it will also provide the opportunity to search for solutions from multiple starting points and jump out from the local optima. According to Kingma and Ba (2014), for a smooth function, Adam will converge as the function gradient vanishes. On the other hand, the genetic algorithm can also converge according to Thierens and Goldberg (1994). Based on these prior knowledge, the convergence of Gadam can be proved as introduced in Zhang and Gouza (2018b). The training of unit models with Gadam can be effectively deployed on parallel/distributed computing platforms, where each unit model involved in Gadam can be learned with a separate process/server. Among the processes/servers, the communication costs are minor, which exist merely in the *crossover* step. Literally, among all the g unit models in each generation, the communication costs among them is $O(k \cdot g \cdot d_\theta)$, where d_θ denotes the dimension of vector $\boldsymbol{\theta}$ and k denotes the required training epochs to achieve convergence by Adam.

7. A Summary

In this paper, we have introduced the gradient descent algorithm together with its various recent variant algorithms for training the deep learning models. Based on the recent developments in this direction, this paper will be updated accordingly later in the near future.

References

Eli David and Iddo Greental. Genetic algorithms for evolving deep neural networks. *CoRR*, abs/1711.07655, 2017. URL http://arxiv.org/abs/1711.07655.

Timothy Dozat. Incorporating Nesterov Momentum into Adam. URL http://cs229.stanford.edu/proj2015/054_report.pdf.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, July 2011. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=1953048.2021068.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL http://arxiv.org/abs/1412.6980.

Yurii Nesterov. A method of solving a convex programming problem with convergence rate O(1/sqr(k)). *Soviet Mathematics Doklady*, 27:372–376, 1983. URL http://www.core.ucl.ac.be/~nesterov/Research/Papers/DAN83.pdf.

Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural Netw.*, 12(1):145–151, January 1999. ISSN 0893-6080. doi: 10.1016/S0893-6080(98)00116-6. URL http://dx.doi.org/10.1016/S0893-6080(98)00116-6.

Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *CoRR*, abs/1712.06567, 2017. URL http://arxiv.org/abs/1712.06567.

Dirk Thierens and David E. Goldberg. Convergence models of genetic algorithm selection schemes. In *Proceedings of the International Conference on Evolutionary Computation. The Third Conference on Parallel Problem Solving from Nature: Parallel Problem Solving from Nature*, PPSN III, pages 119–129, London, UK, UK, 1994. Springer-Verlag. ISBN 3-540-58484-6. URL http://dl.acm.org/citation.cfm?id=645822.670524.

T. Tieleman and G. Hinton. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012. URL http://arxiv.org/abs/1212.5701.

Jiawei Zhang and Fisher B. Gouza. SEGEN: sample-ensemble genetic evolutional network model. *CoRR*, abs/1803.08631, 2018a. URL http://arxiv.org/abs/1803.08631.

Jiawei Zhang and Fisher B. Gouza. GADAM: genetic-evolutionary ADAM for deep neural network optimization. *CoRR*, abs/1805.07500, 2018b. URL http://arxiv.org/abs/1805.07500.