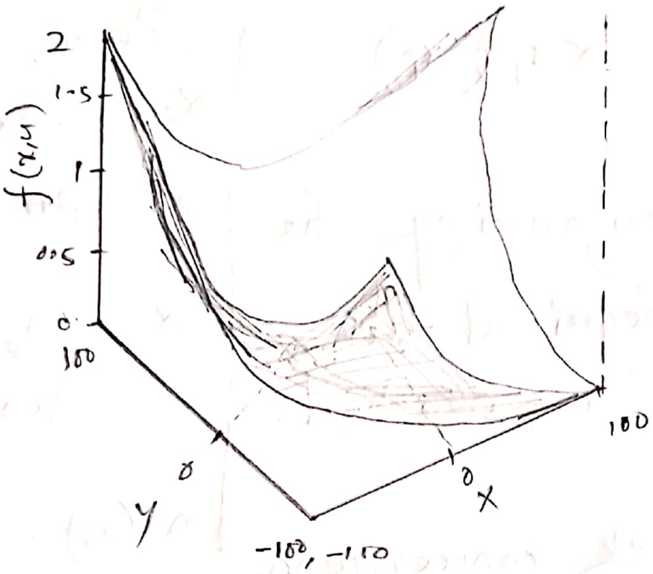


suppose we have a convex cost function of 2 input variables shown above and our goal is to minimize its value and find the value of the parameters (x,y) for which $f(x,y)$ is minimum. What the gradient descent algorithm does is we start at a specific point on the curve and use the negative gradient to find the direction of steepest descent and take a small step in that direction and keep iterating till our values starts converging

Gradient descent is an iterative optimization algorithm for finding the minimum of a function

$$X_{i+1} = x_i - \alpha \Delta f(x_i)$$

Here,

X_{i+1} = next position

x_i = ~~po~~ current position

α = step size or learning rate

$\Delta f(x_i)$ = gradient at current position

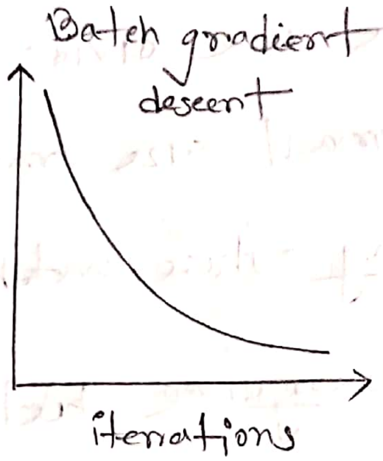
We keep performing the update as required till convergence is reached.

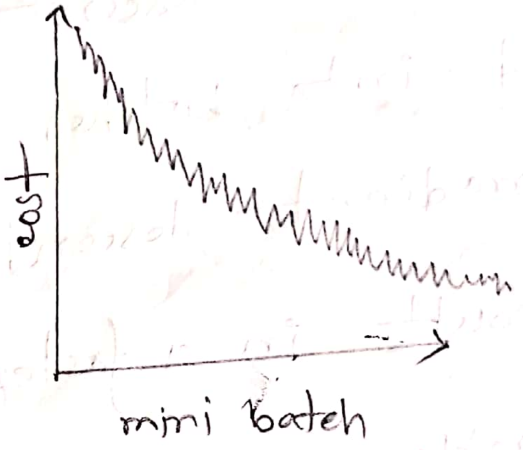
We can check convergence easily by checking whether the difference between $f(x_{i+1})$ and $f(x_i)$ is less than some number, say (0.0001). If so, we say that gradient descent has converged to a local minimum of f.

Where gradient descent fall short

To perform a single step of gradient descent we need to iterate over all training examples this is termed as batch gradient descent. But if we have millions of data then it becomes computationally expensive, so what we done commonly. is something called a mini batch gradient descent where we divide the training set into batches of small size and perform gradient descent using those batches. This often results in a faster convergence but there's major problem here we only look at a fraction of the training set while taking a single step and hence the step may not be towards the steepest decrease of the cost function. This is because we are minimizing the cost based on a subset of total data. Which is not a representative of what's best for the entire training data.

Instead of following a straight path towards the minimum, our algorithm now follows a roundabou path not always even leading to an optimum most commonly. overshooting (going past minimum)





Adam optimization Algorithm

First, let's see the parameters involved

1. α - Learning Rate for gradient descent step.

2. β_1 - Parameter for momentum step. (0.9)

3. β_2 " " RMSprop step. (0.99)

4. ϵ - " " numerical stability (10^{-8})

5. m, v - First and second moment estimates respectly. Initial values of both set to 0.

6. t - The timestep parameter for bias correction step

7. g and f - Gradient and function values of θ.

Adam can essentially be broken down as combination of 2 main algorithms - Momentum and RMSProp.

The momentum step is as follows -

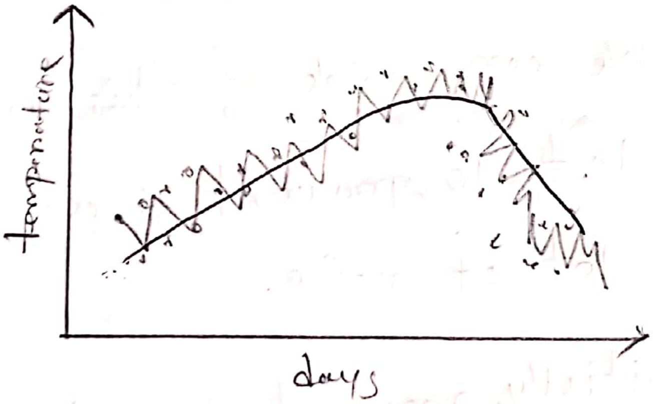
$$m = \beta_1 \times m + (1 - \beta_1) \times g$$

Suppose, $\beta_1 = 0.9$. Then the corresponding step calculates $0.9 *$ current moment $+ 0.1 *$ current gradient. We can think of this is a weighted average over last 10 gradient descent steps, which can cels out a lot of noise.

However initially, moment is set to 0 hence the moment at first step $= 0.9 * 0 + 0.1 *$ gradient $=$ gradient/10 and so on. The moment will fail to keep up with the original gradient and this is known as biased estimate. To correct this we do following, known as bias correction, dividing by 1 - (β_1 raised to the timestep).

m_corrected = m/ (1-np.power(β1,t))

Note that 1- power (β1,t) approaches 1 as t becomes 1 as t becomes higher with each step, decreasing the correction effect later and maximizing it at the first few steps.



The graph alongside picture this perfectly, the pencil line refers to the moment (estimate) obtained with a smaller β1, say 0.5 while the black line refers to a β1 value close to 1, say 0.9

RMSprop does a similar thing, but slightly different

$$V = \beta_2 * V + (1-\beta_2) * np.square(g)$$

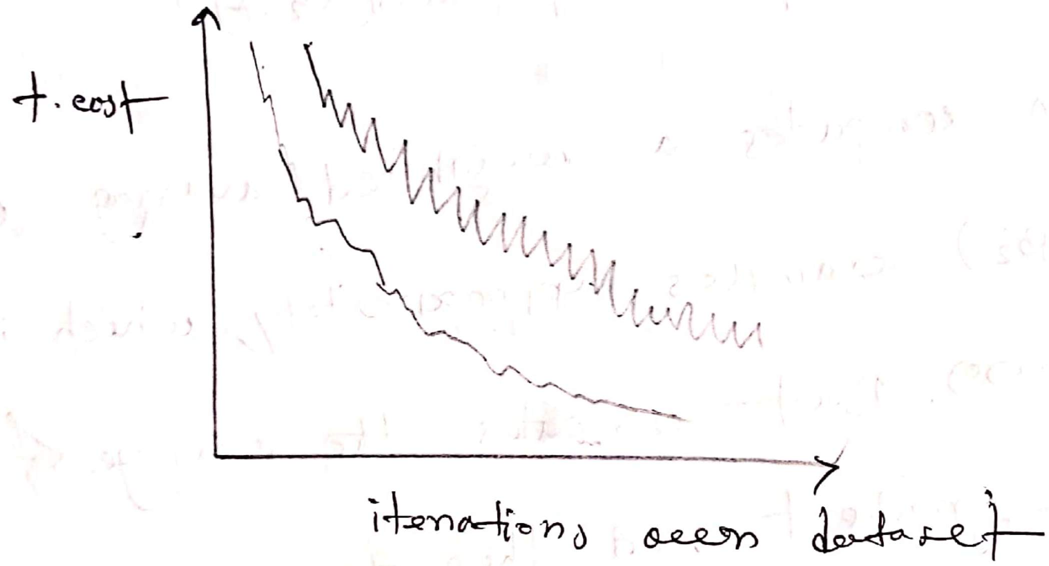
$$V\_corrected = V / (1 - np.power(\beta_2, t))$$

It also computes a weighted avarage over last $1/(1-\beta_2)$ examples approximately, which is 100 when $\beta_2 = 0.999$. But computes the avarage of the square of the gradient and then the same moment of follows,

$$theta = theta - learning\_rate * m\_corrected / np.sqrt(V\_corrected + \epsilon)$$

using m_corrected ensures that our gradient moves in the direction of the general trend and does not oscillate too much. while dividing by the square root of the mean of squared magnitudes ensure that the overall magnitude of the step is close to unit value. Finally, ϵ is added to the denominator to avoid division by 0

Effect on Performance


J. cost
iterations over dataset