Improvements in Algorithm for Approximation using Neural Network

February 23, 2018

1 Task: Approximate the sin function to the following neural network configuration using various gradient descent algorithms.

This will be done using ordinary gradient descent, conjugate gradient descent, stochastic gradient descent, and stochastic conjugate gradient descent.

2 Preliminary Work: Import packages and set up general functions.

```
In [1]: import time
        import math
        from sympy import *
        import numpy as np
        import matplotlib
        import matplotlib.pyplot as plt
        %matplotlib inline
```

Let $\mu(x)$ be defined as the output of the neural network where x is the input value:

$$\mu(x) = \alpha^0\sigma(\theta_0^0 + \theta_1^0 x) + \alpha^1\sigma(\theta_0^1 + \theta_1^1 x) + \alpha^2\sigma(\theta_0^2 + \theta_1^2 x) + \alpha^3\sigma(\theta_0^3 + \theta_1^3 x)$$

and $\sigma(x)$ is the sigmoid function:

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

```
In [2]: def mu(x,var_arr):

            y = 0
            for i in range(4):
                y += var_arr[0][i] * sigmoid(x, var_arr[1][i], var_arr[2][i])

            return y
```

To aid the multiple arguments in the sigmoid and computation of partial derivative of sigmoid function, let's define two exponential function:

$$\text{expOne}(x,y,z) = e^{(y+zx)}$$
$$\text{expTwo}(x,y,z) = e^{-(y+zx)}$$

```
In [3]: def expOne(x,y,z):
            return (math.exp(y + (z*x)))
```

```
In [4]: def expTwo(x,y,z):
            return (math.exp(-1 * (y + z*x)))
```

Therefore the sigmoid function will be defined as a function of 3 arguments:

$$\sigma(x,y,z) = \frac{1}{1+\text{expTwo}(x,y,z)}$$

```
In [5]: def sigmoid(x,y,z):
            return (1 + expTwo(x,y,z))**-1
```

The cost function, which minimizes the distance between $\sin(x)$ and $\mu(x)$ from a total of 100 points, is as follows:

$$C(\alpha^0,\alpha^1,\alpha^2,\alpha^3,\theta_0^0,\theta_0^1,\theta_0^2,\theta_0^3,\theta_1^0,\theta_1^1,\theta_1^2,\theta_1^3,\theta_2^0,\theta_2^1,\theta_2^2,\theta_2^3) = \frac{1}{2}\sum_{x=0}^{99}(\mu(x)-\sin(x))^2$$

```
In [6]: def cost(var_arr):

            output = 0.0
            interval = np.linspace(0,2*np.pi, 100)

            for i in interval:
                output += (mu(i,var_arr) - np.sin(i))**2

            return 0.5*output
```

In the stochastic cost function, only the cost at certain randomized points will be summed together.

```
In [7]: def stocCost(var_arr,xvalues):

            output = 0.0

            for i in xvalues:
                output += (mu(i,var_arr) - np.sin(i))**2

            return 0.5*output
```

The partial derivatives are defined as follows:

$$\frac{\partial C}{\partial \alpha^0} = (\mu(x)-\sin(x))\sigma(\theta_0^0+\theta_1^0 x) \quad \frac{\partial C}{\partial \theta_0^0} = \frac{\alpha^0(\mu(x)-\sin(x))e^{\theta_0^0+x\theta_1^0}}{(1+e^{\theta_0^0+x\theta_1^0})^2} \quad \frac{\partial C}{\partial \theta_1^0} = \frac{x\alpha^0(\mu(x)-\sin(x))e^{\theta_0^0+x\theta_1^0}}{(1+e^{\theta_0^0+x\theta_1^0})^2}$$

$$\frac{\partial C}{\partial \alpha^1} = (\mu(x)-\sin(x))\sigma(\theta_0^1+\theta_1^1 x) \quad \frac{\partial C}{\partial \theta_0^1} = \frac{\alpha^0(\mu(x)-\sin(x))e^{\theta_0^1+x\theta_1^1}}{(1+e^{\theta_0^1+x\theta_1^1})^2} \quad \frac{\partial C}{\partial \theta_1^1} = \frac{x\alpha^0(\mu(x)-\sin(x))e^{\theta_0^1+x\theta_1^1}}{(1+e^{\theta_0^1+x\theta_1^1})^2}$$

$$\frac{\partial C}{\partial \alpha^2} = (\mu(x)-\sin(x))\sigma(\theta_0^2+\theta_1^2 x) \quad \frac{\partial C}{\partial \theta_0^2} = \frac{\alpha^0(\mu(x)-\sin(x))e^{\theta_0^2+x\theta_1^2}}{(1+e^{\theta_0^2+x\theta_1^2})^2} \quad \frac{\partial C}{\partial \theta_1^2} = \frac{x\alpha^0(\mu(x)-\sin(x))e^{\theta_0^2+x\theta_1^2}}{(1+e^{\theta_0^2+x\theta_1^2})^2}$$

$$\frac{\partial C}{\partial \alpha^3} = (\mu(x)-\sin(x))\sigma(\theta_0^3+\theta_1^3 x) \quad \frac{\partial C}{\partial \theta_0^3} = \frac{\alpha^0(\mu(x)-\sin(x))e^{\theta_0^3+x\theta_1^3}}{(1+e^{\theta_0^3+x\theta_1^3})^2} \quad \frac{\partial C}{\partial \theta_1^3} = \frac{x\alpha^0(\mu(x)-\sin(x))e^{\theta_0^3+x\theta_1^3}}{(1+e^{\theta_0^3+x\theta_1^3})^2}$$

```
In [8]: def gradient(var_arr):

            grad_arr = np.zeros((3,4))
            interval = np.linspace(0,2*np.pi, 100)

            for x in interval:
                value = mu(x, var_arr) - np.sin(x)
                for i in range(4):
                    grad_arr[0][i] += (value * sigmoid(x,var_arr[1][i],var_arr[2][i]))
                    temp = expOne(x,var_arr[1][i],var_arr[2][i])
                    tempTwo = (value * temp * var_arr[0][i]) / ((1 + temp)**2)
                    grad_arr[1][i] += tempTwo
                    grad_arr[2][i] += x * tempTwo

            return grad_arr
```

In the stochastic gradient function, only the gradient at certain randomized points will be summed together.

```
In [9]: def stocGrad(var_arr,interval):

            grad_arr = np.zeros((3,4))

            for x in interval:
                value = mu(x, var_arr) - np.sin(x)
                for i in range(4):
                    grad_arr[0][i] += (value * sigmoid(x,var_arr[1][i],var_arr[2][i]))
                    temp = expOne(x,var_arr[1][i],var_arr[2][i])
                    tempTwo = (value * temp * var_arr[0][i]) / ((1 + temp)**2)
                    grad_arr[1][i] += tempTwo
                    grad_arr[2][i] += x * tempTwo

            return grad_arr
```

Note: The 12 variables and its gradient will be stored as follows:

$$\begin{bmatrix} \alpha^0 & \alpha^1 & \alpha^2 & \alpha^3 \\ \theta_0^0 & \theta_0^1 & \theta_0^2 & \theta_0^3 \\ \theta_1^0 & \theta_1^1 & \theta_1^2 & \theta_1^3 \end{bmatrix} \quad \begin{bmatrix} \frac{\partial C}{\partial \alpha^0} & \frac{\partial C}{\partial \alpha^1} & \frac{\partial C}{\partial \alpha^2} & \frac{\partial C}{\partial \alpha^3} \\ \frac{\partial C}{\partial \theta_0^0} & \frac{\partial C}{\partial \theta_0^1} & \frac{\partial C}{\partial \theta_0^2} & \frac{\partial C}{\partial \theta_0^3} \\ \frac{\partial C}{\partial \theta_1^0} & \frac{\partial C}{\partial \theta_1^1} & \frac{\partial C}{\partial \theta_1^2} & \frac{\partial C}{\partial \theta_1^3} \end{bmatrix}$$

3 Algorithm 1: Gradient Descent

```
In [10]: def algorithm1(h,max_steps,tolerance,a_init,printInfo):

             print("For Algorithm 1 (gradient descent)")
             print("Initial Cost: ", cost(a_init))

             if(printInfo == 'True'):
```

```
    print("The initial randomized guesses for the " +
    "constants of the neural network are: ")
    for a in range(4):
        print("alpha_",a, " = ", a_init[0][a])
    for b in range(4):
        print("theta_0^",b, " = ", a_init[1][b])
    for c in range(4):
        print("theta_1^",c, " = ", a_init[2][c])

a_new = a_init - h * gradient(a_init)
steps = 0

for i in range(max_steps):
    if(cost(a_init) - cost(a_new) <= tolerance):
        print("Tolerance has been reached.")
        break
    if(cost(a_init) < cost(a_new)):
        a_new = a_init
        break
    a_init = a_new
    direction = -1 * gradient(a_init)
    a_new = a_init + (h * direction)
    steps += 1

print("Final Cost: ", cost(a_new))

if(printInfo == 'True'):
    print("The constants for the neural network are: ")
    for j in range(4):
        print("alpha_",j, " = ", a_new[0][j])
    for k in range(4):
        print("theta_0^",k, " = ", a_new[1][k])
    for l in range(4):
        print("theta_1^",l, " = ", a_new[2][l])

print(steps, " steps completed.")

x_0 = np.linspace(0, 2 * np.pi,100)
y_0 = []
y_1 = []
for p in x_0:
    y_0.append(mu(p,a_new))
    y_1.append(np.sin(p))
fig = plt.figure()
axes = fig.add_axes([0.1, 0.1, 0.9, 0.7])
axes.plot(x_0,y_0,label ="$\mu$, approximated using GD",c = "red");
axes.plot(x_0,y_1, label = "$\sin(x)$", c = "blue");
axes.legend(loc = 3);
```

4 Algorithm 2: Stochastic Gradient Descent

```
In [11]: def algorithm2(h,max_steps,tolerance,a_init,size,subset,printInfo):


             print("For Algorithm 2 (stochastic gradient descent)")
             print("Initial Cost: ", stocCost(a_init,subset))

             if(printInfo == 'True'):
                 print("The initial randomized guesses for the " +
                 "constants of the neural network are: ")
                 for a in range(4):
                     print("alpha_",a, " = ", a_init[0][a])
                 for b in range(4):
                     print("theta_0^",b, " = ", a_init[1][b])
                 for c in range(4):
                     print("theta_1^",c, " = ", a_init[2][c])

             a_new = a_init - h * stocGrad(a_init,subset)
             steps = 0

             for i in range(max_steps):
                 if(stocCost(a_init,subset) - stocCost(a_new,subset) <= tolerance):
                     print("Tolerance has been reached.")
                     break
                 if(stocCost(a_init,subset) < stocCost(a_new,subset)):
                     a_new = a_init
                     break
                 a_init = a_new
                 direction = -1 * stocGrad(a_init,subset)
                 a_new = a_init + (h * direction)
                 steps += 1

             print("Final Cost: ", stocCost(a_new,subset))

             if(printInfo == 'True'):
                 print("The constants for the neural network are: ")
                 for j in range(4):
                     print("alpha_",j, " = ", a_new[0][j])
                 for k in range(4):
                     print("theta_0^",k, " = ", a_new[1][k])
                 for l in range(4):
                     print("theta_1^",l, " = ", a_new[2][l])

             print(steps, " steps completed using ", size,
                   " randomized points from 0 to 2pi.")

             x_0 = np.linspace(0, 2 * np.pi,100)
```

```
y_0 = []
y_1 = []
for p in x_0:
    y_0.append(mu(p,a_new))
    y_1.append(np.sin(p))
fig = plt.figure()
axes = fig.add_axes([0.1, 0.1, 0.9, 0.7])
axes.plot(x_0,y_0,label ="$\mu$, approximated using SGD",c = "red");
axes.plot(x_0,y_1, label = "$\sin(x)$", c = "blue");
axes.legend(loc = 3);
```

5 Algorithm 3: Conjugate Gradient Descent

In conjugate gradient descent,

$$\min_x = \frac{\left(\frac{b_2}{2}\right)(a_0+a_1) - b_1(a_0+a_2) + \left(\frac{b_0}{2}\right)(a_1+a_2)}{b_2 - 2b_1 + b_0}$$

where (a_0, b_0) , (a_1, b_2) , (a_2, b_2) are evenly spaced and go in order from smallest to largest. It represent points on a parabola determined by

$$a_1 = a_0 - h\nabla g(a_0) \text{ and } a_2 = a_1 - h\nabla g(a_0)$$

In [12]:
```
def algorithm3(h,max_steps,tolerance,a_init,printInfo):

    steps = 0

    print("For Algorithm 3 (conjugate gradient descent)")
    print("Initial Cost: ", cost(a_init))

    if(printInfo == 'True'):
        print("The initial guesses for the constants " +
        "of the neural network are: ")
        for a in range(4):
            print("alpha_",a, " = ", a_init[0][a])
        for b in range(4):
            print("theta_0^",b, " = ", a_init[1][b])
        for c in range(4):
            print("theta_1^",c, " = ", a_init[2][c])

    for i in range(max_steps):
        a_0 = a_init
        b_0 = cost(a_init)
        a_1 = a_0 - (h * gradient(a_0))
        b_1 = cost(a_1)
        a_2 = a_1 - (h * gradient(a_0))
        b_2 = cost(a_2)
        min_a = (0.5*b_2*(a_0 + a_1) - b_1*(a_0 + a_2)
```

```
                + 0.5*b_0*(a_1 + a_2))/(b_2 - 2*b_1 +b_0)
        a_init = min_a
        steps += 1

    print("Final Cost: ", cost(a_init))

    if(printInfo == 'True'):
        print("The constants for the neural network are: ")
        for j in range(4):
            print("alpha_",j, " = ", a_init[0][j])
        for k in range(4):
            print("theta_0^",k, " = ", a_init[1][k])
        for l in range(4):
            print("theta_1^",l, " = ", a_init[2][l])

    print(steps, " steps completed.")

    x_0 = np.linspace(0, 2 * np.pi,100)
    y_0 = []
    y_1 = []
    for p in x_0:
        y_0.append(mu(p,a_init))
        y_1.append(np.sin(p))
    fig = plt.figure()
    axes = fig.add_axes([0.1, 0.1, 0.9, 0.7])
    axes.plot(x_0,y_0,label ="$\mu$, approximated using CGD",c = "red");
    axes.plot(x_0,y_1, label = "$\sin(x)$", c = "blue");
    axes.legend(loc = 3);
```

6 Algorithm 4: Conjugate Stochastic Gradient Descent

```
In [13]: def algorithm4(h,max_steps,tolerance,a_init,size,subset,printInfo):

    steps = 0

    print("For Algorithm 4 (stochastic conjugate gradient descent)")
    print("Initial Cost: ", stocCost(a_init,subset))

    if(printInfo == 'True'):
        print("The initial randomized guesses for the " +
              "constants of the neural network are: ")
        for a in range(4):
            print("alpha_",a, " = ", a_init[0][a])
        for b in range(4):
            print("theta_0^",b, " = ", a_init[1][b])
        for c in range(4):
            print("theta_1^",c, " = ", a_init[2][c])
```

```
    for i in range(max_steps):
        a_0 = a_init
        b_0 = stocCost(a_init,subset)
        a_1 = a_0 - (h * stocGrad(a_0,subset))
        b_1 = stocCost(a_1,subset)
        a_2 = a_1 - (h * stocGrad(a_0,subset))
        b_2 = stocCost(a_2,subset)
        min_a = (0.5*b_2*(a_0 + a_1) - b_1*(a_0 + a_2)
                + 0.5*b_0*(a_1 + a_2))/(b_2 - 2*b_1 +b_0)
        a_init = min_a
        steps += 1

    print("Final Cost: ", stocCost(a_init,subset))

    if(printInfo == 'True'):
        print("The constants for the neural network are: ")
        for j in range(4):
            print("alpha_",j, " = ", a_init[0][j])
        for k in range(4):
            print("theta_0^",k, " = ", a_init[1][k])
        for l in range(4):
            print("theta_1^",l, " = ", a_init[2][l])

    print(steps, " steps completed using ", size,
          " randomized points from 0 to 2pi.")

    x_0 = np.linspace(0, 2 * np.pi,100)
    y_0 = []
    y_1 = []
    for p in x_0:
        y_0.append(mu(p,a_init))
        y_1.append(np.sin(p))

    fig = plt.figure()
    axes = fig.add_axes([0.1, 0.1, 0.9, 0.7])
    axes.plot(x_0,y_0,label ="$\mu$, approximated using SCGD",c = "red");
    axes.plot(x_0,y_1, label = "$\sin(x)$", c = "blue");
    axes.legend(loc = 3);
```

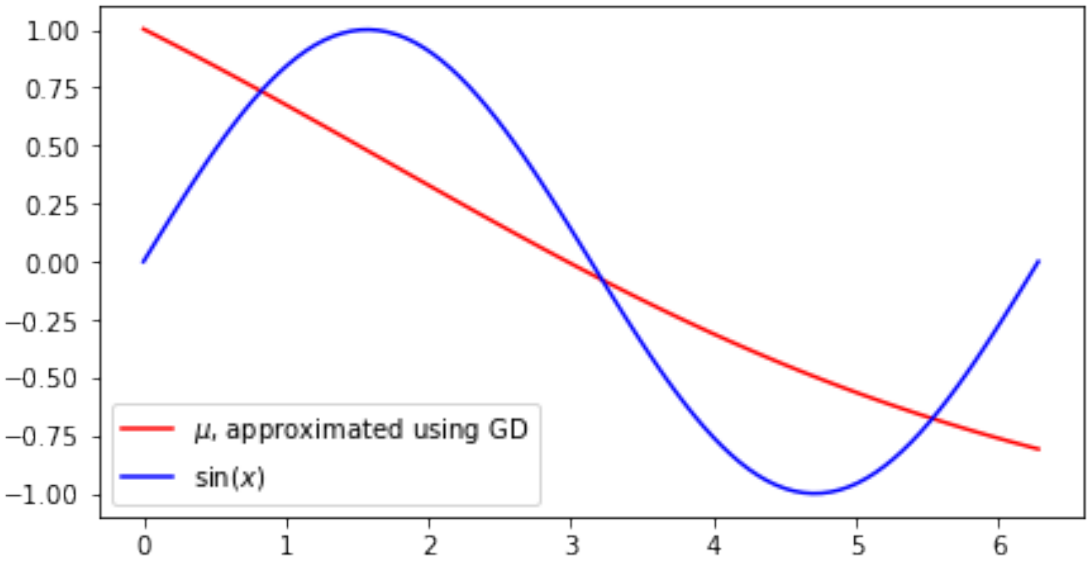
7 Run all algorithms and see which finishes the fastest.

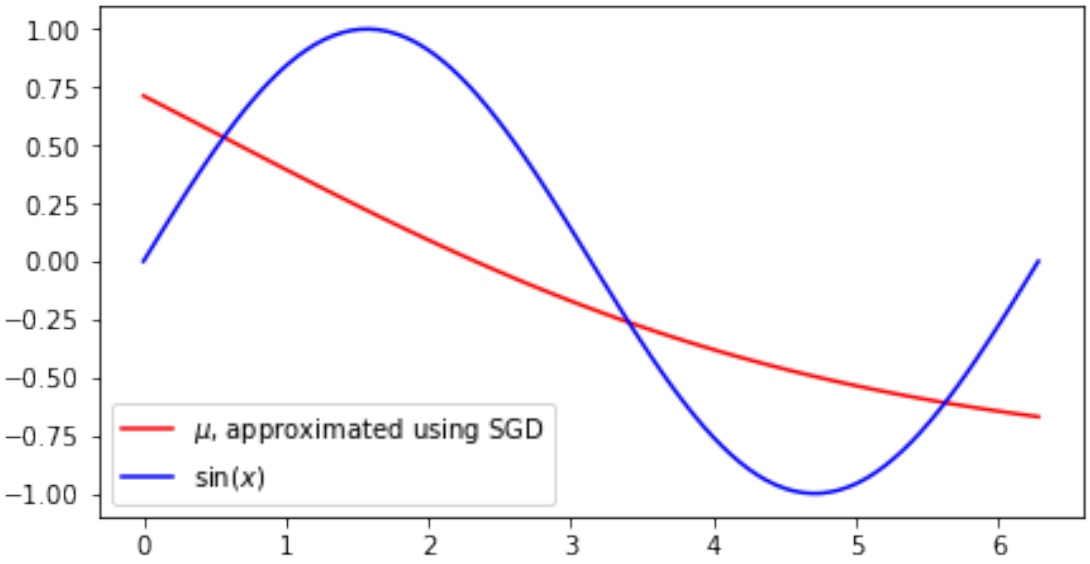
```
In [15]: a_init = np.random.rand(3,4)
         xcoord = np.linspace(0,2*np.pi,100)
         np.random.shuffle(xcoord)
         batchsize = 25
         subset = xcoord[0:batchsize]
```

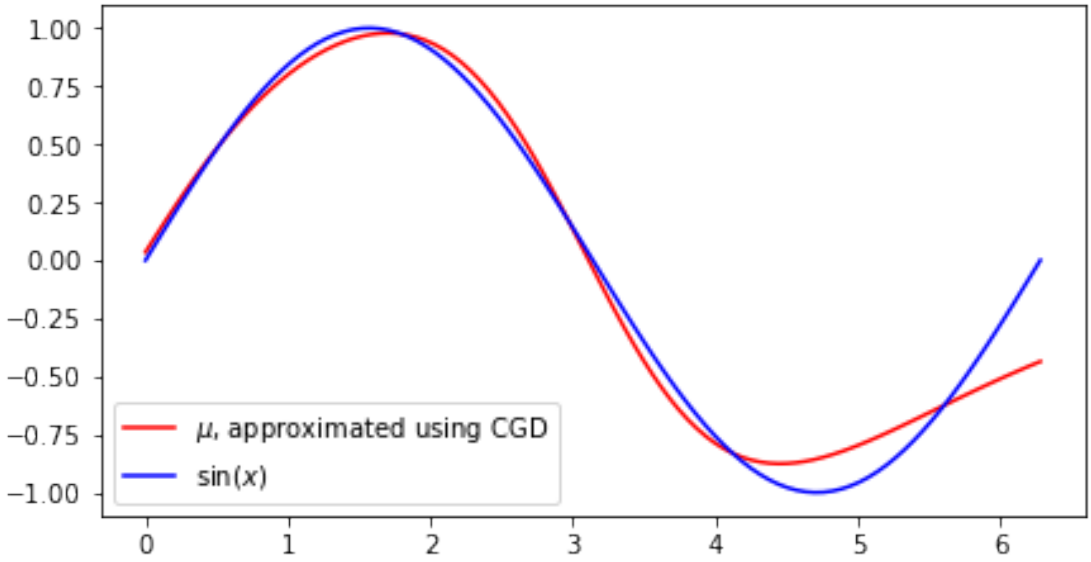
```
h = 0.001
max_steps = 1000
tolerance = 0.0001
showInfo = False

startFirst = time.time()
algorithm1(h,max_steps,tolerance,a_init,showInfo)
endFirst = time.time()

startSecond = time.time()
algorithm2(h,max_steps,tolerance,a_init,batchsize,subset,showInfo)
endSecond = time.time()

startThird = time.time()
algorithm3(h,max_steps,tolerance,a_init,showInfo)
endThird = time.time()

startFourth = time.time()
algorithm4(h,max_steps,tolerance,a_init,batchsize,subset,showInfo)
endFourth = time.time()

print("Using gradient descent, the algorithm took:",
      endFirst - startFirst, " seconds.")
print("Using stochastic gradient descent, the algorithm took:",
      endSecond - startSecond, " seconds.")
print("Using conjugate gradient descent, the algorithm took:",
      endThird - startThird, " seconds.")
print("Using stochastic conjugate gradient descent, the algorithm took:",
      endFourth - startFourth, " seconds.")
```

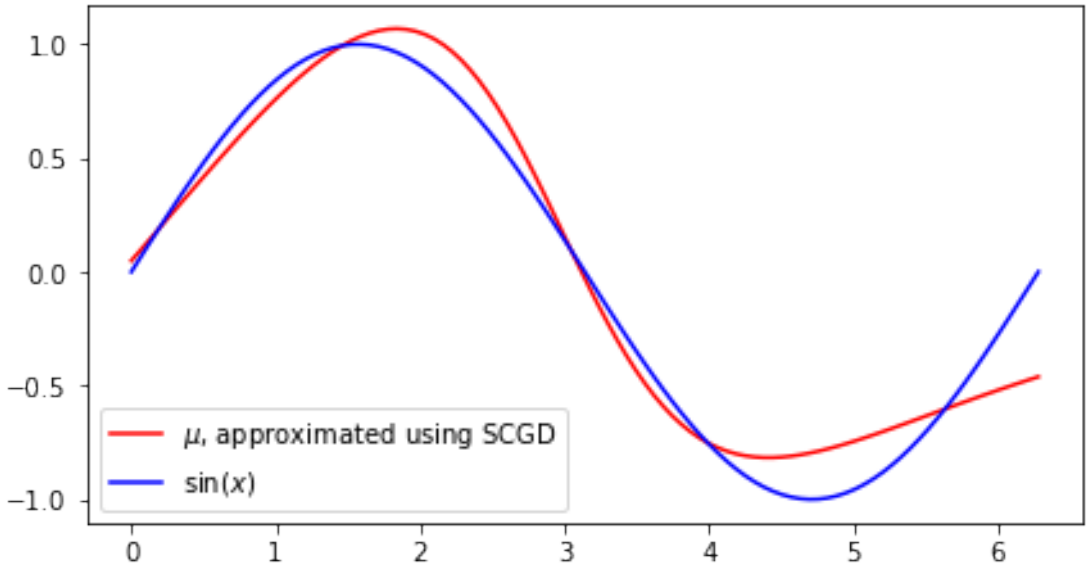
```
For Algorithm 1 (gradient descent)
Initial Cost:  139.476987431
Final Cost:  9.6255393471
1000  steps completed.
For Algorithm 2 (stochastic gradient descent)
Initial Cost:  38.4397556719
Final Cost:  2.8759300089
1000  steps completed using  25  randomized points from 0 to 2pi.
For Algorithm 3 (conjugate gradient descent)
Initial Cost:  139.476987431
Final Cost:  0.575147414347
1000  steps completed.
For Algorithm 4 (stochastic conjugate gradient descent)
Initial Cost:  38.4397556719
Final Cost:  0.242058381044
1000  steps completed using  25  randomized points from 0 to 2pi.
Using gradient descent, the algorithm took: 15.556030988693237  seconds.
```

```
Using stochastic gradient descent, the algorithm took: 3.3607819080352783  seconds.
Using conjugate gradient descent, the algorithm took: 17.989335775375366  seconds.
Using stochastic conjugate gradient descent, the algorithm took: 5.619760274887085  seconds.
```









8 Conclusion:

In all the randomized trial cases, we see that using a stochastic approach (with and without conjugate) created an approximation faster than if it were to use all x values from 0 to 2π. On the other hand, using a conjugate approach did not create a noticeable improvement in speed compared to if the algorithm was run without a conjugate approach. Looking at the generated graphs gives a

different interpretation of the algorithms. The algorithm that best approximated the sin function was conjugate gradient descent. The flaws of the two conjugacy algorithms is that it tends to not approximate well at the endpoints of the range given. Nonetheless, it creates a better approximation to the sin function than the gradient descent and stochastic gradient descent which both show huge variations at certain intervals. In fact, using both conjugate and stochastic approaches at the same time can create drastic problems in rare cases, as was seen in one simulation done (result shown here).

To conclude, if time is a priority and you don't care so much for accuracy, use the stochastic gradient descent approach whereas if time is no problem for you, using the conjugate gradient descent approach will give better approximations. But it is worth noting that out of all four algorithms, the SCGD algorithm was able to best minimize the distances between the sin function and the neural network constants, creating a cost of less than 1 in less than 5 seconds. An improvement that can be made to the algorithms is having less steps taken so as to not have the user get impatient.