30

Optimization Techniques

A Preprint

Kushagra Gupta (Mentor)
Department of Mathematics and Statistics
IIT Kanpur
kushgpt@iitk.ac.in

Mayuri Gedam
Department of Mechanical Engineering
IIT Kanpur
mvgedam@iitk.ac.in

Ayush Jain
Department of Electrical Engineering
IIT Kanpur
ayushj@iitk.ac.in

Ankur Banga
Department of Electrical Engineering
IIT Kanpur
ankurb@iitk.ac.in

Nitin Garg
Department of Mathematics and Statistics
IIT Kanpur
nitingrg@iitk.ac.in

Dhruvil Sangani
Department of Mathematics and Statistics
IIT Kanpur
vikrant@iitk.ac.in

Vikrant Malik
Department of Mechanical Engineering
IIT Kanpur
vikrant@iitk.ac.in

August 5, 2020

1 Three Parameter Weibull Distribution

We were supposed to maximize the likelihoofunction for the weibull distribution. The Ideal value of three parameters β, γ, η was estimated using a python script that effectively equated the partial derivative of MLE to 0.

MLE:

$$N\log(\beta) - N\beta\log(\eta) + (\beta-1)\sum_{i=1}^{n}\ln(x) - \sum_{i=1}^{n}(\frac{x}{\eta})^{\beta} \tag{1}$$

If γ approaches minimum value of sample

$$(\beta-1)\log(min(t)-\gamma) \rightarrow \infty \tag{2}$$

$$\frac{\partial f}{\partial \beta} = 0 \qquad \frac{\partial f}{\partial \eta} = 0 \tag{3}$$

$$\eta = [\frac{1}{N}\sum x^{\beta}]^{\frac{1}{\beta}} \tag{4}$$

Complete Documentation and Scripts can be found at :

`https://github.com/Ayushjain9501/convexOptim/tree/master/Assignments/Assignment%201`

2 Batch Gradient Descent

Batch Gradient Descent computes the gradient of the cost function w.r.t. the parameters for the entire training set. The update can be represented as follows:

$$\theta = \theta - \eta * \Delta_\theta J(\theta) \tag{5}$$

This algorithm can be very slow as it iterates over the whole dataset just to perform a single update. It also can not update our model online where new examples are being added. Batch gradient descent is guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces.

It can be visualized as:

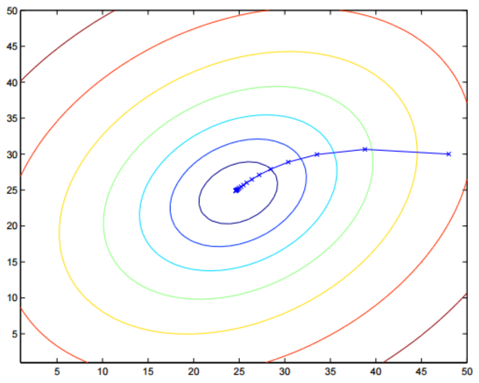

Figure 1: Batch Gradient Descent

Below is an implementation of this algorithm:

`https://github.com/nitingarg1000/Optimization/tree/master/Batch_Gradient_Descent`

3 Stochastic Gradient Descent

Batch gradient descent performs redundant computations for large datasets, as it recomputes gradients for similar examples before each parameter update. SGD does away with this redundancy by performing one update at a time. It is therefore usually much faster and can also be used to learn online.

We try to fit a straight line to some randomly generated dataset using SGD.

Parameter update for a randomly selected training example (x,y):

$$\theta = \theta - \eta * \Delta_\theta J(\theta, x, y) \tag{6}$$

Complete Documentation and Scripts can be found at :

`https://github.com/Ayushjain9501/convexOptim/tree/master/Assignments/stochasticGD`

4 Mini Batch Gradient Descent

Mini-batch gradient descent performs an update for every mini-batch of n training examples. The update can be represented as follows:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}) \tag{7}$$

This reduces the variance of the parameter updates, which can lead to more stable convergence. Common mini-batch sizes range between 50 and 256, but can vary for different applications. Mini-batch gradient descent is typically the algorithm of choice when training a neural network and the term SGD usually is employed also when mini-batches are used. However it does not guarantee good convergence.

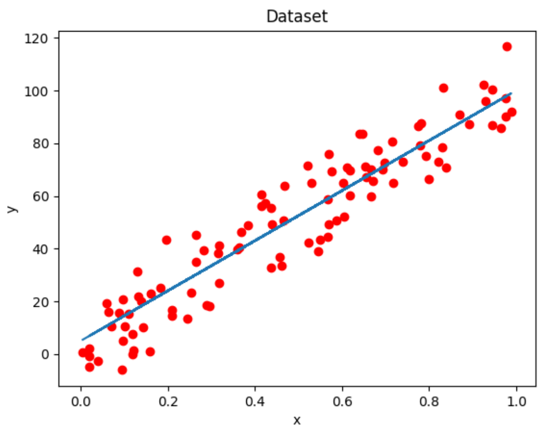

Figure 2: Final Output

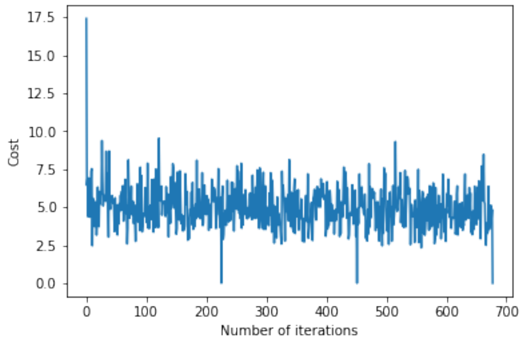

Figure 3: Final Output

Below is an implementation of this algorithm:

`https://github.com/V1krant/Optimisation-Techhniques.git`

5 Matrix Factorization

We find the solution of Bx=b simply by calculating x=B-1b, but sometimes calculating B-1 is cumbersome or when B is nearly singular. So, we use factorization or decomposition methods to break B into easy to use matrices. Some of the factorisation methods are discussed below…

5.1 LU and PLU Factorization for basis B

It's simply the gaussian elimination method that we have learnt already in 12th class and also in mth102. We convert B matrix to upper triangular matrix U by performing a series of operations. We multiply a series of matrix to obtain the same. Name those matrices as Gi and permutations Pi after every operation.

(Gr Pr). . . (G2 P2)(G1 P1)B = U

Multiplying this matrices on either side of the equation Bx=b will give Ux = b' Where b'=(Gr Pr). . . (G2 P2)(G1 P1)b If no permutations are performed, Gr . . . .Gi is lower triangular, and denoting its (lower triangular) inverse as L, we have the factored form B = LU for B. Also, if PT is a permutation matrix that is used to a priori rearrange the rows of B and we then apply the Gaussian triangularization operation to derive L-1PTB = U, we can write B = (PT)-1LU = PLU, noting that PT = P-1. Hence, It's also known as PLU decomposition.

5.2 QR and QRP Factorization for a Basis B

This factorization is most suitable and is used frequently for solving dense equation systems. Here the matrix B is reduced to an upper triangular form R by premultiplying it with a sequence of square, symmetric orthogonal matrices Qi. Bi-1 = Qi-1 ....Q1B The matrix Qi is a square, symmetric orthogonal matrix of the form Qi = 1 – yi qi qiT, where qi = (0, ..., 0, qii ,...,qni)' and yi E RT are suitably chosen to perform the foregoing operation. Defining Q = Qn-1……Q1 , we see that Q is also a symmetric orthogonal matrix and that QB = R, or that B = QR, since Q = QT = Q-1 that is, Q is an involutory matrix. Now, to solve Bx = b, we equivalently solve QRx = b or Rx = Qb by finding b' = Qb first and then solving the upper triangular system Rx = b' via back-substitution. Note that since ||Qv|| = ||v|| for any vector v, we have ||R|| = ||QR|| = ||B||, so that R preserves the relative magnitudes of the elements in B, maintaining stability. This is its principal advantage.

5.3 Cholesky Factorization LLT and LDLT for Symmetric, Positive Definite Matrices B

This method is only applicable for square symmetric matrix B. We will represent this matrix as B=L*LT, where L is a Lower-triangular matrix. So that yuvaL*LT becomes a symmetric matrix and then by comparing the elements of both the symmetric matrices, we could figure out the elements of L. The equation system Bx=b, can now be solved via LT(Lx)=b, through the solution of two triangular systems of equations. We first find y to satisfy Ly = b and then compute x via the system LTx = y. Sometimes we write B = LDLT for the sake of accuracy as it avoids calculating the square root which otherwise was to be calculated in finding the elements of B.

6 Gradient Descent Optimization Algorithms

There are several challenges faced by traditional gradient descent variants like choosing proper learning rates and different learning rates for different parameters. Some of the techniques are as discussed Below.

6.1 Momentum

Around Local Optima, the surface curve much more steeply in one dimension than others. Also SGD oscillates around slopes making very slow progress to the bottom. In such cases we add a fraction of the previous update vector to current update. This acts similar to a ball pushed downhill, accumulating momentum on the way. By using momentum, different dimensions receive different updates. More steep dimensions receive a greater update and hence faster convergence is achieved.

Let v_t be the update vector.

$$v_t = \gamma v_{t-1} + \eta \Delta_\theta J(\theta) \tag{8}$$

$$\theta = \theta - v_t \tag{9}$$

It is an optimization to the traditional SGD and is not found to have any specific bottlenecks.

Code can be found at :

`https://github.com/Ayushjain9501/convexOptim/blob/master/Assignments/Momentum/momentum.py`

6.2 Nesterov accelerated gradient

Continuing on the Momentum optimisation, our parameter speed up based on the previous update, but we aim to slow down at optimal point. We can assume that the next position of our vector is roughly going to be $\theta - \gamma v_{t-1}$.

We now let our momentum drive updates in the direction based on previous updates but also apply a fraction of brake by calculating the the second term at the future position and not the current position. More Clearly,

$$v_t = \gamma v_{t-1} + \eta \Delta_\theta (\theta - \gamma v_{t-1}) \tag{10}$$

$$\theta = \theta - v_t \tag{11}$$

This correction prevents us from going too fast as we reach the optimal point. This leads to increased responsiveness. Both NAG and momentum, are basic variations to look ahead and achieve more natural path to the local optima. Code can be found at :

`https://github.com/Ayushjain9501/convexOptim/blob/master/Assignments/NAG/nag.py`

6.3 Adagrad

Adagrad is an algorithm in which we do not have to worry about the learning rate. It adjusts learning rate according to the frequency of the occuring feature.

Adagrad uses a different learning rate for every θ_i at every time step instead of using the same learning rate for every θ_i.

we will first show adagrad's per-parameter update which we will then vectorize.

In it's update rule adagrad modifies the learning rate according to its past gradients that have been computed for that particular θ.

$$\theta_{t+1,i} \ = \theta_{t,i} - (\eta g_{t,i}) / \sqrt[2]{G_{t,ii} + eps} \tag{12}$$

where $G_{t,ii}$ is the sum of the squares of gradients upto t th time step and eps is set in the range of 1e-4 to 1e-8 to prevent division by zero.

`https://github.com/dhruvilsangani/optimization_techniques_in_ML/blob/master/adagrad.py`

6.4 Adadelta

Adadelta is just an improvement of Adagrad. In Adagrad, due to the accumulation of the squared gradients in the denominator, the learning rate ends up becoming extremely small and no more knowledge can be gained. that seeks to reduce its aggressive, monotonically decreasing learning rate.

Adadelta limits the window of the previously accumulated gradients to a fixed size, say w.

Its update rule is:

$$\Delta\theta_t \ = -g_{t*} RMS[\Delta\theta]_t - 1 / RMS[g]_{t(13)}$$

$$\theta_t + 1 \ = \theta_t - g_{t+} \Delta\theta_{t(14)}$$

6.5 RMSProp

RMSprop have been developed around the time, stemming from the need to resolve Adagrad's radically diminishing learning rates.

Doing same as done in adagrad. we just need to change the function G_t

$$G_{t,ii} = (dr) * (G_{t-1,ii}) + (1 - dr) * (g_{ti})^2 \tag{15}$$

where dr is the decay rate generally set to 0.99

RMSprop divides the learning rate by an exponentially decaying average of squared gradients.

`https://github.com/dhruvilsangani/optimization_techniques_in_ML/blob/master/RMSprop.py`

6.6 Adam

Adaptive Moment Estimation (Adam) is another m

$$m_t = \beta_1 * m_(t-1) + (1-\beta_1) * g_t v_t = \beta_2 * v_(t-1) + (1-\beta_2) * g_t^2 \tag{16}$$

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method. As m_t and v_t are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β_1 and β_2 are close to 1) They counteract these biases by computing bias-corrected first and second moment estimates:

$$by\hat{m}_t = m_t 1 - \beta_1^t \hat{v}_t = v_t 1 - \beta_2^t \tag{17}$$

They then use these to update the parameters just as we have seen in Adadelta and RMSprop, which yields the Adam update rule:

$$\theta_{t+1} = \theta_t - \eta\sqrt{\hat{v}_t} + \eta \tag{18}$$

The Adam works well in practice and compares favorably to other adaptive learning-method algorithms.

`https://github.com/V1krant/Optimisation-Tbyechhniques/blob/master/adam.py`

`https://github.com/mvgedam/Optimisation_Techniques/blob/master/adam.py`

6.7 AdaMax

The v_t factor in the Adam update rule scales the gradient inversely proportionally to the l_2 norm of the past gradients (via the v_{t-1} term) and current gradient $|g_t|^2$:

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)|g_t|^2 \tag{19}$$

We can generalize this update to the l_p norm. β_2 is then parameterized as β_p^2:

$$v_t = \beta_2^p v_{t-1} + (1-\beta_2^p)|g_t|^p \tag{20}$$

Norms for large p values generally become numerically unstable, which is why l_1 and l_2 norms are most common in practice. However, l_∞ also generally exhibits stable behavior. For this reason, we use AdaMax and show that v_t with l_∞ converges to the following more stable value. To avoid confusion with Adam, we use u_t to denote the infinity norm-constrained v_t:

$$u_t = \beta_2^\infty v_{t-1} + (1-\beta_2^\infty)|g_t|^\infty = max(\beta_2.v_{t-1}, |g_t|) \tag{21}$$

We can now plug this into the Adam update equation by replacing $\sqrt{\hat{v}_t} + \epsilon$ with u_t to obtain the AdaMax update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{u_t}\hat{m}_t \tag{22}$$

`https://github.com/nitingarg1000/Optimization/blob/master/adamax.py`

6.8 AMSGrad

As adaptive learning rate methods have become the norm in training neural networks, it was noticed that in some cases, e.g. for object recognition or machine translation they fail to converge to an optimal solution and are outperformed by SGD with momentum.
The exponential moving average of past squared gradients serves as a reason for the poor generalization behaviour of adaptive learning rate methods. It prevents the learning rates from becoming infinitesimally small as training progresses (solved the flaw in Adagrad) but the short term memory of the gradients become an obstacle in other scenarios.
In settings where Adam converges to a suboptimal solution, it has been observed that some minibatches provide large and informative gradients, but as these minibatches only occur rarely, exponential averaging diminishes their influence, which leads to poor convergence.
To fix this behaviour, AMSGrad uses the maximum of past squared gradients rather than the exponential average to update the parameters.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t \tag{23}$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t^2 \tag{24}$$
$$\hat{v}_t = max(\hat{v}_{t-1}, v_t) \tag{25}$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}m_t \tag{26}$$

This way, AMSGrad results in a non-increasing step size, which avoids the problems suffered by Adam. The debiasing step that we have seen in Adam is also removed for simplicity.

`https://github.com/ankurbanga/Optimization-Techniques/blob/master/amsgrad.py`

References

[1] Stephen Boyd and Lieven Vandenberghe. Convex Optimization. Appendix A

[2] MOKHTAR S. BAZARAA, HANIF D. SHERALI, C. M. SHETTY : Non-linear programming theory and algorithms. pages 756-759

[3] `http://ruder.io/optimizing-gradient-descent/index.html#tensorflow`

[4] `https://towardsdatascience.com/types-of-optimization-algorithms-used-in-neural-networks-and-ways-to-`

[5] `https://towardsdatascience.com/demystifying-optimizations-for-machine-learning-c6c6405d3eea`