Submitted to the Annals of the Department of Statistics and Operations Research

GRADIENT DESCENT BASED OPTIMIZATION ALGORITHMS

BY ANO NYMOUS

University of Vienna

Gradient descent is an important technique to iteratively optimize a function. It is very popular thanks to the importance of neural networks, but can be used in other contexts as well. The key idea is to calculate the gradient of a function at the current position and update it using some stepsize (called learning rate in machine learning). There are a lot of gradient descent based optimization algorithms, some of them will be explored in this paper.

1. Introduction. To perform gradient descent we calculate the gradient of the function we want to optimize. In this paper we only consider the problem of minimizing a function $f : \mathbb{R}^d \mapsto \mathbb{R}$, parameterized by $\theta \in \mathbb{R}^d$.

$$\min_{\theta} f(\theta) \tag{1.1}$$

To get to the minimum, we start at some random position and update this position by going some step size η in the negative direction of the gradient. This is done because the gradient points to direction of the steepest ascent and the negative gradient to the steepest descent. We iteratively should get closer to a minimum until we converge to it. The step size η is a hyperparameter which needs to be chosen wisely, because if it is too big, the function might not converge, whereas if it is too small the function is more likely to get stuck and takes a long time to converge. Another important role is assigned to the starting point, since some unlucky choice might cause getting stuck at a local minimum. The closer we get to the optimum, the smaller gets the gradient and the smaller are the steps, as we can see in figure 1. Our objective function to iteratively update our position (later on called parameters) is given by:

$$\theta^{(t)} = \theta^{(t-1)} + \eta \nabla f(\theta^{(t-1)}) \tag{1.2}$$

where $\theta^{(t)}$ and $\theta^{(t-1)}$ are some vectors from some vector space $\mathbb{R}^d$, also called parameters later on. The t stands for the iteration of the algorithm. The optimization process doesn't always converge to the optimal value, because we can get stuck in a local optimum as mentioned before. But there are conditions for our function which guarantee convergence to the global minimum, as described by Shalev-Shwartz and Ben-David (2014) in chapter 14 they are:

- convexity
- differentiability
- lipschitz continuity

In theory gradient descent gives you a fast path to the minimum of a function. In practice we don't have all these conditions and we also don't know our function, we

only estimate it with some sampled data. This can help escaping a local optimum, since the function estimated from some samples will be noisy.

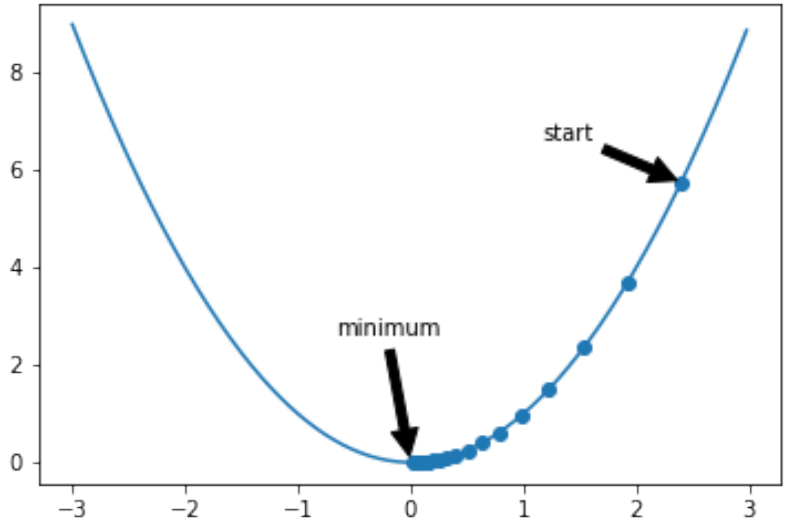

FIG 1. *Example of* $f(x) = x^2$ *converging to it's minimum*

2. Types of gradient descent. There are three variants of gradient descent, which will be discussed shortly in this section. It depends, on how much data we have and how much time we want to spend calculating one update to our position. We choose different variants of gradient descent for different tasks. Since they are heavily used for machine learning tasks, we often find ourselves minimizing some cost function, with some sampled data. This means we try to find the minimum of a function we don't know, which gets modelled by a neural net. Some more context how such cost function (1.1), in a machine learning problem looks like shall be given. In the ml domain, we are given some dataset with N data points and sum over the losses for each sample, f_n is the loss for the n^{th} sample (Deisenroth, Faisal and Ong (2020)). An example for f_n would be the negative log-likelihood.

$$f(\theta) = \sum_{n=1}^{N} f_n(\theta) \tag{2.1}$$

2.1. *Stochastic gradient descent - SGD.* is a variant, which takes one sample from our dataset and updates our parameters (called x in 1.2). SGD is very fast when computing the gradient, because we use only one sample. On the other hand,

as described before, it makes our function very noisy and it might take more time until the function converges (2), but it can prevent getting stuck in some local minimum. Stochastic comes from the fact that we don't know the gradient exactly, we introduce some randomness to the process of finding an optimum.

2.2. *Batch gradient descent.* is often called vanilla gradient descent and uses all of our data to calculate the gradient. It is computationally more expensive than SGD and thus takes more time till convergence. The path we take till convergence is also more stable, than when using SGD.

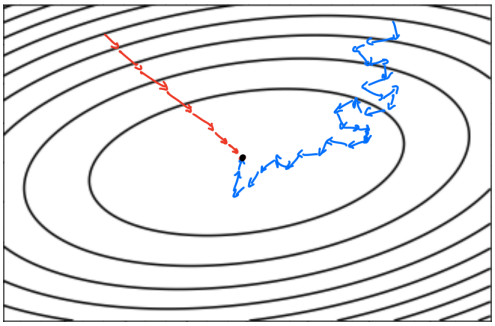

FIG 2. *contourplot of a function where the red path comes from batch gradient descent and blue from SGD*

2.3. *Mini-batch gradient descent.* could be described as a method which is a mix of both variants described before. For mini-batch GD we take a portion of our data (mini-batch) and perform one update to the parameters. The path till convergence wouldn't be as extreme as for SGD, but not as smooth as for the batch variant like in figure 2. It takes the advantages from both SGD and batch gradient descent, but without their disadvantages.

3. Gradient descent based optimization algorithms. In this section we would like to explore some optimization algorithms, which are based on gradient descent and give some intuition on when to use them. There are a lot more than which will be presented, the focus will be on the most popular ones at the time of writing this paper.

3.1. *Momentum based GD.* We can imagine the idea behind this algorithm best if we think of a ball rolling down some hill. The ball starts slowly but quickly picks up momentum and accelerates until it reaches the valley, where it will slow down and eventually stop as described by Géron (2019). Even if due to momentum, the valley will be exceeded, eventually it will be reached after some oscillation. The idea is originally proposed by Boris Polyak in 1964 (Polyak (1964)).
Naturally the closer we get to some minimum, the smaller become the steps, that's why momentum can be quite handy to escape local minima. To get some momentum

in our iterative algorithm, we not only need to consider the current gradient, but also past gradients. Mathematically we make this work by calculating some momentum vector v, which is updated by the gradient each iteration, and this vector then updates our parameters. The formula for momentum based gradient descent is given by:

$$v = \beta v - \eta \nabla f(\theta) \tag{3.1}$$

$$\theta = \theta + v \tag{3.2}$$

In 3.1 β is a hyperparameter called momentum, which controls how much we depend our update v on past gradients. The value of β is between 0 and 1 and a typical default value, for example in a library like tensorflow is 0.9.

3.2. *Adagrad.* When using gradient descent, we go in the direction of the steepest slope, this means we don't directly go in the direction of the optimum. This is what adagrad, which stands for adaptive gradient wants to address. With this method, we want to correct our path, so we end up sooner at the optimum. To make this work as described by Aggarwal (2018), we need to keep track of the aggregated squared magnitude of the partial derivative with respect to each parameter over the course of the algorithm. The square-root of this value is proportional to the root-mean-square slope for that parameter.

$$v_i = v_i + \left(\frac{\partial f(\theta)}{\partial \theta_i}\right)^2 \tag{3.3}$$

$$\theta_i = \theta_i - \frac{\eta}{\sqrt{v_i}} \left(\frac{\partial f(\theta)}{\partial \theta_i}\right) \tag{3.4}$$

In equation 3.3 we keep track of the aggregated squared magnitude as mentioned before and in equation 3.4 we update our parameters like in gradient descent scaled by $\sqrt{v_i}$. Note that in this section all the formulas are element-wise, before they were vector-wise.
As we can see in 3.3 v_i gets bigger and bigger and it's square root scales the learning rate down. This means, if the gradient is steep, the learning rate decays very quickly. This can become a problem for reaching the minimum, since at some point the updates will become so small that they won't change our parameters. In the next methods this problem will be fixed, by using exponential averaging.

3.3. *RProp.* Resilient propagation is an optimization algorithm, which uses the signs of the gradient to compute the updates. It gets mentioned in this paper, because it is necessary to understand why RMSprop was introduced by Geoff Hinton. As described by Riedmiller and Braun (1993) RProp performs a direct adaptation of the weight step based on local gradient information. The adaption is not blurred by gradient behaviour whatsoever. To make this work, we keep track of a separate learning rate η_i for each dimension to update the parameter θ_i. We need to initialize η for the first and second iteration. The adaption rule works as follows: Every

time the the partial derivative of a parameter θ_i changes sign, this indicates that the last update was too big and the algorithm has jumped over a local minimum, the step size η_i gets decreased. In equation 3.6 we can see this behaviour, because $0 < \beta < 1 < \alpha$ and this means that if the partial derivatives have a different sign, we multiply with β which is less than one.

$$\theta_i^{(t)} = \theta_i^{(t-1)} - \eta_i^{(t-1)} * sign\left(\frac{\partial f(\theta^{(t-1)})}{\partial \theta_i^{(t-1)}}\right) \tag{3.5}$$

$$\eta_i^{(t)} = \begin{cases} min(\eta_i^{(t-1)} * \alpha, \eta_{min}), & if \frac{\partial f(\theta^{(t)})}{\partial \theta_i^{(t)}} * \frac{\partial f(\theta^{(t-1)})}{\partial \theta_i^{(t-1)}} > 0 \\ max(\eta_i^{(t-1)} * \beta, \eta_{max}), & if \frac{\partial f(\theta^{(t)})}{\partial \theta_i^{(t)}} * \frac{\partial f(\theta^{(t-1)})}{\partial \theta_i^{(t-1)}} < 0 \\ \eta^{(t-1)}, & otherwise \end{cases} \tag{3.6}$$

To make sure the learning rates won't get too big or too small we use clipping values η_{min} and η_{max} in 3.6. The equations are inspired from this website. It is worth to note, that we used subscript t to indicate the iteration of the algorithm, especially in 3.6 we need this information to determine the partial derivatives in the if condition.
The big advantage of RProp is that the updates get calculated individually for each parameter and, this means if one is very close and another parameter is still far off, this is no problem.

3.4. *RMSprop.* Root Mean Square Propagation was first proposed by Geoff Hinton during the Coursera course "Neural Network for Machine Learning". There was no official paper published, but the slides of the lecture, which introduce the algorithm are available and give a good overview. In lecture 6e of the slides, RMSProp is explained.

$$v_i = \beta v_i + (1 - \beta)\left(\frac{\partial f(\theta)}{\partial \theta_i}\right)^2 \tag{3.7}$$

$$\theta_i = \theta_i - \frac{\eta}{\sqrt{v_i}}\left(\frac{\partial f(\theta)}{\partial \theta_i}\right) \tag{3.8}$$

RMSprop was developed by Hinton, because RProp doesn't work well with mini-batches. For example, when we consider the situation that the partial derivative of one parameter is 0.1 on nine mini-batches and -0.9 on the tenth mini-batch (using a mini-batch size of 10), the value should stay roughly where it is. RProp would increment its value 9 times and only decrease it once by about the same value. This means the parameter grows a lot. RMSprop also addresses the problems of adaptive methods, like adagrad which where described at the end of 3.2.
The problem that varying gradients can cause in RProp get mitigated in RMSprop, because we use a moving average of the squared gradient. The root of this moving average is then used to scale the gradient update accordingly as we can see in equation 3.8. In equation 3.7 we use a parameter β for the moving average which is set to 0.9 in the lecture slides.

3.5. *Adam.* The Adaptive Moment Estimation algorithm is very popular and is another method that computes adaptive learning rates for each parameter. It is basically a combination of RMSprop and momentum. We keep track of an exponentially decaying average of past gradients (3.9) like in momentum and we also keep track of the squared average of past gradients (3.10), which we do in RMSprop. As described by Kingma and Ba (2014), some of the advantages of this algorithm are, that the magnitudes of parameter updates are invariant to rescaling the gradient, the step-sizes are bounded by the stepsize of the hyperparameter and it works for sparse gradients.

$$u = \beta_1 u + (1 - \beta_1)\nabla f(\theta) \tag{3.9}$$

$$v = \beta_2 v + (1 - \beta_2)\left(\nabla f(\theta)\right)^2 \tag{3.10}$$

$$\hat{u} = \frac{u}{1 - \beta_1^t} \tag{3.11}$$

$$\hat{v} = \frac{v}{1 - \beta_2^t} \tag{3.12}$$

$$\theta = \theta - \alpha \frac{\hat{u}}{\sqrt{\hat{v}} + \epsilon} \tag{3.13}$$

In equation 3.10 the square of the gradient is an element-wise operation and in 3.11 and 3.12 β_1 and β_2 are taken to the t'th power, where t is the iteration of the algorithm. A common value for the hyperparameter β_1 is 0.9 and for β_1 0.999. and β_2 The adam optimizer bias corrects u and v, the bias corrected versions are $\hat{u}$ and $\hat{v}$, which are used to update our parameters. We need this bias correction, because at the start of the algorithm u and v will be initialized with zero and therefore biased towards the initialized value. The ϵ in 3.13 is a smoothing term and usually some small value like $10^(-8)$ is chosen. Géron (2019) mentions that adam requires less tuning of η than the algorithms presented before.

4. Visualizing optimization algorithms. The goal for this section, is to compare the discussed optimization algorithms with some plots, which show their paths to the minimum. The implementation was done in python using pyplot from matplotlib for plotting and tensorflow to calculate the gradients of the functions. The code

5. Discussion.

References.

Aggarwal, C. C. (2018). *Neural Networks and Deep Learning.* Springer, Cham.

Deisenroth, M. P., Faisal, A. A. and Ong, C. S. (2020). *Mathematics for Machine Learning.* Cambridge University Press.

Géron, A. (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems.* O'Reilly Media.

Kingma, D. P. and Ba, J. (2014). Adam: A Method for Stochastic Optimization.

Polyak, B. (1964). Some methods of speeding up the convergence of iteration methods. *Ussr Computational Mathematics and Mathematical Physics* **4** 1-17.

Riedmiller, M. and Braun, H. (1993). A direct adaptive method for faster backpropagation learning: the RPROP algorithm. In *IEEE International Conference on Neural Networks* 586–591 vol.1. IEEE.

Shalev-Shwartz, S. and Ben-David, S. (2014). *Understanding Machine Learning - From Theory to Algorithms.* Cambridge University Press.