Lecture 02 Notes [21/08/2019]
[Aman Choudhary 15MI408]

[Lekhraj Kumar 15MI449]

[Sumit Kumar 15MI446]

1. What is Gradient Descent?

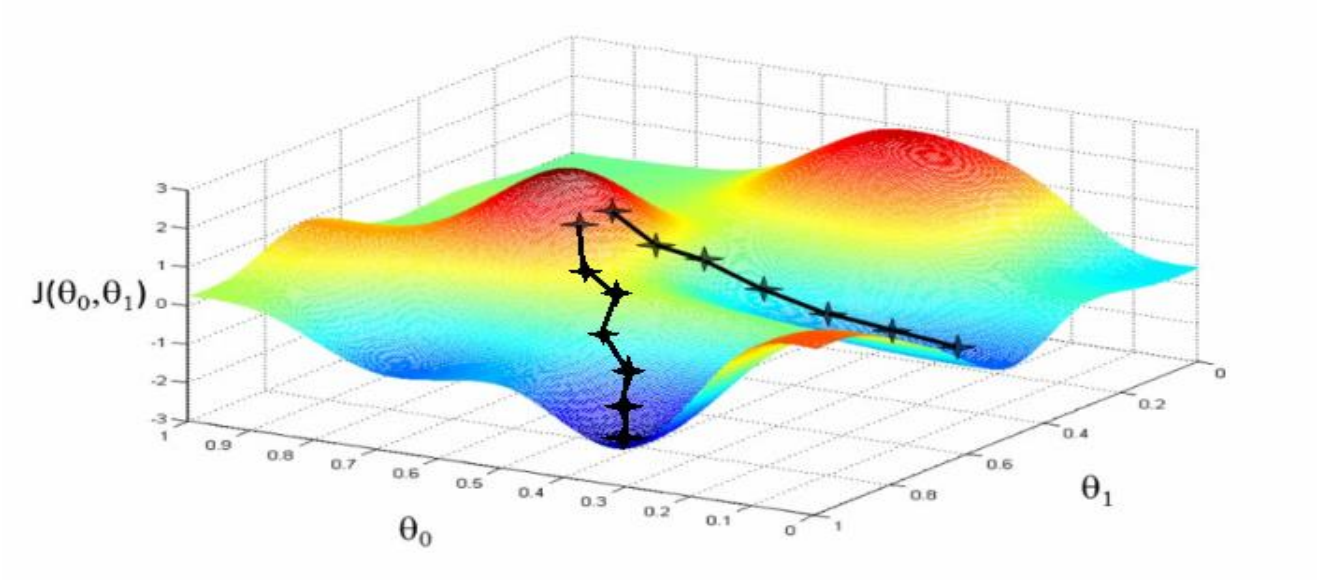
Gradient Descent is the most common optimization algorithm in machine learning and deep learning. It is a first-order optimization algorithm. This means it only takes into account the first derivative when performing the updates on the parameters.

To explain Gradient Descent I'll use the classic mountaineering example.

Suppose you are at the top of a mountain, and you have to reach a lake which is at the lowest point of the mountain (a.k.a valley). A twist is that you are blindfolded and you have zero visibility to see where you are headed. So, what approach will you take to reach the lake?

The best way is to check the ground near you and observe where the land tends to descend. This will give an idea in what direction you should take your first step. If you follow the descending path, it is very likely you would reach the lake.

To represent this graphically, notice the below graph.



Let us now map this scenario in mathematical terms.

Suppose we want to find out the best parameters (θ1) and (θ2) for our learning algorithm. Similar to the analogy above, we see we find similar mountains and valleys when we plot our "cost space". Cost space is nothing but how our algorithm would perform when we choose a particular value for a parameter.

So on the y-axis, we have the cost J(θ) against our parameters θ1 and θ2 on x-axis and z-axis respectively. Here, hills are represented by red region, which have high cost, and valleys are represented by blue region, which have low cost.

Now there are many types of gradient descent algorithms. They can be classified by two methods mainly:

- **On the basis of data ingestion**
    1. Full Batch Gradient Descent Algorithm
    2. Stochastic Gradient Descent Algorithm
    3. Mini Batch Gradient Descent Algorithm

In full batch gradient descent algorithms, you use whole data at once to compute the gradient, whereas in stochastic you take a sample while computing the gradient.

- **On the basis of differentiation techniques**
    1. First order Differentiation
    2. Second order Differentiation

Gradient descent requires calculation of gradient by differentiation of cost function. We can either use first order differentiation or second order differentiation.

2. Challenges in executing Gradient Descent

Gradient Descent is a sound technique which works in most of the cases. But there are many cases where gradient descent does not work properly or fails to work altogether. There are three main reasons when this would happen:

1. Data challenges
2. Gradient challenges
3. Implementation challenges

2.1 Data Challenges

- If the data is arranged in a way that it poses a **non-convex optimization problem.** It is very difficult to perform optimization using gradient descent. Gradient descent only works for problems which have a well-defined convex optimization problem.

- Even when optimizing a convex optimization problem, there may be numerous minimal points. The lowest point is called global minimum, whereas rest of the points are called local minima. Our aim is to go to global minimum while avoiding local minima.
- There is also a saddle point problem. This is a point in the data where the gradient is zero but is not an optimal point. We don't have a specific way to avoid this point and is still an active area of research.

2.2 Gradient Challenges

- If the execution is not done properly while using gradient descent, it may lead to problems like vanishing gradient or exploding gradient problems. These problems occur when the gradient is too small or too large. And because of this problem the algorithms do not converge.

2.3 Implementation Challenges

- Most of the neural network practitioners don't generally pay attention to implementation, but it's very important to look at the resource utilization by networks. For eg: When implementing gradient descent, it is very important to note how many resources you would require. If the memory is too small for your application, then the network would fail.
- Also, it's important to keep track of things like floating point considerations and hardware / software prerequisites.

4. Gradient Descent in Linear Regression

In linear regression, the model targets to get the best-fit regression line to predict the value of y based on the given input value (x). While training the model, the model calculates the cost function which measures the Root Mean Squared error between the predicted value (pred) and true value (y). The model targets to minimize the cost function.

To minimize the cost function, the model needs to have the best value of θ_1 and θ_2. Initially model selects θ_1 and θ_2 values randomly and then itertively update these value in order to minimize the cost function untill it reaches the minimum. By the time model achieves the minimum cost function, it will have the best θ_1 and θ_2 values. Using these finally updated values of θ_1 and θ_2 in the hypothesis equation of linear equation, model predicts the value of x in the best manner it can.

Therefore, the question arises – **How θ_1 and θ_2 values get updated ?**
Linear Regression Cost Function:

$$J = \frac{1}{n}\sum_{i=1}^{n}(pred_i - y_i)^2$$

$$minimize \frac{1}{n} \sum_{i=1}^{n} (pred_i - y_i)^2$$

Gradient Descent Algorithm For Linear Regression

Cost Function

$$J(\Theta_0, \Theta_1) = \frac{1}{2m} \sum_{i=1}^{m} [h_\Theta(x_i) - y_i]^2$$

$h_\Theta(x_i)$: Predicted Value; y_i: True Value

Gradient Descent

$$\Theta_j = \Theta_j - \alpha \frac{\partial}{\partial \Theta_j} J(\Theta_0, \Theta_1)$$

α: Learning Rate

Now,

$$\frac{\partial}{\partial \Theta} J_\Theta = \frac{\partial}{\partial \Theta} \frac{1}{2m} \sum_{i=1}^{m} [h_\Theta(x_i) - y]^2$$

$$= \frac{1}{m} \sum_{i=1}^{m} (h_\Theta(x_i) - y) \frac{\partial}{\partial \Theta_j} (\Theta x_i - y)$$

$$= \frac{1}{m} (h_\Theta(x_i) - y) x_i$$

Therefore,

$$\Theta_j := \Theta_j - \frac{\alpha}{m} \sum_{i=1}^{m} [(h_\Theta(x_i) - y) x_i]$$

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

Now,

$$\frac{\partial}{\partial \theta} J_\theta = \frac{\partial}{\partial \theta} \frac{1}{2m} \sum_{i=1}^{m} \left[h_\theta(x_i) - y_i \right]^2$$

$$\frac{\partial}{\partial \theta} J_\theta = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x_i) - y_i \right) \cdot \frac{\partial}{\partial \theta_j} \left(\theta x_i - y_i \right)$$

$$\frac{\partial}{\partial \theta} J_\theta = \frac{1}{m} \sum_{i=1}^{m} \left[\left(h_\theta(x_i) - y \right) x_i \right]$$

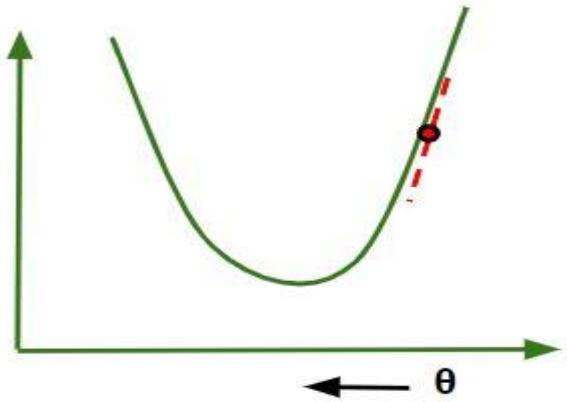
Therefore,

$$\theta_j := \theta_j - \frac{\alpha}{m} \sum_{i=1}^{m} \left[\left(h_\theta(x_i) - y_i \right) x_i \right]$$

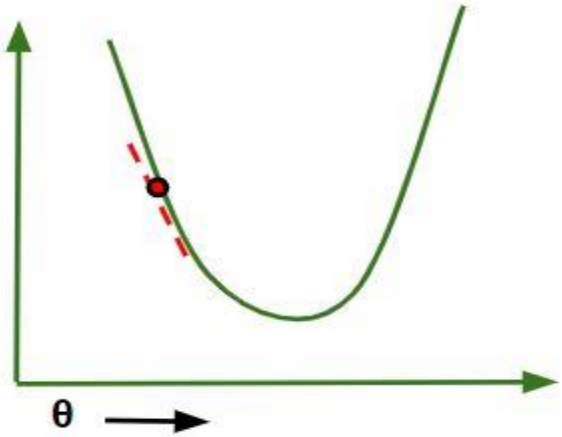
-> θ_j **:** Weights of the hypothesis.
-> $h_\theta(x_i)$ **:** predicted y value for i^{th} input.
-> **j** **:** Feature index number (can be 0, 1, 2, ..., n).
-> α **:** Learning Rate of Gradient Descent.

We graph cost function as a function of parameter estimates i.e. parameter range of our hypothesis function and the cost resulting from selecting a particular set of parameters. We move downward towards pits in the graph, to find the minimum value. Way to do this is taking derivative of cost function as explained in the above figure. Gradient Descent step downs the cost function in the direction of the steepest descent. Size of each step is determined by parameter **α** known as **Learning Rate**.

In the Gradient Descent algorithm, one can infer two points:



1. **If slope is +ve** : $\theta_j = \theta_j$ – (+ve value). Hence value of θ_j decreases.

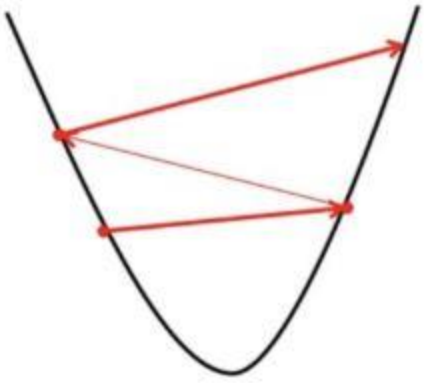


2. **If slope is -ve** : $\theta_j = \theta_j$ – (-ve value). Hence value of θ_j increases.

The choice of correct learning rate is very important as it ensures that Gradient Descent converges in a reasonable time. :

1. If we choose **α to be very large**, Gradient Descent can overshoot the minimum. It may fail to converge or even diverge.

2.



3. If we choose α to be very small, Gradient Descent will take small steps to reach local minima and will take a longer time to reach minima.

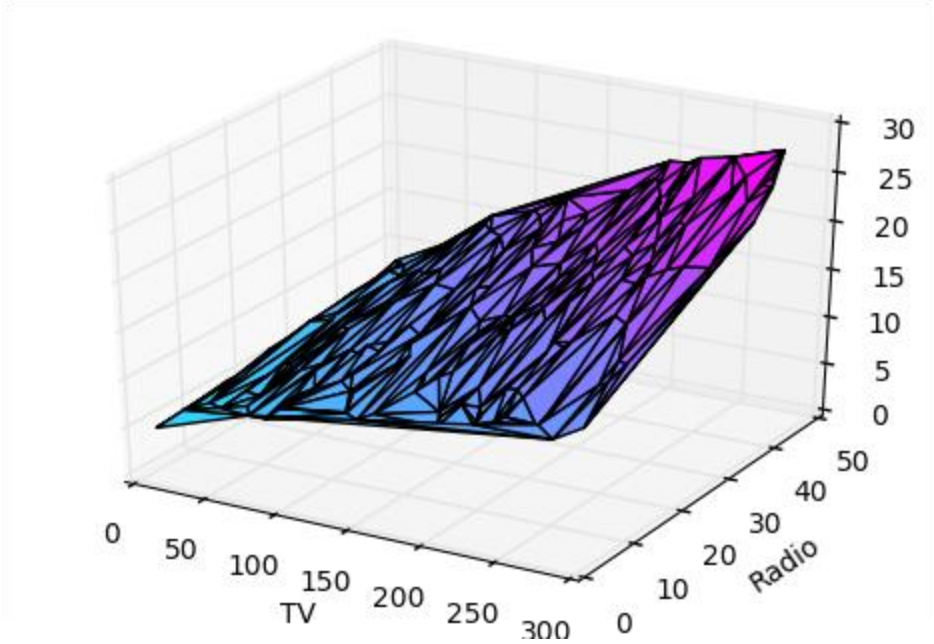
Example:

Let's say we are given data on TV, radio, and newspaper advertising spend for a list of companies, and our goal is to predict sales in terms of units sold.

Company	TV	Radio	News	Units
Amazon	230.1	37.8	69.1	22.1
Google	44.5	39.3	23.1	10.4
Facebook	17.2	45.9	34.7	18.3
Apple	151.5	41.3	13.2	18.5

Growing complexity

As the number of features grows, the complexity of our model increases and it becomes increasingly difficult to visualize, or even comprehend, our data.



One solution is to break the data apart and compare 1-2 features at a time. In this example we explore how Radio and TV investment impacts Sales.

Normalization

As the number of features grows, calculating gradient takes longer to compute. We can speed this up by "normalizing" our input data to ensure all values are within the same range. This is especially important for datasets with high standard deviations or differences in the ranges of the attributes. Our goal now will be to normalize our features so they are all in the range -1 to 1.

Note

Matrix math. Before we continue, it's important to understand basic Linear Algebra concepts as well as numpy functions like numpy.dot().

Making predictions

Our predict function outputs an estimate of sales given our current weights (coefficients) and a company's TV, radio, and newspaper spend. Our model will try to identify weight values that most reduce our cost function.

$$Sales = W1TV + W2Radio + W3Newspaper$$

Cost function

Now we need a cost function to audit how our model is performing. The math is the same, except we swap the ***mx+b*** expression for ***W1x1+W2x2+W3x3***. We also divide the expression by 2 to make derivative calculations simpler.

$$MSE = \frac{1}{2N}\sum_{i=1}^{n}(y_i - (W_1x_1 + W_2x_2 + W_3x_3))^2$$

Gradient descent

Again using the Chain rule we can compute the gradient–a vector of partial derivatives describing the slope of the cost function for each weight.

$$\begin{aligned} f'(W_1) &= -x_1(y - (W_1x_1 + W_2x_2 + W_3x_3)) \\ f'(W_2) &= -x_2(y - (W_1x_1 + W_2x_2 + W_3x_3)) \\ f'(W_3) &= -x_3(y - (W_1x_1 + W_2x_2 + W_3x_3)) \end{aligned}$$

Simplifying with matrices

The gradient descent code above has a lot of duplication. Can we improve it somehow? One way to refactor would be to loop through our features and weights–allowing our function to handle any number of features. However there is another even better technique: vectorized gradient descent.

Math

We use the same formula as above, but instead of operating on a single feature at a time, we use matrix multiplication to operative on all features and weights simultaneously. We replace the xixiterms with a single feature matrix XX.

Gradient = −X(targets−predictions)gradient=−X(targets−predictions)

Bias term

Our train function is the same as for simple linear regression, however we're going to make one final tweak before running: add a bias term to our feature matrix.

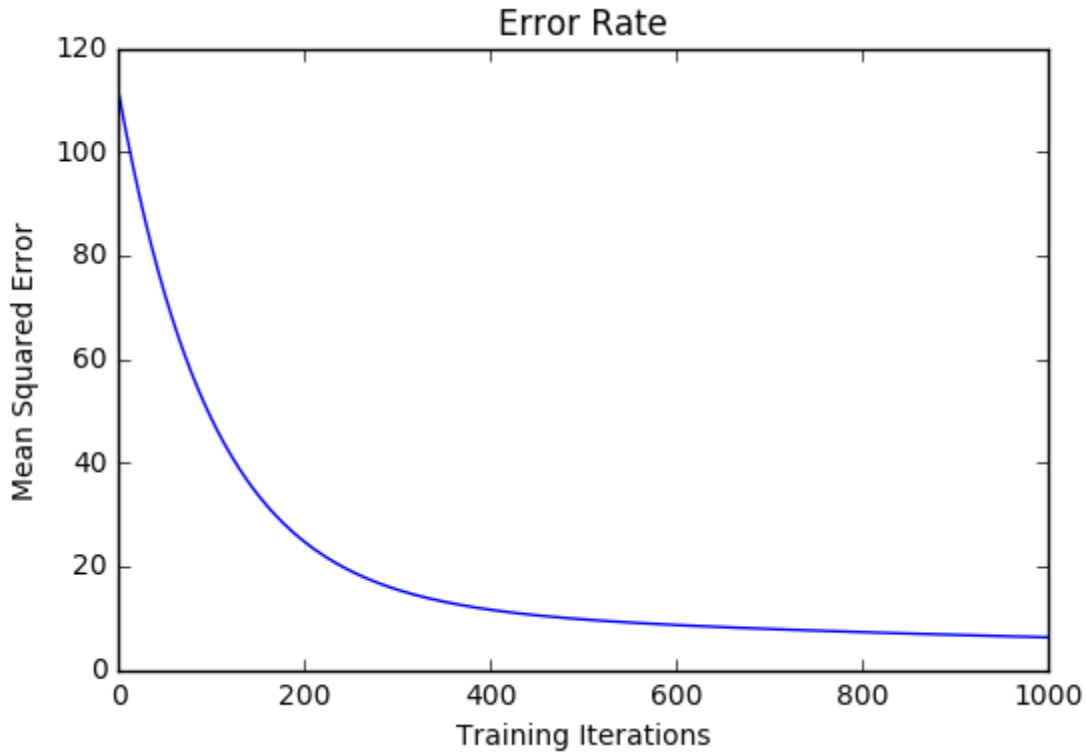
In our example, it's very unlikely that sales would be zero if companies stopped advertising. Possible reasons for this might include past advertising, existing customer relationships, retail locations, and salespeople. A bias term will help us capture this base case.

Model evaluation

After training our model through 1000 iterations with a learning rate of .0005, we finally arrive at a set of weights we can use to make predictions:

Sales=4.7TV+3.5Radio+.81Newspaper+13.9

Our MSE cost dropped from 110.86 to 6.25.



Stochastic Gradient Descent (SGD):

The word '*stochastic*' means a system or a process that is linked with a random probability. Hence, in Stochastic Gradient Descent, a few samples are selected randomly instead of the whole data set for each iteration. In Gradient Descent, there is a term called "batch" which denotes the total number of samples from a dataset that is used for calculating the gradient for each iteration. In typical Gradient Descent optimization, like Batch Gradient Descent, the batch is taken to be the whole dataset. Although, using the whole dataset is really useful for getting to the minima in a less noisy or less random manner, but the problem arises when our datasets get really huge. Suppose, you have a million samples in your dataset, so if you use a typical Gradient Descent optimization technique, you will have to use all of the one million samples for completing one iteration while performing the Gradient Descent, and it has to be done for every iteration until the minima is reached. Hence, it becomes computationally very expensive to perform.

This problem is solved by Stochastic Gradient Descent. In SGD, it uses only a single sample, i.e., a batch size of one, to perform each iteration. The sample is randomly shuffled and selected for performing the iteration.

SGD algorithm:

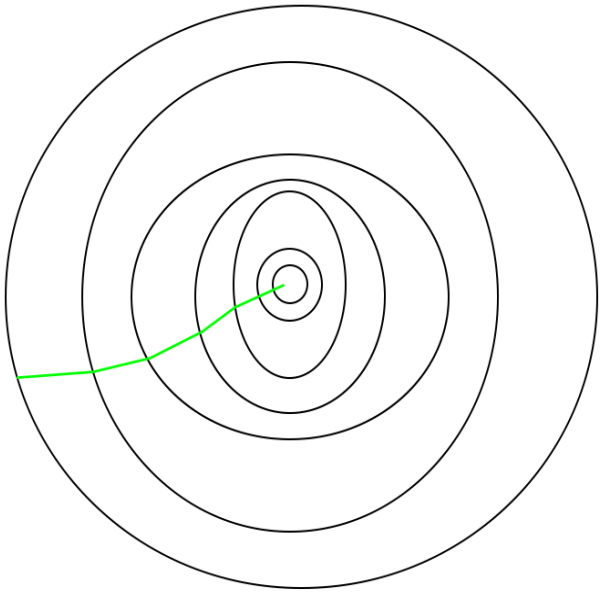
$$for\ i\ in\ range\ (m):$$

$$\theta_j = \theta_j - \alpha\,(\hat{y}^i - y^i)\,x_j^i$$

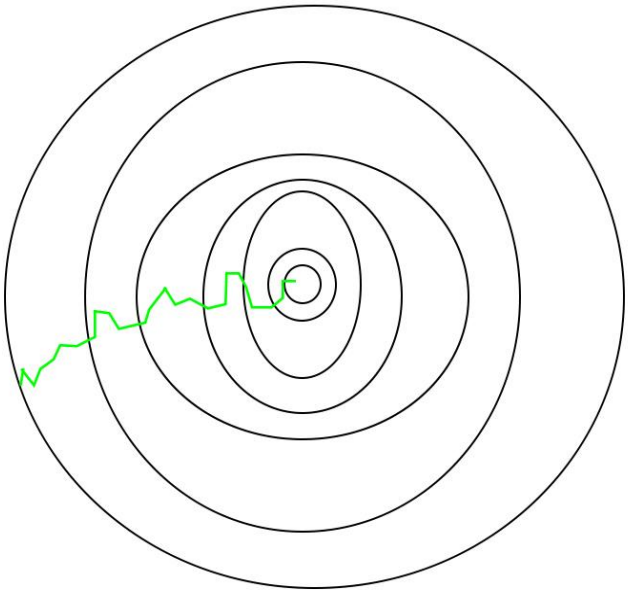
So, in SGD, we find out the gradient of the cost function of a single example at each iteration instead of the sum of the gradient of the cost function of all the examples.

In SGD, since only one sample from the dataset is chosen at random for each iteration, the path taken by the algorithm to reach the minima is usually noisier than your typical Gradient Descent algorithm. But that doesn't matter all that much because the path taken by the algorithm does not matter, as long as we reach the minima and with significantly shorter training time.

Path taken by Batch Gradient Descent –



Path taken by Stochastic Gradient Descent –



One thing to be noted is that, as SGD is generally noisier than typical Gradient Descent, it usually took a higher number of iterations to reach the minima, because of its randomness in its descent. Even though it requires a higher number of iterations to reach the minima than typical Gradient Descent, it is still computationally much less expensive than typical Gradient Descent. Hence, in most scenarios, SGD is preferred over Batch Gradient Descent for optimizing a learning algorithm.

Mini-Batch Gradient Descent

Mini-batch gradient descent is a variation of the gradient descent algorithm that splits the training dataset into small batches that are used to calculate model error and update model coefficients.

Implementations may choose to sum the gradient over the mini-batch which further reduces the variance of the gradient.

Mini-batch gradient descent seeks to find a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent. It is the most common implementation of gradient descent used in the field of deep learning.

Upsides

- The model update frequency is higher than batch gradient descent which allows for a more robust convergence, avoiding local minima.
- The batched updates provide a computationally more efficient process than stochastic gradient descent.
- The batching allows both the efficiency of not having all training data in memory and algorithm implementations.

Downsides

- Mini-batch requires the configuration of an additional "mini-batch size" hyperparameter for the learning algorithm.
- Error information must be accumulated across mini-batches of training examples like batch gradient descent.

How to Configure Mini-Batch Gradient Descent

Mini-batch gradient descent is the recommended variant of gradient descent for most applications, especially in deep learning.

Mini-batch sizes, commonly called "batch sizes" for brevity, are often tuned to an aspect of the computational architecture on which the implementation is being executed. Such as a power of two that fits the memory requirements of the GPU or CPU hardware like 32, 64, 128, 256, and so on.

Batch size is a slider on the learning process.

- Small values give a learning process that converges quickly at the cost of noise in the training process.
- Large values give a learning process that converges slowly with accurate estimates of the error gradient.

Differences:-

Batch Gradient Descent	Stochastic Gradient Descent	Mini-Batch Gradient Descent
Since entire training data is considered before taking a step in the direction of gradient, therefore it takes a lot of time for making a single update.	Since only a single training example is considered before taking a step in the direction of gradient, we are forced to loop over the training set and thus cannot exploit the speed associated with vectorizing the code.	Since a subset of training examples is considered, it can make quick updates in the model parameters and can also exploit the speed associated with vectorizing the code.

It makes smooth updates in the model parameters	It makes very noisy updates in the parameters	Depending upon the batch size, the updates can be made less noisy – greater the batch size less noisy is the update