6.867 : Homework 1

Anonymous authors

I. Gradient Descent

In this section, we implemented Gradient Descent in Python to find the minimum of a function.

A. Part 1

Our Gradient Descent (GD) implementation begins with an initial guess, x_0, of the argument x_{min} that minimizes the function, $f(x)$. The implementation is applied to two functions, a negative Gaussian and a quadratic bowl. The gradient, dfdx, can be calculated in closed form (for simple functions), and the next estimate of x_{min} is updated according to the rule:

$$x_{i+1} = x_i - \alpha \cdot \frac{\partial f(x_i)}{\partial x} \tag{1}$$

Here, α is the step size, or learning rate, which controls how much the gradient affects the next estimate. The rule is applied until either a pre-determined maximum number of iterations, n_{iters}, occurs, or the function's value at the current iteration differs from the previous iteration's estimate by less than ϵ. Each of these paramters, (α, n_{iters}, and ϵ) impact the GD solution in different ways.

In Fig. 3, the intial guess, x_0, is varied to show that a close intial guess leads to convergence in only a few steps. As the norm of difference between initial and final estimates grows, the number of iterations increases. For certain initial guesses, we observed that the algorithm did not converge in some maximum number of iterations. For the negative Gaussian, this is because the gradient is very small far from the mean.

The convergence limit determines when the GD algorithm stops, seen in Fig. 2. The error increases if ϵ is too large, as on the right of Fig. 1.

In GD, the norm of the gradient decreases as the solution converges, which is shown for the quadratic bowl function in Fig. 5.

B. Part 2

For more complicated functions, the gradient is not always as easy to compute. Instead, we can use finite difference approximation instead of a the closed form gradient. The gradient approximation for a function of two variables is:

$$\nabla f = \begin{bmatrix} f_x & f_y \end{bmatrix} \approx \begin{bmatrix} \frac{f(x+\delta x,y)-f(x,y)}{\delta x} & \frac{f(x,y+\delta y)-f(x,y)}{\delta y} \end{bmatrix} \tag{2}$$

While Section I-B shows that the error is rather small across a range of (x,y) values for the two functions, the choice of difference step δx is what led to this small error. In Fig. 6, the norm of the difference between the real and approximate gradient gets quite large when δx is also large.

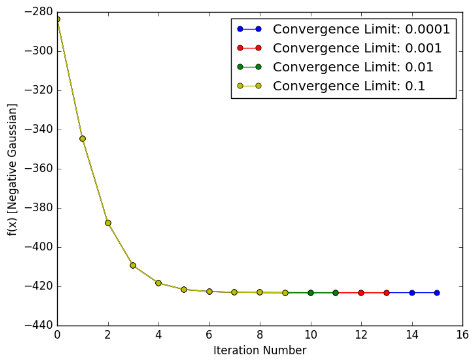

Fig. 1: As convergence limit, ϵ, increases, the iteration stops earlier. From this view, the four solutions differ very little.

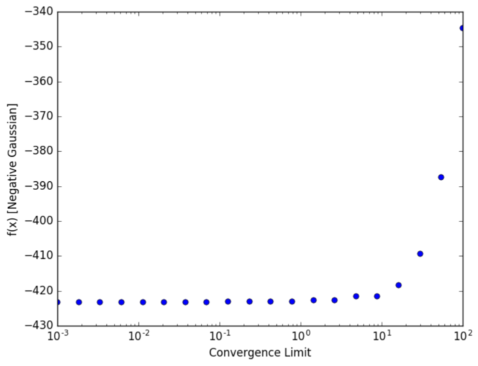

Fig. 2: The final value returned by GD can differ substantially if convergence limit is set too loosely.

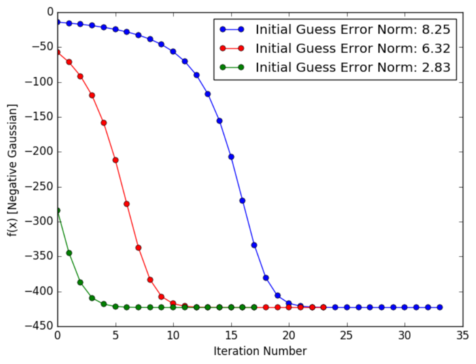

Fig. 3: When initial guess is far from true minimum, it takes many iterations (blue) to converge. As initial guess approaches final solution, number of iterations decreases.

Gradient vs Finite Difference ($\delta x = 0.01$)				
Function	Gradient	Approximation	Error	Error Norm
Neg. Gaussian: $f(3,4)$	[-4.2258, -3.6221]	[-4.2375, -3.6299]	[-0.01178, -0.00785]	0.0141
Neg. Gaussian: $f(12,4)$	[11.455, -34.365,]	[11.472, -34.440,]	[0.017, -0.074,]	0.0764
Quad. Bowl: $f(6,2)$	[-330, -350]	[-329.95, -349.95]	[0.05 0.05]	0.0707
Quad. Bowl: $f(10^3,90)$	[10050, 5500]	[10050.05, 5500.05]	[0.05, 0.05]	0.0707

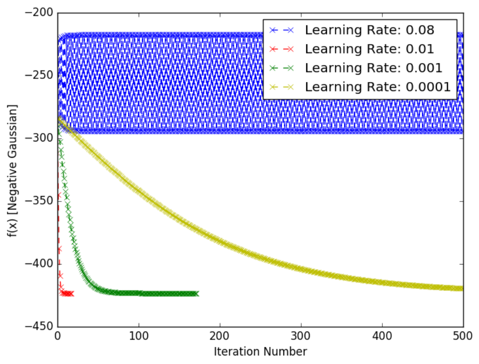

Fig. 4: If learning rate, α, is too large (blue), the solution will oscillate or diverge each iteration. If α is too small, it will take many iterations to converge (yellow). Convergence takes only a few iterations for proper α (red).

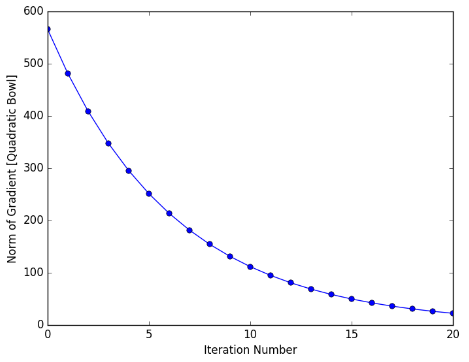

Fig. 5: Norm of gradient decreases with each iteration in GD. This example minimizes the Quadratic Bowl function.

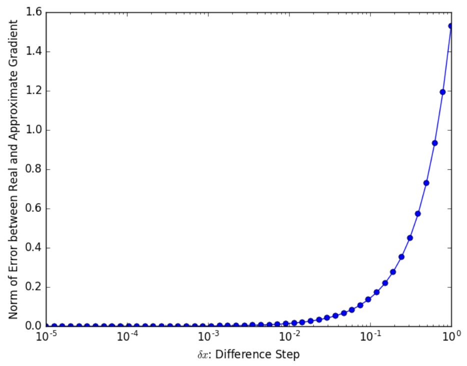

Fig. 6: Norm of gradient approximation error increases as δx increases. This plot is for the Negative Gaussian evaluated at (3,4).

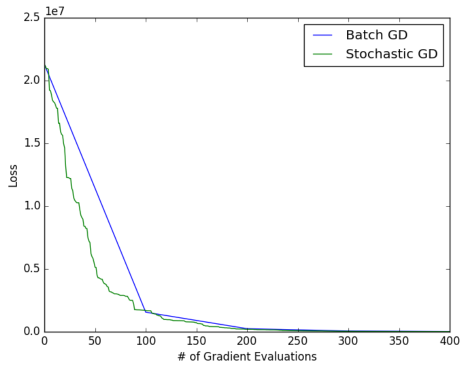

Fig. 7: Stochastic Gradient Descent (green) initially approaches convergence faster than Batch Gradient Descent (blue). Batch GD monotonically decreases in norm error, but SGD randomly oscillates and sometimes increases. Batch GD looks piecewise linear, because each iteration involves 100 (size of dataset) evaluations of the gradient, whereas SGD only evaluates once per iteration.

C. *Part 3*

Next, we consider two types of gradient descent on a dataset. Batch gradient descent (GD) calculates the gradient at each step using every sample in the data set. Stochastic gradient descent (SGD), instead uses a single sample each iteration, looping through a random ordering of the dataset to add a stochastic element. We implemented both algorithms and evaluated them on the same data set, with the same initial guess. For SGD, the equation (1) is modified to accept a single sample, (x^j, y^j) as input to the gradient:

$$\theta_{i+1} = \theta_i - \eta_i \cdot \nabla_\theta J(\theta_i; x^{(j)}, y^{(j)}) \tag{3}$$

We use a common form of learning rate parameter $\eta_i = (\tau_0 + i)^{-k}$ to guarantee convergence with $\tau_0 = 10^8$ and $k = 0.6$.

In Fig. 7, Stochastic Gradient Descent (green) initially approaches convergence faster than Batch Gradient Descent (blue). Batch GD monotonically decreases in norm error, but SGD randomly oscillates and sometimes increases. Batch GD looks piecewise linear, because each iteration involves 100 (size of dataset) evaluations of the gradient, whereas SGD only evaluates once per iteration. So, SGD intially learns faster than Batch GD.

After many iterations Fig. 8, Batch Gradient Descent catches up to Stochastic Gradient Descent and is more accurate. SGD seems to converge within a range, but continues to oscillate as long as learning rate is non-zero. So, Batch GD has better accuracy than SGD. The tradeoff here is speed vs. accuracy.

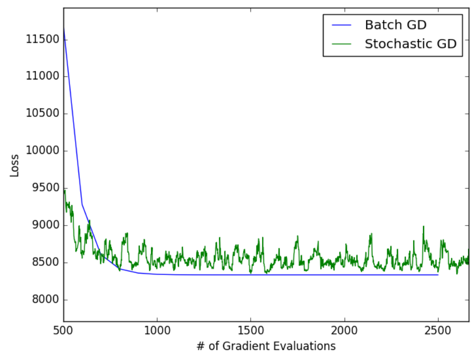

Fig. 8: After many iterations, Batch Gradient Descent catches up to Stochastic Gradient Descent and is more accurate. SGD seems to converge within a range, but continues to oscillate as long as learning rate is non-zero.

II. Linear Basis Function Regression

In this section, we consider two types of linear basis functions for regression: polynomial and cosine. We try different model complexities to compare the performance.

A. Part 1

At first, we don't know the true basis for our dataset, and want to try different complexities of polynomial models. Given an array of data points X and a vector Y of values, a simple polynomial basis,

$$\phi(x) = (1, x, x^2, ..., x^M) \tag{4}$$

is applied to the X points, for several values of M.

Fig. 9 shows the data as cyan dots, and the true function from which the data is generated in green. The blue in each subplot is the polynomial basis model, and red is a cosine model (described in [pt 4]). The model complexity is insufficient to capture the variation in the data with M=0 Fig. 9 (leftmost) since the model is a line with zero slope. For M=1, the linear-plus-constant basis, is still insufficient to capture the concave structure in the dataset. M=2 seems like a good choice visually, as it does not intersect every data point exactly but captures the concave shape. On the rightmost plot, M=10 overfits the data, which means it evaluates exactly to the correct values in the dataset but does not look like the smooth, green line from which the data was generated.

B. Part 4

Since the dataset was generated from a sum of cosines, a set of cosine basis functions should lead to a better fit than polynomials. The cosine basis functions are:

$$\phi(x) = (1, cos(x), cos(2x), ..., cos(Mx)) \tag{5}$$

. Again referring to Fig. 9, a similar trend can be seen of underfitting and overfitting for small and large model complexities, respectively. The M=2 cosine model fits the data pretty well, similar to the M=2 polynomial. This is no coincidence, because the data was generated from a pair of cosines (M=2), specifically $y = cos(\pi x) + 1.5cos(2\pi x)$.

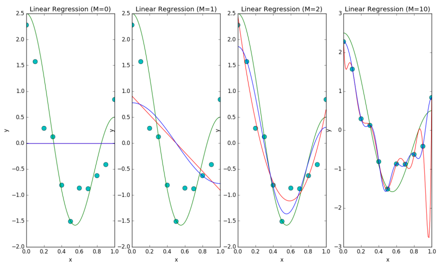

Fig. 9: For different M values (0,1,2,3,10) a polynomial (red) and cosine (blue) basis is used to fit the dataset (cyan points). The true function used to generate the data is shown in green in each plot. For low model complexity (left plot), the model doesn't capture the variation in the data. For high model complexity (right plot), the model overfits the data.

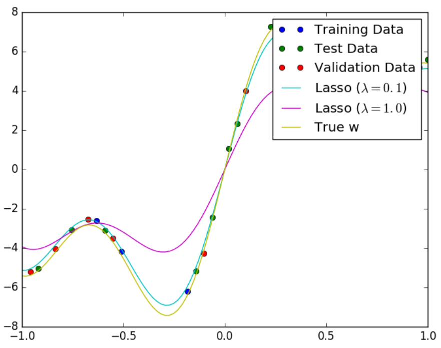

Fig. 10: The test, training, and validation data points are plotted along with models using true w values, LASSO's learned w values for two different regularization levels.

The actual values of the weight vector's elements can also be compared to the true function in Section II-B. The M=2 case does not give the exact same values as the actual function's coefficients, but this could possibly be improved using regularization, more data, data with lower noise, or removing the bias (w_0) term. With the complex model (bottom row, M=8), the high frequency components (7, 8) have very low magnitude.

III. Sparsity and LASSO

In this section, we used the sklearn Python implementation of the LASSO algorithm to model a dataset.

The raw x values are transformed into a feature vector with basis:

$$\phi(x) = (x, sin(0.4\pi x \cdot 1), sin(0.4\pi x \cdot 2), ..., sin(0.4\pi x \cdot 12)) \tag{6}$$

The weights w^T are used to map inputs x to outputs y through the basis $\phi(x)$, according to $y = w^T \phi(x)$. The dataset was generated using a true weight w_{true}^T with small noise ϵ, as $y = w_{true}^T \phi(x) + \epsilon$.

Cosine Basis Function Weights									
M	0	1	2	3	4	5	6	7	8
Actual	0	1	1.5	-	-	-	-	-	-
1	0.0	0.78	-	-	-	-	-	-	-
2	-0.11	0.78	1.19	-	-	-	-	-	-
4	-0.13	0.76	1.16	0.09	0.22	-	-	-	-
8	-0.15	0.77	1.1	0.1	0.16	-0.05	0.38	0.01	0.03