Gradient Descent Optimizations

Gradient descent

Gradient descent is a way to minimize an objective function $J(\theta)$ parameterized by a model's parameters $\theta \in \mathbb{R}^d$ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)$ with respect to the parameters.

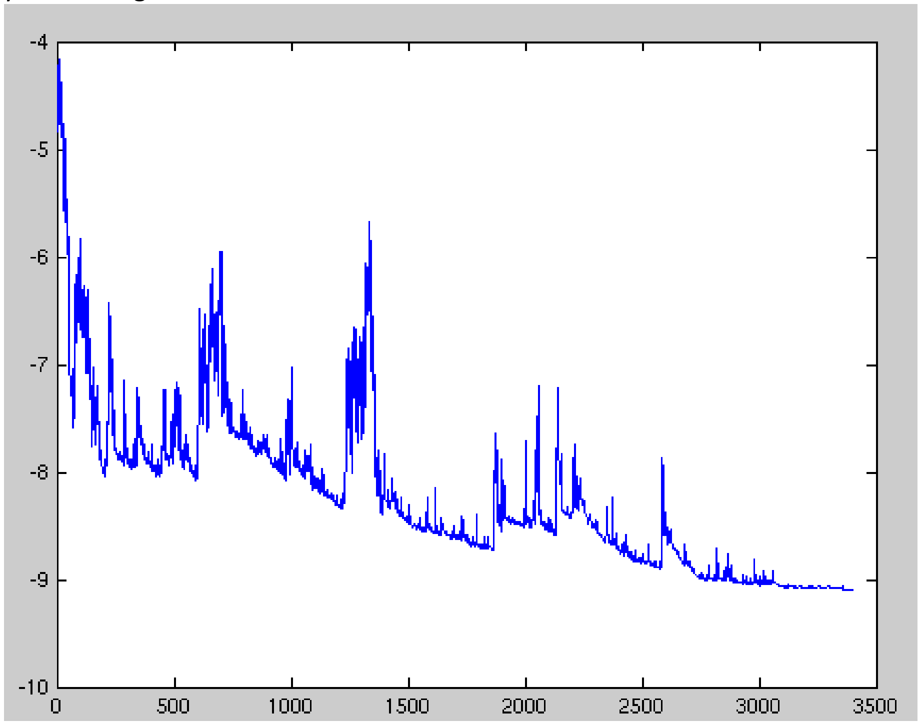
Three gradient descent variants

1. ***Batch gradient descent/Batch gradient descent***
   - ❖ Vanilla gradient descent computes the gradient of the cost function with respect to the parameters θ of the entire training dataset.
   - ❖ *Update rule:*

     $$\theta_{t+1} = \theta_t - \eta \, \nabla_\theta J(\theta_t)$$

     Here θ_{t+1} is the parameter value at $t+1$ timestep
     θ_t is the parameter value at t timestep
     η is the learning rate, a small positive number that determines the step size we take towards a local minima.
   - ❖ Batch gradient descent is very slow and is intractable for large dataset because we need to calculate the gradients for the whole dataset to perform one update.
   - ❖ Batch gradient descent does not allow us to update our model online i.e. with new examples on the fly.
   - ❖ Batch gradient is guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex error surfaces(this is because for convex surfaces if a local minimum exists, then it is a global minimum).
2. ***Stochastic gradient descent(SGD)***
   - ❖ SGD performs a parameter update for each training example x_i and label y_i.
   - ❖ *Update rule:*

     $$\theta_{t+1} = \theta_t - \eta \, \nabla_\theta J(\theta_t; x_i; y_i)$$
   - ❖ SGD is much faster and can be used to train online.
   - ❖ SGD performs frequent update with a high variance that cause the objective function to fluctuate heavily like the figure below



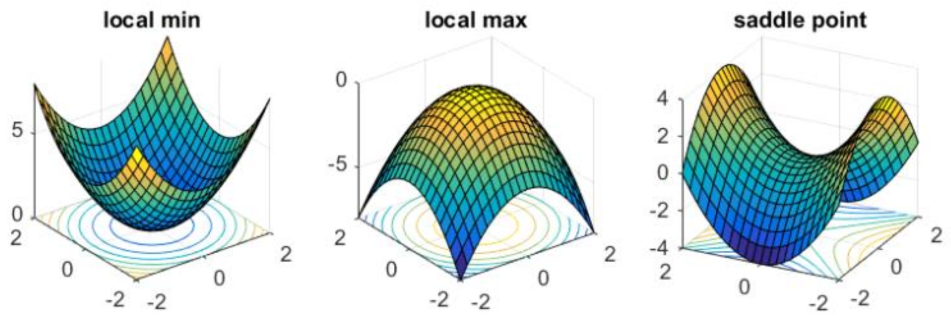
   - ❖ SGD's fluctuation enables it to jump to new and potentially better local minima. On the other hand, this complicates convergence to the exact minimum, as SGD will keep overshooting.

However, it has been shown that when we slowly decrease the learning rate, SGD shows the same convergence behavior as batch gradient descent, almost certainly converging to a local or the global minimum for non-convex and convex optimization respectively.

3. ***Mini-batch gradient descent***
   - ❖ Mini-batch gradient descent preforms an update for every mini-batch of n training examples.
   - ❖ *Update rule:*
$$\theta_{t+1} = \theta_t - \eta \, \nabla_\theta J(\theta_t; x_{i:i+n}; y_{i:i+n})$$
   - ❖ It reduces the variance of parameter updates which can lead to more stable convergence.
   - ❖ It can make use of highly optimized matrix optimizations common to state-of-the-art deep learning libraries that make computing the gradient w.r.t. a mini-batch very efficient.

Challenges with mini-batch vanilla gradient descent

1. Choosing a proper learning rate can be difficult. A learning rate that is too small leads to painfully slow convergence, while a learning rate that is too large can hinder convergence and cause the loss function to fluctuate around the minimum or even to diverge.
2. The same learning rate applies to all parameter updates. If our data is sparse and our features have very different frequencies, we might not want to update all of them to the same extent, but perform a larger update for rarely occurring features.
3. It's very difficult for SGD to escape numerous suboptimal local minima especially the saddle points, i.e. points where one dimension slopes up and another slopes down.



Gradient descent extensions and variants

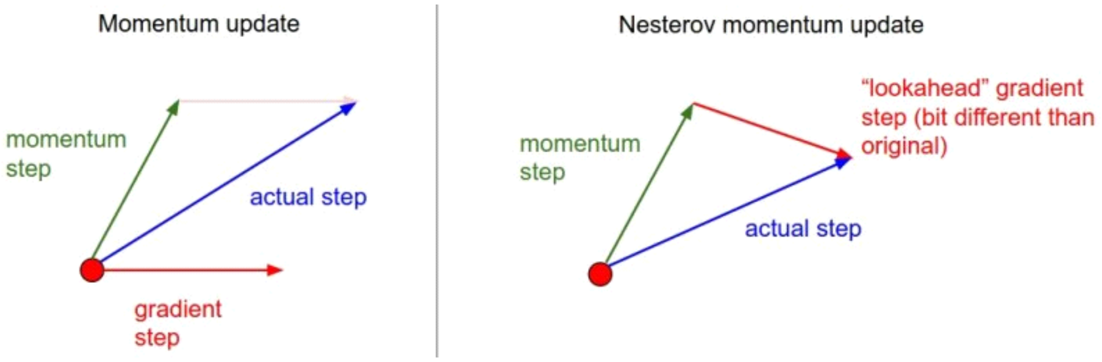
1. ***Momentum***
   - ❖ Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations. Here the current update depend on all past gradients.
   - ❖ *Update rule:*
$$v_{t+1} = \gamma v_t + \eta \nabla_\theta J(\theta_t)$$
$$\theta_{t+1} = \theta_t - v_{t+1}$$
     Here γ is the momentum value which is usually set to 0.9 or similar value.
   - ❖ How the update depend on previous gradients:
$$\begin{aligned} v_{t+1} &= \gamma v_t + \eta \nabla_\theta J(\theta_t) \\ &= \gamma[\gamma v_{t-1} + \eta \nabla_\theta J(\theta_{t-1})] + \eta \nabla_\theta J(\theta_t) \\ &= \gamma^2 v_{t-1} + \gamma\eta \nabla_\theta J(\theta_{t-1}) + \eta \nabla_\theta J(\theta_t) \\ &= \gamma^2[\gamma v_{t-2} + \eta \nabla_\theta J(\theta_{t-2})] + \gamma\eta \nabla_\theta J(\theta_{t-1}) + \eta \nabla_\theta J(\theta_t) \\ &= \gamma^3 v_{t-2} + \gamma^2 \eta \nabla_\theta J(\theta_{t-2}) + \gamma\eta \nabla_\theta J(\theta_{t-1}) + \gamma^0 \eta \nabla_\theta J(\theta_t) \\ &\dots \\ &= \sum_{i=0}^{t} \gamma^i \eta \nabla_\theta J(\theta_{t-i}) \end{aligned}$$
     So the update depends on all of the previous gradients with exponentially decaying contributions from past updates (since $\gamma < 1$)
   - ❖ Essentially, when using momentum, we push a ball down a hill. The ball accumulates momentum

as it rolls downhill, becoming faster and faster on the way (until it reaches its terminal velocity, if there is air resistance, i.e. $\gamma < 1$). The same thing happens to our parameter updates: The momentum
term increases for dimensions whose gradients point in the same directions and reduces updates for
dimensions whose gradients change directions. As a result, we gain faster convergence and reduced
oscillation.

2. ***Nesterov accelerated gradient(NAG)***
   - We know that we will use our momentum term γv_t to move the parameters θ. Computing $\theta_t - \gamma v_t$ thus gives us an approximation of the next position of the parameters. With NAG we compute gradients w.r.t. the approximate future positions of our parameters. This prevents us from going too fast becausew we can make necessary correction if the slope changes direction and ensures better responsiveness.
   - *Update rule:*

$$v_{t+1} = \gamma v_t + \eta \nabla_\theta J(\theta_t - \gamma v_t)$$
$$\theta_{t+1} = \theta_t - v_{t+1}$$

   - *Example*



With momentum SGD we compute momentum term and gradient step and move towards the direction of the resultant vector of these two. Instead of evaluating gradient at the current position (red circle), we know that our momentum is about to carry us to the tip of the green arrow. With Nesterov momentum we therefore instead evaluate the gradient at this "looked-ahead" position. Thus we make necessary adjustment based on the direction of this "look-ahead" gradient.

3. ***Adagrad(Adaptive gradient)***
   - Adagrad adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters.
   - *Update rule:*
     Per parameter update:

$$g_{t+1,i} = \nabla_\theta J(\theta_{t,i})$$
$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t+1,i}$$

Here $g_{t,i}$ is the gradient of the objective function w.r.t. the parameter θ_i, $G_t \in \mathbb{R}^{d \times d}$ is a diagonal matrix where each diagonal element is the sum of the squares of the gradients w.r.t. θ_i up to time step t and ϵ is a smoothing term that avoids division by zero()usually on the order of $1e - 8$.
Vectorized update:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} g_t$$

   - One of Adagrad's main benefits is that it eliminates the need to manually tune the learning rate. Most
     implementations use a default value of 0.01 and leave it at that.
   - Adagrad's main weakness is its accumulation of the squared gradients in the denominator: Since

every added term is positive, the accumulated sum keeps growing during training. This in turn causes
the learning rate to shrink and eventually become infinitesimally small, at which point the algorithm
is no longer able to acquire additional knowledge.

4. ***RMSprop***
   - ❖ The RMSprop update adjusts the Adagrad method in a very simple way in an attempt to reduce its aggressive, monotonically decreasing learning rate. In particular, it uses an exponentially decaying moving average of squared gradients instead.
   - ❖ *Update rule:*

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

   Here $E[g^2]_t$ is the moving average at time step t.
   - ❖ Hinton suggests γ to be set to 0.9, while a good default value for the learning rate η is 0.001.

5. ***Adadelta***
   - ❖ Like RMSprop it uses an exponentially decaying moving average of squared gradients. With Adadelta it is possible to perform update without setting any learning rate at all.
   - ❖ *Update rule:*

$$RMS[g_t] = \sqrt{E[g^2]_t + \epsilon}$$
$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon}$$
$$\Delta\theta_t = \frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$
$$\theta_{t+1} = \theta_t - \Delta\theta_t$$

   Here $E[\Delta\theta^2]$ is exponentially decaying average of swuared parameter updates(not gradients).

6. ***Adam(Adaptive Momentum Estimation***
   - ❖ In addition to storing an exponentially decaying average of past squared gradients v_t like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients m_t, similar to momentum.
   - ❖ *Update rule:*

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$
$$\widehat{m}_t = \frac{m_t}{1-\beta_t^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}+\epsilon}} \widehat{m}_t$$

   Here m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively. 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. β1 and β2 are close to 1). So they counteract these biases by computing bias-corrected first and second moment estimates $\widehat{m}_t$ and $\hat{v}_t$ respectively.
   - ❖ The authors propose default values of 0.9 for β_1,0.999 for β_2, and 10^{-8} for ϵ.
   - ❖ Adam works well in practice and compares favorably to other adaptive learning-method algorithms.

References

1. ***https://arxiv.org/abs/1609.04747***
2. ***https://en.wikipedia.org/wiki/Convex optimization***
3. ***https://en.wikipedia.org/wiki/Gradient descent***

4. ***https://en.wikipedia.org/wiki/Stochastic_gradient_descent***
5. ***https://www.offconvex.org/2016/03/22/saddlepoints/***
6. ***https://wiseodd.github.io/techblog/2016/06/22/nn-optimization/***
7. ***http://cs231n.github.io/neural-networks-3/***