A new optimization algorithm

Dávid Tollner
Dept. of Differential Equations
Technical Univeristy of Budapest
Budapest, Hungary

1. Introduction

Technically the learning of a neural network is a very complex global optimization problem. In training phase we want to minimize the loss function and find parameters which match predictions with reality. To do this is to feed a training data set into the model and adjust the parameters iteratively to make the cost function as small as possible.

There are built-in optimization algorithms in Tensorflow and Keras (SGD, Momentum, RMSprop, Adam), but all of these are based on gradient descent. The choice of optimizer influences both the speed of convergence and whether it occurs. My goal is to implement global optimization algorithm from the mathematical literature. Some well-known datasets will be used for testing, such as MNIST, Boston housing and CFAR10. I want to compare the algorithms and examine the results.

1.1. Gradient descent

First we have to initialize the parameters, which will be the starting position on the error surface. Iteratively, we adjust the parameter values to reduce the value of the cost function. In every iteration we update the parameters based on the opposite direction of the gradient of the cost. This direction will reduce the cost. The updating formula:

$$W^{(i)}(t+1) = W^{(i)}(t) - \alpha \frac{\partial C}{\partial W^{(i)}(t)}$$

where $W^{(i)}(t)$ is the weight vector in layer i and iteration t, α is the learning rate and C is the cost function.
Pro:

- Guaranteed to convergence to global minimum for convex error surfaces and to a local minimum for non-convex surfaces

Contra:

- Very slow convergence
- No online learning

1.2. Stochastic gradient descent

In gradient descent we feedforward all the train data to do a single update, in SGD we can use the subset of the train data to do a single update.

$$W^{(i)}(t+1) = W^{(i)}(t) - \alpha \frac{\partial C}{\partial W^{(i)}(t)}$$

Pro:

- Faster convergence than GD
- Allows online learning

Contra:

- Mini-batch size is a hyperparameter

1.3. Momentum

The learning with SGD can sometimes be slow. The momentum method is designed to accelerate learning. This algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction. The name momentum derives from a physical analogy, in which the negative gradient is a force moving a particle through parameter space, according to Newton's laws of motion. Momentum in physics is mass times velocity. In the momentum learning algorithm, we assume unit mass, so the velocity vector v may also be regarded as the momentum of the particle.

$$\Delta W^{(i)}(t) = -\mu \frac{\partial C}{\partial W^{(i)}(t)} + \alpha \Delta W^{(i)}(t-1)$$

$$W^{(i)}(t+1) = W^{(i)}(t) + \Delta W^{(i)}(t)$$

Pro:

- Speeds up the learning
- Help to come out from a local minimum

Contra:

- More memory usage, than SGD

1.4. AdaGrad

Previous methods: same learning rate for all parameters. AdaGrad adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all the historical squared values of the gradient. The parameters with the largest partial derivative of the loss have a correspondingly rapid decrease in their learning rate, while parameters with small partial derivatives have a relatively small decrease in their learning rate. The net effect is greater progress in the more gently sloped directions of parameter space.

$$\alpha(t) = \alpha(t-1) + \left(\frac{\partial C}{\partial W^{(i)}(t)}\right)^2$$

$$W^{(i)}(t+1) = W^{(i)}(t) - \mu \frac{\frac{\partial C}{\partial W^{(i)}(t)}}{\sqrt{\alpha(t)} + \varepsilon}$$

Pro:

- Well-suited for dealing with sparse data
- Significantly improves robustness of SGD
- Lesser need to manually tune the learning rate

Contra:

- Accumulates squared gradients in denominator. Causes the learning rate to shrink and become infinitesimally small.

1.5. RMSprop

The RMSProp algorithm modifies AdaGrad to perform better in the nonconvex setting by changing the gradient accumulation into an exponentially weighted moving average. RMSProp uses an exponentially decaying average to discard history from the extreme past so that it can converge rapidly after finding a convex bowl.

$$\alpha(t) = \lambda \alpha(t-1) + (1-\lambda)\left(\frac{\partial C}{\partial W^{(i)}(t)}\right)^2$$

$$W^{(i)}(t+1) = W^{(i)}(t) - \mu \frac{\frac{\partial C}{\partial W^{(i)}(t)}}{\sqrt{\alpha(t)} + \varepsilon}$$

Pro:

- The adaprive learning rate usually prevents the learning rate decay from diminishing too slowly or too fast.

Contra:

-

1.6. Adam

Adam also stores running average of past squared gradients like RMSprop. Like Momentum, stores running average of past gradients. In the context of the earlier algorithms, it is perhaps best seen as a variant on the combination of RMSProp and momentum with a few important distinctions. First, in Adam, momentum is incorporated directly as an estimate of the first-order moment (with exponential weighting) of the gradient. The most straightforward way to add momentum to RMSProp is to apply momentum to the rescaled gradients. The use of momentum in combination with rescaling does not have a clear theoretical motivation. Second, Adam includes bias corrections to the estimates of both the first-order moments and the second-order moments to account for their initialization at the origin.

$$m(t) = \beta_1 m(t-1) + (1-\beta_1)\frac{\partial C}{\partial W^{(i)}(t)}$$

$$v(t) = \beta_2 v(t-1) + (1-\beta_2)\left(\frac{\partial C}{\partial W^{(i)}(t)}\right)^2$$

$$W^{(i)}(t+1) = W^{(i)}(t) - \mu \frac{m(t)}{\sqrt{v(t)} + \varepsilon}$$

Pro:

- Momentum + adaptive learning rate

Contra:

- Too much memory usage

2. Other optimization methods

2.1. Newton method

Newton's method is a root-finding algorithm which produces successively better approximations to the roots of a real-valued function.

$$W^{(i)}(t+1) = W^{(i)}(t) - \left[\left(\frac{\partial C}{\partial W^{(i)}(t)}\right)^2\right]^{-1} \frac{\partial C}{\partial W^{(i)}(t)}$$

2.2. Gauss-Newton method

Unlike Newton's method, the Gauss–Newton algorithm can only be used to minimize a sum of squared function values, but it has the advantage that second derivatives, which can be challenging to compute, are not required

$$W^{(i)}(t+1) = W^{(i)}(t) - (J(t)^T J(t))^{-1} J(t)^T e(t)$$

where $J(t)$ is the Jacobian matrix and $e(t)$ is the error vector: $e_i(t) = d_i - y_i(t)$.

Irodalomjegyzék:

Sebastioan Ruder - Optimization for deep learning.
Fazekas István - Neurális hálózatok
Ian Goodfellow, Yoshua Bengio and Aaron Courville - Deep learning