**Types of Gradient Descent**

**What is a Gradient?**



***dy = change in y***

***dx = change in x***

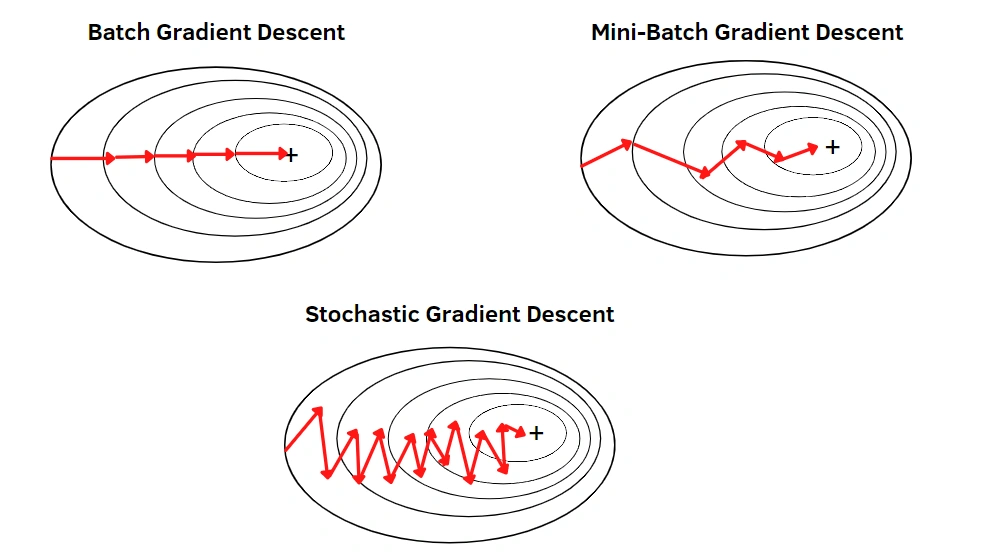
1. A gradient measures how much the output of a function changes if you change the inputs a little bit.
2. In machine learning, a gradient is a derivative of a function that has more than one input variable. Known as the slope of a function in mathematical terms, the gradient simply measures the change in all weights about the change in error.

**Learning Rate:**

The algorithm designer can set the learning rate. If we use a learning rate that is too small, it will cause us to update very slowly, requiring more iterations to get a better solution.

Types of Gradient Descent:

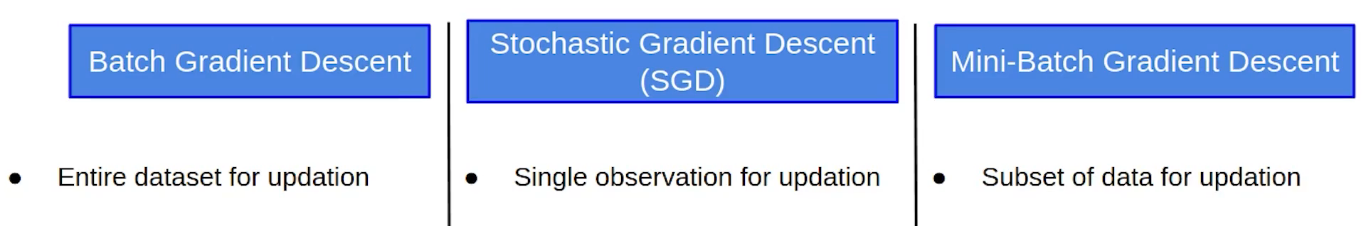
There are three popular types  that mainly differ in the amount of data they use:



**Comparison between Batch GD, SGD, and Mini-batch GD:**

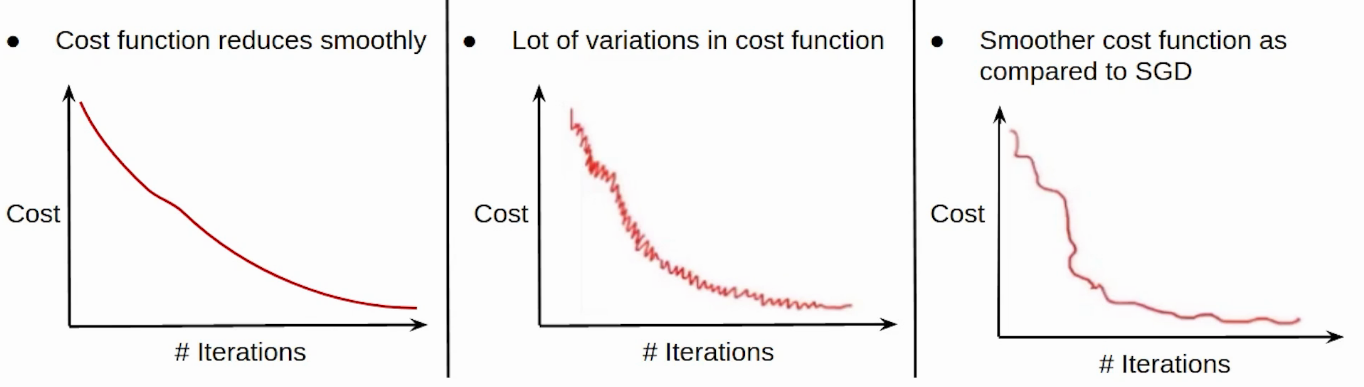
This is a brief overview of the different variants of Gradient Descent. Now let’s compare these different types with each other:

Comparison: Number of observations used for Updation



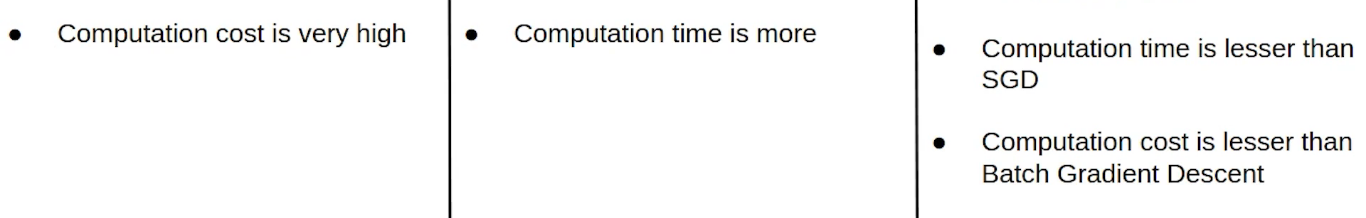
* In batch gradient Descent, as we have seen earlier as well, we take the entire dataset > calculate the cost function > update parameter.
* In the case of Stochastic Gradient Descent, we update the parameters after every single observation and we know that every time the weights are updated it is known as an iteration.
* In the case of Mini-batch Gradient Descent, we take a subset of data and update the parameters based on every subset.

Comparison: Cost function



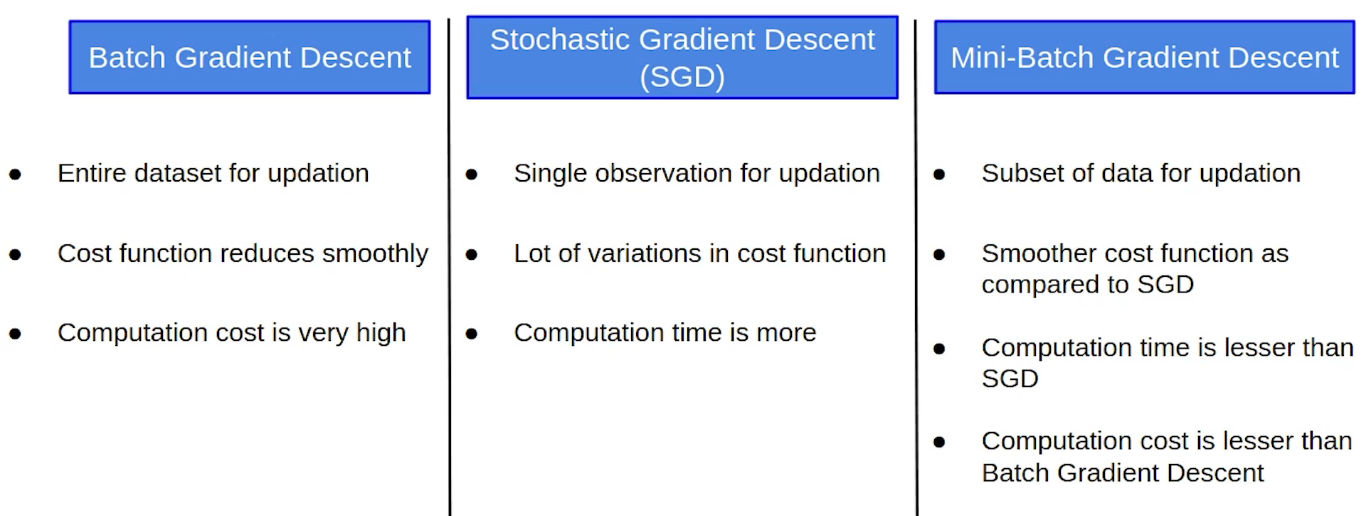
* Now since we update the parameters using the entire data set in the case of the Batch GD, the cost function, in this case, reduces smoothly.
* On the other hand, this updation in the case of SGD is not that smooth. Since we’re updating the parameters based on a single observation, there are a lot of iterations. It might also be possible that the model starts learning noise as well.
* The updation of the cost function in the case of Mini-batch Gradient Descent is smoother as compared to that of the cost function in SGD. Since we’re not updating the parameters after every single observation but after every subset of the data.

Comparison: Computation Cost and Time



* Now coming to the computation cost and time taken by these variants of Gradient Descent. Since we’ve to load the entire data set at a time, perform the forward propagation on that and calculate the error and then update the parameters, the computation cost in the case of Batch gradient descent is very high.
* Computation cost in the case of SGD is less as compared to the Batch Gradient Descent since we’ve to load every single observation at a time but the Computation time here increases as there will be more number of updates which will result in more number of iterations.
* In the case of Mini-batch Gradient Descent, taking a subset of the data there are a lesser number of iterations or updations and hence the computation time in the case of mini-batch gradient descent is less than SGD. Also, since we’re not loading the entire dataset at a time whereas loading a subset of the data, the computation cost is also less as compared to the Batch gradient descent. This is the reason why people usually prefer using Mini-batch gradient descent. Practically whenever we say Stochastic Gradient Descent we generally refer to Mini-batch Gradient Descent.

Here is the complete Comparison Chart:



FAQs

**Q1.Which is the fastest gradient descent?**

The fastest gradient descent algorithm is stochastic gradient descent (SGD), as it updates the model parameters after processing each training example, leading to faster convergence.

**Q2. Why is batch gradient descent better?**

Batch gradient descent is better because it computes the gradient using the entire training dataset, leading to more accurate updates and smoother convergence. However, it can be slower than stochastic gradient descent, especially for large datasets

**Q3. How is batch gradient descent different from normal equation?**

Batch gradient descent is an iterative algorithm that updates the model parameters after processing the entire training dataset, while the normal equation is a closed-form solution that directly computes the optimal parameters without iteration.

**1. BATCH GRADIENT DESCENT:**

Batch gradient descent, also known as vanilla gradient descent, calculates the error for each example within the training dataset. Still, the model is not changed until every training sample has been assessed. The entire procedure is referred to as a cycle and a training epoch.

Some benefits of batch are its computational efficiency, which produces a stable error gradient and a stable convergence. Some drawbacks are that the stable error gradient can sometimes result in a state of convergence that isn’t the best the model can achieve. It also requires the entire training dataset to be in memory and available to the algorithm.

class GDRegressor:  
   
 def \_\_init\_\_(self,learning\_rate=0.01,epochs=100):  
   
 self.coef\_ = None  
 self.intercept\_ = None  
 self.lr = learning\_rate  
 self.epochs = epochs  
   
 def fit(self,X\_train,y\_train):  
 # init your coefs  
 self.intercept\_ = 0  
 self.coef\_ = np.ones(X\_train.shape[1])  
   
 for i in range(self.epochs):  
 # update all the coef and the intercept  
 y\_hat = np.dot(X\_train,self.coef\_) + self.intercept\_  
 #print("Shape of y\_hat",y\_hat.shape)  
 intercept\_der = -2 \* np.mean(y\_train - y\_hat)  
 self.intercept\_ = self.intercept\_ - (self.lr \* intercept\_der)  
   
 coef\_der = -2 \* np.dot((y\_train - y\_hat),X\_train)/X\_train.shape[0]  
 self.coef\_ = self.coef\_ - (self.lr \* coef\_der)  
   
 print(self.intercept\_,self.coef\_)  
   
 def predict(self,X\_test):  
 return np.dot(X\_test,self.coef\_) + self.intercept\_

**Advantages**

1. Fewer model updates mean that this variant of the steepest descent method is more computationally efficient than the stochastic gradient descent method.
2. Reducing the update frequency provides a more stable error gradient and a more stable convergence for some problems.
3. Separating forecast error calculations and model updates provides a parallel processing-based algorithm implementation.

**Disadvantages**

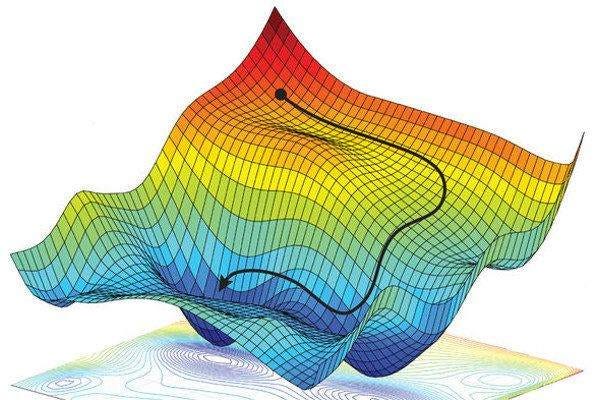
1. A more stable error gradient can cause the model to prematurely converge to a suboptimal set of parameters.
2. End-of-training epoch updates require the additional complexity of accumulating prediction errors across all training examples.
3. The batch gradient descent method typically requires the entire training dataset in memory and is implemented for use in the algorithm.
4. Large datasets can result in very slow model updates or training speeds.
5. Slow and require more computational power.

**2. STOCHASTIC GRADIENT DESCENT:**

**Introduction**

Stochastic Gradient Descent (SGD) is an effective and popular optimization algorithm for machine learning. Its key strength is its ability to process large datasets and reach convergence quickly. It also has high computational efficiency due to the fact that the gradient can be estimated with a random sample of data points instead of requiring the full dataset.

Unlike traditional Gradient Descent, SGD relies on a single example or samples at each iteration, introducing randomness into the learning process. This makes your model learn faster as it converges quicker than other optimization methods such as Mini-Batch or Batch Gradient Descent. When applied to models with thousands of features, SGD performs extremely well because of its significant computational speed and accuracy.



[Credits](https://dtmvamahs40ux.cloudfront.net/gl-academy/course/course-1281-Non-convex-optimization-We-utilize-stochastic-gradient-descent-to-find-a-local-optimum.jpg)

Overall, Stochastic Gradient Descent presents an excellent opportunity for machine learning engineers to simplify their training process and improve model performance significantly.

“SGD is the workhorse of machine learning.” — Yoshua Bengio

**What was the need for SGD?**

There was a need for SGD (Stochastic Gradient Descent) in ML (Machine Learning) because of the following reasons:

1. Large datasets: In ML, it is common to have very large datasets with millions or even billions of data points. Using traditional gradient descent algorithms on such large datasets is computationally expensive and impractical. In such cases, SGD works well because it uses randomly selected data points to update the model parameters, making the process faster and more efficient.

2. Non-Convex optimization problems: In many ML problems, the objective function is not convex, which means there are multiple local minima. Traditional gradient descent algorithms may get stuck in one of the local minima, resulting in suboptimal models. SGD, on the other hand, is less likely to get stuck because it updates the parameters using only a few data points at a time, making it more likely to find the global minimum.

3. Online learning: In some ML applications, new data is constantly being generated and added to the dataset. In such cases, it is necessary to update the model parameters in real-time, as new data becomes available. SGD is well-suited for such tasks because it updates the model parameters using small batches of data and can be applied to data as it arrives.

Overall, SGD is a more efficient and practical alternative to traditional gradient descent algorithms in many ML applications, particularly for large datasets, non-convex optimization problems, and online learning.

**The Math Behind SGD**

SGD works by iteratively updating the model parameters in the direction of the negative gradient of the loss function. The gradient of the loss function is a vector that points in the direction of the steepest ascent of the loss function. By moving in the direction of the negative gradient, SGD is able to find the minimum of the loss function.

The loss function is a measure of how well the model fits the data. The goal of SGD is to find the set of model parameters that minimizes the loss function.

The gradient of the loss function can be calculated using the following formula:

gradient = ∂L/∂θ

where L is the loss function and θ is the vector of model parameters.

The gradient can be used to update the model parameters using the following formula:

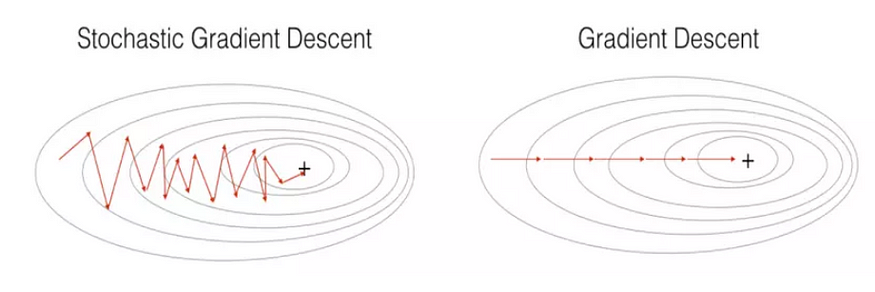
θ = θ - η \* gradient

where η is the learning rate. The learning rate is a hyperparameter that controls how much the model parameters are updated each iteration.

**Stochastic Gradient Descent vs. Gradient Descent**

Here are some of the benefits of using SGD over GD:

* Speed: SGD is much faster than GD, especially when the dataset is large. This is because SGD only updates the model parameters after each individual data point, while GD updates the parameters after each batch of data points.
* Robustness to noise: SGD is more robust to noise in the data than GD. This is because SGD only uses a single data point to update the parameters, so it is less likely to be affected by outliers or noise in the data.
* Efficiency: SGD is more efficient than GD in terms of memory usage. This is because SGD only needs to store the current parameters and the gradient of the loss function, while GD needs to store the entire dataset.



<https://towardsdatascience.com/stochastic-gradient-descent-math-and-python-code-35b5e66d6f79/>

<https://medium.com/@gayatrinikam5103/stochastic-gradient-descent-d7833e6fc88d>

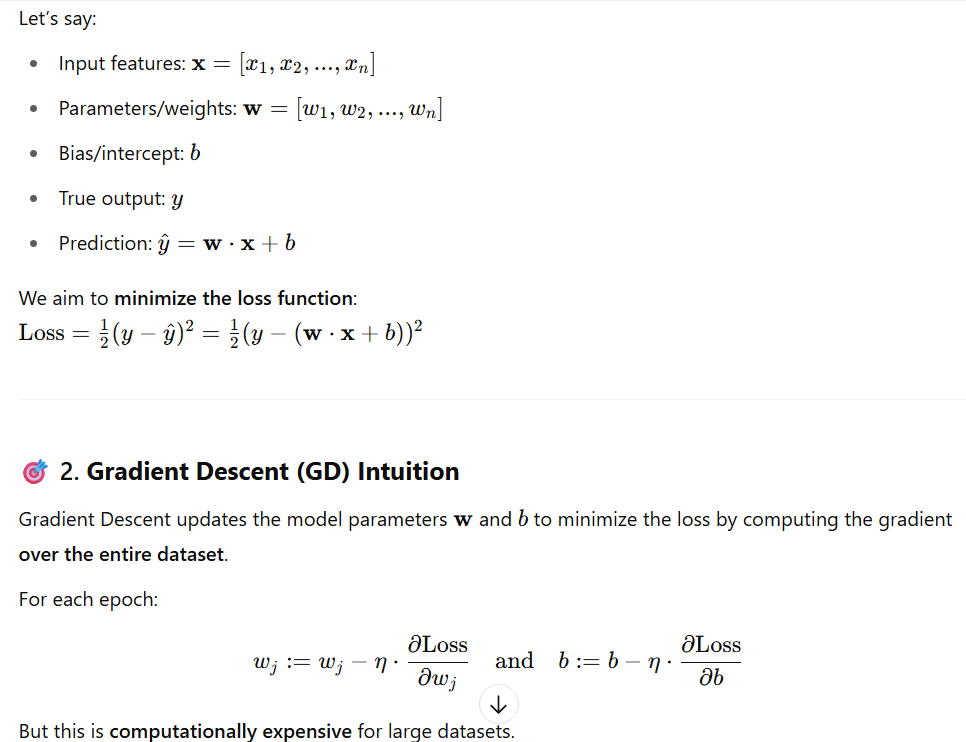
[Credits](https://i.stack.imgur.com/G7BBG.png)

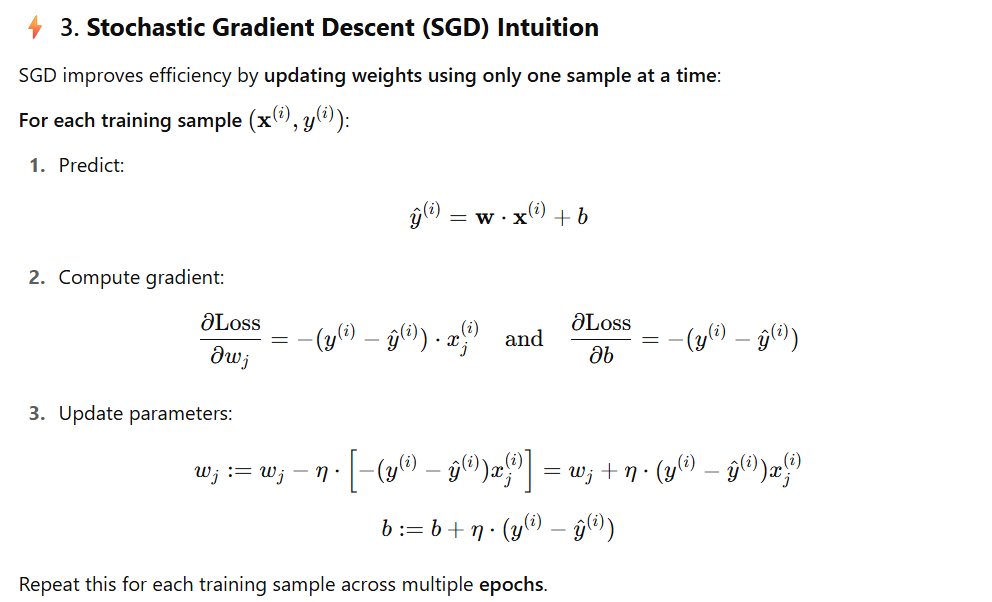
Here is a mathematical proof that shows that SGD converges to the same minimum as GD, but at a slower rate. Let *f*(*x*) be a differentiable function and let *x*0​ be an initial guess for the minimum of *f*(*x*).

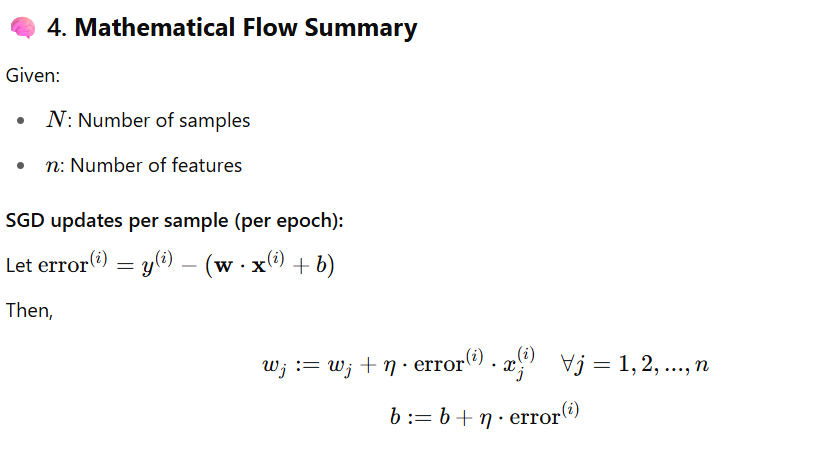
It can be shown that both GD and SGD converge to the same minimum of *f*(*x*), but SGD converges at a slower rate. This is because SGD only uses a single data point to update the parameters, while GD uses the entire dataset.

Here are some statistical facts about SGD:

* SGD is a stochastic algorithm, which means that it is not guaranteed to converge to the same minimum every time it is run. However, it is typically very close to the minimum after a large number of iterations.
* The learning rate *η* is a hyperparameter that controls the speed of convergence. A smaller learning rate will converge more slowly, but it will be more stable. A larger learning rate will converge more quickly, but it may be more likely to diverge.
* The choice of batch size can also affect the speed of convergence. A smaller batch size will converge more slowly, but it will be more accurate. A larger batch size will converge more quickly, but it may be less accurate.







**The Implementation of SGD**

SGD can be implemented in any programming language that supports numerical computation. In Python, SGD can be implemented using the scipy.optimize library.

The following code shows how to implement SGD in Python:

import numpy as np  
from scipy.optimize import minimize  
  
def loss\_function(θ):  
 # Calculate the loss function  
 return np.sum((θ - x)\*\*2)  
  
def gradient\_descent(θ, η, iterations):  
 # Initialize the parameters  
 θ\_new = θ  
  
 # Iterate for the specified number of iterations  
 for i in range(iterations):  
  
 # Calculate the gradient  
 gradient = np.gradient(loss\_function(θ))  
  
 # Update the parameters  
 θ\_new = θ - η \* gradient  
  
 # Return the updated parameters  
 return θ\_new  
  
 # Initialize the parameters  
 θ = np.array([1, 2])  
  
 # Set the learning rate  
 η = 0.01  
 # Set the number of iterations  
 iterations = 1000  
  
 # Run SGD  
 θ\_new = gradient\_descent(θ, η, iterations)  
 # Print the updated parameters  
 print(θ\_new)

**The Pros and Cons of SGD**

SGD is a simple and efficient algorithm, but it can be sensitive to hyperparameters. It can also be slow to converge for large datasets.

“SGD is the most important algorithm in machine learning.” — Andrew Ng

Here are some of the pros and cons of SGD:

**Advantages**

* Simple to implement
* Efficient
* Can be used with any type of loss function.

1. You can instantly see your model’s performance and improvement rates with frequent updates.
2. This variant of the steepest descent method is probably the easiest to understand and implement, especially for beginners.
3. Increasing the frequency of model updates will allow you to learn more about some issues faster.
4. The noisy update process allows the model to avoid local minima (e.g., premature convergence).
5. Faster and require less computational power.
6. Suitable for the larger dataset.

**Disadvantages**

* Sensitive to hyperparameters
* Can be slow to converge for large datasets.

1. Frequent model updates are more computationally intensive than other steepest descent configurations, and it takes considerable time to train the model with large datasets.
2. Frequent updates can result in noisy gradient signals. This can result in model parameters and cause errors to fly around (more variance across the training epoch).
3. A noisy learning process along the error gradient can also make it difficult for the algorithm to commit to the model’s minimum error.

**Conclusion**

SGD is a powerful optimization algorithm that can be used to train machine learning models. It is a simple algorithm, but it can be difficult to understand how it works. This blog post has provided a detailed explanation of SGD, including its math, implementation, and pros and cons.

**Implementation of sgd classifier in sklearn:**

from sklearn.linear\_model import SGDClassifier

X = [[0., 0.], [1., 1.]]

y = [0, 1]

clf = SGDClassifier(loss="hinge", penalty="l2", max\_iter=5)

clf.fit(X, y)

SGDClassifier(max\_iter=5)

**3. MINI-BATCH GRADIENT DESCENT:**

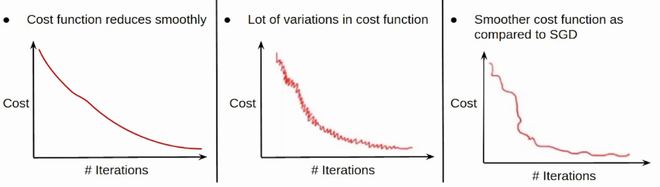
Since mini-batch gradient descent combines the ideas of batch gradient descent with SGD, it is the preferred technique. It divides the training dataset into manageable groups and updates each separately. This strikes a balance between batch gradient descent’s effectiveness and stochastic gradient descent’s durability.

Mini-batch sizes typically range from 50 to 256, although, like with other machine learning techniques, there is no set standard because it depends on the application. The most popular kind in deep learning, this method is used when training a neural network.

**3. Mini-Batch Gradient Descent (MBGD)**

Mini-batch gradient descent combines the benefits of both batch and stochastic gradient descent. Instead of using the entire dataset or a single sample, MBGD processes small batches of data at a time. This balances computational efficiency and model stability.

* **Pros**: Faster training than batch gradient descent, more stable than SGD
* **Cons**: Choosing the right batch size can be challenging



**How Mini-Batch Gradient Descent Works**

The key idea behind mini-batch gradient descent is to split the training dataset into small batches and compute the gradient for each batch. The process can be broken down into the following steps:

1. **Shuffle the dataset** to ensure each mini-batch is representative of the entire data distribution.
2. **Divide the dataset** into mini-batches of a specified size.
3. **Compute the gradient** for each mini-batch by calculating the loss and its derivative.
4. **Update model parameters** based on the computed gradient.
5. **Repeat the process** until convergence is achieved or a stopping criterion is met.

**Implementing Mini-Batch Gradient Descent in Python**

**Step 1: Import Required Libraries**

import numpy as np  
import matplotlib.pyplot as plt

**Step 2: Define the Mini-Batch Gradient Descent Function**

def mini\_batch\_gradient\_descent(X, y, batch\_size=32, learning\_rate=0.01, epochs=100):  
 m, n = X.shape  
 weights = np.zeros(n)  
 bias = 0  
 loss\_history = []  
   
 for epoch in range(epochs):  
 # Shuffle data to ensure randomness  
 indices = np.random.permutation(m)  
 X\_shuffled = X[indices]  
 y\_shuffled = y[indices]  
   
 for i in range(0, m, batch\_size):  
 X\_batch = X\_shuffled[i:i+batch\_size]  
 y\_batch = y\_shuffled[i:i+batch\_size]  
   
 # Compute predictions  
 predictions = np.dot(X\_batch, weights) + bias  
   
 # Compute error  
 error = predictions - y\_batch  
   
 # Compute gradients  
 weights\_gradient = np.dot(X\_batch.T, error) / batch\_size  
 bias\_gradient = np.sum(error) / batch\_size  
   
 # Update parameters  
 weights -= learning\_rate \* weights\_gradient  
 bias -= learning\_rate \* bias\_gradient  
   
 # Compute loss (Mean Squared Error)  
 loss = np.mean(error \*\* 2)  
 loss\_history.append(loss)  
   
 return weights, bias, loss\_history

**Step 3: Generate Sample Data and Train the Model**

# Generate synthetic data  
np.random.seed(42)  
X = np.random.rand(1000, 1) \* 10 # 1000 samples, 1 feature  
y = 3 \* X.squeeze() + np.random.randn(1000) \* 2 # y = 3X + noise

# Train using mini-batch gradient descent  
weights, bias, loss\_history = mini\_batch\_gradient\_descent(X, y)

**Step 4: Plot the Loss Curve**

plt.plot(loss\_history)  
plt.xlabel('Epochs')  
plt.ylabel('Loss')  
plt.title('Mini-Batch Gradient Descent Loss Curve')  
plt.show()

**Practical Considerations**

**1. Choosing the Right Batch Size**

The batch size is a crucial hyperparameter. Typical values range from **16 to 512**. A small batch size provides more frequent updates but increases variance, while a large batch size improves stability but slows down training.

**2. Learning Rate Tuning**

The learning rate determines the step size in parameter updates. If the learning rate is too high, the model may not converge. If it’s too low, convergence can be slow. A common approach is to use an **adaptive learning rate** with techniques like Adam or RMSprop.

**3. Regularization**

To prevent overfitting, regularization techniques such as **L1 (Lasso)** and **L2 (Ridge)** regularization can be applied. These methods penalize large weight values, improving model generalization.

**Advantages**

1. The model is updated more frequently than the stack gradient descent method, allowing for more robust convergence and avoiding local minima.
2. Batch updates provide a more computationally efficient process than stochastic gradient descent.
3. Batch processing allows for both the efficiency of not having all the training data in memory and implementing the algorithm.

**Disadvantages**

1. Mini-batch requires additional hyperparameters “mini-batch size” to be set for the learning algorithm.
2. Error information should be accumulated over a mini-batch of training samples, such as batch gradient descent.
3. it will generate complex functions.

**Configure Mini-Batch Gradient Descent:**

The mini-batch steepest descent method is a variant of the steepest descent method recommended for most applications, intense learning.

Mini-batch sizes, commonly called “batch sizes” for brevity, are often tailored to some aspect of the computing architecture in which the implementation is running. For example, a power of 2 that matches the memory requirements of the GPU or CPU hardware, such as 32, 64, 128, and 256.

The stack size is a slider for the learning process.

Smaller values ​​allow the learning process to converge quickly at the expense of noise in the training process. Larger values ​​result in a learning process that slowly converges to an accurate estimate of the error gradient.

Conclusion

In this article, we learned about different types of gradient descent. The key takeaways from the article are:

* The mini-batch steepest descent method is the recommended method because it combines the concept of batch steepest descent with SGD. Simply divide your training dataset into manageable groups and update each individually. This balances the effectiveness of batch gradient descent with the durability of stochastic gradient descent.
* When using batch gradient descent, adjustments are made after calculating the error for a certain batch. One advantage of the batch gradient descent method is its computational efficiency, which produces a stable error gradient and a stable convergence.
* Stochastic Gradient Descent (SGD) sequentially modifies the parameters of each training sample in each training sample of the dataset. This allows SGD to be faster than batch gradient descent. One benefit is that the regular updates give us a fairly accurate idea of the rate of improvement.
* In general, the higher the learning rate, the faster the model can learn at the expense of the non-optimal final set of weights. With a low learning rate, the model can learn a more optimal or globally optimal set of weights, but it can take considerable time to train.