MATH AND INSURANCE PROJECT REPORT

Stochastic Gradient

Produced by:

Iheb Amri

University Year : 2021-2022

Contents

List of Figures

List of acronyms

- **OCR** : Optical Character Recognition
- **SGD** : Stochastic Gradient Descent
- **GD** : Gradient Descent

1 Introduction

1.1 Purpose

This report will cover the following topics:

- A literature Analysis and presentation about Numerical stochastic gradient descent.
- Numerical implementation using Python.
- Simulation using Python and interpretation

1.2 Motivation and background

The increase of the size of the databases that need to be processed daily leads to the increase of the interest put on large scale optimization problems, which increase remarkably. This, in fact, represents a fundamental issue for machine learning algorithms. In this case, the tasks requisite solving an optimization problem of a composite loss function, whereby the number of terms scales with the number of observations. Alternative learning strategies that can handle large databases need to be developed, since the traditional techniques grounded in full gradient information at each update step are no longer time-sufficient. It is, thus, crucial to do away with traditional plain methods used to build strategies that keep a light 'per update iteration' cost.This is the main focus

of stochastic gradient methods which is not a new concept (cf. ADALINE, 1960), but is of crucial importance today, mainly for training machine learning and resolving minimization problems in finance.

2

Analysis of optimization algorithms and Implementation

2.1 Optimization Concepts

Optimization refers to the process of minimizing/maximizing an objective function f(x) parameterized by x. In machine/deep learning terminology, it consists of the process of minimizing the cost/loss function J(w) under the framework of the model's parameters $w \in R^d$.
Optimization algorithms (in case of minimization) consist of one of the below goals:

1. Delineating the global minimum of the objective function. This is feasible if the objective function is convex : any local minimum is a global minimum.

2. Depicting the lowest possible value of the objective function within its neighborhood, presuming that the objective function is not convex as is the case in most deep learning problems.

There are three kinds of optimization algorithms:

1. If a differentiable function and convexe are used. This consists of an optimization algorithm that is not iterative and simply solves for one point.

2. An iterative approach needs to be approximated with if the analytical solution is not possible; either due the high number of parameters, or the high cost of calculation

    - Optimization algorithm that is iterative in nature and converges to acceptable solutions, regardless of the parameters initialization such as gradient descent applied to logistic regression.
    - Optimization algorithm that is iterative in nature and applied to a set of problems that have non-convex cost functions such as neural networks.Hence, the initialization of parameters plays a crucial role in the speeding up of convergence and the achievement of lower error rates.

2.2 Gradient descent

Gradient descent is defined as the iterative algorithm that initiates from an arbitrary point on a function and declines its slope in steps until it attains the lowest point of that function.

2.2.1 Pseudo Algorithm

The first step is to start with a random point and find a way to update this point with each iteration such that one descends the slope.
To understand how gradient descent works, consider a multi-variable function f(w), where $w = [w1, w2, \ldots, wn]^T$.
The below steps must be followed to find the w at which this function reaches the minimum, gradient descent:

1. Choose an initial random value of w
2. Choose the number of maximum iterations T
3. Choose the learning rate value : $\eta \in [a, b]$
4. Repeat those 2 steps until iterations exceed T or f does not change

    - Compute the step sizes per feature : $\Delta w = -\eta \nabla_w f(w)$

- Update the new parameter w as follows :$w \leftarrow w + \Delta_w$

  ∇w f indicates the gradient of the function f as designated by:

  $\nabla w f(w) = [\partial f(w)/\partial w_1, \partial f(w)/\partial w_2 :: \partial f(w)/\partial w_n]$

The learning rate η adjusts the local minimum. Hence, we need to follow the direction of the slope downhill until we reach a local minimum.

2.2.2 Adding Momentum

The below problems are encountered, when using gradient descent:

1. Being stacked in a local minimum due the greedy nature of the algorithm results.
2. Moving too fast towards the gradient direction leads to overshooting and to miss the global optimum.
3. The oscillation phenomenon that happens when the value of the function slightly changes. It is like a navigating plateau, i.e. the height is constant regardless of your direction.

Inorder to counter these issues, a momentum term α has to be added to the expression for Δ_w to maintain the rate of learning when moving towards the value of the global optimum.

2.2.3 Implementing Gradient Descent in Python

The **gradient-descent()** function is function wherein the loop ends when either:

1. The iterations number surpasses a maximum value
2. The difference between function values betwixt two consecutive iterations falls below a particular threshold

Based on the objective function's gradient, the parameters are updated per iteration.

Inputs:

The function will take the following parameters as inputs:

- **max-iterations**: Maximum iterations to do

- **threshold**: Stop, in case the difference in function varies between two consecutive iterations falls below the aforementioned threshold.
- **w-init**:The Initial starting point of the gradient descent.
- **obj-func**:Function that calculates the objective function.
- **grad-func**:Function that calculates the gradient of the function.
- **extra-param**:Extra parameters if needed for the objective function and gradient funcion.
- **learning-rate**:The step size for gradient descent should be in [0,1].
- **momentum**:The momentum to use should be in [0,1].

Output:

- **w-history**:All points in space, visited by gradient descent wherein the objective function was assessed.
- **f-history**:Corresponding value of the objective function calculated per point.

```
def gradient_descent(max_iterations,threshold,w_init,
                     obj_func,grad_func,extra_param = [],
                     learning_rate=0.05,momentum=0.8):

    w = w_init
    w_history = w
    f_history = obj_func(w,extra_param)
    delta_w = np.zeros(w.shape)
    i = 0
    diff = 1.0e10

    while  i<max_iterations and diff>threshold:
        delta_w = -learning_rate*grad_func(w,extra_param) + momentum*delta_w
        w = w+delta_w

        # store the history of w and f
        w_history = np.vstack((w_history,w))
        f_history = np.vstack((f_history,obj_func(w,extra_param)))

        # update iteration number and diff between successive values
        # of objective function
        i+=1
        diff = np.absolute(f_history[-1]-f_history[-2])

    return w_history,f_history
```

Figure 2.1: The gradient descent function Implementation

2.2.4 Gradient Descent for Minimizing Mean Square Error

GradientDescent is considered as an efficient and simple technique for minimizing the mean square error in a supervised classification or regression problem.

Suppose we are given m training examples [Xij] with i=1... m, wherein each example has n features, i.e., j=1... n. In case, the corresponding target and output values for each example are Ti and Oi respectively, hence the mean square error function E (in this case our object function) is defined as:

$E = \frac{1}{m} \sum_{i=1}^{m} (t_i - o_i)^2$

Wherein the output o_i is conditioned by a weighted linear combination of inputs, given by:

$o_i = w_0 + w_1 x_{i1} + w_2 x_{i2} + ... + w_n x_{in}$

The undisclosed variable in the above equation is the weight vector $w = [w1, w2, \ldots, wn]^T$

The objective function in this case is the mean square error with a gradient designated by:

$\nabla_w E(w) = -\sum_{i=1}^{m} (t_i - o_i) X_i$

Where x_i is the i-th example

A function to compute the gradient and a function to compute the mean square error are needed in this context.

```
# Input argument is weight and a tuple (train_data, target)
def grad_mse(w,xy):
    (x,y) = xy
    (rows,cols) = x.shape

    # Compute the output
    o = np.sum(x*w,axis=1)
    diff = y-o
    diff = diff.reshape((rows,1))
    diff = np.tile(diff, (1, cols))
    grad = diff*x
    grad = -np.sum(grad,axis=0)
    return grad

# Input argument is weight and a tuple (train_data, target)
def mse(w,xy):
    (x,y) = xy

    # Compute output
    # wer're using mse and not mse/m
    o = np.sum(x*w,axis=1)
    mse = np.sum((y-o)*(y-o))
    mse = mse/2
    return mse
def error(w,xy):
    (x,y) = xy
    o = np.sum(x*w,axis=1)

    #map the output values to 0/1 class labels
    ind_1 = np.where(o>0.5)
    ind_0 = np.where(o<=0.5)
    o[ind_1] = 1
    o[ind_0] = 0
    return np.sum((o-y)*(o-y))/y.size*100
```

Figure 2.2: Objective, gradient and error functions for mean square error Implementation

2.2.5 Running Gradient Descent on OCR (Optical character recognition) Dataset

The digits datasets included in sklearn.datasets were chosen in order to illustrate gradient descent on a classification problem

Shortly, a test run of gradient descent will be done on a two-class problem (digit 0 vs. digit 1).

The code below loads the digits. The method train-test-split from sklearn.model-selection is also needed to divide the training data into a train and a test set. The code below runs gradient descent on the training set, learns the weights, and schemes the mean square error at different iterations.

```
# Load the digits dataset with two classes
digits,target = dt.load_digits(n_class=2,return_X_y=True)
# Split into train and test set
x_train, x_test, y_train, y_test = train_test_split(
                      digits, target, test_size=0.2, random_state=10)

# Add a column of ones to account for bias in train and test
x_train = np.hstack((np.ones((y_train.size,1)),x_train))
x_test  = np.hstack((np.ones((y_test.size,1)),x_test))
```

Figure 2.3: Loading and splitting the OCR dataset

The below code demonstrates how the results are for Gradient descent:

```
# Initialize the weights and call gradient descent
rand = np.random.RandomState(19)
w_init = rand.uniform(-1,1,x_train.shape[1])*.000001
w_history,mse_history = gradient_descent(100,0.1,w_init,
                              mse,grad_mse,(x_train,y_train),
                             learning_rate=1e-6,momentum=0.7)

# Plot the MSE
plt.plot(np.arange(mse_history.size),mse_history)
plt.xlabel('Iteration No.')
plt.ylabel('Mean Square Error')
plt.title('Gradient Descent on Digits Data (Batch Version)')
plt.show()
```

Figure 2.4: Running gradient descent on the training set

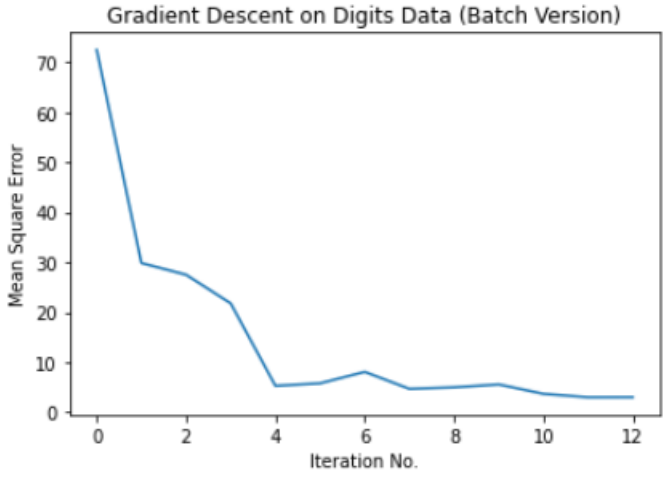

Figure 2.5: Plot of running the gradient descent function on the digits dataset

Below is a small function to compute the error rate of classification, which is called on the training and test set, which would help us check the error rate of our OCR on the training and test data :

```python
def error(w,xy):
    (x,y) = xy
    o = np.sum(x*w,axis=1)

    #map the output values to 0/1 class labels
    ind_1 = np.where(o>0.5)
    ind_0 = np.where(o<=0.5)
    o[ind_1] = 1
    o[ind_0] = 0
    return np.sum((o-y)*(o-y))/y.size*100
```

Figure 2.6: Function that computes the error between observations and results implementation

```python
train_error = error(w_history[-1],(x_train,y_train))
test_error = error(w_history[-1],(x_test,y_test))

print("Train Error Rate: " + "{:.2f}".format(train_error))
print("Test Error Rate: " + "{:.2f}".format(test_error))
```

```
Train Error Rate: 0.69
Test Error Rate: 1.39
```

Figure 2.7: Training and testing error rates

2.3 Stochastic Gradient Descent (SGD)

The word ‘stochastic‘ is defined as a system or a process that is interrelated with a random probability. For instance, in Stochastic Gradient Descent, some samples are selected randomly out of the whole data set in each iteration. In the case of Gradient Descent, the total number of samples from a dataset that is used for calculating the gradient per iteration. In typical Gradient Descent optimization, such as Batch Gradient Descent, the batch is considered the whole dataset. If the dataset size increases, the use of the whole dataset is controversial. It is important to note though that the use of the whole dataset is really useful for getting to the minimum in a less arbitrary and less random manner.

Assuming that a number of a million samples is available in our dataset. In case we use a typical Gradient Descent optimization technique, all of the one million samples have to be used in order to complete one iteration, while performing the Gradient Descent. The process needs to be replicated until the minimum is reached. This would, as a result, become computationally very expensive to perform.

The Stochastic Gradient Descent would solve this problem. In SGD, only a single sample is used,

a batch size of one, to perform each iteration. The sample is randomly stumbled and selected to perform the iteration.

Therefore, in SGD, it is easier to depict 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 selected randomly per iteration, the path taken by the algorithm to reach the minimum is usually noisier than the classical Gradient Descent algorithm. However, as long as the minimum is reached with a significantly shorter training time, the path taken by the algorithm does not matter

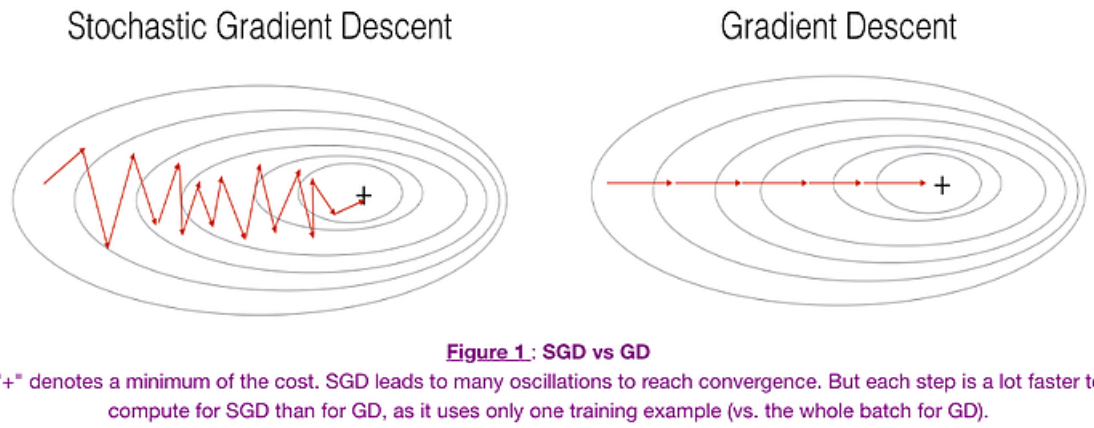

Figure 2.8: SGD vs GD Osillations

Ref image : Medium article

SGD usually took a higher number of iterations to reach the minimum , due to the fact that it is generally noisier than typical Gradient Descent. This could be explained by the arbitrariness in its descent. It is true that it requires a higher number of iterations to reach the minimum than typical Gradient Descent, but it is still computationally much less expensive than typical Gradient Descent. SGD is, overall, preferred over Batch Gradient Descent for learning algorithms optimization

2.3.1 Stochastic gradient algorithm

Stochastic gradient descent consists of the idea of local descent: we iteratively alter to decrease f(ω), until it becomes impossible. In this case, we have arrived at a local and possibly global minimum by chance. In stochastic gradient descent, the true value of the gradient of f(ω) is thus

approximated by the gradient of a single component of the sum; a single individual, selected at random, in our dataset. The method can then be much quicker than the classical gradient descent. This is particularly relevant for large datasets, where the number of updates is much higher

Pseudo Algorithm:

Input initial vector ω, j mini-batch size, learning rate η

1. **while** until convergence and k $\leq$ maximum number of iterations **do**
   (a) shuffle the dataset
   (b) **for** i= 1, 2, 3, . . ., n **do**
      - build a mini-lot with the individuals i to j
      - compute the scores for each individual of the mini-lot
      - compute the step with the truncated conjugate gradient from $\nabla f(w)$, the scores and η
      - w=w+pas
      - i=i+j
   (c) **end for**
2. **end while**

Output w,k

2.3.2 Stochastic Descent Python implementation

Inputs:

- **max-iterations**: Maximum number of iterations to run.
- **threshold**:Stop, in case the difference in function varies between two consecutive iterations falls below the aforementioned threshold.
- **w-init**:The Initial starting point of the gradient descent.
- **obj-func**:Function that calculates the objective function.

- **grad-func**:Function that calculates the gradient of the function.
- **extra-param**:Extra parameters if needed for the objective function and gradient function.
- **learning-rate**:Step size for gradient descent. It should be in [0,1].
- **momentum**:Momentum to use. It should be in [0,1].

Output:

- **w-history**:All points in space, visited by gradient descent wherein the objective function was assessed.
- **f-history**:Corresponding value of the objective function calculated per point.

```
# (xy) is the (training_set,target) pair
def stochastic_gradient_descent(max_epochs,threshold,w_init,
                                obj_func,grad_func,xy,
                                learning_rate=0.05,momentum=0.8):
    (x_train,y_train) = xy
    w = w_init
    w_history = w
    f_history = obj_func(w,xy)
    delta_w = np.zeros(w.shape)
    i = 0
    diff = 1.0e10
    rows = x_train.shape[0]

    # Run epochs
    while  i<max_epochs and diff>threshold:
        # Shuffle rows using a fixed seed to reproduce the results
        # Use the seed() method to customize the start number of the random number generator.
        np.random.seed(i)
        p = np.random.permutation(rows)

        # Run for each instance/example in training set
        for x,y in zip(x_train[p,:],y_train[p]):
            delta_w = -learning_rate*grad_func(w,(np.array([x]),y)) + momentum*delta_w
            w = w+delta_w

        i+=1
        w_history = np.vstack((w_history,w))
        f_history = np.vstack((f_history,obj_func(w,xy)))
        diff = np.absolute(f_history[-1]-f_history[-2])

    return w_history,f_history
```

Figure 2.9: The stochastic gradient descent function Implementation

2.3.3 Running Stochastic Descent on OCR(Optical character recognition) for Minimizing Mean Square Error:

The below code demonstrates how the results are for stochastic descent:

```
rand = np.random.RandomState(19)
w_init = rand.uniform(-1,1,x_train.shape[1])*.000001
w_history_stoch,mse_history_stoch = stochastic_gradient_descent(
                                    100,0.1,w_init,
                                  mse,grad_mse,(x_train,y_train),
                                 learning_rate=1e-6,momentum=0.7)

# Plot the MSE
plt.plot(np.arange(mse_history_stoch.size),mse_history_stoch)
plt.xlabel('Iteration No.')
plt.ylabel('Mean Square Error')
plt.title('Gradient Descent on Digits Data (Stochastic Version)')

plt.show()
```

Figure 2.10: Running the stochastic gradient descent function on the training set

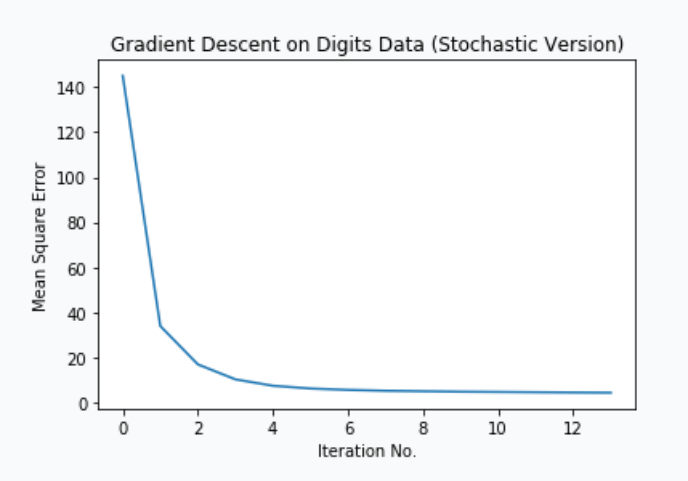

Figure 2.11: Plot of running the stochastic gradient descent function on the digits dataset

Let's also check the error rate:

```
train_error_stochastic = error(w_history_stoch[-1],(x_train,y_train))
test_error_stochastic = error(w_history_stoch[-1],(x_test,y_test))

print("Train Error rate with Stochastic Gradient Descent: " +
      "{:.2f}".format(train_error_stochastic))
print("Test Error rate with Stochastic Gradient Descent: "
      + "{:.2f}".format(test_error_stochastic))
```

```
Train Error rate with Stochastic Gradient Descent: 0.35
Test Error rate with Stochastic Gradient Descent: 1.39
```

Figure 2.12: Training and testing error rates

2.4 Comparing Batch and Stochastic Versions

2.4.1 Implementation of the comparison between GD and SGD

We will compare both the stochastic and batch versions of the gradient descent.

The learning rate will be fixed for both to the same value and we will vary the momentum to speed of convergence.We will keep the initial weights and the stopping criteria:

```
fig, ax = plt.subplots(nrows=3, ncols=1, figsize=(10,3))

rand = np.random.RandomState(11)
w_init = rand.uniform(-1,1,x_train.shape[1])*.000001
eta = 1e-6
for alpha,ind in zip([0,0.5,0.9],[1,2,3]):
    w_history,mse_history = gradient_descent(
                                100,0.01,w_init,
                              mse,grad_mse,(x_train,y_train),
                             learning_rate=eta,momentum=alpha)
    w_history_stoch,mse_history_stoch = stochastic_gradient_descent(
                                100,0.01,w_init,
                              mse,grad_mse,(x_train,y_train),
                             learning_rate=eta,momentum=alpha)

    # Plot the MSE
    plt.subplot(130+ind)
    plt.plot(np.arange(mse_history.size),mse_history,color='green')
    plt.plot(np.arange(mse_history_stoch.size),mse_history_stoch,color='blue')
    plt.legend(['batch','stochastic'])

    # Display total iterations
    plt.text(3,-30,'Batch: Iterations='+
             str(mse_history.size) )
    plt.text(3,-45,'Stochastic: Iterations='+
             str(mse_history_stoch.size))
    plt.title('Momentum = ' + str(alpha))

    # Display the error rates
    train_error = error(w_history[-1],(x_train,y_train))
    test_error = error(w_history[-1],(x_test,y_test))

    train_error_stochastic = error(w_history_stoch[-1],(x_train,y_train))
    test_error_stochastic = error(w_history_stoch[-1],(x_test,y_test))

    print ('Momentum = '+str(alpha))

    print ('\tBatch:')
    print ('\t\tTrain error: ' + "{:.2f}".format(train_error) )
    print ('\t\tTest error: ' + "{:.2f}".format(test_error) )

    print ('\tStochastic:')
    print ('\t\tTrain error: ' + "{:.2f}".format(train_error_stochastic) )
    print ('\t\tTest error: ' + "{:.2f}".format(test_error_stochastic) )


plt.show()
```

Figure 2.13: Comparison of the stochastic vs batch version Implementation

```
Momentum = 0
        Batch:
                Train error: 0.35
                Test error: 1.39
        Stochastic:
                Train error: 0.35
                Test error: 1.39
Momentum = 0.5
        Batch:
                Train error: 0.35
                Test error: 1.39
        Stochastic:
                Train error: 0.35
                Test error: 1.39
Momentum = 0.9
        Batch:
                Train error: 0.00
                Test error: 1.39
        Stochastic:
                Train error: 0.00
                Test error: 1.39
```

Figure 2.14: The comparison results of error rates with variant momentum

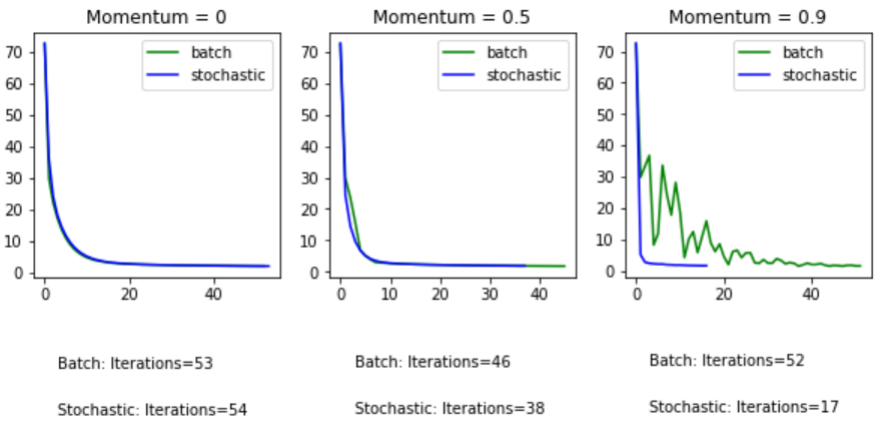

Figure 2.15: Plots of the comparison results of error rates with variant momentum

While there is no substantial difference in accuracy between the two versions of the classifier, the stochastic version clearly wins in terms of convergence speed. Fewer Iterations are required to accomplish the same outcome as in batch.

2.4.2 Summary of the comparison

The following table presents the result of the comparison and the implementation established above between both of the algorithms

	Batch Gradient Descent	Stochastic Gradient Descent
1	- Computes gradient using the whole Training sample	+ Computes gradient using a single Training sample
2	- Slow and computationally expensive algorithm	+ Faster and less computationally expensive than Batch GD
3	- Not suggested for huge training samples.	+ Can be used for large training samples.
4	+ Deterministic in nature.	- Stochastic in nature.
5	+ Gives optimal solution given sufficient time to converge.	- Gives good solution but not optimal.
6	- No random shuffling of points are required.	+ The data sample should be in a random order, and this is why we want to shuffle the training set for every epoch.
7-	- Can't escape shallow local minima easily.	+ SGD can escape shallow local minima more easily.
8-	- Convergence is slow.	+Reaches the convergence much faster.

Figure 2.16: Pros and Cons of batch and stochastic version

3

Simulation

3.1 Stochastic Processes Simulation

3.1.1 Brownian Motion or Wiener Process

Both names are used to describe the process. The process is a continuous-time stochastic process Lets represent the process with W_t, with the t index as the time

1. It starts with 0 or W_0=0.
2. The increments are independent from the past values $W_{t+u} - W_t$ for u > s > 0 are independent form W_s.
3. The increments have normal distribution $W_{t+u} - W_t \sim N(O, u) \sim N(O, u)$
4. W has a.s. continuous path
   Some of the basic properties are:
   - The mathematical expectation is 0 or E(Wt)=0.
   - The variance is 0 or Var(W_t)=t
   - The covariance is cov(W_s, W_t)=min(s,t)

- The correlation is corr(W_s, W_t)=min(s,t) / $\sqrt{st}$

Now we generate 50 values for x_t

```
for i in range(0, 50):
    xt = 0 + norm.rvs(scale=1**2* 4)
```

Figure 3.1: Generate 50 values for X_t

The GenerateBrownian Function is defined, with **Inputs**:

- x0
- n
- dt
- delta

And the formulais:

$X(t + dt) = X(t) + N(0, (delta)^2 dt; t, t + dt)$

```
def generate_brownian(x0, n, dt, delta, output=None):
    x0 = np.asarray(x0)
    r = norm.rvs(size=x0.shape + (n,), scale=delta* np.sqrt(dt))
    if output is None:
        output = np.empty(r.shape)

    np.cumsum(r, axis=-1, out = output)
    output += np.expand_dims(x0, axis=-1)

    return output
```

Figure 3.2: Generating Brownians Function

```
: #ploting the 50 simulations
x = np.empty((50,101))
x[:, 0] = 0

generate_brownian(x[:, 0], 100, 1, 4, output=x[:,1:])

time = np.linspace(0, 100, 101)
for k in range(50):
    plt.plot(time, x[k])
plt.title('Brownian Motion Simulation Multiple')
plt.xlabel('time')
plt.ylabel('x_values')
plt.grid(True)
plt.show()
```

Figure 3.3: Plot the 50 simulations

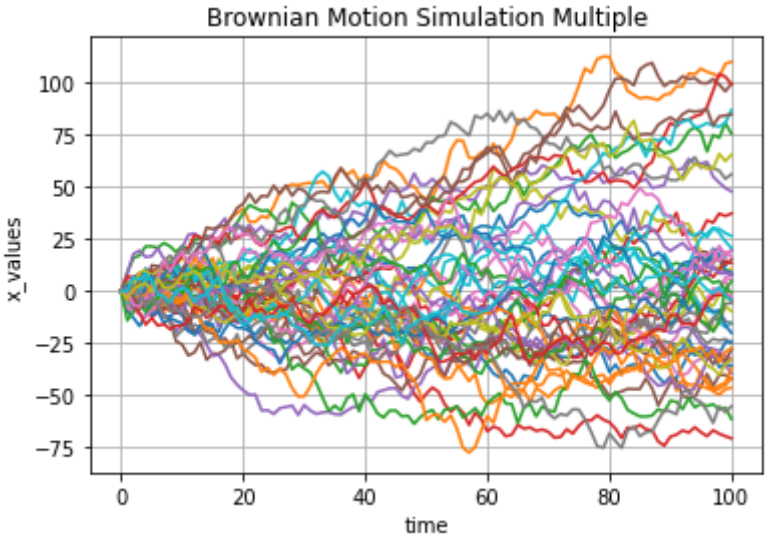

Figure 3.4: Brownian Motion multiple simulation

Finally we plot the 50 simulations, we start with $x_0 = 0$ and go on for 100 periods. Most simulations stay close to the baseline of 0, but some stray further away.

3.1.2 Poisson Process

We define a Poisson Process as random process when it satisfies the following conditions:

1. The first value is 0 or X(0)=0.

2. The increments are independent. This is also named lack of memory or Markov Property, meaning the the Markov Process is more general.

3. The probabilities of x = k follows the poisson distribution (a type of discrete probability distribution) with a probability mass function for k = 0, 1, 2 ...:
   $P(X = k) = \frac{\lambda^k e^{-\lambda}}{k!}$

The distribution can be visualized in the following picture. Lambda is also known as an intensity. The higher the lambda the more symmetric the distribution

Now we define the poisson process function with 2 arguments lambdas and number. The lambdas show different intensities, while the number is the time periods that are generated
We will check different outcomes.The output is generated following this function:
$X(t) = X0 + \sum_{k=1}^{t} Xk$

```
: def poisson_process(lambdas, number):
      X_T = np.random.poisson(lambdas, size=number)
      S = [np.sum(X_T[0:i]) for i in range(number)]
      X = np.linspace(0, number, number)
      graphs = [plt.step(X, S, label="Lambda = %d"%lambdas)[0] for i in range(lambdas)]
      graph = plt.step(X, S, label="Lambda = %d"%lambdas)
      plt.legend(handles=graph, loc=2)
      plt.title('Poisson Process')
      plt.xlabel('time')
      plt.ylabel('intensity')
```

Figure 3.5: Poisson process function

```
poisson_process(1, 20)
plt.show()
```

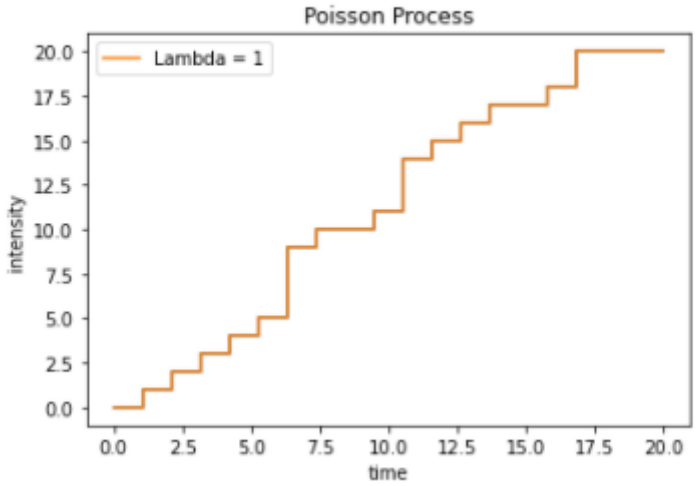

Figure 3.6: Generate the process

```
poisson_process(2, 20)
plt.show()
```

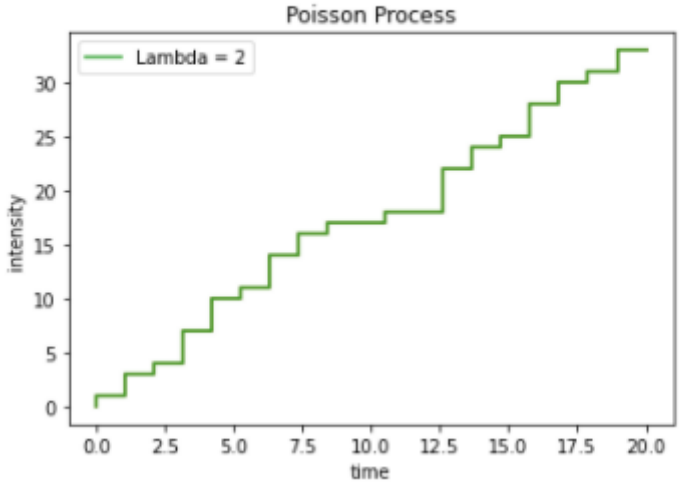

Figure 3.7: Increase the value of lamda

```
poisson_process(10, 50)
plt.show()
```

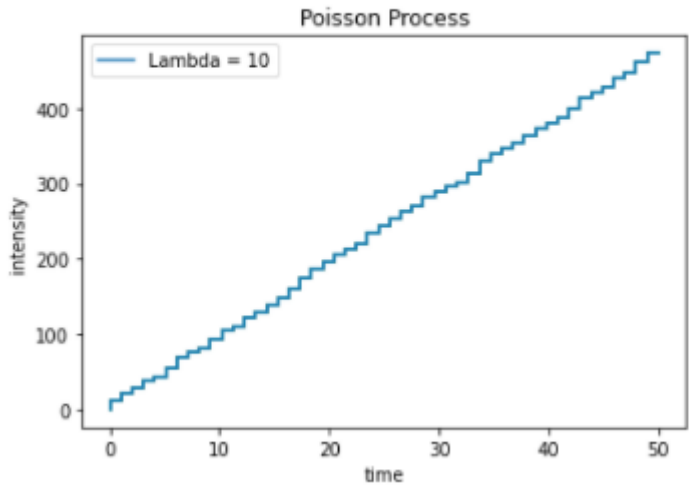

Figure 3.8: Increase the value of lamda and number

After generating the process we get several stepping functions. We can see that the higher the intensity the higher the values reached, same can be said for the time. The higher the intensity the higher the steps.

4

Conclusion

In this report , we have initially looked at the principles of two variants of gradient descent : Batch and Stochastic version.

We have then compared them on OCR Dataset where we tried to minimize the mean squared error .Based on the given result we have established the table of comparison of both of the Algorithms .It is obvious that the stochastic version is a clear winner when we are interested in the speed of convergence while maintaining the same accuracy level as for the batch version .

The SGD version can also be optimized and reach higher rates by the use of other algorithms for optimizing like : Momentum(like we did in the comparison by changing the momentum value for both of the algorithms), Nesterov accelerated gradient, Adagrad, Adadelta, RMSprop, Adam, as well as different algorithms to optimize asynchronous SGD.

Finally, We can also consider other strategies to improve SGD such as shuffling and curriculum learning, batch normalization, and early stopping.

Bibliography

https://medium.com/analytics-vidhya/journey-of-gradient-descent-from-local-to-global-c851eba3d367