Queen's University, Kingston, Ontario

: Convergence of Stochastic Gradient Descent

Making Gradient Descent Optimal for Strongly Convex Stochastic Optimization

Professor Bahman Gharesifard

May 1, 2021

Amean Asad

February 2021

Introduction

This paper [1] provides results for recovering an $O(\frac{1}{T})$ convergence rate for Stochastic Gradient Descent applied to smooth and non-smooth strongly convex functions. I will start this by motivating an example to show why this result is significant in the topic of optimization. Let's consider the finite-sum optimization problem:

$$\arg\min_{x \in C} F(x) = \frac{1}{n} \sum_{i=1}^{N} f_i(x), \ C \subseteq \mathbb{R}^d$$

$f_i(x)$ is loss function, where d is known as the number of features and N is the number of data samples. The task is to find an optimal $x^* \in C$ that minimizes this loss function with respect to all our data samples. Solving this optimization problem is one of the core tasks in machine learning. One of the algorithms discussed in class, Gradient Descent (GD), is widely used to approach this problem [2]. We know using the **Descent Lemma** that under certain conditions for $F(x)$, Gradient Descent converges to a minimizer with convergence rate $O(\frac{1}{T})$. However as the number of data samples n grows, the time and space complexity of computing the gradient grows linearly with the size of the data-set. With a convergence rate of $O(\frac{1}{T})$, to get an error of $\epsilon > 0$, it requires $t = \frac{1}{\epsilon}$ iterations. Let's assume the cost of computing the gradient $\nabla f_i(x)$ is D, then for every iteration of gradient descent we have to compute $n \cdot D$ iterations. Hence for error rate ϵ, the time complexity is $O(D \cdot n \cdot t)$. Datasets are now in the order of millions of samples, which makes it a difficult task for gradient descent.

Stochastic Gradient Descent (SGD) is a stochastic variation of Gradient Descent that addresses this issue. In SGD, a random variable distribution is used to pick i from $\{1, \cdots, N\}$ and the iteration step becomes:

$$x_{k+1} = P_C(x_k - \alpha_k \nabla f_i(x_k))$$

$P_C(x)$ is a projection operator used to address constrained problems. SGD makes the cost of computing the derivative constant, so for t iterations, SGD has a cost of $O(D * t)$, which is n times less than GD. For $F(x)$ that is strongly-convex and smooth, the authors show that SGD does converge to a minimizer. Other Stochastic Optimization algorithms recently developed have a convergence of rate of $O(\frac{1}{T})$, while it is known that for strongly-convex functions, SGD has a sub-optimal convergence rate of $O(\frac{log(T)}{T})$ [3]. This paper aims to recover the $O(\frac{1}{T})$ rate for strongly convex functions (smooth and non-smooth) for SGD. This result renders SGD as an optimal algorithm that is able to reduce the time complexity of GD while maintain the same convergence rate.

Convergence Rate Proof

I will prove that we can recover a $O(\frac{1}{T})$ convergence rate for functions that are strongly convex and smooth with respect to the minimizer as per [1] . The theorem to be proved is the following:

Theorem 1. *Let $f : C \mapsto \mathbb{R}$, $C \subseteq \mathbb{R}^n$ is a convex set and let x^* be a global minimizer for f over C. Suppose the following:*

(i) f is λ-strongly convex:

$$f(y) \geq f(x) + \langle g(x), y - x \rangle + \frac{\lambda}{2}||y - x||^2,\ x, y \in C,\ \lambda \in \mathbb{R}_{>0}$$

(ii) μ-smooth with respect to x^*:

$$f(x) - f(x^*) \leq \frac{\mu}{2}||x - x^*||^2,\ \in \mathbb{R}_{>0}$$

(iii) $\mathbb{E}[||\hat{g}_t(x)||^2] \leq B^2$ for $B \in \mathbb{R}_{>0}$ where $\hat{(g)}(x)$ is a stochastic gradient oracle of f at the point x

(iv) The learning rate $\alpha(t) = \frac{1}{\lambda t}$

Then $\forall\ T \in \mathbb{R}_{\geq 0}$ we have that:

$$\mathbb{E}[f(x_T) - f(x^*)] \leq \frac{2\mu B^2}{\lambda^2 T}$$

I will use the following proposition as part of the sketch of the proof. This proposition shows how some results from Optimization Theory are directly involved in proving the results in presented in the paper.

Proposition 1. *Let $C \subseteq \mathbb{R}^d$ be a convex set and let $P_C(y)$ be the projection of the point $y \in \mathbb{R}^d$ onto the set C. Then we have that: $||P_C(x) - x|| \leq ||x - y||$ for all $x, y \in \mathbb{R}^d$*

Proof. The projection is a continuous and differentiable function defined on a convex set, hence we can use the **Variational Inequality**. We have that:

$$\begin{aligned}
0 &\leq \langle P_C(y) - y, x - P_C(y) \rangle \\
\implies 0 &\leq \langle y - P_C(y), P_C(y) - x \rangle \\
&= \langle (y - x) - (P_C(y) - x), P_C(y) - x \rangle = \langle y - x, P_C(y) - x \rangle - ||P_C(y) - x||^2 \\
\implies ||P_C(y) - x||^2 &\leq \langle y - x, P_C(y) - x \rangle \leq ||y - x|| \cdot ||P_C(y) - x|| \text{ By } Cauchy - Schwarz \\
\implies ||P_C(y) - x|| &\leq ||y - x||
\end{aligned}$$

□

Another important result from Optimization Theory that is used for this proof is the choice of the learning rate α_t. We know that if the sequence $\{\alpha_t\}_{t=1}^{\infty} \longrightarrow 0$ and the series $\sum_{t=1}^{\infty} \alpha_t = +\infty$, then our descent steps converge to a critical point. Also, putting a condition on the learning rate such that $\sum_{t=1}^{\infty} \alpha_t^2 < \infty$. The choice of $\alpha_t = \frac{1}{\lambda t}$ satisfies all three of these conditions because the sequence $\{\frac{1}{t}\}_{t=1}^{\infty} \longrightarrow 0$ and we have that $\sum_{t=1}^{\infty} (\frac{1}{t})^p$ is a $p-$series that diverges when $p = 1$ and converges when $p = 2$. This result guarantees that our choice of learning rate is suitable and the descent steps do take us to the minimizer using this step size.

Definition 1. ***Stochastic Gradient Oracle***
Let $f : C \mapsto R$ where $C \subseteq \mathbb{R}^n$ and let $x \in C$. A stochastic gradient oracle is a maps the point x to a vector $\hat{g} \in C$ such that the expectation $\mathbb{E}[\hat{g}] = g(x)$ where $g(x)$ is a sub-gradient of f evaluated at the point x [4]

Proof. The authors use an induction argument to first prove a **Lemma** that states that $\mathbb{E}[x_T - x^*] \leq \frac{4B^2}{\lambda^2 T}$. Once this Lemma is proven, the authors simply apply the definition of $\mu-$ smoothness to conclude the proof. I will go over the induction proof since it is not clearly demonstrated in the paper and it uses results from the course.

(i) Base Case $T = 1$:
Here we use the properties of strong convexity. From the definition of λ-strongly convex, we have that:

$$f(x^*) \geq f(x_1) + \langle g(x_1), x^* - x_1 \rangle + \frac{\lambda}{2}||x^* - x_1||^2 \tag{1}$$

where $g(x_1)$ is a sub-gradient of the function f at the point x_1. Since x^* is a global minimizer, we have that $f(x^*) \geq f(y) \implies f(x_1) - f(x^*) \geq 0$. Therefore, we can re-arrange (1) to get the following:

$$\begin{aligned}
&0 \leq f(x_1) - f(x^*) \leq -\langle g(x_1), x^* - x_1 \rangle - \frac{\lambda}{2}||x^* - x_1||^2 \\
&\implies 0 \leq -\langle g(x_1), x^* - x_1 \rangle - \frac{\lambda}{2}||x^* - x_1||^2 \\
&\implies \frac{\lambda}{2}||x^* - x_1||^2 \leq -\langle g(x_1), x^* - x_1 \rangle = \langle g(x_1), x_1 - x^* \rangle \\
&\implies \frac{\lambda}{2}||x^* - x_1||^2 \leq \langle g(x_1), x_1 - x^* \rangle \leq ||g(x_1)|| \cdot ||x_1 - x^*|| \text{ By } Cauchy - Schwarz \\
&\implies \frac{\lambda^2}{4}||x^* - x_1||^2 \leq ||g(x_1)||^2
\end{aligned}$$

The authors then apply the expected value to both sides, and use the property that $\mathbb{E}[||g(x)||] \leq \mathbb{E}[||\hat{g}(x)||]$ along with assumption (iii) in the theorem statement above to get that:

$$\mathbb{E}[||x^* - x_1||^2] \leq \mathbb{E}[\frac{4}{\lambda^2}||g(x_1)||^2] = \frac{4}{\lambda^2}\mathbb{E}[||g(x_1)||^2] \leq \frac{4}{\lambda^2}\mathbb{E}[||\hat{g}(x_1)||^2] \leq \frac{B^2 4}{\lambda^2}$$

(ii) We assume true for step $T = k$ that $\mathbb{E}[||(x_k - x^*)||^2] \leq \frac{4B^2}{\lambda^2 k}$

(iii) For step $T = k + 1$, we have that

$$\begin{aligned}
\mathbb{E}[||x_{k+1} - x^*||^2] &= \mathbb{E}[||P_C(x_k - \alpha_k \hat{g}_k) - x^*||^2] \\
&\leq \mathbb{E}[||x_k - \alpha_k \hat{g}_k - x^*||^2] \text{ By } \textbf{Proposition 1} \\
&= \mathbb{E}[||(x_k - x^*) - \alpha_k \hat{g}_k||^2] \\
&= \mathbb{E}[||(x_k - x^*)||^2] - 2\alpha_k \mathbb{E}[\langle g(x_k), x^* - x_k \rangle] + \mathbb{E}[\alpha_k \hat{g}_k||^2] \text{ By } \textbf{Definition 1} \\
&\leq \mathbb{E}[||(x_k - x^*)||^2] - 2\alpha_k \mathbb{E}[f(x_k) - f(x^*) + \frac{\lambda}{2}||x^* - x_1||] + \alpha_k^2 \mathbb{E}[||\hat{g}_k||^2] \\
&\leq \mathbb{E}[||(x_k - x^*)||^2] - 2\alpha_k \mathbb{E}[\lambda||x^* - x_1||^2] + \alpha_k^2 \mathbb{E}[||\hat{g}_k||^2]
\end{aligned}$$

The last two inequalities follow from applying the definition of strong convexity and smoothness. The authors then use the assumptions from the theorem statement combined with the assumption on induction step $T = K$

$$\mathbb{E}[||x_{k+1} - x^*||^2] \leq \mathbb{E}[||(x_k - x^*)||^2](1 - 2\alpha_k \lambda) + \alpha_k^2 B^2 \leq \frac{2\mu B^2}{\lambda^2 k}(1 - \frac{2}{k}) + \frac{B^2}{\lambda^2 k^2} = \frac{4B^2}{\lambda^2 k}$$

As claimed, the authors conclude the proof by applying the definition μ-smoothness to the result of the induction of proof to get that:

$$\mathbb{E}[f(x_t) - f(x^*)] \leq \frac{\mu}{2}\mathbb{E}[||x_{t+1} - x^*||^2] \leq \frac{2\mu B^2}{\lambda^2 t}$$

□

Application of Stochastic Gradient Descent

Stochastic Gradient Descent is one of the central algorithms that is used for Machine Learning [2]. There are countless ways this algorithm can be applied in the machine learning world. I will focus on the area of applying this algorithm in Neural Networks for Computer Vision. I be using the example of recognizing handwritten digits using the MNIST dataset. The MNIST dataset is a sufficient benchmark dataset that is excellent at demonstrating the power and speed of SGD. I will be using an Least Squares regression as the cost function for this network:

$$C_0(W) = \frac{1}{2n}[\sum_{i=1}^{n}(\sigma_w(x^{(i)}) - y^{(i)})^2 + \lambda\sum_{j=1}^{m} w_j^2]$$

We can regularize Least Squares using $L-2$ Regularization to give us a strongly convex cost function [5] defined as:

$$C(W) = C_0(W) + \frac{1}{2n}[\lambda\sum_{j=1}^{m} w_j^2]$$

Here W is characterized as our weights, σ_w is the activation function, and x and y are the sets of training data. I will now go over some of the machinery of neural networks. A neural network with L layers has $L-1$ weight matrices. Suppose layer L has a_j neurons and layer $L+1$ has k neurons then the weight matrix W^{L+1} mapping $a_j^L \mapsto a_k^{L+1}$ has dimensions $(L+1) \times (L)$. We define a_j^L as the j^th activation neuron in layer L, w_{jk}^L as the weight from the k^{th} activation in the $L-1$ layer to the j^{th} in layer L.
For this example, the choice of activation function will be a sigmoid activation function defined as:

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

To forward propagate in the network we have that:

1. $z_j^L = \sum_{k=1}^{n} w_{jk}^L \cdot a_k^{L-1}$ where n is the number of neurons in layer $L-1$
2. $a_j^L = \sigma(z_j^L)$
3. $a^L = \sigma(W^L \cdot a^{L-1})$

To retrieve the derivatives for the weights in the network, we use the following 3 computations

1. $\delta_j = \frac{\partial C}{\partial z_j}$ is defined as the error in layer j
2. $\delta_j = (W^T \cdot \delta_{j+1}) \odot \sigma(z_j)$

3. $\frac{\partial C}{\partial w^L_{jk}} = a^{L-1}_k \delta_j$

Our update rule for gradient descent becomes then:

$$w_{t+1} = w_t - \alpha_t \frac{\partial C_0}{\partial w_t} - \frac{\alpha_t \lambda}{n} w_t$$

I will be using a version of SGD called mini-batch SGD where instead of querying 1 random sample out of our dataset, we query m random variables out of the and dataset.

Here is an overview of SGD that I will be applying:

Algorithm 1: Mini Batch Stochastic Gradient Descent

Data: Initialize input data x_0, α_t, Batch Size m
while *x_t not converged* **do**
  $t = t + 1$;
  Query m random samples from the data x_0;
  for *sample in samples* **do**
    Forward Propagate to find the cost using $a^L = \sigma(W^L \cdot a^{L-1})$;
    Back-propagate to find the errors in each layer using $\delta_j = (W^T \cdot \delta_{j+1}) \odot \sigma(z_j)$;
  end
  Update the weights using the sample gradients: $w_{t+1} = w_t - \alpha_t \sum_{i=1}^{m} \frac{\partial C_0(x_i)}{\partial w_t}$;
end

I wrote a code implementation from scratch in order to demonstrate the effectiveness of SGD and the speed of SGD versus regular gradient descent. I did not use any external machine learning libraries. I initialized a neural network with 3 layers to predict the hand-written digits. I ran a version of SGD for 20 epochs with a batch size of 20 and and $\alpha_t = 8$. Unfortunately, due to my computer limits, I am unable to run the complete version of gradient descent due to how much time it would take, but I am able to mimick gradient descent by vastly increasing the batch size. To mimick gradient descent, I ran the algorithm with a batch size of 200, which increases the gradient computation by 10 folds each iteration. This should be enough to show us if mini batch SGD is able to provide the same convergence results for way less computational overhead.

Figure 1 and Figure 2 show the results of my neural network training steps. We see that the cost function for GD and SGD converges almost to the same value. SGD converges to 0.92 while *GD* converges to 0.93. The classification accuracy of SGD ends up being 94.3% and GD is 94.4%. So in terms of convergence results, the two algorithms seem identical. The average time per epoch for SGD was 7.45 seconds while the average time per epoch for GD was 820 seconds. This example has shown two important things. Firstly is that SGD is a robust algorithm to use for machine learning tasks. Secondly is that we are able to recover the same convergence rate for SGD as GD while significantly reducing the time complexity of our neural network.

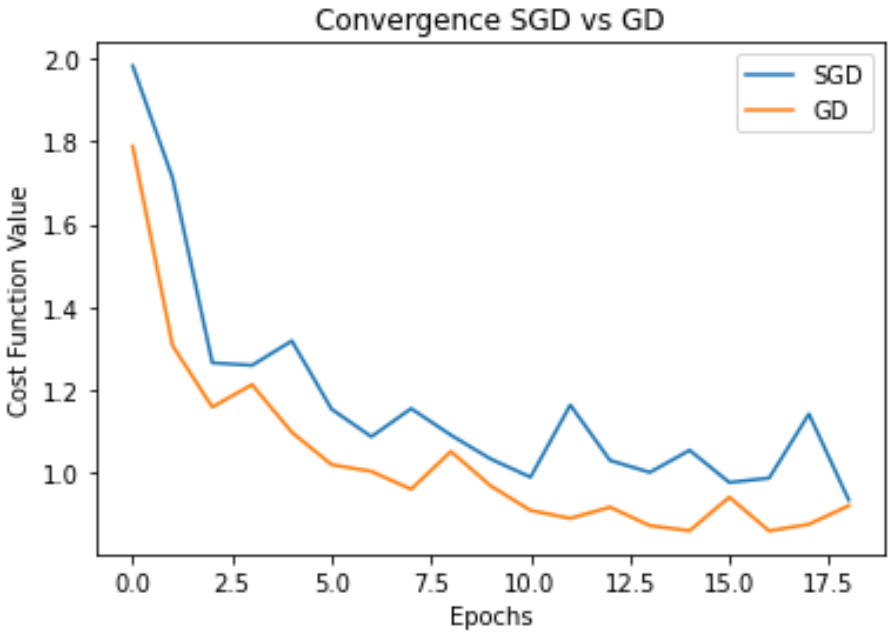

Figure 1: Convergence Comparison

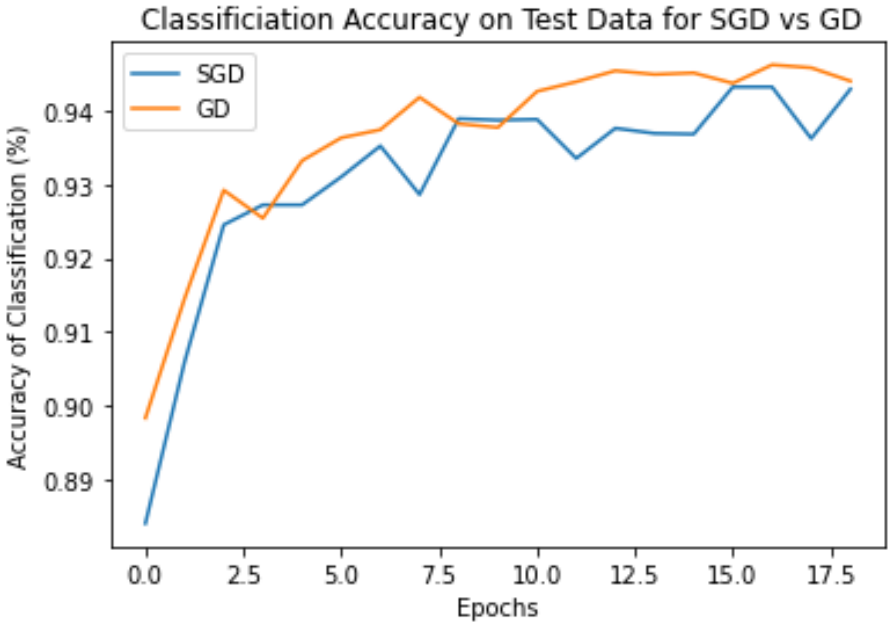

Figure 2: Classification Accuracy Comparison

Bibliography

[1] Alexander Rakhlin, Ohad Shamir, and Karthik Sridharan. "Making Gradient Descent Optimal for Strongly Convex Stochastic Optimization". In: *Proceedings of the 29th International Conference on Machine Learning, ICML 2012* 1 (Sept. 2011), pp. 449–456. URL: http://arxiv.org/abs/1109.5647.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[3] Ohad Shamir and Tong Zhang. *Stochastic Gradient Descent for Non-smooth Optimization: Convergence Results and Optimal Averaging Schemes*.

[4] Amitabh Basu. *Notes on stochastic gradient descent*. 2020.

[5] Mark Schmidt. *Rates of Convergence Linear Convergence of Gradient Descent CPSC 540: Machine Learning Rates of Convergence*. University of British Columbia, 2019.

Code for Neural Network

```
# −∗− coding: utf−8 −∗−
"""
Created on Wed April 20 23:07:05 2021

@author: amean
"""
import numpy as np
import random
import mnistLoader
import time
import matplotlib.pyplot as plt

class Network(object):

    def __init__(self, size, activationFunction):

        self.layers = len(size)
        self.size = size
        self.activationFunction = activationFunction
        self.biases = [np.random.randn(x,1) for x in size[1:]]
        self.weights = [np.random.randn(m,n) for m,n in zip(size[1:], size[:−1])]



    def costFunction(self, X, Y):
        row, col = np.shape(X)
        MSE = (0.5)∗(X−Y)∗∗2
        return (1.0/row)∗np.sum(MSE)



    def forwardPropagation(self,inputs):
        current_activation = inputs
        activations = [current_activation]
        deltaSigmoids = []
        for layer in range(self.layers−1):

            Z = np.dot(self.weights[layer], current_activation) + self.biases[layer]
            activation = self.sigmoid(Z)
            activations.append(activation)
            deltaSigmoids.append(sigmoidDash(Z))
            current_activation = activation
        return current_activation, activations, deltaSigmoids
```

```
    def backPropagation(self, deltaSigmoids, activations, y):
        propagations = self.layers - 2
        deltaWeights =[np.zeros(weightMatrix.shape, float) for weightMatrix in self.
            weights]
        deltaBiases = [np.zeros(biasVector.shape, float) for biasVector in self.biases]

        #Activation layer is always 1 + wieght layers
        currentActivationLayer = activations[propagations + 1][:][-1]
        currentDeltaSigmoidLayer = deltaSigmoids[propagations][:][-1]
        currentDeltaLayer = np.multiply((currentActivationLayer - y),
            currentDeltaSigmoidLayer)

        deltaWeights[-1] = np.matmul(currentDeltaLayer, np.transpose(activations
            [-2]))
        deltaBiases[-1] = currentDeltaLayer

        for layer in range(0, propagations):
            weightTranspose = np.transpose(self.weights[-layer - 1])
            intermediateStep = np.matmul(weightTranspose, currentDeltaLayer)
            delta = np.multiply(intermediateStep, deltaSigmoids[-layer])
            deltaWeights[-layer-2] = np.matmul(delta, np.transpose(activations[-
                layer - 3]))
            deltaBiases[-layer-2] = delta
            currentDeltaLayer = delta

        return (deltaWeights, deltaBiases)

    def evaluate(self, test_data):
        n = len(test_data)
        test_results = [(np.argmax(self.forwardPropagation(x)), y)
                        for (x, y) in test_data]

        cost = (1/(n+1))*sum((x-y)**2 for (x,y) in test_results)
        accuracy = sum(int(x == y) for (x, y) in test_results)/n
        return (cost, accuracy)

    def sigmoid(self,z):
        return 1.0/(1.0 + np.exp(-z))

    def updateWeights(self, batch, eta):

        totalDeltaWeights = [np.zeros(weightMatrix.shape, float) for weightMatrix in
            self.weights]
#
        totalDeltaBiases = [np.zeros(biasVector.shape, float) for biasVector in self.
            biases]

        for x,y in batch:
```

```
            deltaWeights, deltaBiases = self.backPropagation(x,y)
            totalDeltaWeights = [tdw + dw for tdw, dw in zip(totalDeltaWeights,
                deltaWeights)]
            totalDeltaBiases = [tdb + db for tdb, db in zip(totalDeltaBiases,
                deltaBiases)]

        self.weights = [w − (eta/len(batch))∗tdw for w, tdw in zip(self.weights,
            totalDeltaWeights)]
        self.biases = [w − (eta/len(batch))∗tdb for w, tdb in zip(self.biases,
            totalDeltaBiases)]


def stochasticGD(self, trainingData, epochs, batchSize, eta, testData):

    data = list(trainingData)
    testData = list(testData)
    times = []
    costs = []
    accuracies = []
    for i in range(epochs):
        random.shuffle(data)
        batches = [
                    data[k: k + batchSize]
                    for k in range(0, len(data), batchSize)]
        start = time.time()
        for batch in batches:
            self.updateWeights(batch, eta)
        finish = round(time.time() − start,4);
        cost, accuracy = self.evaluate(testData)
        times.append(finish)
        costs.append(cost)
        accuracies.append(accuracy)
        print("Epoch␣{}␣:␣{}␣/␣{}".format(i,accuracy, len(testData)));
    return times, costs, accuracies


def gradientDescent(self, trainingData, epochs, batchSize, eta, testData):

    data = list(trainingData)
    testData = list(testData)
    times = []
    costs = []
    accuracies = []
    for i in range(epochs):
        random.shuffle(data)
        batches = [
                    data[k: k + 200]
                    for k in range(0, len(data), 20)]
```

```
            print(len(batches))
            start = time.time()
            for batch in batches:
                self.updateWeights(batch, eta)
            finish = round(time.time() - start,4);
            cost, accuracy = self.evaluate(testData)
            times.append(finish)
            print(finish)
            costs.append(cost)
            accuracies.append(accuracy)
            print("Epoch {} : {} / {}".format(i,accuracy, len(testData)));
        return times, costs, accuracies

def sigmoid(z):
    return 1.0/(1.0+np.exp(-z))

def sigmoidDash(z):
    return sigmoid(z)*(1.0-sigmoid(z))



GdNet = Network([784,30,10, 10], sigmoid)
data = mnistLoader.load_data_wrapper()


epochs = list(range(20))
times, costs, accuracy = GdNet.stochasticGD(data[0], 20, 20, 8, data[2])

SGDNet = Network([784,30,10, 10], sigmoid)

times2, costs2, accuracy2 = SGDNet.gradientDescent(data[0], 20, 200, 8, data[2])

plt.figure()
plt.plot(epochs, accuracy, label="SGD");
plt.plot(epochs, accuracy2, label="GD");
plt.legend()
plt.xlabel("Epochs")
plt.ylabel("Accuracy of Classification (%)")
plt.title(" Classificiation Accuracy on Test Data for SGD vs GD")



plt.figure()
plt.plot(epochs, costs, label="SGD");
plt.plot(epochs, costs2, label="GD");
plt.legend()
plt.xlabel("Epochs")
plt.ylabel("Cost Function Value")
plt.title("Convergence SGD vs GD")
```

```
plt.show()


timeAvg = (1/len(epochs))*sum(times)
time2Avg = (1/len(epochs))*sum(times2)
```