CS 189 Introduction to Machine Learning
Spring 2018 HW7

Your self-grade URL is http://eecs189.org/self_grade?question_ids=1_1,1_2,2_1,2_2,2_3,2_4,2_5,3_1,3_2,3_3,3_4,3_5,3_6,3_7,4_1,4_2,4_3,4_4,4_5,4_6,5_1,5_2,5_3,5_4,5_5,5_6,5_7,5_8,5_9,5_10,5_11,6.

This homework is due **Friday, March 9 at 10pm.**

2 Step Size in Gradient Descent

By this point in the class, we know that gradient descent is a powerful tool for moving towards local minima of general functions. We also know that local minima of convex functions are global minima. In this problem, we will look at the convex function $f(\mathbf{x}) = ||\mathbf{x} - \mathbf{b}||_2$. Note that we are using "just" the regular Euclidean ℓ_2 norm, *not* the norm squared! This problem illustrates the importance of understanding how gradient descent works and choosing step sizes strategically. In fact, there is a lot of active research in variations on gradient descent. Throughout the question we will look at different kinds of step-sizes. Constant step size vs. decreasing step size. We will also look at the the rate at which the different step sizes decrease and draw some conclusions about the rate of convergence. Notice that we want to make sure the we get to some local minimum and we want to do it as quickly as possible.

You have been provided with a tool in `step_size.py` which will help you visualize the problems below.

(a) Let $\mathbf{x}, \mathbf{b} \in \mathbb{R}^d$. **Prove that $f(\mathbf{x}) = \|\mathbf{x} - \mathbf{b}\|_2$ is a convex function of $\mathbf{x}$.**

Solution: Recall that a function is convex if for all $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^d$ we have

$$f(\lambda \mathbf{x}_1 + (1-\lambda)\mathbf{x}_2) \leq \lambda f(\mathbf{x}_1) + (1-\lambda) f(\mathbf{x}_2)$$

for all $0 \leq \lambda \leq 1$. For $f(\mathbf{x}) = \|\mathbf{x} - \mathbf{b}\|_2$, the triangle inequality gives us:

$$\begin{aligned} \|\lambda \mathbf{x}_1 + (1-\lambda)\mathbf{x}_2 - \mathbf{b}\|_2 &= \|\lambda(\mathbf{x}_1 - \mathbf{b}) + (1-\lambda)(\mathbf{x}_2 - \mathbf{b})\|_2 \\ &\leq \|\lambda(\mathbf{x}_1 - \mathbf{b})\|_2 + \|(1-\lambda)(\mathbf{x}_2 - \mathbf{b})\|_2 \\ &= \lambda\|\mathbf{x}_1 - \mathbf{b}\|_2 + (1-\lambda)\|\mathbf{x}_2 - \mathbf{b}\|_2, \end{aligned}$$

which shows that f is convex.

(b) We are minimizing $f(\mathbf{x}) = \|\mathbf{x} - \mathbf{b}\|_2$, where $\mathbf{x} \in \mathbb{R}^2$ and $\mathbf{b} = [4.5, 6] \in \mathbb{R}^2$, with gradient descent. We use a constant step size of $t_i = 1$. That is,

$$\mathbf{x}_{i+1} = \mathbf{x}_i - t_i \nabla f(\mathbf{x}_i) = \mathbf{x}_i - \nabla f(\mathbf{x}_i).$$

We start at $\mathbf{x}_0 = [0, 0]$. **Will gradient descent find the optimal solution? If so, how many steps will it take to get within** 0.01 **of the optimal solution? If not, why not?** Prove your answer. (Hint: use the tool to compute the first ten steps.) **What about general** $\mathbf{b} \neq \mathbf{0}$**?**

Solution: Using the tool provided (or computing gradients by hand), we get

$$\mathbf{x}_6 = \mathbf{x}_8 = [4.2, 5.6]$$

and

$$\mathbf{x}_7 = \mathbf{x}_9 = [4.8, 6.4].$$

Examine the formula

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \nabla f(\mathbf{x}_i)$$

and notice that $\mathbf{x}_{i+1}$ only depends on $\mathbf{x}_i$, regardless of what step we are on (this is only the case because we are using a constant step size). It means that if $\mathbf{x}_i = \mathbf{x}_j$, then $\mathbf{x}_{i+1} = \mathbf{x}_{j+1}$. Thus, this pattern will repeat indefinitely and we do not reach the optimal solution. It is also helpful to notice that

$$\|\mathbf{x}_6\|, \|\mathbf{x}_7\| \in \mathbb{Z}.$$

In general, notice that

$$-\nabla f(\mathbf{x}_i) = \frac{\mathbf{b} - \mathbf{x}_i}{\|\mathbf{x}_i - \mathbf{b}\|}$$

will always have unit length. In fact, we can prove by induction that $\mathbf{x}_k$ is always an integer multiple of the unit vector $\frac{\mathbf{b}}{\|\mathbf{b}\|}$. Thus,

$$\|\mathbf{x}_k\| \in \mathbb{Z}.$$

If we are lucky and $\|\mathbf{b}\|$ happens to be an integer (or very close to integer), then we will reach or get near the optimal solution. Otherwise, our step size is too coarse to hit it.

In fact, for the function $\|\mathbf{x} - \mathbf{b}\|$, a constant step size will rarely work. We are too prone to "jumping over" our solution. A decreasing step size is necessary and we will investigate decreasing stepsizes next.

(c) We are minimizing $f(\mathbf{x}) = \|\mathbf{x} - \mathbf{b}\|_2$, where $\mathbf{x} \in \mathbb{R}^2$ and $\mathbf{b} = [4.5, 6] \in \mathbb{R}^2$, now with a decreasing step size of $t_i = (\frac{5}{6})^i$ at step i. That is,

$$\mathbf{x}_{i+1} = \mathbf{x}_i - t_i \nabla f(\mathbf{x}_i) = \mathbf{x}_i - (\frac{5}{6})^i \nabla f(\mathbf{x}_i).$$

We start at $\mathbf{x}_0 = [0, 0]$. **Will gradient descent find the optimal solution? If so, how many steps will it take to get within** 0.01 **of the optimal solution? If not, why not?** Prove your answer. (Hint: examine $\|\mathbf{x}_i\|_2$.) **What about general** $\mathbf{b} \neq \mathbf{0}$**?**

Solution: Notice that we can express $\mathbf{x}_{i+1}$ as the sum of all the steps taken, so we can express its norm as

$$\|\mathbf{x}_{i+1}\| = \|\mathbf{x}_0 - \sum_{j=0}^{i} (\frac{5}{6})^j \nabla f(\mathbf{x}_i)\| \leq \|\mathbf{x}_0\| + \sum_{j=0}^{i} \|(\frac{5}{6})^j \nabla f(\mathbf{x}_i)\|$$

But all of $\|\nabla f(\mathbf{x}_i)\|$ are exactly 1, so we can write this as

$$\|\mathbf{x}_{i+1}\| \leq \|\mathbf{x}_0\| + \sum_{j=0}^{i} (\frac{5}{6})^j \|\nabla f(\mathbf{x}_i)\| = 0 + \sum_{j=0}^{i} (\frac{5}{6})^j \leq 6.$$

In words, we can never "travel" a distance of more than 6 from $\mathbf{0}$, so we will never get to our optimal solution $\mathbf{b}$ if $\|\mathbf{b}\| > 6$. So, while we need a decreasing step size, we also have to be sure it does not decrease too quickly.

(d) We are minimizing $f(\mathbf{x}) = \|\mathbf{x} - \mathbf{b}\|_2$, where $\mathbf{x} \in \mathbb{R}^2$ and $\mathbf{b} = [4.5, 6] \in \mathbb{R}^2$, now with a decreasing step size of $t_i = \frac{1}{i+1}$ at step i. That is,

$$\mathbf{x}_{i+1} = \mathbf{x}_i - t_i \nabla f(\mathbf{x}_i) = \mathbf{x}_i - \frac{1}{i+1} \nabla f(\mathbf{x}_i).$$

We start at $\mathbf{x}_0 = [0, 0]$. **Will gradient descent find the optimal solution? If so, how many steps will it take to get within** 0.01 **of the optimal solution? If not, why not?** Prove your answer. (Hint: examine $\|\mathbf{x}_i\|_2$, and use $\sum_{i=1}^{n} \frac{1}{i}$ is of the order $\log n$.) **What about general** $\mathbf{b} \neq \mathbf{0}$**?**

Solution: The key realization here is that all our gradient steps have $\|\nabla f(\mathbf{x}_i)\| = 1$ and move along the same vector towards $\mathbf{b}$ (to be precise, we could prove by induction that our $\mathbf{x}_i$ and $\nabla f(\mathbf{x}_i)$ will always be in the span of $\mathbf{b}$). We can write

$$\|\mathbf{x}_{i+1}\| = \|\mathbf{x}_0 - \sum_{j=0}^{i} (\frac{1}{j+1}) \nabla f(\mathbf{x}_i)\| = \sum_{j=0}^{i} \|(\frac{1}{j+1}) \nabla f(\mathbf{x}_i)\| = \sum_{j=1}^{i} \frac{1}{j+1} \approx \ln i.$$

This sum diverges, so we will reach the optimal solution $\mathbf{b}$ eventually, but it could take quite a while. For $\mathbf{b} = [4.5, 6]$ it takes about 1000 steps. In generally, the number of steps it takes will be exponential in $\|\mathbf{b}\|$, since we have

$$\ln i \approx \|\mathbf{b}\| \implies i \approx e^{\|\mathbf{b}\|}.$$

We may not hit $\mathbf{b}$ exactly. Once we exceed $\mathbf{b}$, we start oscillating around it. Let i_0 be the first step where we "cross over" $\mathbf{b}$. After that, we know we are always moving towards $\mathbf{b}$ (whether we are stepping backwards or forwards), so after step i (assuming $i > i_0$), we can be at most $\frac{1}{i+1}$ from the optimal solution (if we are more than $\frac{1}{i+1}$ away, that means our previous step went in the wrong direction).

(e) Now, say we are minimizing $f(\mathbf{x}) = \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2$. Use the code provided to test several values of $\mathbf{A}$ with the step sizes suggested above. Make plots to visualize what is happening. We suggest trying $\mathbf{A} = [[10, 0], [0, 1]]$ and $\mathbf{A} = [[15, 8], [6, 5]]$. **Will any of the step sizes above work for all choices of** $\mathbf{A}$ **and** $\mathbf{b}$**?** You do not need to prove your answer, but you should briefly explain your reasoning.

Solution: The first two step sizes did not work for $\mathbf{A} = \mathbf{I}$, so they will not work in general. The last step sizes *does* always work, though it may take a very long time to converge.

A perspective to notice is that the $\mathbf{x}_i$ are still ultimately moving towards $\mathbf{x}^*$, but not on a straight line but instead on a curved path. The path has finite length, so we will eventually reach $\mathbf{x}^*$. Once we are close to $\mathbf{x}^*$, successive steps will oscillate around $\mathbf{x}^*$.

Formally: This was not asked in the question, but here is the proof: We have

$$\nabla f(\mathbf{x}) = \frac{\mathbf{A}^\top(\mathbf{A}\mathbf{x} - \mathbf{b})}{\|\mathbf{A}\mathbf{x} - \mathbf{b}\|}$$

and therefore $\|\nabla f(\mathbf{x})\|_2^2 \leq \sigma_{max}^2(\mathbf{A})$. From convexity, we furthermore have

$$\nabla f(\mathbf{x}_i)^\top(\mathbf{x}_i - \mathbf{x}^*) \leq f(\mathbf{x}_i) - f(\mathbf{x}^*)$$

which yields

$$\begin{aligned}
\|\mathbf{x}_{i+1} - \mathbf{x}^*\|_2^2 &= \|\mathbf{x}_i - t_i\nabla f(\mathbf{x}_i) - \mathbf{x}^*\|_2^2 \\
&= \|\mathbf{x}_i - \mathbf{x}^*\|_2^2 - 2t_i\nabla f(\mathbf{x}_i)^\top(\mathbf{x}_i - \mathbf{x}^*) + t_i^2\|\nabla f(\mathbf{x}_i)\|_2^2 \\
&\leq \|\mathbf{x}_i - \mathbf{x}^*\|_2^2 - 2t_i(f(\mathbf{x}_i) - f(\mathbf{x}^*)) + t_i^2\sigma_{max}^2(\mathbf{A}) \\
&\leq \|\mathbf{x}_0 - \mathbf{x}^*\|_2^2 - 2\sum_{k=0}^{i} t_k(f(\mathbf{x}_k) - f(\mathbf{x}^*)) + \sigma_{max}^2(\mathbf{A})\sum_{k=0}^{i} t_k^2 \\
&\leq \|\mathbf{x}_0 - \mathbf{x}^*\|_2^2 - 2(f(\mathbf{x}_{best}) - f(\mathbf{x}^*))\sum_{k=0}^{i} t_k + \sigma_{max}^2(\mathbf{A})\sum_{k=0}^{i} t_k^2
\end{aligned}$$

where $\mathbf{x}_{best} = \arg\min_{0\leq k\leq i} f(\mathbf{x}_k)$. We denote $\|\mathbf{x}_0 - \mathbf{x}^*\|_2 = d_0$. This yields

$$0 \leq d_0^2 - 2(f(\mathbf{x}_{best}) - f(\mathbf{x}^*))\sum_{k=0}^{i} t_k + \sigma_{max}^2(\mathbf{A})\sum_{k=0}^{i} t_k^2$$

Hence,

$$f(\mathbf{x}_{best}) - f(\mathbf{x}^*) \leq \frac{d_0^2 + \sigma_{max}^2(\mathbf{A})\sum_k t_k^2}{\sum_k t_k} = \frac{d_0^2 + \sigma_{max}^2(\mathbf{A})\sum_k \frac{1}{(1+k)^2}}{\sum_k \frac{1}{1+k}} \xrightarrow[i\to\infty]{} 0$$

3 Convergence Rate of Gradient Descent

In the previous problem, you examined $||\mathbf{A}\mathbf{x} - \mathbf{b}||_2$ (without the square). You showed that even though it is convex, getting gradient descent to converge requires some care. In this problem, you will examine $\frac{1}{2}||\mathbf{A}\mathbf{x} - \mathbf{b}||_2^2$ (with the square). You will show that now gradient descent converges quickly.

For a matrix $\mathbf{A} \in \mathbb{R}^{n\times d}$ and a vector $\mathbf{b} \in \mathbb{R}^n$, consider the quadratic function $f(\mathbf{x}) = \frac{1}{2}\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2$ such that $\mathbf{A}^\top\mathbf{A}$ is positive definite.

Throughout this question the *Cauchy-Schwarz inequality* might be useful: Given two vectors $\mathbf{u}, \mathbf{v}$:

$$|\mathbf{u}^\top \mathbf{v}| \leq \|\mathbf{u}\|_2 \|\mathbf{v}\|_2,$$

with equality only when $\mathbf{v}$ is a scaled version of $\mathbf{u}$.

(a) First, consider the case $\mathbf{b} = \mathbf{0}$, and think of each $\mathbf{x} \in \mathbb{R}^d$ as a "state". Performing gradient descent moves us sequentially through the states, which is called a "state evolution". **Write out the state evolution for n iterations of gradient descent using step-size $\gamma > 0$. i.e. express $\mathbf{x}_n$ as a function of $\mathbf{x}_0$.** Use $\mathbf{x}_0$ to denote the initial condition of where you start gradient descent from.

Solution: *Quick note:* First let us clarify why gradient descent on the norm squared converges faster than gradient descent on the norm. The norm is convex but not strictly convex. Therefore gradient descent will only obtain the slow error rate $O(\frac{1}{\sqrt{k}})$ where k is the number of iterations (we did not prove this, but our step size selection in the last part of the last problem led to a worse bound $O(\frac{1}{\log k})$ and with a more refined analysis, we can get $O(\frac{1}{\sqrt{k}})$). On the other hand, the squared norm is strictly convex and therefore gradient descent can obtain the fast linear error rate $O(\beta^k)$ for $\beta < 1$ as we show in the remainder of this problem.

The general formula for the gradient update is

$$\mathbf{x}_n = \mathbf{x}_{n-1} - \gamma \nabla f(\mathbf{x}_{n-1})$$

which in this case is

$$\mathbf{x}_n = \mathbf{x}_{n-1} - \gamma(\mathbf{A}^\top \mathbf{A}\mathbf{x}_{n-1} - \mathbf{A}^\top \mathbf{b}) = (\mathbf{I} - \gamma \mathbf{A}^\top \mathbf{A})\mathbf{x}_{n-1} - \gamma \mathbf{A}^\top \mathbf{b}.$$

Since we assumed $\mathbf{b} = \mathbf{0}$, this works out to

$$\mathbf{x}_n = (\mathbf{I} - \gamma \mathbf{A}^\top \mathbf{A})\mathbf{x}_{n-1},$$

so by induction we can show

$$\mathbf{x}_n = (\mathbf{I} - \gamma \mathbf{A}^\top \mathbf{A})^n \mathbf{x}_0.$$

(b) A state evolution is said to be stable if it does not blow up arbitrarily over time. Specifically, if state n is

$$\mathbf{x}_n = \mathbf{B}^n \mathbf{x}_0$$

then we need *all* the eigenvalues of $\mathbf{B}$ to be less than or equal to 1 in absolute value, otherwise $\mathbf{B}^n$ might blow up $\mathbf{x}_0$ for large enough n.

When is the state evolution of the iterations you calculated above stable when viewed as a dynamical system?

Solution: An important fact in linear algebra is that if we take a matrix $\mathbf{B}$ with eigenvalues λ_i to the power of k, then the resulting matrix $\mathbf{B}^k$ has eigenvalues λ_i^k with the same eigenvectors. The proof of this fact (inductively) is

$$\mathbf{B}^k \mathbf{v} = \mathbf{B}^{k-1}\mathbf{B}\mathbf{v} = \mathbf{B}^{k-1}\lambda \mathbf{v} = \lambda \mathbf{B}^{k-1}\mathbf{v} = \lambda \lambda^{k-1}\mathbf{v} = \lambda^k \mathbf{v}.$$

Let $\mathbf{B} = (\mathbf{I} - \gamma \mathbf{A}^\top \mathbf{A})$. In order for the state evolution to be stable, we need *all* the eigenvalues of $\mathbf{B}$ to be less than or equal to 1 in absolute value. More formally, this connects to an identity we have previously reviewed, which is

$$\|\mathbf{B}^k \mathbf{v}\|_2 \leq |(\lambda_{\max}(\mathbf{B}))^k| \|\mathbf{v}\|_2 = (|\lambda_{\max}(\mathbf{B})|)^k \|\mathbf{v}\|_2.$$

So if the largest eigenvalue of $(\mathbf{I} - \gamma \mathbf{A}^\top \mathbf{A})$ (in absolute value) is bounded by 1, then the system is stable.

(c) We want to bound the progress of gradient descent in the general case, when $\mathbf{b}$ is arbitrary. To do this, we first show a slightly more general bound, which relates how much the spacing between two points changes if they *both* take a gradient step. If this spacing shrinks, this is called a contraction. Define $\varphi(\mathbf{x}) = \mathbf{x} - \gamma \nabla f(\mathbf{x})$, for some constant step size $\gamma > 0$. **Show that for any** $\mathbf{x}, \mathbf{x}' \in \mathbb{R}^d$,

$$\|\varphi(\mathbf{x}) - \varphi(\mathbf{x}')\|_2 \leq \beta \|\mathbf{x} - \mathbf{x}'\|_2$$

where $\beta = \max\left\{|1 - \gamma \lambda_{\max}(\mathbf{A}^\top \mathbf{A})|, |1 - \gamma \lambda_{\min}(\mathbf{A}^\top \mathbf{A})|\right\}$. Note that $\lambda_{\min}(\mathbf{A}^\top \mathbf{A})$ denotes the smallest eigenvalue of the matrix $\mathbf{A}^\top \mathbf{A}$; similarly, $\lambda_{\max}(\mathbf{A}^\top \mathbf{A})$ denotes the largest eigenvalue of the matrix $\mathbf{A}^\top \mathbf{A}$.

Solution: We start off by expanding the left side of the equation we are interested in:

$$\begin{aligned}
\|\varphi(\mathbf{x}) - \varphi(\mathbf{x}')\|_2 &= \|(\mathbf{x} - \gamma \mathbf{A}^\top (\mathbf{A}\mathbf{x} - \mathbf{b})) - (\mathbf{x}' - \gamma \mathbf{A}^\top (\mathbf{A}\mathbf{x}' - \mathbf{b}))\|_2 \\
&= \|(\mathbf{x} - \mathbf{x}') - \gamma \mathbf{A}^\top \mathbf{A}(\mathbf{x} - \mathbf{x}')\|_2 \\
&= \|(\mathbf{I} - \gamma \mathbf{A}^\top \mathbf{A})(\mathbf{x} - \mathbf{x}')\|_2
\end{aligned}$$

Let $\mathbf{B} = \mathbf{I} - \gamma \mathbf{A}^\top \mathbf{A}$. We can now square both sides to obtain

$$\|\varphi(\mathbf{x}) - \varphi(\mathbf{x}')\|_2^2 \leq (\mathbf{x} - \mathbf{x}')^\top \mathbf{B}^\top \mathbf{B} (\mathbf{x} - \mathbf{x}').$$

Recall the definition of the Rayleigh quotient, by which we know that for a symmetric matrix $\mathbf{X}$, we have

$$R(\mathbf{v}) = \frac{\mathbf{v}^\top \mathbf{X} \mathbf{v}}{\mathbf{v}^\top \mathbf{v}} \leq R(\mathbf{v}_1) = \lambda_1(\mathbf{X}),$$

when $\mathbf{v}_1$ is the eigenvector corresponding to the maximum eigenvalue $\lambda_1(\mathbf{X})$ of the matrix $\mathbf{X}$. Using this fact substituting this in our result, we see that

$$\|\varphi(\mathbf{x}) - \varphi(\mathbf{x}')\|_2^2 \leq \lambda_1(\mathbf{B}^\top \mathbf{B}) \|\mathbf{x} - \mathbf{x}'\|_2^2,$$

and consequently, we have

$$\|\varphi(\mathbf{x}) - \varphi(\mathbf{x}')\|_2 \leq \sqrt{\lambda_1(\mathbf{B}^\top \mathbf{B})} \|\mathbf{x} - \mathbf{x}'\|_2.$$

We must now compute the maximum eigenvalue of $\mathbf{B}^\top \mathbf{B}$. We know that the eigenvalues of $\mathbf{B}^\top \mathbf{B}$ are the squared eigenvalues of $\mathbf{B}$. In particular, that means that the maximum eigenvalue

of $\mathbf{B}^\top\mathbf{B}$ is either the square of the maximum eigenvalue of $\mathbf{B}$ (when that is large and positive), or the minimum eigenvalue of $\mathbf{B}$ (when that is large and negative). Since the maximum and minimum eigenvalues of $\mathbf{I} - \gamma\mathbf{A}^\top\mathbf{A}$ are given by $1 - \gamma\lambda_{\min}(\mathbf{A}^\top\mathbf{A})$, and $1 - \gamma\lambda_{\max}(\mathbf{A}^\top\mathbf{A})$, we have

$$\lambda_1(\mathbf{B}^\top\mathbf{B}) = \max\left\{(1 - \gamma\lambda_{\min}(\mathbf{A}^\top\mathbf{A}))^2, (1 - \gamma\lambda_{\max}(\mathbf{A}^\top\mathbf{A}))^2\right\}.$$

Taking the square root and combining the pieces, we have

$$\|\varphi(\mathbf{x}) - \varphi(\mathbf{x}')\|_2 \leq \beta\|\mathbf{x} - \mathbf{x}'\|_2.$$

(d) Now we give a bound for progress after k steps of gradient descent. Define

$$\mathbf{x}^* = \arg\min_{\mathbf{x}\in\mathbb{R}^d} f(\mathbf{x}).$$

Show that

$$\|\mathbf{x}_{k+1} - \mathbf{x}^*\|_2 = \|\varphi(\mathbf{x}_k) - \varphi(\mathbf{x}^*)\|_2$$

and conclude that

$$\|\mathbf{x}_{k+1} - \mathbf{x}^*\|_2 \leq \beta^{k+1}\|\mathbf{x}_0 - \mathbf{x}^*\|_2.$$

Solution: Note that $\varphi(\mathbf{x}^*) = \mathbf{x}^*$ at optimality and $\varphi(\mathbf{x}_k) = \mathbf{x}_k - \gamma\nabla f(\mathbf{x}_k) = \mathbf{x}_{k+1}$. Then,

$$\|\mathbf{x}_{k+1} - \mathbf{x}^*\|_2 = \|\varphi(\mathbf{x}_k) - \varphi(\mathbf{x}^*)\|_2.$$

Applying the previous part, we have

$$\|\mathbf{x}_{k+1} - \mathbf{x}^*\|_2 \leq \beta\|\mathbf{x}_k - \mathbf{x}^*\|_2.$$

Applying the same bound k times, we have

$$\|\mathbf{x}_{k+1} - \mathbf{x}^*\|_2 \leq \beta^{k+1}\|\mathbf{x}_0 - \mathbf{x}^*\|_2.$$

(e) However, what we actually care about is progress in the objective value $f(\mathbf{x})$. That is, we want to show how quickly $f(\mathbf{x})$ is converging to $f(\mathbf{x}^*)$. We can do this by relating $f(\mathbf{x}) - f(\mathbf{x}^*)$ to $\|\mathbf{x} - \mathbf{x}^*\|_2$; or even better, relating $f(\mathbf{x}) - f(\mathbf{x}^*)$ to $\|\mathbf{x}_0 - \mathbf{x}^*\|_2$, for some starting point $\mathbf{x}_0$. First, **show that**

$$f(\mathbf{x}) - f(\mathbf{x}^*) = \frac{1}{2}\|\mathbf{A}(\mathbf{x} - \mathbf{x}^*)\|_2^2.$$

Solution: Note that at $\nabla f(\mathbf{x}^*) = \mathbf{A}^\top(\mathbf{A}\mathbf{x}^* - \mathbf{b}) = \mathbf{0}$ (the gradient evaluated at $\mathbf{x}^*$), implying that $\mathbf{A}^\top\mathbf{A}\mathbf{x}^* = \mathbf{A}^\top\mathbf{b}$. From here we do a bit of algebra to reach our answer, canceling terms and then substituting for $\mathbf{A}^\top\mathbf{b}$ and rearranging:

$$f(\mathbf{x}) - f(\mathbf{x}^*) = \frac{1}{2}\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2 - \frac{1}{2}\|\mathbf{A}\mathbf{x}^* - \mathbf{b}\|_2^2$$

$$
\begin{aligned}
&= \frac{1}{2}\left((\mathbf{x}^\top\mathbf{A}^\top\mathbf{A}\mathbf{x} - 2\mathbf{b}^\top\mathbf{A}\mathbf{x} + \mathbf{b}^\top\mathbf{b}) - (\mathbf{x}^{*\top}\mathbf{A}^\top\mathbf{A}\mathbf{x}^* - 2\mathbf{b}^\top\mathbf{A}\mathbf{x}^* + \mathbf{b}^\top\mathbf{b})\right)\\
&= \frac{1}{2}\left((\mathbf{x}^\top\mathbf{A}^\top\mathbf{A}\mathbf{x} - 2\mathbf{b}^\top\mathbf{A}\mathbf{x}) - (\mathbf{x}^{*\top}\mathbf{A}^\top\mathbf{A}\mathbf{x}^* - 2\mathbf{b}^\top\mathbf{A}\mathbf{x}^*)\right)\\
&= \frac{1}{2}\left((\mathbf{x}^\top\mathbf{A}^\top\mathbf{A}\mathbf{x} - 2\mathbf{x}^{*\top}\mathbf{A}^\top\mathbf{A}\mathbf{x}) - (\mathbf{x}^{*\top}\mathbf{A}^\top\mathbf{A}\mathbf{x}^* - 2\mathbf{x}^{*\top}\mathbf{A}^\top\mathbf{A}\mathbf{x}^*)\right)\\
&= \frac{1}{2}\left(\mathbf{x}^\top\mathbf{A}^\top\mathbf{A}\mathbf{x} - 2\mathbf{x}^{*\top}\mathbf{A}^\top\mathbf{A}\mathbf{x} + \mathbf{x}^{*\top}\mathbf{A}^\top\mathbf{A}\mathbf{x}^*\right)\\
&= \frac{1}{2}\|\mathbf{A}\mathbf{x} - \mathbf{A}\mathbf{x}^*\|_2^2\\
&= \frac{1}{2}\|\mathbf{A}(\mathbf{x} - \mathbf{x}^*)\|_2^2
\end{aligned}
$$

(f) **Show that**

$$f(\mathbf{x}_k) - f(\mathbf{x}^*) \leq \frac{\alpha}{2}\|\mathbf{x}_k - \mathbf{x}^*\|_2^2,$$

for $\alpha = \lambda_{\max}(\mathbf{A}^\top\mathbf{A})$, and conclude that

$$f(\mathbf{x}_k) - f(\mathbf{x}^*) \leq \frac{\alpha}{2}\beta^{2k}\|\mathbf{x}_0 - \mathbf{x}^*\|_2^2.$$

Solution: Starting from the previous part, we have

$$
\begin{aligned}
f(\mathbf{x}_k) - f(\mathbf{x}^*) &= \frac{1}{2}\|\mathbf{A}(\mathbf{x}_k - \mathbf{x}^*)\|_2^2\\
&= (\mathbf{x}_k - \mathbf{x}^*)^\top\mathbf{A}^\top\mathbf{A}(\mathbf{x}_k - \mathbf{x}^*)\\
&\leq \frac{1}{2}\lambda_{\max}(\mathbf{A}^\top\mathbf{A})\|\mathbf{x}_k - \mathbf{x}^*\|_2^2\\
&= \frac{\alpha}{2}\|\mathbf{x}_k - \mathbf{x}^*\|_2^2
\end{aligned}
$$

Combining this with part (d), we get the desired geometric convergence rate for least squares (quadratic functions)

$$f(\mathbf{x}_k) - f(\mathbf{x}^*) \leq \frac{\alpha}{2}\beta^{2k}\|\mathbf{x}_0 - \mathbf{x}^*\|_2^2$$

(g) Finally, the convergence rate is a function of β, so it's desirable for β to be as small as possible. Recall that β is a function of γ, so we want to pick γ such that β is as small as possible, as a function of $\lambda_{\min}(\mathbf{A}^\top\mathbf{A}), \lambda_{\max}(\mathbf{A}^\top\mathbf{A})$. **Write the resulting convergence rate as a function of $\kappa = \frac{\lambda_{\max}(\mathbf{A}^\top\mathbf{A})}{\lambda_{\min}(\mathbf{A}^\top\mathbf{A})}$, That is, show that**

$$f(\mathbf{x}_k) - f(\mathbf{x}^*) \leq \frac{\alpha}{2}\left(\frac{\kappa - 1}{\kappa + 1}\right)^{2k}\|\mathbf{x}_0 - \mathbf{x}^*\|_2^2$$

Solution: We would like to solve

$$\min_{\gamma} \beta(\gamma) = \min_{\gamma} \max\{|1 - \gamma\lambda_{\max}(\mathbf{A}^\top\mathbf{A})|, |1 - \gamma\lambda_{\min}(\mathbf{A}^\top\mathbf{A})|\}$$

Recall that $0 < \lambda_{\min}(\mathbf{A}^\top\mathbf{A}) \leq \lambda_{\max}(\mathbf{A}^\top\mathbf{A})$ and that $\gamma > 0$. Then, the optimal solution is when the two values are equal in absolute value but opposite in value.

$$-(1 - \gamma\lambda_{\max}(\mathbf{A}^\top\mathbf{A})) = 1 - \gamma\lambda_{\min}(\mathbf{A}^\top\mathbf{A})$$

This gives $\gamma = \frac{2}{\lambda_{\max}(\mathbf{A}^\top\mathbf{A}) + \lambda_{\min}(\mathbf{A}^\top\mathbf{A})}$.

Then, the convergence result from the previous part can be written as

$$\begin{aligned} f(\mathbf{x}_k) - f(\mathbf{x}^*) &= \frac{\alpha}{2}\left(\frac{\lambda_{\max}(\mathbf{A}^\top\mathbf{A}) - \lambda_{\min}(\mathbf{A}^\top\mathbf{A})}{\lambda_{\max}(\mathbf{A}^\top\mathbf{A}) + \lambda_{\min}(\mathbf{A}^\top\mathbf{A})}\right)^{2k} \|\mathbf{x}_0 - \mathbf{x}^*\|_2^2 \\ &= \frac{\alpha}{2}\left(\frac{\kappa - 1}{\kappa + 1}\right)^{2k} \|\mathbf{x}_0 - \mathbf{x}^*\|_2^2 \end{aligned}$$

4 Sensors, Objects, and Localization

In this problem, we will be using gradient descent to solve the problem of figuring out where objects are, given noisy distance measurements. (This is roughly how GPS works and students who have taken EE16A have seen a variation on this problem in lecture and lab.)

First, the setup. Let us say there are m sensors and n objects located in a two dimensional plane. The m sensors are located at the points $(a_1, b_1), \ldots, (a_m, b_m)$. The n objects are located at the points $(x_1, y_1), \ldots, (x_n, y_n)$. We have measurements for the distances between the sensors and the objects: D_{ij} is the measured distance from sensor i to object j. The distance measurement has noise in it. Specifically, we model

$$D_{ij} = ||(a_i, b_i) - (x_j, y_j)|| + Z_{ij},$$

where $Z_{ij} \sim N(0, 1)$. The noise is independent across different measuments.

Code has been provided for data generation to aid your explorations.

For this problem, all Python libraries are permitted.

(a) Consider the case where $m = 7$ and $n = 1$. That is, there are 7 sensors and 1 object. Suppose that we know the exact location of the 7 sensors but not the 1 object. We have 7 measurements of the distances from each sensor to the object $D_{i1} = d_i$ for $i = 1, \ldots, 7$. Because the underlying measurement noise is modeled as iid Gaussian, the interesting part of the log likelihood function is

$$L(x_1, y_1) = -\sum_{i=1}^{7} (\sqrt{(a_i - x_1)^2 + (b_i - y_1)^2} - d_i)^2, \tag{1}$$

ignoring the constant term. **Manually compute the symbolic gradient of the log likelihood function, with respect to x_1 and y_1.**

Solution: We identify x, y with x_1, y_1 respectively for notational simplicity. We have

$$D_{i1} \sim N(\sqrt{(a_i - x)^2 + (b_i - y)^2}, 1). \tag{2}$$

The log likelihood function is

$$\begin{aligned} L(x, y) &= \sum_{i=1}^{7} \log P(D_{i1} = d_i) \\ &= -\sum_{i=1}^{7} (\sqrt{(a_i - x)^2 + (b_i - y)^2} - d_i)^2 + \text{Constant} \end{aligned}$$

For notational simplicity, define $\phi_i(x, y) = \sqrt{(a_i - x)^2 + (b_i - y)^2}$. Then the gradient of $\phi_i(x, y)$ with respect to x, y is

$$\nabla \phi_i(x, y) = (\frac{x - a_i}{\phi_i(x, y)}, \frac{y - b_i}{\phi_i(x, y)})^T.$$

The gradient of $L(x, y)$ is

$$\begin{aligned} \nabla L(x, y) &= \nabla[-\sum_{i=1}^{7} (\phi_i(x, y) - d_i)^2] \\ &= -2 \sum_{i=1}^{7} (\phi_i(x, y) - d_i) \nabla \phi_i(x, y) \\ &= -2 \sum_{i=1}^{7} \frac{\phi_i(x, y) - d_i}{\phi_i(x, y)} (x - a_i, y - b_i)^T, \end{aligned}$$

where the second line follows by the chain rule and the third line follows from plugging in the gradient of ϕ_i directly.

(b) The provided code generates

- $m = 7$ sensor locations (a_i, b_i) sampled from $N(\mathbf{0}, \sigma_s^2 \mathbf{I})$
- $n = 1$ object locations (x_1, y_1) sampled from $N(\boldsymbol{\mu}, \sigma_o^2 \mathbf{I})$
- $mn = 7$ distance measurements $D_{i1} = ||(a_i, b_i) - (x_1, y_1)|| + N(0, 1)$.

for $\boldsymbol{\mu} = [0, 0]^T$, $\sigma_s = 100$ and $\sigma_o = 100$. **Solve for the maximum likelihood estimator of (x_1, y_1) by gradient descent on the negative log-likelihood. Report the estimated (x_1, y_1) for the given sensor locations.** Try two approaches for initializing gradient descent: starting at $\mathbf{0}$ and starting at a random point. Which of the following step sizes is a reasonable one, $1, 0.01, 0.001$ or 0.0001?

Solution: Please refer to the solution code. We should use a step size that is not too large to make sure that the algorithm converges and also avoid choosing a step size that is too small so that the algorithm converges faster.

```
1
2  ########################################################################
3  #########  Results ######################################################
4  ########################################################################
5  The real object location is
6  [[ 44.38632327  33.36743274]]
7  The estimated object location with zero initialization is
8  [[ 43.07188426  32.71217807]]
9  The estimated object location with random initialization is
10 [[ 43.07188426  32.71217807]]
```

(c) (Local Mimima of Gradient Descent) In this part, we vary the location of the single object among different positions:

$$(x_1, y_1) \in \{(0,0), (100,100), (200,200), \ldots, (900,900)\}.$$

For each choice of (x_1, y_1), **generate the following data set** 10 **times**:

- Generate $m = 7$ sensor locations (a_i, b_i) from $N(\mathbf{0}, \sigma_s^2 \mathbf{I})$ (Use the same σ_s from the previous part.)
- Generate $mn = 7$ distance measurements $D_{i1} = ||(a_i, b_i) - (x_1, y_1)|| + N(0,1)$.

For each data set, run the gradient descent methods 100 **times to find a prediction for** (x_1, y_1). We are pretending we do not know (x_1, y_1) and are trying to predict it. For each gradient descent, take 1000 iterations with step-size 0.1 and a random initialization of (x, y) from $N(\mathbf{0}, \sigma^2 \mathbf{I})$, where $\sigma = x_1 + 1$.

- **Draw the contour plot of the log likelihood function of a particular data set for** $(x_1, y_1) = (0,0)$ **and** $(x_1, y_1) = (200, 200)$**.**
- For each of the ten data sets and each of the ten choices of (x_1, y_1), calculate the number of distinct points that gradient descent converges to. Then, for each of the ten choices of (x_1, y_1), calculate the average of the number of distinct points over the ten data sets. **Plot the average number of local minima against** x_1**.** For this problem, two local minima are considered identical if their distance is within 0.01.

  Hint: `np.unique` and `np.round` will help.
- For each of the ten data sets and each of the ten choices of (x_1, y_1), calculate the proportion of gradient descents which converge to what you believe to be a global minimum (that is, the minimum point in the set of local minima that you have found). Then, for each of the ten choices of (x_1, y_1), calculate the average of the proportion over the ten data sets. **Plot the average proportion against** x_1**.**
- For the object location of (500, 500) and one trail out of 10 of the data generation, plot the sensor locations, the ground truth object location and the MLE object locations found by 100 times of gradient descent. Do you find any patterns?

Please be aware that the code might take a while to run.

Solution: Please refer to the solution code and figures 1, 2 and 3. You will notice that as we get to the larger values for the true location of the object, gradient descent converges to more and more local minima. This is happening for two reasons. First, we are creating more local minima because the distance of the object to the sensors is slowly dominating the distance between the sensors themselves. Image the extreme case where sensors are within a radius of 1 while the object is about 1×10^8 away. Then the distances of the object to each of the sensor are almost the same. And the estimated location of the object can be anywhere around the circle of radius 1×10^8. In other words, all of these places would be local minima. Second, gradient descent is starting further away from the true optimal, which increases the chances that gradient descent get diverted to a local minima along the way. From figure 3, we can see that the recovered local minima roughly lies on a circle. Thus, in this case, most of the local minimas are caused by the large distance between the object and the sensors.

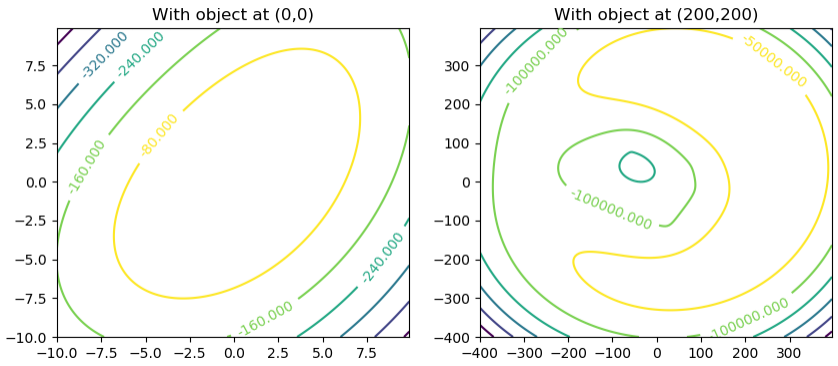


Figure 1

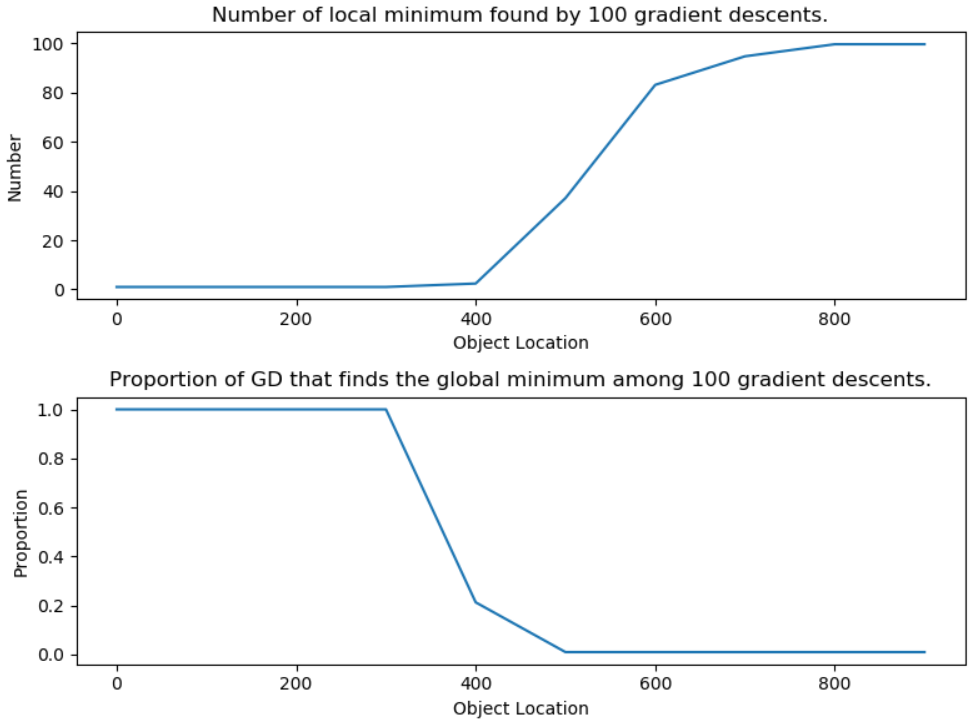


Figure 2

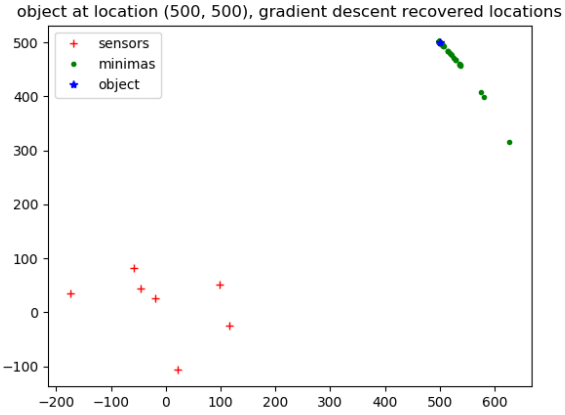


Figure 3

(d) Repeat the previous part, except explore what happens as you reduce the variance of the measurement noise. **Comment with the same plots justifying your comments.**

For the sake of saving time, you can experiment with only one smaller noise, such as $\mathcal{N}(0, 0.01^2)$. You might need to change the learning rate to make the gradient descent converge.

Solution: For the sake of this solution, it is important to distinguish between the process of generating the D_{i1} and the optimization process for a fixed set of D_{i1}.

First, think about how incorrect local minima might arise from the data generation process. The first way that local minima are created is by noise in the neighborhood of the global minimum. This creates local minima that are near the true object location, but their distance from the true object location will grow with the noise σ. The second way local minima are created is by bad geometry, meaning that there is not a unique global minimum (because the sensors are not located in a way that can disambiguate certain points). Decreasing noise helps with the first case by making the objective value at the true object location deeper, while it does not help in the second case. To be more concrete, imagine the extreme case where the noise σ is very small (practically 0). The true object location is a global minimum. The only other minima are points which are almost exactly D_{i1} from each of the i; the geometric ambiguities.

The argument above explains how having smaller noise in data generation can create an optimization landscape with fewer local minima in the neighborhood of the global minimum. The argument below will show that once the optimization landscape is fixed, the choice of σ only scales our gradient descent algorithm.

Consider the optimization process once the D_{i1} are fixed. Reducing the variance of measurement to $\sigma^2 < 1$ will scale the gradient at every point by $\frac{1}{\sigma^2}$ (check this by re-writing the log likelihood function). We can counteract this by making our step size significantly smaller. The result is that these two changes cancel each other out and we end up taking the same set of steps as before if we start at the same initialization.

Overall, we do not see significant changes in our results in practice. If you did see an effect, hopefully GD improved with smaller noise σ. If you saw erratic behavior when you changed σ^2, it is possible that you did not change the step size and thus your gradient was huge and your GD diverged.

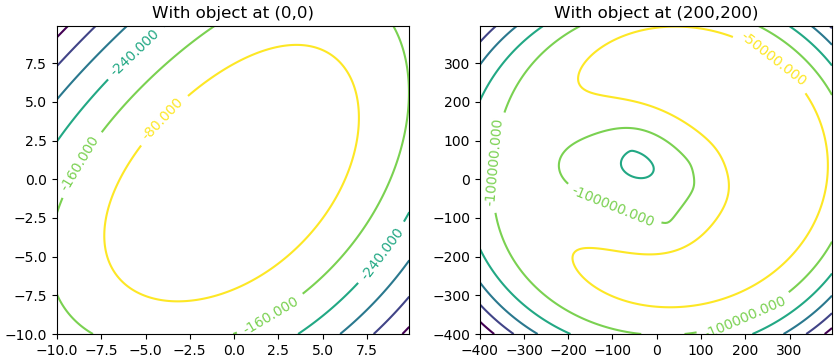


Figure 4

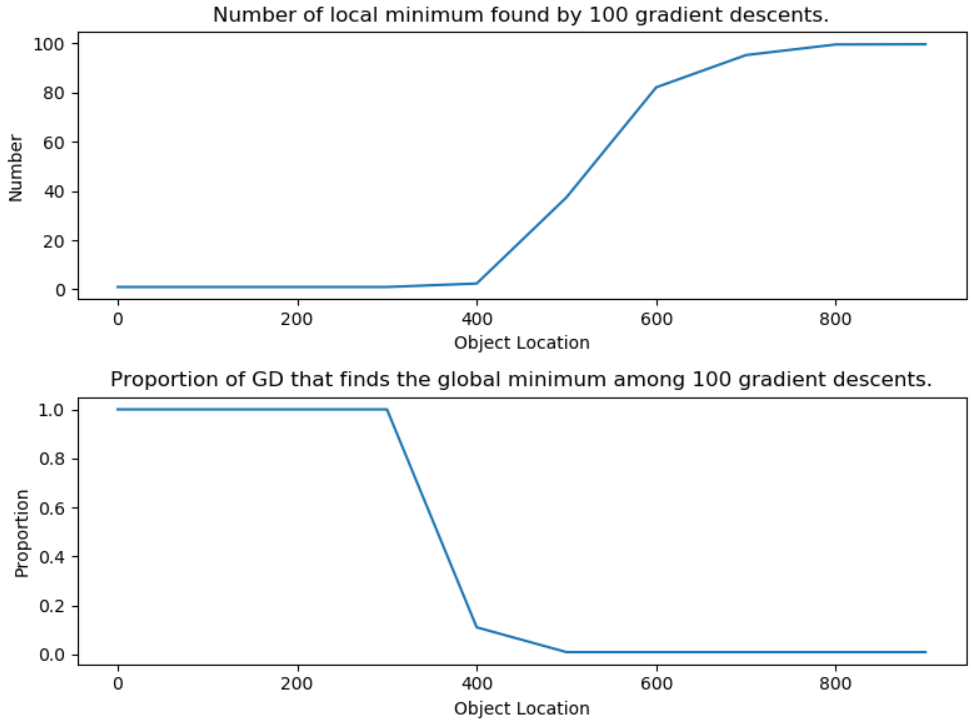


Figure 5

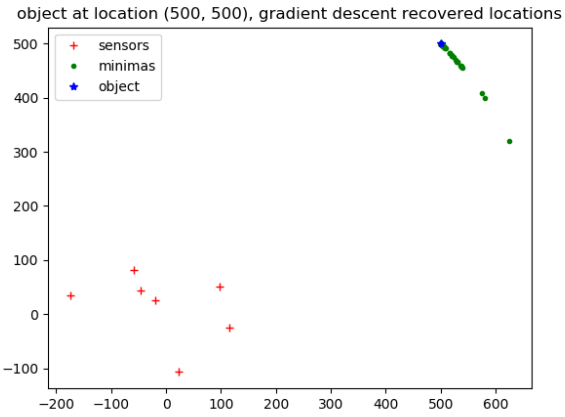


Figure 6

See figures 4, 5 and 6.

(e) Repeat the part (c) again, except explore what happens as you increase the number of sensors. For the sake of saving time, you can experiment with only one number of sensors, such as 20. **Comment with appropriate plots justifying your comments.**

Solution: Intuitively, more sensors lead to more measurements from different locations. This can help reduce the negative effects of measurement errors and locate the object.

Let's be a little more precise and explore what happens to local minima. When we increase the number of sensors, the log likelihood will change to include more information about the location of the object. Imagine that we examine the global minimum and a local minimum before and after adding a new sensor. When we add the new sensor, the local minimum will be pulled upwards more than the global minimum. That is because the global minimum already roughly "agrees" with the data from the new sensor (that is, adds a small penalty to the likelihood) while the local minimum might "disagree" (that is, adds a strong penalty to the likelihood). Ultimately, this addition of sensors will eliminate some of the local minima and so we expect gradient descent to work better. See figures 7 and 8.

Consider a concrete example where more sensors can help determine the location of the object better geometrically. Imagine the case where we only have two sensors at $(-1, 0)$ and $(1, 0)$ and each report a distance of $\sim \sqrt{2}$ from the object. Then we cannot decide whether an object is around $(0, 1)$ or around $(0, -1)$. A new sensor, preferably one which is far from the x-axis, will likely resolve this issue.

For the (500, 500) object, 20 sensors help to resolves the local minima problem.

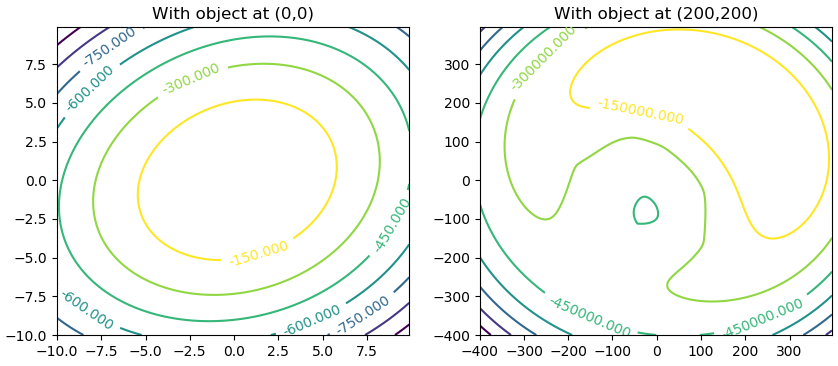


Figure 7

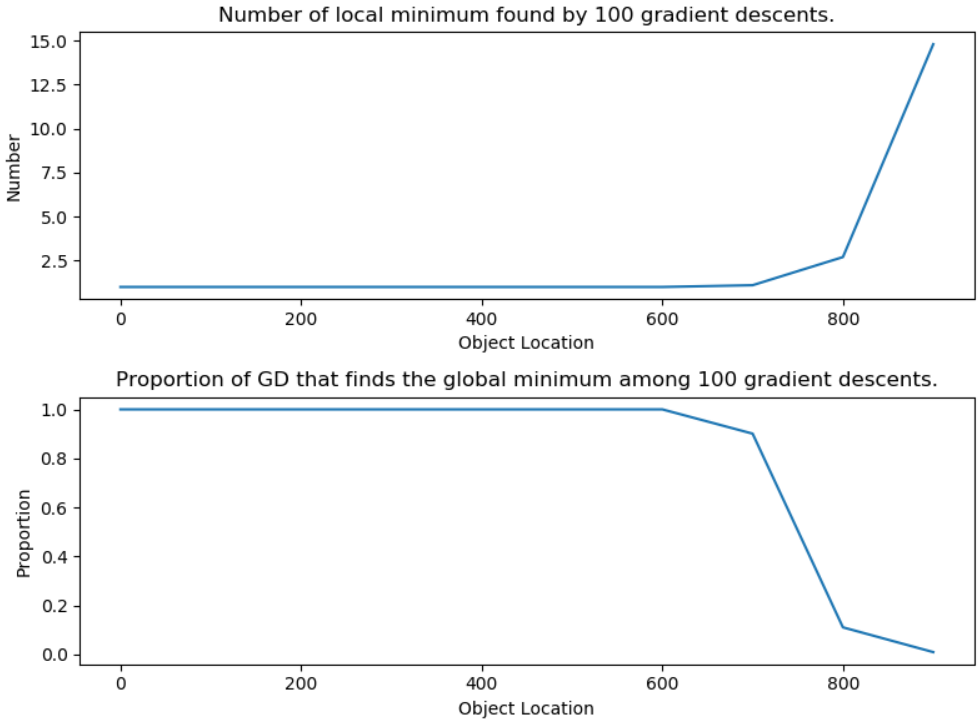
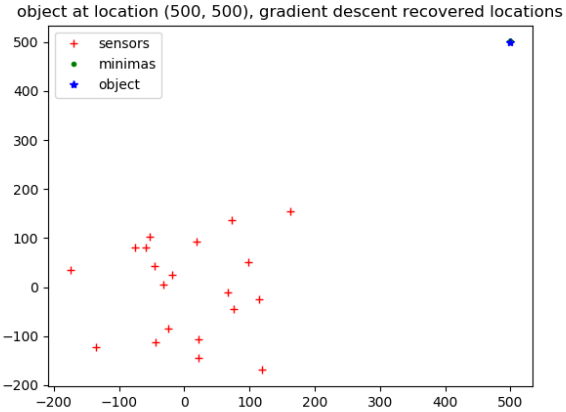


Figure 8



(f) Now, we are going to turn things around. Instead of assuming that we know where the sensors are, suppose that the sensor locations are unknown. But we get some training data for 100 object locations that are known. We want to use gradient descent to estimate the sensor locations, and then use these estimated sensor locations on new test data for objects.

Consider the case where $m = 7$ sensors and the training data consists of $n = 100$ object positions. We have 7 noisy measurements of the distances from each sensor to the object $D_{i1} = d_{ij}$ for $i = 1, \ldots, 7; j = 1, 2, \ldots, 100$.

Use the provided code to generate

- $m = 7$ sensor locations (a_i, b_i) sampled from $N(\mathbf{0}, \sigma^2\mathbf{I})$
- $n = 100$ object locations (x_j, y_j) sampled from $N(\boldsymbol{\mu}, \sigma^2\mathbf{I})$ in 3 datasets: (1) Training data with $\boldsymbol{\mu} = \mathbf{0}$, (2) Interpolating Test data with $\boldsymbol{\mu} = \mathbf{0}$, and (3) Extrapolating Test data with $\boldsymbol{\mu} = [300, 300]^T$.
- $mn = 700$ distance measurements $D_{ij} = ||(a_i, b_i) - (x_j, y_j)|| + N(0, 1)$ for each of the data sets.
- Use the same σ as before, i.e. $\sigma = 100$.

Use the first dataset as the training data and the second two as two kinds of test data: points drawn similarly to the training data, and points drawn in different way.

Use gradient descent to calculate the MLE for the sensor locations $(\hat{a}_i, \hat{b}_i)$ given the training object locations (x_j, x_j) and all the pairwise training distance measurements $(D_{ij} = d_{ij})$. (Use gradient descent with multiple random starts, picking the best estimates as your estimate.)

Use these estimated sensor locations as though they were true sensor locations to compute object locations for both sets of test data. (Use gradient descent with multiple random starts, picking the best estimate as your estimated position.) **Report the mean-squared error in object positions on both test data sets. Also report the MSE on the second test set if we know the testing mean is (300, 300) (such that we can have a better initial guess in the gradient descent).**

Solution: See the solution code. This problem is closely tied to the idea of parameter estimation, where we estimate the parameters of a model (in this case the sensors) and then use those parameters to predict more outputs of the model in an iterative fashion.

To solve this problem, we need to make a few key observations. First, the likelihood is symmetric. If we want to get the location of 7 unknown sensors from 50 known objects, it is the same as getting the location of 7 unknown objects from 50 known sensors. For that reason, learning the location of the 7 sensors from the 50 training objects is a problem we have already solved. Second, we notice that when the sensors are in a fixed location, finding the location of 50 objects can be broken down into finding the location of each of the 50 objects one at a time. More formally, we could say the likelihood function is separable.

The MSE reported are composed of two parts. First, there is the error that results from the difference between the estimated parameters and the true sensor locations. Second, there is the error that results from the noise of distance measurements. The first part can decrease when more training data is added. The second part cannot.

Since we trained the model with object locations around the origin, testing on the objects with similar location results in a smaller error (case 1) than the objects that are located further away.

The MSE for Case 1 is 0.9374

The MSE for Case 2 is 3903.712

The MSE for Case 2 (if we knew mu is [300,300]) is 2.862

5 Backpropagation Algorithm for Neural Networks

In this problem, we will be implementing the backprop algorithm to train a neural network to approximate the function

$$f(x) = \sin(x).$$

To establish notation for this problem, the output of layer i given the input $\mathbf{a}_i$ in the neural network is given by

$$\mathbf{a}_{i+1} = l_i(\mathbf{a}_i) = \sigma(\mathbf{z}_i) = \sigma(\mathbf{W}_i\mathbf{a}_i + \mathbf{b}_i).$$

In this equation, $\mathbf{W}_i$ is a $n_{i+1} \times m_i$ matrix that maps the input $\mathbf{a}_i$ of dimension m_i to a vector of dimension n_{i+1}, where n_{i+1} is the size of layer $i+1$ and we have that $m_i = n_i$. The vector $\mathbf{b}_i$ is the bias vector added after the matrix multiplication, and σ is the nonlinear function applied element-wise to the result of the matrix multiplication and addition. $\mathbf{z}_i = \mathbf{W}_i\mathbf{a}_i + \mathbf{b}_i$ is a shorthand for the intermediate result within layer i before applying the nonlinear activation function σ. Each layer is computed sequentially where the output of one layer is used as the input to the next. To compute the derivatives with respect to the weights $\mathbf{W}_i$ and the biases $\mathbf{b}_i$ of each layer, we use the chain rule starting with the output of the network and work our way backwards through the layers, which is where the backprop algorithm gets its name.

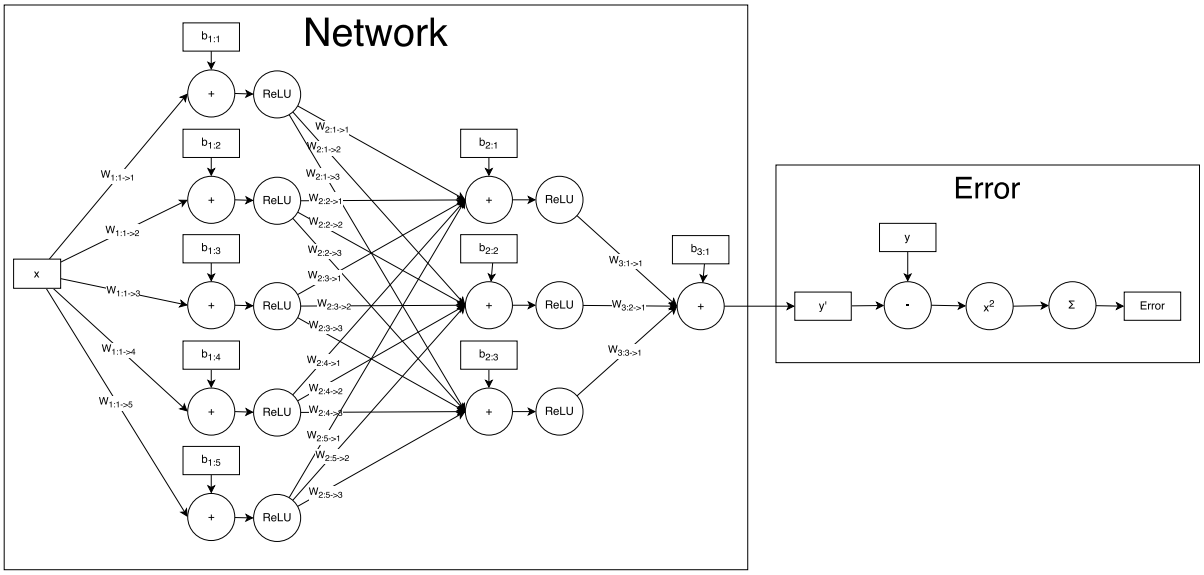
You are given starter code with incomplete function implementations. For this problem, you will fill in the missing code so that we can train a neural network to learn the function $f(x) = \sin(x)$. The code currently trains a network with two hidden layers with 100 nodes per layer. Later in this problem, you will be exploring how the number of layers and the number of nodes per layer affects the approximation.

(a) **Start by drawing a small example network with three computational layers, where the last layer has a single scalar output.** The first layer should have a single external input corresponding to the input x. The computational layers should have widths of 5, 3, and 1 respectively. The final "output" layer's "nonlinearity" should be a linear unit that just returns its input. The earlier "hidden" layers should have ReLU units. Label all the n_i and m_i as well as all the $\mathbf{a}_i$ and $\mathbf{W}_i$ and $\mathbf{b}_i$ weights. You can consider the bias terms to be weights connected to a dummy unit whose output is always 1 for the purpose of labeling. You can also draw and label the loss function that will be important during training — use a squared-error loss.

Here, the important thing is for you to understand your own clear ways to illustrate neural nets. You can follow conventions seen online or in lecture or discussion, or you can modify those conventions to pick something that makes the most sense to you. The important thing is to have your illustration be unambiguous so you can use it to help understand the forward flow of information during evaluation and the backward flow during gradient computations. Since you're going to be implementing all this during this problem, it is good to be clear.

Solution: The following diagram is a sample of what you could have drawn for your answer. Here, the weights a marked as $W_{i:j->k}$ which indicates the weight in layer i going from the

jth input to the kth output. The weights are drawn along the lines connecting graph nodes to indicate multiplication along the connection. Operations are indicated with circles, and model parameters and inputs are indicated with rectangles.



(b) Let's start by implementing the cost function of the network. This function is used to assign an error for each prediction made by the network during training. The implementation will be using the mean squared error cost function, which is given by

$$\text{MSE}(\hat{\mathbf{y}}) = \frac{1}{2}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

where y_i is the observation that we want the neural network to output and $\hat{y}_i$ is the prediction from the network.

Write the derivative of the mean squared error cost function with respect to the predicted outputs $\hat{\mathbf{y}}$. In `backprop.py` implement the functions `QuadraticCost.fx` and `QuadraticCost.dx`

Solution: The derivative is given by

$$\begin{aligned}\frac{\partial \text{MSE}}{\partial \hat{\mathbf{y}}} &= 2\frac{1}{2}(-1)(\mathbf{y} - \hat{\mathbf{y}}) \\ &= \hat{\mathbf{y}} - \mathbf{y}\end{aligned}$$

Please see the solution code for the implementation.

(c) Now, let's take the derivatives of the nonlinear activation functions used in the network. **Implement the following nonlinear functions in the code and their derivatives:**

$$\sigma_{\text{linear}}(z) = z$$

$$\sigma_{\text{ReLU}}(z) = \begin{cases} 0 & z < 0 \\ z & \text{otherwise} \end{cases}$$

$$\sigma_{\tanh}(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

For the `tanh` function, feel free to use the `tanh` function in `numpy`. We have provided the sigmoid function as an example activation function.

Solution:

$$\frac{\partial \sigma_{\text{linear}}}{\partial z} = 1$$

$$\frac{\partial \sigma_{\text{ReLU}}}{\partial z} = \begin{cases} 0 & z < 0 \\ 1 & \text{otherwise} \end{cases}$$

$$\frac{\partial \sigma_{\tanh}}{\partial z} = 1 - \sigma_{\tanh}(z)^2$$

Note in particular for $\sigma_{\tanh}$ there are multiple valid ways to express its derivative.

Please see the code for the implementation.

(d) We have implemented the forward propagation part of the network for you (see `Model.evaluate` in the code). We now need to compute the derivative of the cost function with respect to the weights $\mathbf{W}$ and the biases $\mathbf{b}$ of each layer in the network. We will be using all of the code we previously implemented to help us compute these gradients. **Assume that $\frac{\partial \text{MSE}}{\partial \mathbf{a}_{i+1}}$ is given, where $\mathbf{a}_{i+1}$ is the input to layer $i+1$. Write the expression for $\frac{\partial \text{MSE}}{\partial \mathbf{a}_i}$ in terms of $\frac{\partial \text{MSE}}{\partial \mathbf{a}_{i+1}}$. Then implement these derivative calcualtions in the function `Model.compute_gradient`.** Recall, $\mathbf{a}_{i+1}$ is given by

$$\mathbf{a}_{i+1} = l_i(\mathbf{a}_i) = \sigma(\mathbf{z}_i) = \sigma(\mathbf{W}_i \mathbf{a}_i + \mathbf{b}_i)$$

Solution: We start with looking at the formula that takes the activations from layer i and computes the activations for layer $i+1$:

$$\mathbf{a}_{i+1} = \sigma(W_i \mathbf{a}_i + \mathbf{b}_i).$$

We know that the error is computed as a composition of the layers in the network, so we can use the chain rule to help us compute $\frac{\partial \text{MSE}}{\partial \mathbf{a}_i}$ by using

$$\frac{\partial \text{MSE}}{\partial \mathbf{a}_i} = \frac{\partial \text{MSE}}{\partial \mathbf{a}_{i+1}} \frac{\partial}{\partial \mathbf{a}_i} \left(\sigma(W_i \mathbf{a}_i + \mathbf{b}_i)\right)$$
$$= \frac{\partial \text{MSE}}{\partial \mathbf{a}_{i+1}} \sigma'(W_i \mathbf{a}_i + \mathbf{b}_i) W_i$$

Please see the code for the implementation.

(e) To help you debug, we have implemented a numerical gradient calculator. **Use the starter code to compare and verify your gradient implementation with the numerical gradient calculator. Include the output numbers comparing the differences between the two gradient calculations in your writeup.**

Solution:

```
('squared difference of layer 0:', 2.8053187673930506e-10)
('squared difference of layer 1:', 1.848900869808921e-10)
('squared difference of layer 2:', 1.0137849593546456e-10)
```

If you changed the random seed / made other slight modifications, the squared differences may be a bit different but should be close to 0.

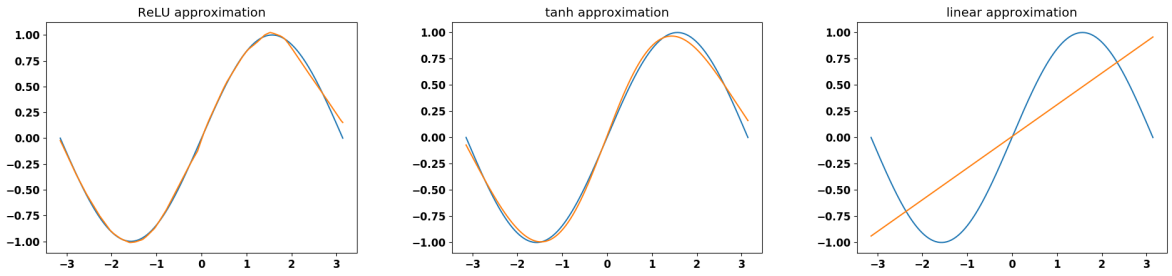
(f) Finally, we use these gradients to update the model parameters using gradient descent. **Implement the function `GDOptimizer.update` to update the parameters in each layer of the network.** You will need to use the derivatives $\frac{\partial \text{MSE}}{\partial \mathbf{z}_i}$ and the outputs of each layer $\mathbf{a}_i$ to compute the derivates $\frac{\partial \text{MSE}}{\partial \mathbf{W}_i}$ and $\frac{\partial \text{MSE}}{\partial \mathbf{b}_i}$. Use the learning rate η, given by `self.eta` in the function, to scale the gradients when using them to update the model parameters. Normalize the learning rate by the number of inputs to the network.

Solution: Please see the code for the implementation.

(g) **Show your results by reporting the mean squared error of the network when using the** $ReLU$ **nonlinearity, the** $\tanh$ **nonlinearity, and the linear activation function. Additionally, make a plot of the approximated** $\sin$ **function in the range** $[-\pi, \pi]$**.** Use the model parameters, the learning rate, and the training iterations provided in the code to train the models. When you have all of the above parts implemented, you will just need to copy the output of the script when you run `backprop.py`.

Solution:

```
ReLU MSE: 0.000408666485428
linear MSE: 0.0967346963549
tanh MSE: 0.000991349026287
```



(h) Let's now explore how the number of layers and the number of hidden nodes per layer affects the approximation. **Train a models using the tanh and the ReLU activation functions with** 5**,** 10**,** 25**, and** 50 **hidden nodes per layer (width) and** 1**,** 2**, and** 3 **hidden layers (depth)**. Use the same training iterations and learning rate from the starter code. **Report the resulting error on the training set after training for each combination of parameters.**

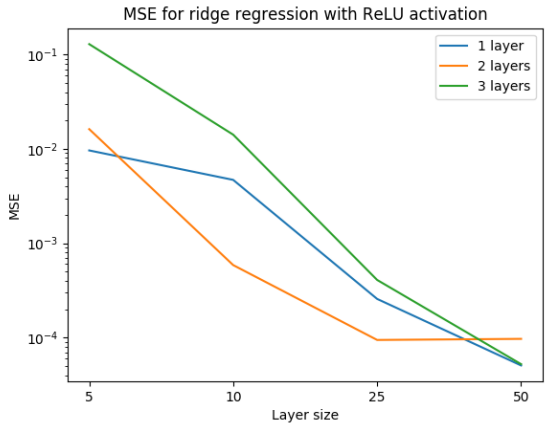
Solution: See the implementation for details.

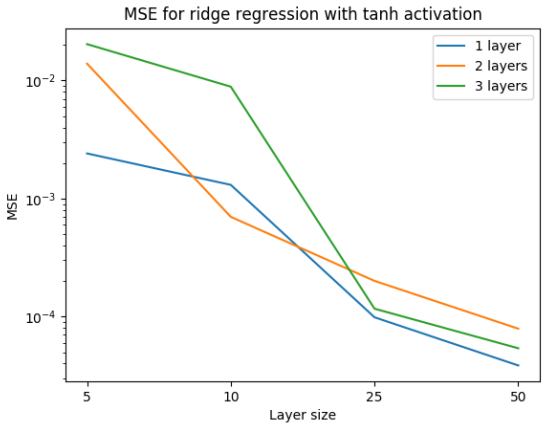
```
ReLU MSE Error
 Layers  5 nodes  10 nodes 25 nodes 50 nodes
      1 0.01252  0.01695  0.01880  0.01047
      2 0.02023  0.00246  0.00296  0.00189
      3 0.02919  0.01120  0.00116  0.00138
tanh MSE Error
 Layers  5 nodes  10 nodes 25 nodes 50 nodes
      1 0.01474  0.02170  0.00378  0.00514
      2 0.03058  0.00762  0.00429  0.00166
      3 0.02152  0.01160  0.00178  0.00023
```

(i) **Run a shortcut-training approach that doesn't bother to update any of the weights that are inputs to the hidden layers and just leaves them at the starting random values. All it does is treat the outputs of the hidden layers as random features, and does OLS+Ridge to set the final weights. Compare the resulting approximation by both plots and mean-squared-error values for the 24 cases above (2 nonlinearities times 4 widths times 3 depths). Comment on what you see.**

Solution:

```
ReLU MSE Error
 Layers  5 nodes  10 nodes 25 nodes 50 nodes
      1 0.00333  0.00348  0.00135  0.00006
      2 0.01612  0.00114  0.00018  0.00008
      3 0.02919  0.00580  0.00017  0.00001
tanh MSE Error
 Layers  5 nodes  10 nodes 25 nodes 50 nodes
      1 0.01321  0.00250  0.00029  0.00013
      2 0.01237  0.00028  0.00006  0.00003
      3 0.02161  0.00189  0.00025  0.00008
```





From these plots, we can see that the general trend is that the MSE error is smaller when we increase the number of nodes per layer. While we might expect more layers would also decrease MSE, that apparently does not occur here (in general though, we would expect it to).

(j) **Bonus:** Modify the code to implement stochastic gradient descent where the batch size is given as a parameter to the function `Model.train`. **Choose several different batch sizes and report the final error on the training set given the batch sizes.**

Solution: See the code for the SGD implementation. The example output given from the solution code is presented in the form `activation(batch size) MSE: error` These errors are smaller than those from part b for `ReLU` and `tanh` because a larger total number of gradient steps were taken during training.

```
ReLU(1) MSE: 7.5930232028e-05
ReLU(10) MSE: 4.87527661144e-06
ReLU(100) MSE: 8.32726468519e-06
tanh(1) MSE: 5.01032586154e-05
tanh(10) MSE: 6.38810697749e-06
tanh(100) MSE: 1.2967554014e-05
```

(k) **Bonus:** Experiment with using different learning rates. Try both different constant learning rates and rates that change with the iteration number such as one that is proportional to 1/iteration. Feel free to change the number of training iterations. **Report your results with the number of training iterations and the final error on the training set.**

Solution: Answers will vary based on which learning rates were tested and how many training iterations were used.

6 Your Own Question

Write your own question, and provide a thorough solution.

Writing your own problems is a very important way to really learn the material. The famous "Bloom's Taxonomy" that lists the levels of learning is: Remember, Understand, Apply, Analyze, Evaluate, and Create. Using what you know to create is the top-level. We rarely ask you any HW questions about the lowest level of straight-up remembering, expecting you to be able to do that

yourself. (e.g. make yourself flashcards) But we don't want the same to be true about the highest level.

As a practical matter, having some practice at trying to create problems helps you study for exams much better than simply counting on solving existing practice problems. This is because thinking about how to create an interesting problem forces you to really look at the material from the perspective of those who are going to create the exams.

Besides, this is fun. If you want to make a boring problem, go ahead. That is your prerogative. But it is more fun to really engage with the material, discover something interesting, and then come up with a problem that walks others down a journey that lets them share your discovery. You don't have to achieve this every week. But unless you try every week, it probably won't happen ever.