Optimization in Deep Learning

R.C. Luo

Oct 2020

"Optimization is more of an art than a science." This note is a summary of the 8th chapter of the textbook *Deep Learning* [1].

1 Neural Network Optimization

Neural network optimizations are in general more indirect than pure optimizations. In practice, bunches of methods are introduced regarding how to shuffle, pack and feed the training data in order to make them function more like test data.

Deep learning wants to minimize the **generalization error**

$$J^*(\theta) = E_{(x,y)\sim p_{\text{data}}}\left[L(f(x;\theta), y)\right],$$

but can only observe and minimize the **empirical error**

$$J(\theta) = E_{(x,y)\sim \hat{p}_{\text{data}}}\left[L(f(x;\theta), y)\right] = \frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)};\theta), y^{(i)}).$$

2 Challenges

Some critical challenges in optimizations of neural networks:

- **Ill Conditioning.** First-order optimizations chop off the second-order term in Taylor expansion, but for ill conditioned Hessian matrix, this kind of chopping may be problematic ($\frac{1}{2}\epsilon^2 g^T H g > \epsilon g^T g$).
- **Local Minima.** It remains open questions for neural networks of practical interest whether local minima can be the decent approximation of global minima, whether trainings are easily attracted to local minima of high cost, and whether training dynamics can escape poor local minima.
- **Plateaus, Saddle Points.** For large neural networks and higher dimensional parameter space, it's even more common to encounter saddle points in comparison with local minima. Unmodified Newton method can be easily trapped by saddle points, while gradient descent dynamics seems

to be able to rapidly escape from saddle points. Other flat regions of high cost (plateaus) pose major problems for almost all numerical optimization algorithms.

- **Cliffs and Exploding Gradients.** Established techniques: gradient clipping.
- **Long-Term Dependencies.** Established techniques: LSTM, GRU, Attention.
- **Inexact Gradients.** Most modern neural networks are designed to keep robustness under gradient noises. If one really wants to avoid this problem, choosing a surrogate loss function is suggested.
- **Poor Correspondence between Local and Global Structure.** Gradient descent and essentially all optimizers take small, local moves. However, this sometimes may not define a well path to a solution (due to ill conditionings, flat regions, as well as greediness). The most ideal thing may be to initialize the parameters perfectly onto a path to a solution.
- **Theoretical Limits.** Last but not least, the theoretical proofs are very limited in this field and the bounds are too loose and of few practical uses.

3 Algorithms

3.1 Gradient Descent

Gradient descent and its variants are probably the most used optimization algorithms. Algorithm 1 shows how to follow the gradient downhill.

Algorithm 1 Minibatch SGD update at training iteration k

Require: Learning rate ϵ_k.
Require: Initial parameter $\boldsymbol{\theta}$.

while stopping criterion not met **do**
  Sample a minibatch of b examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(b)}\}$ with corresponding targets $y^{(i)}$.
  Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{b} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{g}}$
end while

The **batch size** b is a hyper-parameter. When $b = \#\textit{training samples}$, it is **full-batch gradient descent**. When $b = 1$, it is **stochastic gradient descent**. Conventionally, b is set to some power of 2 between 32 and 256.

Another crucial paramter is the **learning rate** ϵ. The simplest way to set ϵ to some small constant is workable. It's more common to gradually decay the

learning rate, for example

$$\epsilon_k = (1 - \frac{k}{r})\epsilon_0 + \frac{k}{r}\epsilon_\tau.$$

3.2 Momentum

The method of **momentum** is designed to accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients. The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction.

The SGD algorithm with momentum accumulation is shown below.

Algorithm 2 SGD with momentum

Require: Learning rate ϵ, momentum parameter $\alpha(= 0.5, 0.9, 0.99)$.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.

while stopping criterion not met **do**
  Sample a minibatch of b examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(b)}\}$ with corresponding targets $y^{(i)}$.
  Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{b}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$
  Compute velocity update: $\boldsymbol{v} \leftarrow \alpha\boldsymbol{v} - \epsilon\boldsymbol{g}$
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

Nesterov momentum is attempting to add a *correction factor* to the standard method of momentum.

Algorithm 3 SGD with Nesterov momentum

Require: Learning rate ϵ, momentum parameter $\alpha(= 0.5, 0.9, 0.99)$.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.

while stopping criterion not met **do**
  Sample a minibatch of b examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(b)}\}$ with corresponding targets $y^{(i)}$.
  Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha\boldsymbol{v}$
  Compute gradient (at interim point): $\boldsymbol{g} \leftarrow \frac{1}{b}\nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), y^{(i)})$
  Compute velocity update: $\boldsymbol{v} \leftarrow \alpha\boldsymbol{v} - \epsilon\boldsymbol{g}$
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

3.3 Adaptive Gradient

Learning rate is a crucial hyperperameter in neural networks. Some methods automatically adapt the learning rates through the course of learning. This section will give a brief review over related algorithms.

The **AdaGrad** algorithm is designed to decay the learning rate. It has desirable theoretical properties in the context of convex optimizations, but only fairly works for deep learning optimizations.

Algorithm 4 The AdaGrad algorithm

Require: Global Learning rate ϵ.
Require: Initial parameter $\boldsymbol{\theta}$.
Require: Small constant $\delta(= 10^{-7})$, for numerical stability.
Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$
while stopping criterion not met **do**
Sample a minibatch of b examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(b)}\}$ with corresponding targets $y^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{b}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$
Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$
Compute update: $\boldsymbol{\Delta\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wisely)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{\Delta\theta}$
end while

The **RMSProp** algorithm modifies Adagrad to perfrom better in the non-convex setting by changing the gradient accumulation into an exponentially weighted moving average.

Algorithm 5 The RMSProp algorithm

Require: Global Learning rate ϵ, decay rate ρ.
Require: Initial parameter $\boldsymbol{\theta}$.
Require: Small constant $\delta(= 10^{-6})$, for numerical stability.
Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$
while stopping criterion not met **do**
Sample a minibatch of b examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(b)}\}$ with corresponding targets $y^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{b}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$
Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho\boldsymbol{r} + (1-\rho)\boldsymbol{g} \odot \boldsymbol{g}$
Compute update: $\boldsymbol{\Delta\theta} \leftarrow -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wisely)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{\Delta\theta}$
end while

The **Adam** algorithm is another adaptive learning rate optimization algorithm with its name derived from "adaptive moments".

3.4 Second-Order Method

The most widely used second-order method is Newton's method. However, it seems not very appropriate in the context of deep neural network trainings regarding computational cost, saddle points and numerical stability.

Algorithm 6 The Adam algorithm

Require: Step size $\epsilon(= 0.001)$.
Require: Expoential decay rates $\rho_1(= 0.99)$ and $\rho_2(= 0.999)$ in $[0, 1)$.
Require: Small constant $\delta(= 10^{-8})$, for numerical stability.
Require: Initial parameter $\boldsymbol{\theta}$.
Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
Initialize time step $t = 0$
while stopping criterion not met **do**
Sample a minibatch of b examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(b)}\}$ with corresponding targets $y^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{b}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$
$t \leftarrow t + 1$
Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1^t}$
Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2^t}$
Compute update: $\boldsymbol{\Delta\theta} \leftarrow -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}}+\delta}$. (operations applied element-wisely)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{\Delta\theta}$
end while

Conjugate gradient is a method to efficiently avoid the calculation of the inverse Hessian by iteratively descending conjugate directions. The basic idea is to ensure the orthogonality of two sequential search directions $\boldsymbol{d_t}$ and $\boldsymbol{d_{t-1}}$:

$$\boldsymbol{d}_t^{\mathrm{T}} \boldsymbol{H} \boldsymbol{d}_{t-1} = 0.$$

At the training iteration t, the next direction $\boldsymbol{d}_t$ takes the form

$$\boldsymbol{d}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) + \beta_t \boldsymbol{d}_{t-1}.$$

Two popular ways to compute β_t are:

1. **Fletcher-Reeves.**

$$\beta_t = \frac{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)^{\mathrm{T}} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})^{\mathrm{T}} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})}$$

2. **Polak-Ribière.**

$$\beta_t = \frac{(\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t) - \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1}))^{\mathrm{T}} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})^{\mathrm{T}} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})}$$

The conjugate gradient algorithm is given in algorithm 8.

Algorithm 7 Newton's method

Require: Initial parameter $\boldsymbol{\theta}_0$
Require: Training set of b examples

while stopping criterion not met **do**
  Compute gradient $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$
  Compute Hessian: $\boldsymbol{H} \leftarrow \frac{1}{m}\nabla^2_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$
  Compute Hessian (pseudo)inverse: $\boldsymbol{H}^{-1}$
  Compute update: $\boldsymbol{\Delta\theta} \leftarrow -\boldsymbol{H}^{-1}\boldsymbol{g}$
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{\Delta\theta}$
end while

Algorithm 8 Conjugate gradient method

Require: Initial parameters $\boldsymbol{t}heta_0$.
Require: Training set of m examples.
Initialize $\boldsymbol{\rho}_0 = \boldsymbol{0}$
Initialize $g_0 = 0$
Initialize $t = 1$

while stopping criterion not met **do**
  Initialize the gradient $\boldsymbol{g}_t = \boldsymbol{0}$
  Compute gradient $\boldsymbol{g}_t = \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$
  Compute $\beta_t = \frac{(\boldsymbol{g}_t - \boldsymbol{g}_{t-1})^{\mathrm{T}}\boldsymbol{g}_t}{\boldsymbol{g}_{t-1}^{\mathrm{T}}\boldsymbol{g}_{t-1}}$ (Polak-Ribière)
  Compute search direction: $\boldsymbol{\rho}_t = -\boldsymbol{g}_t + \beta_t\boldsymbol{\rho}_{t-1}$
  Perform line search to find: $\epsilon^* = \mathrm{argmin}_\epsilon \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}_t + \epsilon\boldsymbol{\rho}_t), y^{(i)})$
  Apply update: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \epsilon^*\boldsymbol{\rho}_t$
  $t \leftarrow t + 1$
end while

4 Parameter Initialization

Parameter initialization is of critical importance but not yet well understood. The textbook lists several heuristic methods, most of which are based on experiments and are not theoretically proven.

Some heuristics for initialization of weights are listed as follows:

1. **Break symmetry.** Two children units should have different initial parameters.

2. **Larger/smaller weights.** It's a contradiction. Large weights are favored by optimizations, while smaller weights are favored by regulaizations.

3. **Norm-preserving.** Initialize orthogonal weight matrices and set gain factors to scale the activation functions. This may preserve the norms and allow trainings of deeper networks.

4. **Sparse initialization.** Each unit can have exactly k non-zero weights.

5. **Layerwise initialization.** It may be easier to train a deep network layerwisely with some prior features on each layer.

However, these ideas often fail to lead optimal performance because:

1. We may be using the wrong criteria. These kinds of specific designs may not beneficial to whole network.

2. During training process, the initial properties may get lost.

3. The criteria might succeed at improving the speed of optimization but inadvertently increase generalization error.

Some heuristics for initialization of bias are listed as follows:

1. For output layer, it's usual to set the bias vector $\boldsymbol{b}$ to approximate the marginal class distribution $\boldsymbol{c}$:

$$\text{softmax}(\boldsymbol{b}) = \boldsymbol{c}.$$

2. For ReLU units, it's usual to set the bias to some small positive numbers in case of saturation.

3. Sometimes a unit controls whether other units are able to participate in a function. For instance, for gates ($h \in [0, 1]$), it's common to open the gate at first (set $h = 1$).

Besides these simple constant or random methods of initializing model parameters, it is also possible to initialize model parameters by other related, *pretrained* model parameters.

5 Meta-Algorithms

References

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*, volume 1. MIT Press, 2016.