HW07- CS 189

1.
Who else did you work with on this homework? In case of course events, just describe the group. How did you work on this homework? Any comments about the homework?

I worked on this homework with Ehimare Okoyomon, Prashanth Ganeth, and Daniel Mockaitis. We worked by getting together throughout the week and communicating on facebook.

I certify that all solutions are entirely in my words and that I have not looked at another student's solutions. I have credited all external sources in this write up}

Nicholas Lorio, 26089160

Question 6 Own Question.

Where did machine learning neural networks get their name?

Neural networks are loosely inspired by the biological neural networks of animal brains. These systems progressively learn and improve their functions for certain tasks based off of observational data and feedback. This is opposed to task specific programming.

Our brains take in observational data and process it abstractly. Certain neurons fire, detecting abstract features of the world around us. Its a grouping of abstract features/patterns.

Example Observation:
Yuor abiilty to exaimne hgiher-lveel fteaures is waht aollws yuo to unedrtsand tihs.

Neural networks in machine learning use affine tools to interpret and build models from observational data / inputs .

Nicholas Lorio

HW2 - 189.

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

$$f(x) = \sqrt{(x-b)^T(x-b)} = \sqrt{(x^T - b^T)(x-b)}$$

$$= \sqrt{x^Tx - x^Tb - b^Tx + b^Tb}$$

$$\nabla_x f(x) = \frac{1}{2}\left(x^Tx - x^Tb - b^Tx + b^Tb\right)^{-1/2}\left(2x - b - b\right)$$

$\nabla_x(b^Tx)$, $b^T(x+\Delta) = b^Tx + b^T\Delta$ $\quad \delta(b^Tx)/\delta x = b^T$ $\quad \nabla_x(b^Tx) = (b^T)^T = b$

$\nabla_x(x^Tx)$ $\quad (x+\Delta)^T(x+\Delta) = (x^T+\Delta^T)(x+\Delta) = x^Tx + x^T\Delta + \Delta^Tx + \Delta^T\Delta$

$$= (x^T + x^T)\Delta \qquad (\Delta^Tx)^T = (x^T\Delta)^T$$

$$\delta(2x^T)^T = \nabla_x \; 2x$$

$$= \left(x^Tx - x^Tb - b^Tx + b^Tb\right)^{-1/2}(x-b)$$

$$\nabla_x^2 = \begin{cases} -\left((x-b)^T(x-b)\right)^{-3/2}(x-b)(x-b) \\ +I\left((x-b)^T(x-b)\right)^{-1/2} \end{cases}$$

∴ IF $\nabla_x^2(F(x)) \geq 0$ then our function $F(x)$ is convex.

$$\nabla x^2 = -\left((x-b)^T(x-b)\right)^{(-3/2)}(x-b)^T(x-b) + \left((x-b)^T(x-b)\right)^{-1/2}$$

$$= -\left((x-b)^T(x-b)\right)^{-1/2} + I\left((x-b)^T(x-b)\right)^{-1/2}$$

$$= \frac{\|x-b\|_2^2}{\|x-b\|_2^3} + \frac{1}{\|x-b\|_2} = 0 \geq 0 \quad \text{as required for convexity}$$

2.b. Gradient Descent $x_0 = [0,0]$ $x \in \mathbb{R}^2$, $b \in [4.5, 6]$, $t_i = 1$

$\varepsilon = 0.01$, want $\|x_i - x^*\|_2 \leq \varepsilon$

$x_{i+1} = x_i - \nabla f(x_i)$ w/ $t_i = 1$

- Gradient descent will work for our convex function.

$$\nabla f(x) = \left((x-b)^T(x-b)\right)^{-1/2}(x-b) = \left((x-\begin{bmatrix}4.5\\6\end{bmatrix})^T(x-\begin{bmatrix}4.5\\6\end{bmatrix})\right)^{-1/2}(x-\begin{bmatrix}4.5\\6\end{bmatrix})$$

- Set $\nabla f(x)$ equal to zero to solve for our optimal solution.

Convexity means this local min is a global min.

$$\frac{1}{\|x-b\|_2}(x-b) = 0 \qquad \frac{x}{\|x-b\|_2^2} = \frac{b}{\|x-b\|_2^2} \qquad x^* = b = \begin{bmatrix}4.5\\6\end{bmatrix}$$

- $x_1 = \begin{bmatrix}0\\0\end{bmatrix} - \nabla_x f(0) = \begin{bmatrix}0\\0\end{bmatrix} + \left(\left(\begin{bmatrix}4.5\\6\end{bmatrix}^T\begin{bmatrix}4.5\\6\end{bmatrix}\right)^{-1/2}\right)\left(\begin{bmatrix}4.5\\6\end{bmatrix}\right) = \left(+56.25\right)^{-1/2}\begin{bmatrix}4.5\\6\end{bmatrix}$

$(20.25 + 36)$

$$= \begin{bmatrix}+0.6\\+0.8\end{bmatrix}$$

- $x_2 = \begin{bmatrix}+0.6\\+0.8\end{bmatrix}$ $\nabla_x f(x_1) =$

2.c. $\|x_{i+1}\|_2 = \|x_i - t_i \nabla F(x_i)\|_2 = \|x_i - (5/6)^i \nabla F(x_i)\|_2$

$x_1 = x_0 - 1 \cdot \nabla F(x_0) = 0 + \frac{b}{\|b\|}$, $\nabla F(x_0) = -\frac{b}{\|b\|}$

$x_2 = x_1 - (5/6)\nabla F(x_1) = x_1 + 5/6 \frac{b}{\|b\|} = \frac{b}{\|b\|}(1 + 5/6)$

$x_n = \frac{b}{\|b\|}\left(\sum_{i=0}^{n-1} (5/6)^i\right) = \frac{b}{\|b\|} \cdot \frac{1}{1 - 5/6} = \frac{b}{\|b\|} \cdot 6$

$\left[x_n = \frac{b}{\|b\|} \cdot 6 \right]$ • converges to a point.
• if optimal point is further away than our converged point than this step size for this method will never reach the optimal point.

closest to our starting x_0.

2.d. $x_1 = x_0 - \frac{1}{1} \nabla F(x_0)$, $\nabla F(x_0) = \left((0-b)^T (0-b) \right)^{-1/2} (0-b) = \frac{-b}{\|0-b\|^2} = \frac{-b}{\|b\|}$

$= 0 - \nabla F(x_0) = +b/\|b\|$

$x_2 = x_1 - \frac{1}{2} \nabla F(x_1)$, $\nabla F(x_1) = \left(\left(b/\|b\| - a \right)^T \left(b/\|b\| - a \right) \right)^{-1/2} \left(b/\|b\| - a \right)$

$= b\left(\frac{1}{\|b\|} - 1\right) / \left\| b\left(\frac{1}{\|b\|} - 1\right) a \right\|^2$

$= -b\left(1 - 1/\|b\|\right) a / \left\| -b\left(1 - 1/\|b\|\right) a \right\| = -b/\|b\|$

$x_2 = \frac{b}{\|b\|} + \frac{1}{2} \frac{b}{\|b\|} = \frac{b}{\|b\|} \left(\frac{2}{2} + \frac{1}{2} \right)$

$x_n = \frac{b}{\|b\|} \left(\sum_{i=1}^{n} \frac{1}{i} \right)$

2.d. cont.

$x_n = b/\|b\| \left(\sum_{i=1}^{n} 1/i\right)$

$\|x_i - x^*\|_2 \leq 0.01$, $\left\| \frac{b}{\|b\|} \sum_{g=1}^{i} \frac{1}{g} - b \right\| \leq 0.01$

$\left\| b\left(\frac{\log(i)}{\|b\|} - 1\right) \right\|_2 \leq 0.01$, $\left(\frac{\log(i)}{\|b\|} - 1\right) \|b\| \leq 0.01$

$\log(i) - \|b\| \leq 0.01$, $\log(i) \leq 0.01 + \|b\|$

$i \geq \exp(0.01 + \|b\|)$ | # iterations for convergence to optimal solution

3. $|u^T v| \le \|u\|_2 \|v\|_2$ $A^T A$ pos def

$F(x) = \frac{1}{2}\|Ax-b\|_2^2$

a. $b = \vec{0}$ Step size: γ, each $x \in \mathbb{R}^d$ state

$$F(x) = \frac{1}{2}\|Ax-b\|_2^2 = \frac{1}{2}(Ax-b)^T(Ax-b) = \frac{1}{2}(x^T A^T Ax - x^T A^T b - b^T Ax + b^T b)$$
$$= \frac{1}{2}(x^T A^T Ax - 2x^T A^T b)$$

$$\nabla_x F(x) = \frac{1}{2}\left((A^T A + A^T A)x - 2A^T b\right) = A^T Ax - A^T b$$

$x_{i+1} = x_i - \gamma \nabla F(x_i)$, $x_1 = x_0 - \gamma \nabla F(x_0) = x_0 - \gamma(A^T Ax_0 - A^T b)$
$= (1 - \gamma A^T A)x_0$

$x_2 = x_1 - \gamma \nabla F(x_1) = x_0 - \gamma A^T Ax_0 + \gamma A^T b - \gamma \nabla F(x_1)$

$= x_0 - \gamma(A^T Ax_0 - A^T b) - \gamma\left(A^T A(x_0 - \gamma(A^T Ax_0 - A^T b)) - A^T b\right)$

remember $b = 0$ $-\gamma A^T Ax_0 + \gamma A^T A \gamma A^T Ax_0$

$= x_0 - \gamma A^T Ax_0 - \gamma A^T Ax_0 + \gamma A^T A \gamma A^T Ax_0$

$= x_0 - \gamma(2A^T Ax_0 - A^T A \gamma A^T Ax_0) = (1 - \gamma 2A^T A + \gamma A^T A \gamma A^T A)x_0$
$= (I - \gamma A^T A)^2 x_0$

$x_3 = x_2 - \gamma\left(A^T A(x_0 - \gamma(2A^T Ax_0 - A^T A \gamma A^T Ax_0))\right)$

$= x_0 - \gamma(2A^T Ax_0 - A^T A \gamma A^T Ax_0) - \gamma(A^T A(x_0 - \gamma(2A^T Ax_0 - A^T A \gamma A^T Ax_0)))$

$$\boxed{x_\infty = (I - \gamma A^T A)^\infty x_0}$$

3.b. all e/s $B \leq 1$ in absolute value.

$B = I - \gamma A^T A$ from 3.a.

- $A^T A$ is a PSD matrix so $\lambda_i \geq 0$ ①
- γ is a scalar > 0 ②

① $\therefore A^T A = P D P^T$, $I = P P^T P P^T$

thus $I - \gamma A^T A = P(P^T P - \gamma D) P^T$

$$= P \begin{pmatrix} 1-\gamma\lambda_1 & & \\ & \ddots & \\ & & 1-\gamma\lambda_n \end{pmatrix} P^T$$

- we need $\|1-\gamma\lambda_i\| \leq 1 \;\forall i$ $\Rightarrow$ $-1 \leq 1-\gamma\lambda_i \leq 1$
- $\lambda_i > 0$, $\gamma > 0$ $\Rightarrow$ $\gamma\lambda_i \leq 2$

$$\lambda_i \leq \frac{2}{\gamma} \;\forall i$$ else ②

- the evolution for B is stable if: $0 \leq \lambda_i \leq \frac{2}{\gamma}$ $\forall$ eigenvalue λ_i of $A^T A$

$|u^T v| \leq \|u\| \|v\|$

3.c. $\varphi(x) = x - \gamma \nabla F(x)$ $\gamma > 0$ constant step size

Show $\forall\ x, x' \in \mathbb{R}^d$ $\|\varphi(x) - \varphi(x')\|_2 \leq \beta \|x - x'\|_2$

$\beta = \max\left[\,|1 - \gamma \lambda_{max}(A^TA)|, |1 - \gamma \lambda_{min}(A^TA)|\,\right]$

$\|x - \gamma \nabla F(x) - x' + \gamma \nabla F(x')\|_2 = \|x - x' + \gamma(\nabla F(x') - \nabla F(x))\|_2$

$= \|x - x' + \gamma (A^TAx' - A^Tb - A^TAx + A^Tb)\|$ apply on the gradients

$= \|(I - \gamma A^TA)(x - x')\|$

$= \|(I - \gamma A^TA)(x - x')\| \leq \|I - \gamma A^TA\| \, \|x - x'\|$

$\|I - \gamma A^TA\| = \sqrt{\lambda_{max}(I - \gamma A^TA)} = |1 - \gamma \lambda_{max}(A^TA)|$ σ always posm

* From matrix norm wikipedia

$\|A\|_2 = \sqrt{\lambda_{max}(A^TA)} = \sigma_{max}(A)$ (σ must always be positive)

• From 3b, the eigenvalues of $I - \gamma A^TA$ are bounded by $0 \leq \lambda_i \leq \frac{\gamma}{2}$

$\|I - \gamma A^TA\| = \max\left(|1 - \gamma \lambda_{max}(A^TA)|\right) = \max\left[\,|1 - \gamma \lambda_{max}(A^TA)|, |1 - \gamma \lambda_{min}(A^TA)|\,\right]$

for all eigs

$= \beta$

$\therefore$ $\leq \beta \|x - x'\|_2$ as required

3.d. $x^* = \arg\min_{x \in \mathbb{R}^2} f(x)$, $\varphi(x) = x - \gamma \nabla F(x)$

Show: $\|x_{k+1} - x^*\|_2 = \|\varphi(x_k) - \varphi(x^*)\|_2$

From 3.c. we know $\|\varphi(x_k) - \varphi(x^*)\| \leq \beta \|x_k - x^*\|$

$$\left.\begin{aligned} \varphi(x^*) &= x^* - \gamma \nabla F(x^*) \\ \varphi(x) = x_{k+1} &= x_k - \gamma \nabla F(x_k) \end{aligned}\right\} \therefore \|x_{k+1} - x^*\|_2 = \|\varphi(x_k) - \varphi(x^*)\|_2$$

Conclude that $\|x_{k+1} - x^*\|_2 \leq \beta^{k+1} \|x_0 - x^*\|_2$

[•From 3.c

$\|x_{k+1} - x^*\|_2 \leq \beta \|x_k - x^*\|_2$]

$\|x_1 - x^*\|_2 \leq \beta \|x_0 - x^*\|_2$ - From 3.c.

$\|x_2 - x^*\|_2 \leq \beta \|x_1 - x^*\|_2 \leq \beta \left[\beta \|x_0 - x^*\|_2\right]$

$\leq \beta^2 \|x_0 - x^*\|_2$

$\vdots$

$\|x_{k+1} - x^*\|_2 \leq \beta^{k+1} \|x_0 - x^*\|_2$ as required

3.e ① ②

$$f(x) - f(x^*) = \frac{1}{2}\|A(x-x^*)\|_2^2 = \frac{1}{2}\left((x-x^*)^T A^T A (x-x^*)\right)$$

$Ax = Ax^*$ $(Ax - Ax^*)$

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

$$\frac{1}{2}\left[\|Ax-b\|_2^2 - \|Ax^*-b\|_2^2\right] = \frac{1}{2}(x-x^*)^T A^T A (x-x^*)$$

$$\frac{1}{2}\left[x^T A^T Ax - x^T A^T b - b^T Ax + b^T b - x^{*T} A^T Ax^* + x^{*T} A^T b + b^T Ax^* - b^T b\right]$$

$-2b^T Ax$ $\quad +2b^T Ax^*$

x^* is opt Sol^n for $\frac{1}{2}\|Ax-b\|_2^2$ $\quad x^* = (A^TA)^{-1}A^Tb$ · A^TA psd

$$\frac{1}{2}\left(x^T A^T Ax - 2b^T Ax + 2b^T A(A^TA)^{-1}A^Tb - x^{*T}A^TAx^*\right)$$

$q_1 q_2 = b$ plug $x = x^*$

$\cdot\cdot ((A^TA)^{-1}A^Tb)^T A^TA (A^TA)^{-1}A^Tb = q_1^T A^T A q_1 = -b^Tb$

$\cdot 2b^TA(A^TA)^{-1}A^Tb = 2b^TA\,A^{-1}A^{-T}A^Tb = q_1^T q_2 = b^Tb$

$$= \frac{1}{2}\left(x^TA^TAx - 2b^TAx + 2b^Tb\right) = \frac{1}{2}\left(x^TA^TAx - 2b^TAx + b^Tb\right) \quad ①$$

② $\frac{1}{2}\left(x^TA^TAx - x^TA^TAx^* - x^{*T}A^TAx + x^{*T}A^TAx^*\right)$

$$= \frac{1}{2}\left(x^TA^TAx - x^TA^TA(A^TA)^{-1}A^Tb - b^TA(A^TA)^{-1}A^TAx + b^Tb\right)$$

$-x^TA^Tb$ $\quad$ $(A^TA)^{-1}A^Tb$

$$= \frac{1}{2}\left(x^TA^TAx - x^TA^Tb - b^TAx + b^Tb\right) = \frac{1}{2}\left(x^TA^TAx - 2b^TAx + b^Tb\right) \quad ②$$

$$-\left(b^TA(A^TA^{-T})A^TA\,A^{-1}A^{-T}A^Tb = -b^Tb\right)$$

LHS = RHS ① ②

3.f. $F(x_k) - f(x^*) \leq \frac{\alpha}{2}\|x_k - x^*\|_2^2 \qquad \alpha = \lambda_{max}(A^TA)$

Show the above statement:

From 3.e. $f(x_k) - f(x^*) = \frac{1}{2}\|A(x_k - x^*)\|_2^2$:,

$\|A\|_2^2 = \left(\sqrt{\lambda_{max}(A^TA)}\right)^2 = \lambda_{max}(A^TA) = \alpha$

$\leq \frac{\alpha}{2}\|x_k - x^*\|_2^2$

• From 3.d. we conclude the following:

$$\frac{\alpha}{2}\|x_k - x^*\|_2^2 \leq \frac{\alpha}{2}\beta^{2k}\|x_0 - x^*\|_2^2$$

$$\therefore f(x_k) - F(x^*) \leq \frac{\alpha}{2}\beta^{2k}\|x_0 - x^*\|_2^2 \quad \text{as required}$$

3.g. β is function of γ $\qquad \kappa = \frac{\lambda_{max}(A^TA)}{\lambda_{min}(A^TA)}$

already pick γ s.t $\beta = \frac{\kappa - 1}{\kappa + 1} = \frac{\frac{\lambda_{max}(A^TA)}{\lambda_{min}(A^TA)} - 1}{\frac{\lambda_{max}(A^TA)}{\lambda_{min}(A^TA)} + 1}$

$= \frac{\lambda_{max}(A^TA) - \lambda_{min}(A^TA)}{\lambda_{min}(A^TA)} \cdot \frac{\lambda_{min}(A^TA)}{\lambda_{max}(A^TA) + \lambda_{min}(A^TA)}$

$= \frac{\lambda_{max}(A^TA) - \lambda_{min}(A^TA)}{\lambda_{max}(A^TA) + \lambda_{min}(A^TA)} = \max\left\{\left|1 - \gamma\lambda_{max}(A^TA)\right|, \left|1 - \gamma\lambda_{min}(A^TA)\right|\right\}$

(cont)

e.g. $1-\gamma\lambda_{max}(A^TA) = \frac{\lambda_{max}(A^TA)-\lambda_{min}(A^TA)}{\lambda_{max}(A^TA)+\lambda_{min}(A^TA)}$

$\lambda_{max}(A^TA) = M$
$\lambda_{min}(A^TA) = N$

$$1-\gamma M = \frac{M-N}{M+N}, \quad 1-\frac{(M-N)}{M+N} = \gamma M$$

$$\frac{M+N-M+N}{M+N} = \gamma M, \left(\gamma = \frac{2N}{(M+N)M}\right) \quad \frac{2N}{MM+NM}$$

for $1-\lambda_{max}(A^TA)$

same

$$\boxed{\gamma = \frac{2\lambda_{min}(A^TA)}{\lambda_{max}(A^TA)\lambda_{max}(A^TA) + \lambda_{min}(A^TA)\cdot\lambda_{max}(A^TA)}}$$

$\left(\gamma = \frac{2N}{(M+N)N}\right)$ for $1-\lambda_{min}(A^TA)$ $\quad \frac{2N}{MN+NN}$

$$\boxed{\gamma = \frac{2\lambda_{min}(A^TA)}{\lambda_{max}(A^TA)\lambda_{min}(A^TA) + \lambda_{min}(A^TA)\lambda_{min}(A^TA)}}$$

- γ can be either of the above. $\beta = \max\left[|1-\gamma\lambda_{max}(A^TA)|, |1-\gamma\lambda_{min}(A^TA)|\right]$
- want γ s.t. β is as small as possible

M sensors, N objects Distance: D_{ij}
(a_i, b_i) (x_i, y_i)

4.a. $M = 7$, $N = 2$

$$D_{ij} = \|(a_i, b_i) - (x_j, y_j)\| + z_{ij} \quad \sim \mathcal{N}(0, 1)$$

$$L(x, y) = -\sum_{i=1}^{7} \left(\left((a_i - x_1)^2 + (b_i - y_1)^2 \right)^{1/2} - d_i \right)^2$$

$$\nabla_{x,y} \begin{cases} \partial L(x,y)/\partial x = \left\{ -2 \sum_{i=1}^{7} \left(\left((a_i - x_1)^2 + (b_i - y_1)^2 \right)^{1/2} - d_i \right) \right. \\ \qquad \left. \cdot \frac{1}{2} \left((a_i - x_1)^2 + (b_i - y_1)^2 \right)^{-1/2} \cdot 2 (a_i - x_1) \cdot (-1) \right. \\ \partial L(x,y)/\partial y = -2 \sum_{i=1}^{7} \left((a_i - x_1)^2 + (b_i - y_1)^2 \right)^{+1/2} - d_i \cdot \frac{1}{2} \left((a_i - x_1)^2 + (b_i - y_1)^2 \right)^{-1} \\ \qquad \cdot 2 (b_i - y_1)(-1) \end{cases}$$

$$\nabla_{x,y} = \begin{bmatrix} 2 \sum_{i=1}^{7} \left((a_i - x_1)^2 + (b_i - y_1)^2 \right)^{1/2} - d_i \left((a_i - x_1)^2 + (b_i - y_1)^2 \right)^{-1/2} (a_i - x_1) \\ 2 \sum_{i=1}^{7} (\quad // \quad // \quad (b_i - y_1) \end{bmatrix}$$

5. $f(x) = \sin(x)$ $\quad a_{i+1} = l_i(a_i) = \sigma(z_i) = \sigma(W_i a_i + b_i)$ (weights)

$a_{i+1} =$ (nonlinear function applied elementwise)

$W \in \mathbb{R}^{\Lambda_{i+1} \times M_i}$

Λ_{i+1} : size of layer $i+1$

$\partial l_i / \partial a_i =$

b_i : vector of biases

z_i

$M_i = \Lambda_{i-1}$

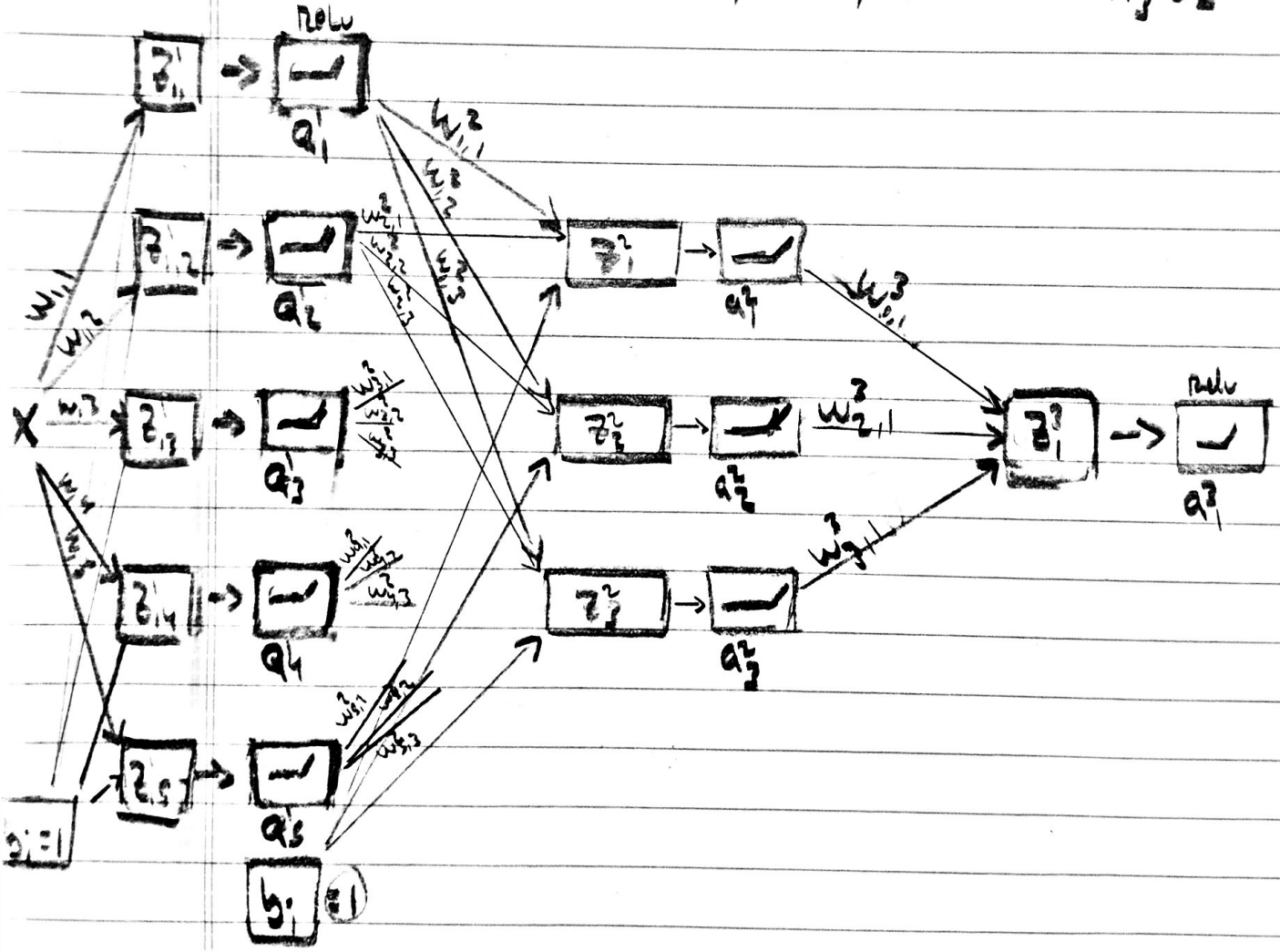
$Wa \to \mathbb{R}^{\Lambda_i \times 2}$ $\quad (W_i a_i + b_i) \; \mathbb{R}^{\Lambda_i \times 2}$

$a_i \in \mathbb{R}^{M_i \times 1}$ $\quad \Lambda_i = 3$

a. example network w/ 3 computational layers $\Lambda_1 = 5$ $b_1 = b_2 = b_3 = 1$

5, 3, 1 $\quad \Lambda_2 = 3$

$\Lambda_3 = 1$



key $\left[w^k_{i,j} \quad k \text{ is the layer applied to} \right.$
$\left. \quad i \text{ - from } i \text{ to } j \text{ elements} \right]$

5.b $MSE(\hat{y}) = \frac{1}{2}\sum_{i=1}^{\hat{}} (y_i - \hat{y}_i)^2$, $f(x) = \sin(x) = y_i$

$\hat{y} = h(x)$

$$\delta(MSE(\hat{y}))/\delta\hat{y} = \frac{1}{2}\sum_{i=1}^{\hat{}} 2(y_i - \hat{y}_i)^1 \cdot (-1) \frac{\delta \hat{y}_i}{\delta \hat{y}}$$

$$= -\sum_{i=1}^{\hat{}} (y_i - \hat{y}_i) \cdot 1 \quad , \quad \boxed{= (\hat{y}_i - y_i)}$$

5.c. • $\sigma_{linear}(z) = z$ • $\delta\sigma_z/\delta z = 1$

• $\sigma_{tanh}(z) = (e^z - e^{-z})(e^z + e^{-z})^{-1} =$

• $\delta\sigma_{tanh}/\delta z = (e^z + e^{-z})(e^z + e^{-z})^{-1} + (e^z - e^{-z})(-1)(e^z + e^{-z})^{-2} \cdot (e^z - e^{-z})$

$= 1 - (e^z - e^{-z})(e^z + e^{-z})^{-2}(e^z - e^{-z})$

$= 1 - \frac{(e^z - e^{-z})^2}{(e^z + e^{-z})^2}$

• $\sigma_{relu}(z) = \begin{cases} 0 & z < 0 \\ z & \text{otherwise} \end{cases}$ • $\frac{\delta\sigma_{relu}(z)}{\delta z} \begin{cases} 0 & z < 0 \\ 1 & \text{o.w.} \end{cases}$

in case they want clearance Nett Sum

5.d. Given: $\delta MSE/\delta a_{i+2}$ a_{i+1} is input to layer $i+1$

$\delta MSE/\delta a_i$ = (in terms of $\frac{\delta MSE}{\delta a_{i+1}}$) or $\left|\frac{\delta MSE}{\delta}\right.$

$a_{i+1} = \sigma(w_i a_i + b_i)$, $\hat{a}_{i+1} = \underbrace{\sigma(w_i a_i + b_i)}_{\text{output}}$

$MSE(\hat{a}_i) = \frac{1}{2}\sum(a_i - \hat{a}_i)$

Back prop so $a_{i+1} \to a_i$

$MSE(\hat{a}_{i+1}) = \frac{1}{2}(a_{i+1} - \hat{a}_{i+1})^2 = \frac{1}{2}(a_{i+1} - \sigma(w_i a_i + b_i))^2$

$MSE(\hat{a}_i) = \frac{1}{2}(a_i - \hat{a}_i)^2 = \frac{1}{2}(a_i - \sigma(w_{i-1} a_{i-1} + b_{i-1}))^2$

$\frac{\delta MSE}{\delta a_{i+1}} = \frac{\delta MSE}{\delta a_{i+1}}$

$\frac{\delta MSE}{\delta a_i} = \frac{1}{2}\frac{\delta((a_i - \hat{a}_{i+1})^2)}{\delta a_{i+1}} \cdot \frac{\delta a_{i+1}}{\delta a_i} =$

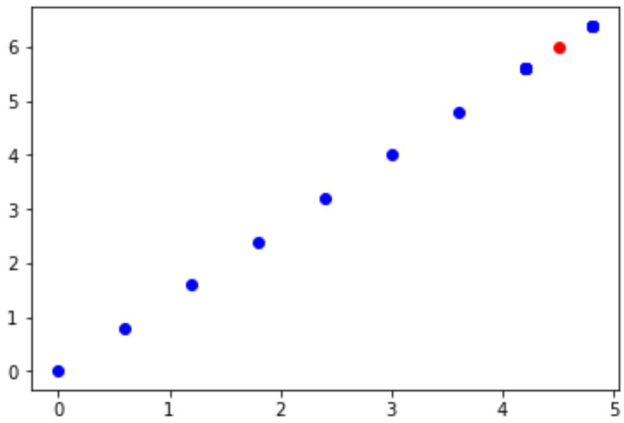
$\frac{\delta a_{i+1}}{\delta a_i} = \frac{\delta(\sigma(z_i))}{\delta z_i} \cdot \frac{\delta z_i}{\delta a_i}$ $\quad z_i = w_i a_i + b_i$

$\frac{\delta MSE}{\delta a_i} = \frac{\delta MSE}{\delta a_{i+1}} \cdot \underbrace{\frac{\delta(\sigma(z_i))}{\delta z_i} \cdot \frac{\delta z_i}{\delta a_i}}_{\frac{\delta a_{i+1}}{\delta a}}$ this is w_i

Question 2 B

In [39]:
```
""" Tools for calculating Gradient Descent for ||Ax-b||. """
import matplotlib.pyplot as plt
import numpy as np



def compute_gradient(A, b, x):
    """Computes the gradient of ||Ax-b|| with respect to x."""
    return np.dot(A.T, (np.dot(A, x) - b)) / np.linalg.norm(np.dot(A, x) - b)


def compute_update(A, b, x, step_count, step_size):
    """Computes the new point after the update at x."""
    return x - step_size(step_count) * compute_gradient(A, b, x)


def compute_updates(A, b, p, total_step_count, step_size):
    """Computes several updates towards the minimum of ||Ax-b|| from p.

    Params:
        b: in the equation ||Ax-b||
        p: initialization point
        total_step_count: number of iterations to calculate
        step_size: function for determining the step size at step i
    """
    positions = [np.array(p)]
    n = 0
    for k in range(total_step_count):
        n += 1
        positions.append(compute_update(A, b, positions[-1], k, step_size))
    print(n)
    return np.array(positions)




################################################################################
# TODO(student): Input Variables
A = np.array([[1, 0], [0, 1]])  # do not change this until the last part
b = np.array([4.5, 6])  # b in the equation ||Ax-b||
initial_position = np.array([0, 0])  # position at iteration 0
total_step_count = 15 # number of GD steps to take
step_size = lambda i: 1  # step size at iteration i
################################################################################

# computes desired number of steps of gradient descent
positions = compute_updates(A, b, initial_position, total_step_count, step_size)

# print out the values of the x_i
print(positions)
print(np.dot(np.linalg.inv(A), b))

# plot the values of the x_i
plt.scatter(positions[:, 0], positions[:, 1], c='blue')
plt.scatter(np.dot(np.linalg.inv(A), b)[0],
            np.dot(np.linalg.inv(A), b)[1], c='red')
plt.plot()
plt.show()
```

```
15
[[ 0.   0. ]
 [ 0.6  0.8]
 [ 1.2  1.6]
 [ 1.8  2.4]
 [ 2.4  3.2]
 [ 3.   4. ]
 [ 3.6  4.8]
 [ 4.2  5.6]
 [ 4.8  6.4]
 [ 4.2  5.6]
 [ 4.8  6.4]
 [ 4.2  5.6]
 [ 4.8  6.4]
 [ 4.2  5.6]]
[ 4.5  6. ]
```



ALthough the function is convex, the gradient descent method will not find the optimal solution.

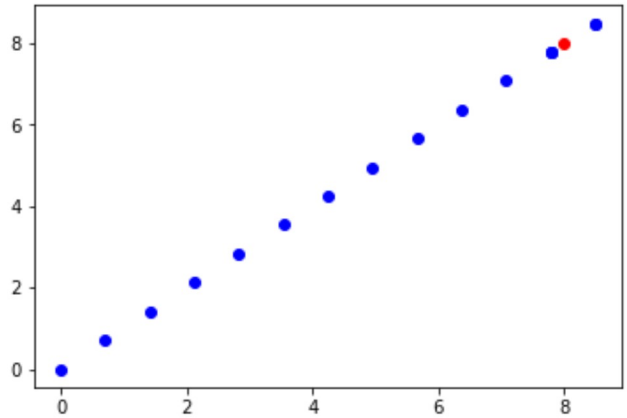
It will never get within 0.01 of the optimal solution. The gradient descent method clearly begins oscillating between [4.8 6.4] and [4.2 5.6]. The error in these step are [0.3, 0.4], which does not satisfy epsilon. Does not converge to the optimal solution.

The step size is constant and is too large.

For General b

In [54]:
```
    ##########################################################################
#
    # TODO(student): Input Variables
    A = np.array([[1, 0], [0, 1]])  # do not change this until the last part
    b = np.array([8, 8])  # b in the equation ||Ax-b||
    initial_position = np.array([0, 0])  # position at iteration 0
    total_step_count = 15  # number of GD steps to take
    step_size = lambda i: 1  # step size at iteration i
    ##########################################################################
#

    # computes desired number of steps of gradient descent
    positions = compute_updates(A, b, initial_position, total_step_count, step_size
)

    # print out the values of the x_i
    print(positions)
    print(np.dot(np.linalg.inv(A), b))

    # plot the values of the x_i
    plt.scatter(positions[:, 0], positions[:, 1], c='blue')
    plt.scatter(np.dot(np.linalg.inv(A), b)[0],
                np.dot(np.linalg.inv(A), b)[1], c='red')
    plt.plot()
    plt.show()
```

```
15
[[ 0.          0.        ]
 [ 0.70710678  0.70710678]
 [ 1.41421356  1.41421356]
 [ 2.12132034  2.12132034]
 [ 2.82842712  2.82842712]
 [ 3.53553391  3.53553391]
 [ 4.24264069  4.24264069]
 [ 4.94974747  4.94974747]
 [ 5.65685425  5.65685425]
 [ 6.36396103  6.36396103]
 [ 7.07106781  7.07106781]
 [ 7.77817459  7.77817459]
 [ 8.48528137  8.48528137]
 [ 7.77817459  7.77817459]
 [ 8.48528137  8.48528137]
 [ 7.77817459  7.77817459]]
[ 8.  8.]
```



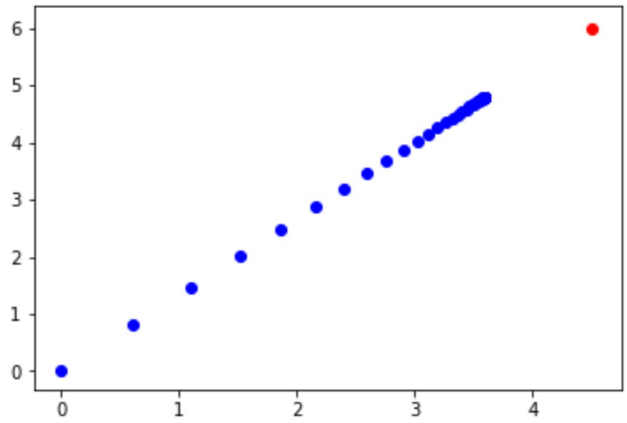
The same can be said for general b. Gradient descent will not always reach the optimal solution.

The step size is constant and is too large. Therefore it will not always converge.

Question 2 C

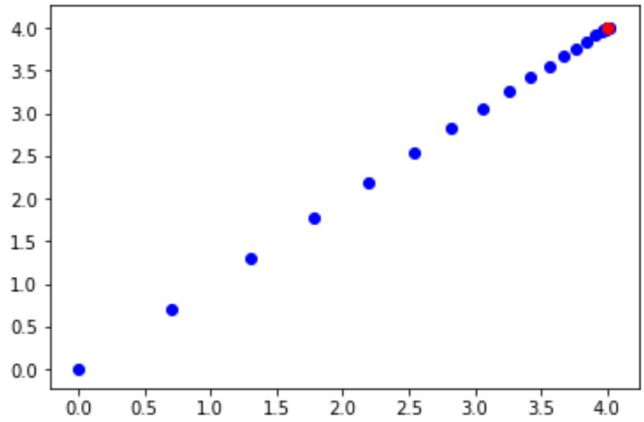
In [60]:
```
################################################################################
# TODO(student): Input Variables
A = np.array([[1, 0], [0, 1]])  # do not change this until the last part
b = np.array([4.5, 6])  # b in the equation ||Ax-b||
initial_position = np.array([0, 0])  # position at iteration 0
total_step_count = 5000  # number of GD steps to take
step_size = lambda i: (5/6)**i # step size at iteration i
################################################################################

# computes desired number of steps of gradient descent
positions = compute_updates(A, b, initial_position, total_step_count, step_size)

# print out the values of the x_i
print(positions)
print(np.dot(np.linalg.inv(A), b))

# plot the values of the x_i
plt.scatter(positions[:, 0], positions[:, 1], c='blue')
plt.scatter(np.dot(np.linalg.inv(A), b)[0],
            np.dot(np.linalg.inv(A), b)[1], c='red')
plt.plot()
plt.show()


# computes desired number of steps of gradient descent
b = [3, 3]
positions = compute_updates(A, b, initial_position, total_step_count, step_size)

# print out the values of the x_i
print(positions)
print(np.dot(np.linalg.inv(A), b))

# plot the values of the x_i
plt.scatter(positions[:, 0], positions[:, 1], c='blue')
plt.scatter(np.dot(np.linalg.inv(A), b)[0],
            np.dot(np.linalg.inv(A), b)[1], c='red')
plt.plot()
plt.show()
```

```
5000
[[ 0.          0.        ]
 [ 0.6         0.8       ]
 [ 1.1         1.46666667]
 ...,
 [ 3.6         4.8       ]
 [ 3.6         4.8       ]
 [ 3.6         4.8       ]]
[ 4.5  6. ]
```



```
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:10: RuntimeWarning:
invalid value encountered in true_divide
  # Remove the CWD from sys.path while we load stuff.
```

```
5000
[[ 0.          0.        ]
 [ 0.70710678  0.70710678]
 [ 1.29636243  1.29636243]
 ...,
 [         nan         nan]
 [         nan         nan]
 [         nan         nan]]
[ 4.  4.]
```



No the gradient descent will not find the optimal solution. This is because our step size gets exponentially smaller as we iterate. It does not converge. The plot does a good job illustrating this point.

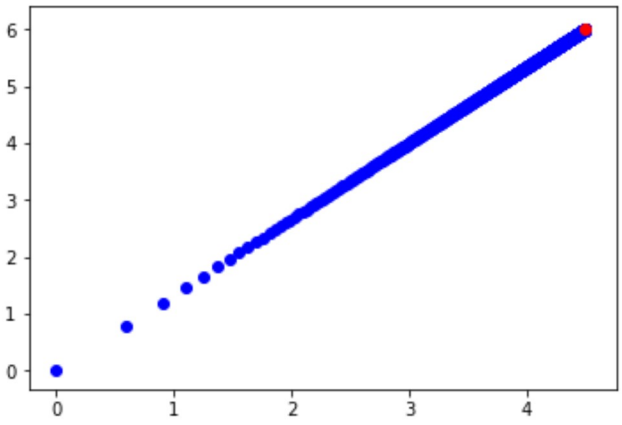
For General b

Conceptually, if our optimal solution x* is close enough to our intial x than our gradient descent method will converge to the optimal solution.

Question 2 D

In [57]:
```
################################################################################
# TODO(student): Input Variables
A = np.array([[1, 0], [0, 1]])  # do not change this until the last part
b = np.array([4.5, 6])  # b in the equation ||Ax-b||
initial_position = np.array([0, 0])  # position at iteration 0
total_step_count = 5000  # number of GD steps to take
step_size = lambda i: 1/(i + 1) # step size at iteration i
################################################################################

# computes desired number of steps of gradient descent
positions = compute_updates(A, b, initial_position, total_step_count, step_size)

# print out the values of the x_i
print(positions)
print(np.dot(np.linalg.inv(A), b))

# plot the values of the x_i
plt.scatter(positions[:, 0], positions[:, 1], c='blue')
plt.scatter(np.dot(np.linalg.inv(A), b)[0],
            np.dot(np.linalg.inv(A), b)[1], c='red')
plt.plot()
plt.show()
```

```
5000
[[ 0.          0.        ]
 [ 0.6         0.8       ]
 [ 0.9         1.2       ]
 ...,
 [ 4.50012003  6.00016004]
 [ 4.50000001  6.00000001]
 [ 4.49988001  5.99984001]]
[ 4.5  6. ]
```



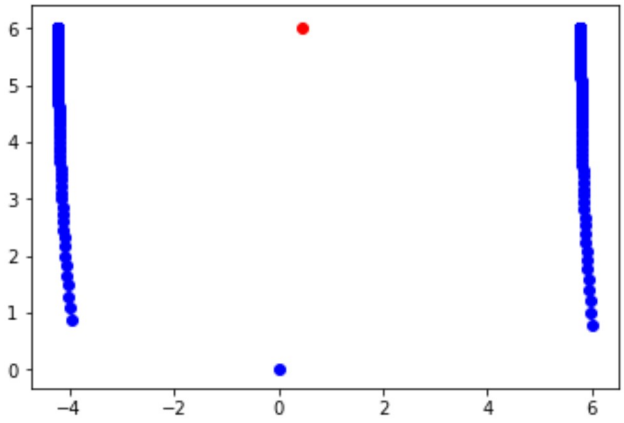
Our gradient descent method converges to the optimal solution. The expression for the number of iterations required for convergence is in the hand written part of this HW submission.

This step size makes our gradient descent method work for general b.

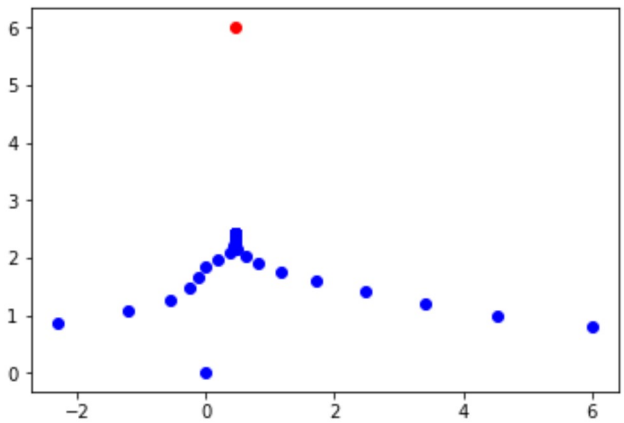
Question 2 E

In [71]:
```
################################################################################
# TODO(student): Input Variables
A = np.array([[10, 0], [0, 1]])  # do not change this until the last part
b = np.array([4.5, 6])  # b in the equation ||Ax-b||
initial_position = np.array([0, 0])  # position at iteration 0
total_step_count = 5000  # number of GD steps to take
steps = [lambda i : 1, lambda i: (5/6)**i, lambda i: 1/(i + 1)] # step size at iter
ation i
################################################################################

# plot the values of the x_i
for i in range(3):
    print(i)
    step_size = steps[i]
    # computes desired number of steps of gradient descent
    positions = compute_updates(A, b, initial_position, total_step_count, step_size
)
    # print out the values of the x_i
    print(positions)
    print(np.dot(np.linalg.inv(A), b))
    plt.scatter(positions[:, 0], positions[:, 1], c='blue')
    plt.scatter(np.dot(np.linalg.inv(A), b)[0],
                np.dot(np.linalg.inv(A), b)[1], c='red')
    print(A, "Step Size Method: " + str(i))
    plt.plot()
    plt.show()




################################################################################
# TODO(student): Input Variables
A = np.array([[15, 8], [6, 5]])  # do not change this until the last part
################################################################################

# plot the values of the x_i
for i in range(3):
    print(i)
    step_size = steps[i]
    # computes desired number of steps of gradient descent
    positions = compute_updates(A, b, initial_position, total_step_count, step_size
)
    # print out the values of the x_i
    print(positions)
    print(np.dot(np.linalg.inv(A), b))
    plt.scatter(positions[:, 0], positions[:, 1], c='blue')
    plt.scatter(np.dot(np.linalg.inv(A), b)[0],
                np.dot(np.linalg.inv(A), b)[1], c='red')
    print(A, "Step Size Method: " + str(i))
    plt.plot()
    plt.show()
```

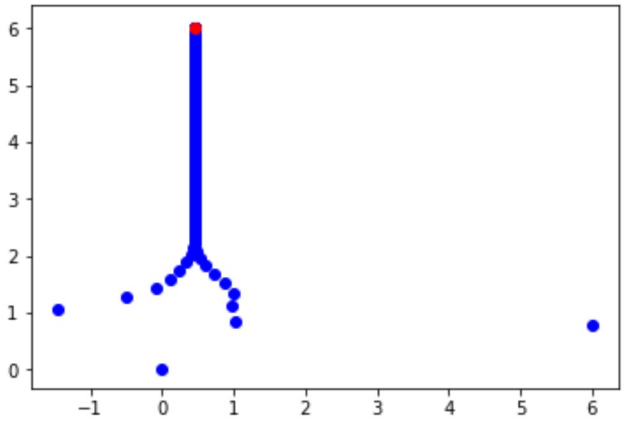
```
0
5000
[[ 0.          0.        ]
 [ 6.          0.8       ]
 [-3.95639434  0.89328514]
 ...,
 [-4.21969358  6.        ]
 [ 5.78030642  6.        ]
 [-4.21969358  6.        ]]
[ 0.45  6.  ]
[[10  0]
 [ 0  1]] Step Size Method: 0
```



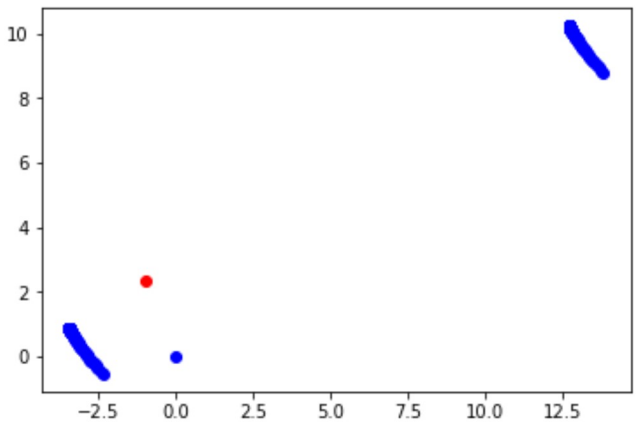

```
1
5000
[[ 0.          0.        ]
 [ 6.          0.8       ]
 [-2.29699529  0.87773761]
 ...,
 [ 0.44999997  2.43095698]
 [ 0.44999997  2.43095698]
 [ 0.44999997  2.43095698]]
[ 0.45  6.  ]
[[10  0]
 [ 0  1]] Step Size Method: 1
```



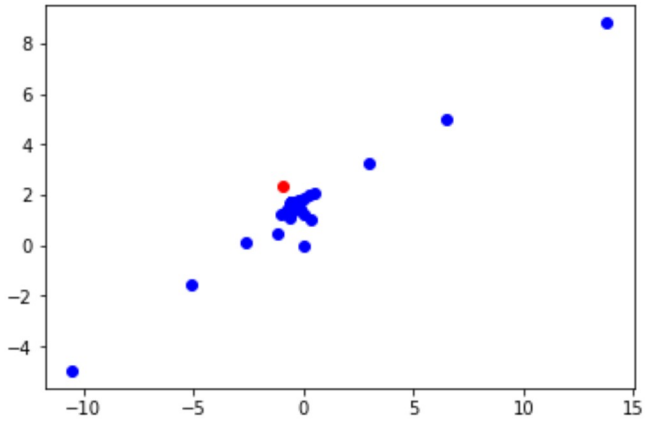
```
2
5000
[[ 0.          0.        ]
 [ 6.          0.8       ]
 [ 1.02180283  0.84664257]
 ...,
 [ 0.45        6.00020008]
 [ 0.45        6.00000004]
 [ 0.45        5.99980004]]
[ 0.45  6.  ]
[[10  0]
 [ 0  1]] Step Size Method: 2
```



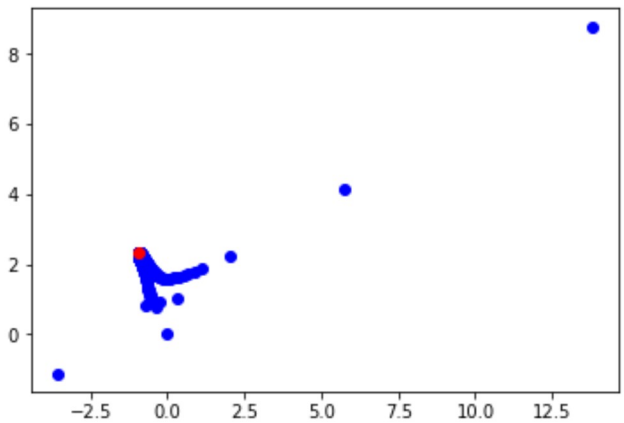

```
0
5000
[[  0.           0.        ]
 [ 13.8          8.8       ]
 [ -2.34489789  -0.53919835]
 ...,
 [ -3.45081002   0.88123538]
 [ 12.68837859  10.23170003]
 [ -3.45081002   0.88123538]]
[-0.94444444  2.33333333]
[[15  8]
 [ 6  5]] Step Size Method: 0
```



```
1
5000
[[  0.           0.        ]
 [ 13.8          8.8       ]
 [  0.34591842   1.01733471]
 ...,
 [ -0.58650742   1.715523  ]
 [ -0.58650742   1.715523  ]
 [ -0.58650742   1.715523  ]]
[-0.94444444  2.33333333]
[[15  8]
 [ 6  5]] Step Size Method: 1
```




```
2
5000
[[  0.           0.        ]
 [ 13.8          8.8       ]
 [  5.72755105   4.13040083]
 ...,
 [ -0.94283004   2.33426866]
 [ -0.94605852   2.33239819]
 [ -0.94283069   2.33426829]]
[-0.94444444  2.33333333]
[[15  8]
 [ 6  5]] Step Size Method: 2
```



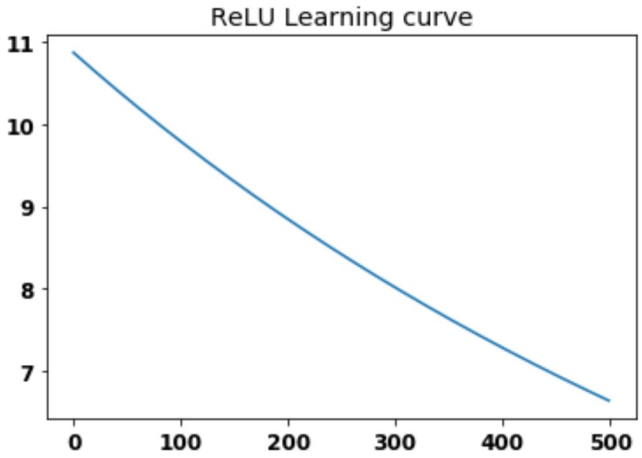
The step size function that we utilized for Q2.d should converge to the optimal solution regardless of the matrix A.

Changing A only changes what we are minimizing.

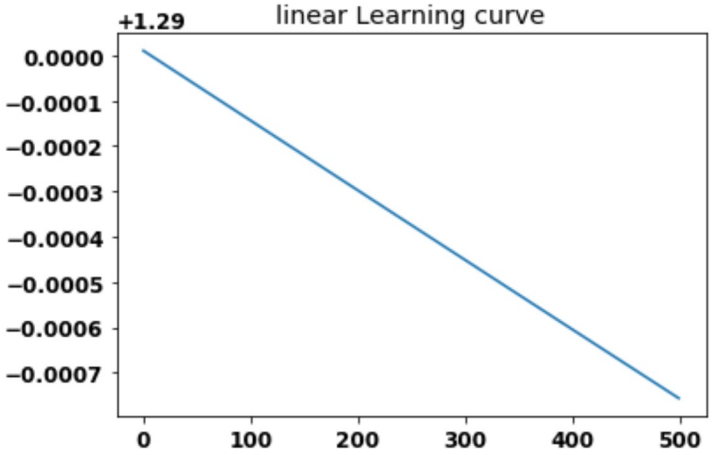
Question 5

In [98]:
```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt

# Gradient descent optimization
# The learning rate is specified by eta
class GDOptimizer(object):
    def __init__(self, eta):
        self.eta = eta

    def initialize(self, layers):
        pass

    # This function performs one gradient descent step
    # layers is a list of dense layers in the network
    # g is a list of gradients going into each layer before the nonlinear activatio
n
    # a is a list of of the activations of each node in the previous layer going
    #
    def update(self, layers, g, a):
        m = a[0].shape[1]
        for layer, curGrad, curA in zip(layers, g, a):
            # TODO: PART F ##########################################################
#################
            # Compute the gradients for layer.W and layer.b using the gradient for
the output of the
            # layer curA and the gradient of the output curGrad
            # Use the gradients to update the weight and the bias for the layer
            #
            # Normalize the learning rate by m (defined above), the number of train
ing examples input
            # (in parallel) to the network.
            #
            # It may help to think about how you would calculate the update if we i
nput just one
            # training example at a time; then compute a mean over these individual
update values.
            # ######################################################################
#################
            Wgrad_layer = curGrad * curA
            bgrad_layer = curGrad

            layer.W = layer.W - self.eta/m * np.mean(Wgrad_layer)
            layer.b = layer.b - self.eta/m * np.mean(bgrad_layer)

# Cost function used to compute prediction errors
class QuadraticCost(object):

    # Compute the squared error between the prediction yp and the observation y
    # This method should compute the cost per element such that the output is the
    # same shape as y and yp
    @staticmethod
    def fx(y,yp):
        # TODO: PART B ##########################################################
#############
        # Implement me
        return 0.5*np.square(y - yp)
        # ######################################################################
#############


    # Derivative of the cost function with respect to yp
    @staticmethod
    def dx(y,yp):
```

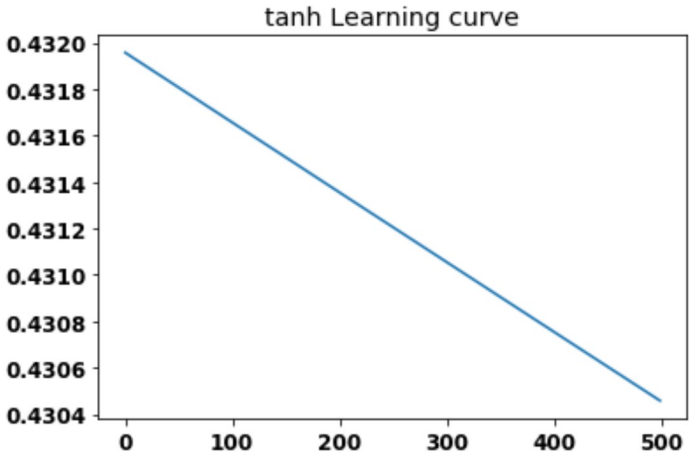
```
Debugging gradients..
squared difference of layer 0: 3.05989714594e-10
squared difference of layer 1: 1.88645317382e-10
squared difference of layer 2: 9.84497101272e-11

----------------------------------------

Standard fully connected network
ReLU MSE: 6.64656178094
```

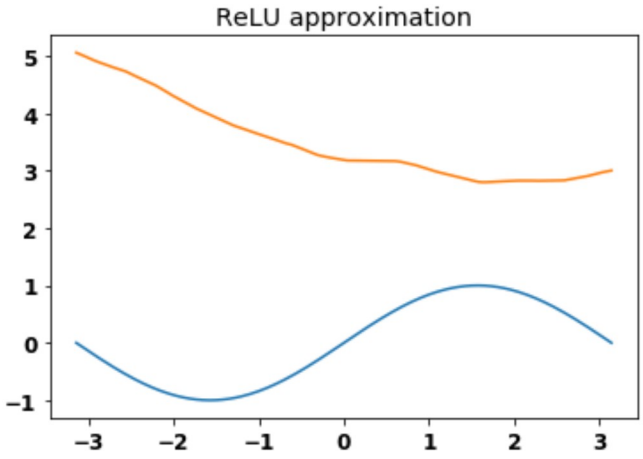


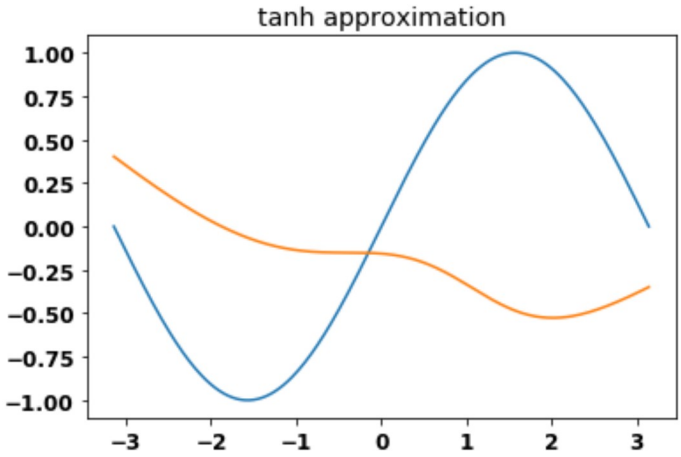

```
linear MSE: 1.28924330257
```

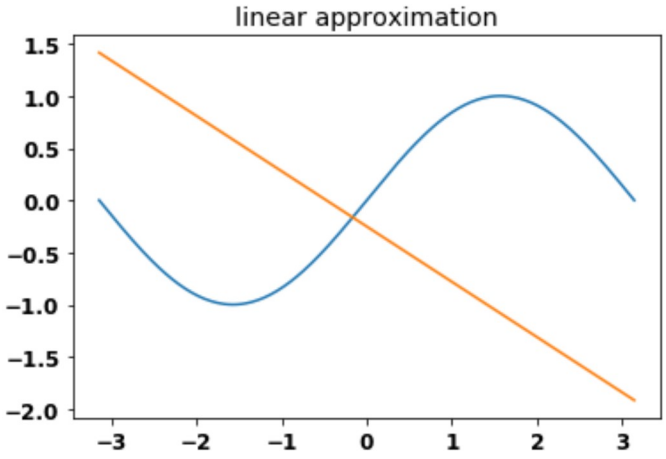



```
tanh MSE: 0.430458640433
```

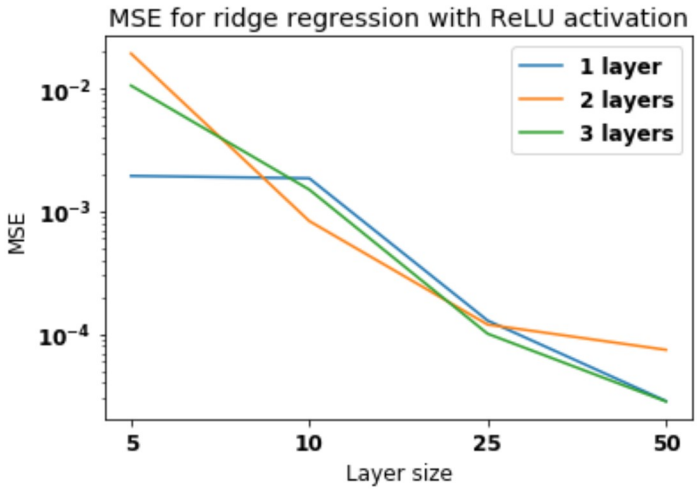


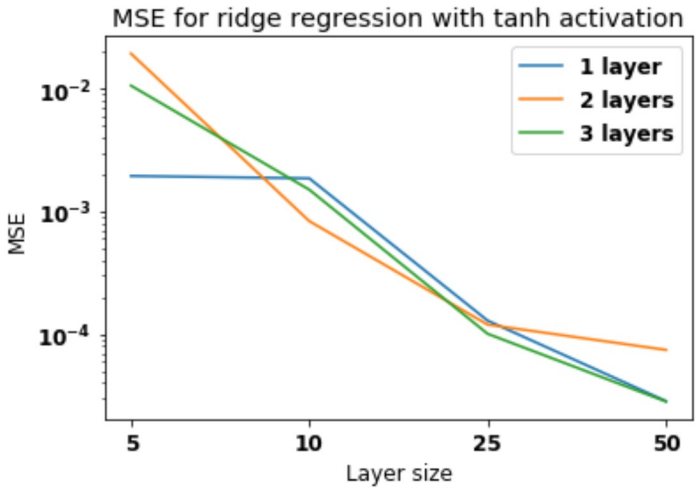

ReLU approximation
5
4
3
2
1
0
−1
−3
−2
−1
0
1
2
3


tanh approximation
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−3
−2
−1
0
1
2
3


linear approximation
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2
3

```
----------------------------------------

Training with various sized network
ReLU MSE Error
 Layers  5 nodes  10 nodes 25 nodes 50 nodes
       1 1.90620  1.64771  0.32309  0.13785
       2 0.28166  0.73836  2.74120  1.16850
       3 5.60639  6.09030  5.19953  2.26372
tanh MSE Error
 Layers  5 nodes  10 nodes 25 nodes 50 nodes
       1 0.68136  2.37873  4.06720  0.25491
       2 1.00403  0.10372  0.48874  3.52292
       3 0.27115  1.76387  0.15810  1.71457

----------------------------------------

Running ridge regression on last layer
ReLU MSE Error
 Layers  5 nodes  10 nodes 25 nodes 50 nodes
       1 0.00696  0.01074  0.00067  0.00005
       2 0.01916  0.00284  0.00018  0.00007
       3 0.03777  0.00106  0.00009  0.00009
tanh MSE Error
 Layers  5 nodes  10 nodes 25 nodes 50 nodes
       1 0.00195  0.00186  0.00013  0.00003
       2 0.01931  0.00083  0.00012  0.00007
       3 0.01058  0.00150  0.00010  0.00003
```





Question 5 Comment

Layer size is proportional to MSE. For both nonlinearities the MSE decreases with layer size. However there is not an appreciable difference as we increase the number of layers.

In [7]:
```
""" Tools for calculating Gradient Descent for ||Ax-b||. """
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
```

4. Sensors, Objects, and Localization

In [8]:
```
import numpy as np
import scipy.spatial
import matplotlib
import matplotlib.pyplot as plt
########################################################################
#########  Data Generating Functions ###################################
########################################################################
VAR_MEASUREMENT_NOISE = 1

def generate_sensors(k = 7, d = 2):
    """
    Generate sensor locations.
    Input:
    k: The number of sensors.
    d: The spatial dimension.
    Output:
    sensor_loc: k * d numpy array.
    """
    sensor_loc = 100*np.random.randn(k,d)
    return sensor_loc

def generate_data(sensor_loc, k = 7, d = 2,
                  n = 1, original_dist = True, sigma_s = 100):
    """
    Generate the locations of n points and distance measurements.

    Input:
    sensor_loc: k * d numpy array. Location of sensor.
    k: The number of sensors.
    d: The spatial dimension.
    n: The number of points.
    original_dist: Whether the data are generated from the original
    distribution.
    sigma_s: the standard deviation of the distribution
    that generate each object location.

    Output:
    obj_loc: n * d numpy array. The location of the n objects.
    distance: n * k numpy array. The distance between object and
    the k sensors.
    """
    assert k, d == sensor_loc.shape

    obj_loc = sigma_s*np.random.randn(n, d)
    if not original_dist:
        obj_loc = sigma_s*np.random.randn(n, d)+([300,300])

    distance = scipy.spatial.distance.cdist(obj_loc,
                                            sensor_loc,
                                            metric='euclidean')
    distance += np.random.randn(n, k)
    return obj_loc, distance

def generate_data_given_location(sensor_loc, obj_loc, k = 7, d = 2):
    """
    Generate the distance measurements given location of a single object and sensor
.

    Input:
    obj_loc: 1 * d numpy array. Location of object
    sensor_loc: k * d numpy array. Location of sensor.
    k: The number of sensors.
    d: The spatial dimension.
```

4. b)

In [9]:
```
##########################################################################
######### Part b ###################################
##########################################################################

##########################################################################
#########  Gradient Computing and MLE ###################################
##########################################################################
def compute_gradient_of_likelihood(single_obj_loc, sensor_loc,
                                   single_distance):
    """
    Compute the gradient of the loglikelihood function for part a.

    Input:
    single_obj_loc: 1 * d numpy array.
    Location of the single object.

    sensor_loc: k * d numpy array.
    Location of sensor.

    single_distance: k dimensional numpy array.
    Observed distance of the object.

    Output:
    grad: d-dimensional numpy array.

    """
#     grad = np.zeros_like(single_obj_loc)
#     #Your code: implement the gradient of loglikelihood
#     d = len(single_obj_loc[0])
#     k = len(single_distance)

#     for i in range(d):
#         #grad[0][i] = np.sum([2*(1 - single_distance[j]*(1/np.linalg.norm(sensor_
loc[j]-single_obj_loc[0])))*(sensor_loc[j][i] - single_obj_loc[0][i]) for j in rang
e(k)])
#         for j in range(k):
#             norm = np.linalg.norm(sensor_loc[j]-single_obj_loc[0]) #np.sqrt(np.sq
uare(sensor_loc[j]-single_obj_loc[0]))
#             scalar = 2*(sensor_loc[j][i] - single_obj_loc[0][i])

#             grad[0][i] += scalar*(norm - single_distance[j])/norm
    grad = np.zeros_like(single_obj_loc)


    for i in range(len(single_distance)):
        actual_dist = np.sqrt(np.sum((sensor_loc[i] - single_obj_loc) ** 2))
        observed_dist = single_distance[i]
        grad_i = single_obj_loc - sensor_loc[i]
        grad_i *= ((actual_dist - observed_dist) / actual_dist)
        grad += grad_i

    grad *= -1

    return grad

def find_mle_by_grad_descent_part_b(initial_obj_loc,
            sensor_loc, single_distance, lr=0.001, num_iters = 10000):
    """
    Compute the gradient of the loglikelihood function for part a.

    Input:
    initial_obj_loc: 1 * d numpy array.
    Initialized Location of the single object.
```

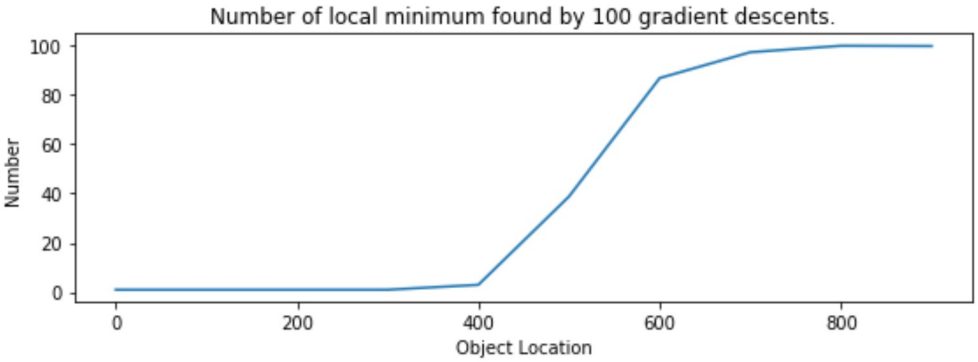
In [10]:
```
##########################################################################
#########  MAIN ##########################################################
##########################################################################

# Your code: set some appropriate learning rate here
def run(lr):
    print("Learning Rate is:", lr)
    np.random.seed(0)
    sensor_loc = generate_sensors()
    obj_loc, distance = generate_data(sensor_loc)
    single_distance = distance[0]
    print('The real object location is')
    print(obj_loc)
    # Initialized as [0,0]
    initial_obj_loc = np.array([[0.,0.]])
    estimated_obj_loc = find_mle_by_grad_descent_part_b(initial_obj_loc,
               sensor_loc, single_distance, lr=lr, num_iters = 10000)
    print('The estimated object location with zero initialization is')
    print(estimated_obj_loc)

    # Random initialization.
    initial_obj_loc = np.random.randn(1,2)
    estimated_obj_loc = find_mle_by_grad_descent_part_b(initial_obj_loc,
               sensor_loc, single_distance, lr=lr, num_iters = 10000)
    print('The estimated object location with random initialization is')
    print(estimated_obj_loc)
    print('\n\n')

lrs = [1.0, 0.01, 0.001, 0.0001]
for lr in lrs:
    run(lr)
```

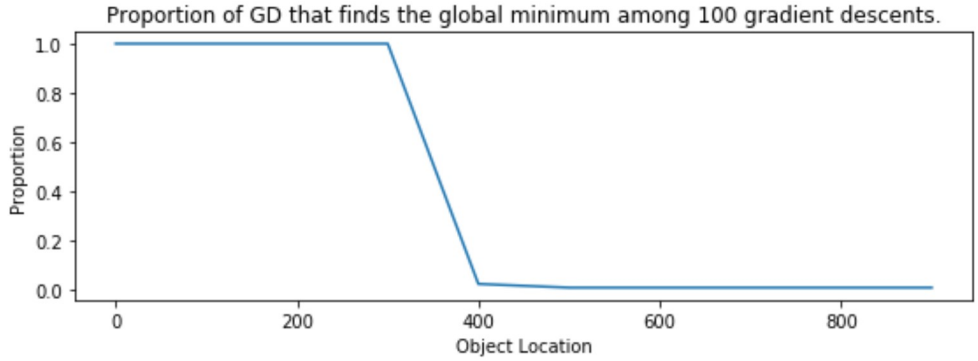
```
Learning Rate is: 1.0
The real object location is
[[ 44.38632327  33.36743274]]

/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:43: RuntimeWarning:
overflow encountered in square
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:46: RuntimeWarning:
invalid value encountered in double_scalars

The estimated object location with zero initialization is
[[ nan  nan]]
The estimated object location with random initialization is
[[ nan  nan]]



Learning Rate is: 0.01
The real object location is
[[ 44.38632327  33.36743274]]
The estimated object location with zero initialization is
[[ 43.07188433  32.71217817]]
The estimated object location with random initialization is
[[ 43.07188433  32.71217817]]



Learning Rate is: 0.001
The real object location is
[[ 44.38632327  33.36743274]]
The estimated object location with zero initialization is
[[ 43.07188426  32.71217807]]
The estimated object location with random initialization is
[[ 43.07188426  32.71217807]]



Learning Rate is: 0.0001
The real object location is
[[ 44.38632327  33.36743274]]
The estimated object location with zero initialization is
[[ 39.61604031  27.77608078]]
The estimated object location with random initialization is
[[ 39.67888838  27.90249244]]
```

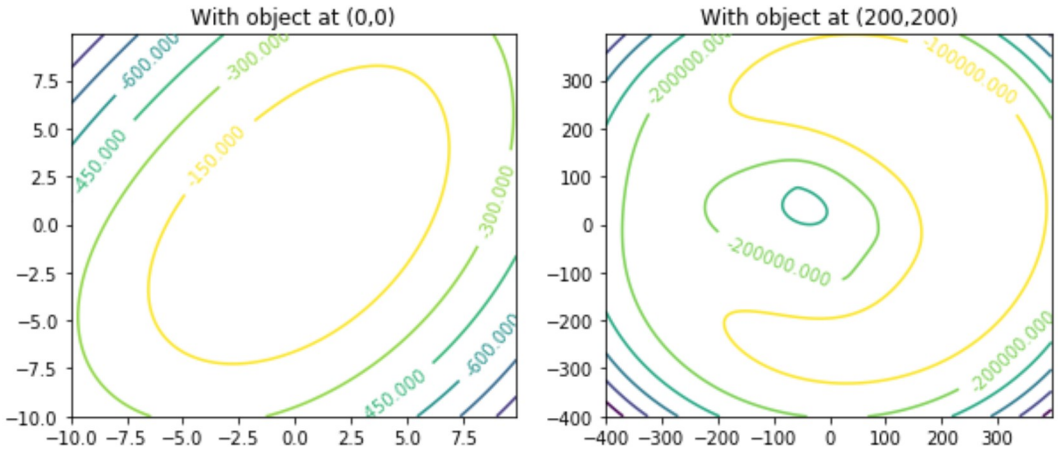
4. c)

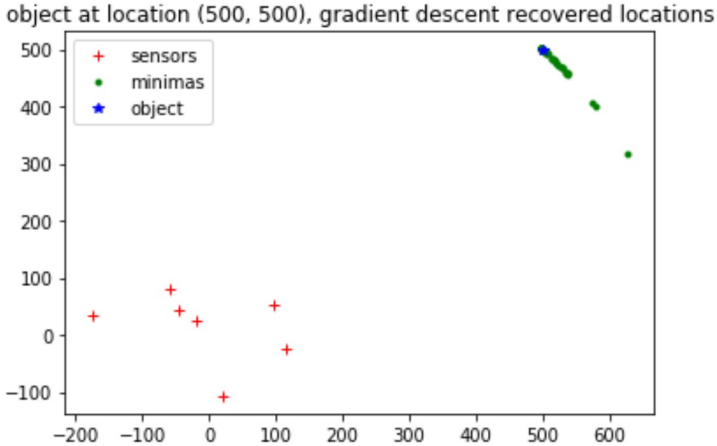
In [11]:
```
########################################################################
######### Part c ###################################################
########################################################################
def log_likelihood(obj_loc, sensor_loc, distance):
    """
    This function computes the log likelihood (as expressed in Part a).
    Input:
    obj_loc: shape [1,2]
    sensor_loc: shape [7,2]
    distance: shape [7]
    Output:
    The log likelihood function value.
    """
    # Your code: compute the log likelihood
#     func_value = -1*np.sum(
#         [np.square(np.linalg.norm(sensor_loc[i] - obj_loc[0]) - distance[i])
#          for i in range(len(distance))])

#     return func_value
    func_value = 0.0

    for i in range(len(distance)):
        actual_dist = np.sqrt(np.sum((sensor_loc[i] - obj_loc) ** 2))
        observed_dist = distance[i]
        func_value += (actual_dist - observed_dist) ** 2

    func_value *= -1

    return func_value
```

In [12]:
```
######################################################################
######## Compute the function value at local minimum for all experiments.###
######################################################################
def run4(num_sensors):
    #num_sensors = 7

    np.random.seed(100)
    sensor_loc = generate_sensors(k=num_sensors)

    # num_data_replicates = 10
    num_gd_replicates = 100

    obj_locs = [[[i,i]] for i in np.arange(0,1000,100)]

    func_values = np.zeros((len(obj_locs),10, num_gd_replicates))
    # record sensor_loc, obj_loc, 100 found minimas
    minimas = np.zeros((len(obj_locs), 10, num_gd_replicates, 2))
    true_object_locs = np.zeros((len(obj_locs), 10, 2))

    for i, obj_loc in enumerate(obj_locs):
        for j in range(10):
            obj_loc, distance = generate_data_given_location(sensor_loc, obj_loc,
                                                             k = num_sensors, d = 2)
            true_object_locs[i, j, :] = np.array(obj_loc)

            for gd_replicate in range(num_gd_replicates):
                initial_obj_loc = np.random.randn(1,2)* (100 * i+1)
                obj_loc = find_mle_by_grad_descent_part_b(initial_obj_loc,
                         sensor_loc, distance[0], lr=0.1, num_iters = 1000)
                minimas[i, j, gd_replicate, :] = np.array(obj_loc)
                func_value = log_likelihood(obj_loc, sensor_loc, distance[0])
                func_values[i, j, gd_replicate] = func_value

    ######################################################################
    ######## Calculate the things to be plotted. ###
    ######################################################################
    local_mins = [[np.unique(func_values[i,j].round(decimals=2)) for j in range(10)
] for i in range(10)]
    num_local_min = [[len(local_mins[i][j]) for j in range(10)] for i in range(10)]
    proportion_global = [[sum(func_values[i,j].round(decimals=2) == min(local_mins[
i][j]))*1.0/100 \
                          for j in range(10)] for i in range(10)]


    num_local_min = np.array(num_local_min)
    num_local_min = np.mean(num_local_min, axis = 1)

    proportion_global = np.array(proportion_global)
    proportion_global = np.mean(proportion_global, axis = 1)

    ######################################################################
    ######## Plots. ######################################################
    ######################################################################
    fig, axes = plt.subplots(figsize=(8,6), nrows=2, ncols=1)
    fig.tight_layout()
    plt.subplot(211)

    plt.plot(np.arange(0,1000,100), num_local_min)
    plt.title('Number of local minimum found by 100 gradient descents.')
    plt.xlabel('Object Location')
    plt.ylabel('Number')
    #plt.savefig('num_obj.png')
    # Proportion of gradient descents that find the local minimum of minimum value.
```


Number of local minimum found by 100 gradient descents.
Number
Object Location

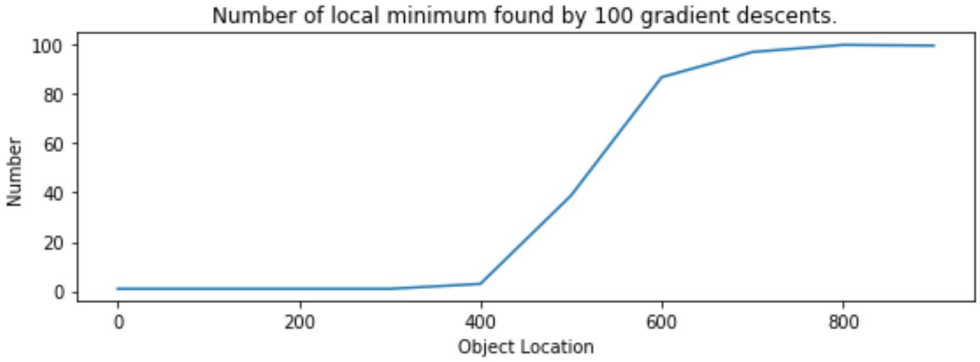

Proportion of GD that finds the global minimum among 100 gradient descents.
Proportion
Object Location

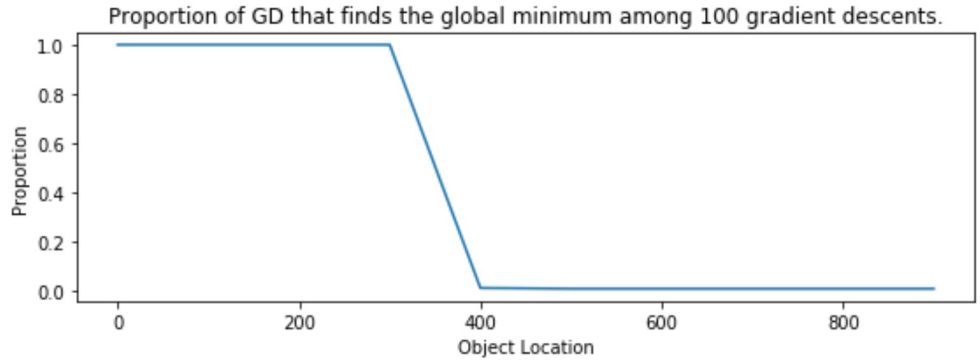

With object at (0,0)
With object at (200,200)

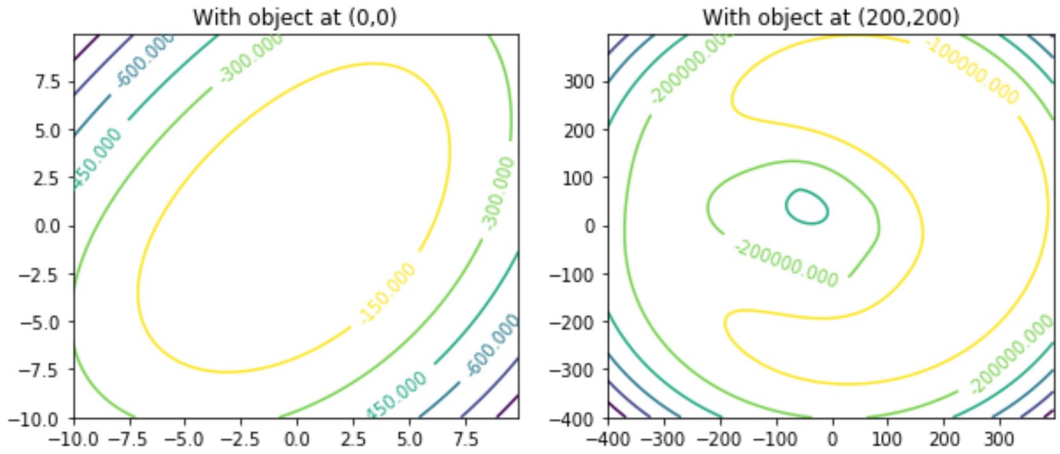

object at location (500, 500), gradient descent recovered locations
sensors
minimas
object

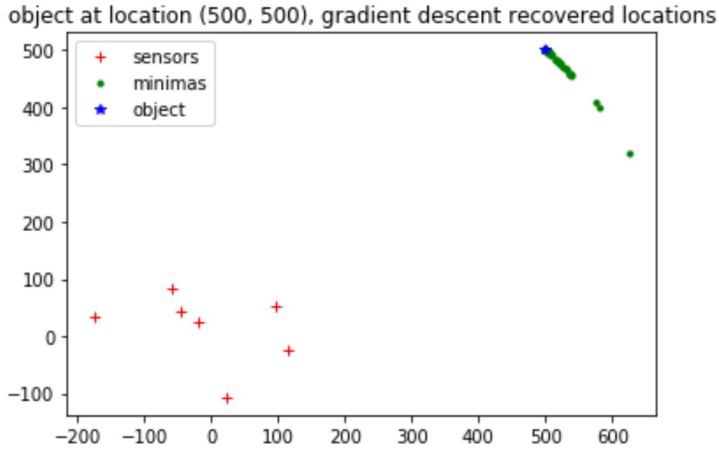
4. d)

In [14]:
```
VAR_MEASUREMENT_NOISE = 0.01 # Used by generate_data_given_location function
run4(num_sensors=7)
```









4. e)

In [15]:
```
VAR_MEASUREMENT_NOISE = 1 # Used by generate_data_given_location function
run4(num_sensors=20)
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-15-e08484daed3e> in <module>()
      1 VAR_MEASUREMENT_NOISE = 1 # Used by generate_data_given_location functio
n
----> 2 run4(num_sensors=20)

<ipython-input-12-8d0fd7e8f169> in run4(num_sensors)
     27                 initial_obj_loc = np.random.randn(1,2)* (100 * i+1)
     28                 obj_loc = find_mle_by_grad_descent_part_b(initial_obj_lo
c,
---> 29                         sensor_loc, distance[0], lr=0.1, num_iters = 10
00)
     30                 minimas[i, j, gd_replicate, :] = np.array(obj_loc)
     31                 func_value = log_likelihood(obj_loc, sensor_loc, distanc
e[0])

<ipython-input-9-a5a12d7e7dcc> in find_mle_by_grad_descent_part_b(initial_obj_lo
c, sensor_loc, single_distance, lr, num_iters)
     72     # Your code: do gradient descent
     73     for i in range(num_iters):
---> 74         grad = compute_gradient_of_likelihood(obj_loc, sensor_loc, singl
e_distance)
     75         obj_loc = obj_loc + grad*lr
     76

<ipython-input-9-a5a12d7e7dcc> in compute_gradient_of_likelihood(single_obj_loc,
sensor_loc, single_distance)
     41
     42     for i in range(len(single_distance)):
---> 43         actual_dist = np.sqrt(np.sum((sensor_loc[i] - single_obj_loc) **
2))
     44         observed_dist = single_distance[i]
     45         grad_i = single_obj_loc - sensor_loc[i]

/opt/conda/lib/python3.6/site-packages/numpy/core/fromnumeric.py in sum(a, axis,
dtype, out, keepdims)
   1812             return sum(axis=axis, dtype=dtype, out=out, **kwargs)
   1813     return _methods._sum(a, axis=axis, dtype=dtype,
-> 1814                          out=out, **kwargs)
   1815
   1816

KeyboardInterrupt:
```

4. f)

In [16]:
```
#########################################################################
########  Gradient Computing and MLE ####################################
#########################################################################
def compute_grad_likelihood(sensor_loc, obj_loc, distance):
    """
    Compute the gradient of the loglikelihood function for part f.

    Input:
    sensor_loc: k * d numpy array.
    Location of sensors.

    obj_loc: n * d numpy array.
    Location of the objects.

    distance: n * k dimensional numpy array.
    Observed distance of the object.

    Output:
    grad: k * d numpy array.
    """
    grad = np.zeros(sensor_loc.shape)
    # Your code: finish the grad loglike
    for i, sens in enumerate(sensor_loc):
        grad[i] = compute_gradient_of_likelihood(sens.reshape((1,-1)), obj_loc, dis
tance[:, i])

    return grad

def find_mle_by_grad_descent(initial_sensor_loc,
           obj_loc, distance, lr=0.001, num_iters = 1000):
    """
    Compute the gradient of the loglikelihood function for part f.

    Input:
    initial_sensor_loc: k * d numpy array.
    Initialized Location of the sensors.

    obj_loc: n * d numpy array. Location of the n objects.

    distance: n * k dimensional numpy array.
    Observed distance of the n object.

    Output:
    sensor_loc: k * d numpy array. The mle for the location of the object.

    """
    sensor_loc = initial_sensor_loc
    # Your code: finish the gradient descent
    for i in range(num_iters):
        grad = compute_grad_likelihood(sensor_loc, obj_loc, distance)
        sensor_loc += lr*grad

    return sensor_loc
#########################################################################
########  Gradient Computing and MLE ####################################
#########################################################################

np.random.seed(0)
sensor_loc = generate_sensors()
obj_loc, distance = generate_data(sensor_loc, n = 100)
print('The real sensor locations are')
print(sensor_loc)
# Initialized as zeros.
initial_sensor_loc = np.zeros((7,2)) #np.random.randn(7,2)
```

```
The real sensor locations are
[[ 176.4052346    40.01572084]
 [  97.87379841  224.08931992]
 [ 186.75579901  -97.72778799]
 [  95.00884175  -15.13572083]
 [ -10.32188518   41.05985019]
 [  14.40435712  145.4273507 ]
 [  76.10377251   12.16750165]]
The predicted sensor locations are
[[ 176.28223286   39.76550241]
 [  97.73185804  224.12127837]
 [ 186.68461116  -97.5920796 ]
 [  94.90639305  -15.16741152]
 [ -10.17511568   40.93691838]
 [  14.19497946  145.42410633]
 [  76.11863016   12.24807699]]
The MSE for Case 1 is 7.863128736936463
The MSE for Case 2 is 7.227391044803935
The MSE for Case 2 (if we knew mu is [300,300]) is 6.914124924780527
```

Comments for Question 4

4. B

Learning rate of 1 is unreasonable. The best learning rate was 0.01 for both random and zero initialization.

4. C

Do you find any patterns?

As our optimal point is moved farther away from the sensors our gradient descent to get to the optimal position becomes worse. The insight we get from slight differences between our sensors which we use in GD to determine optimal direction become useless.

4. D

Do you find any patterns?

“Reduce variance of the measurement noise”

The difference did not amount to much. This is due to the fact that the variance which we have across all of our measurement noise only scales our gradient descent. It is worth noting that our algorithm may have to take smaller steps to guarantee converge due to this scaling potential.

4. E

Do you find any patterns?

More sensors means that it is easier to locate our object. Having a greater number of sensors means that our gradient descent is better able to distinguish an optimal direction to find the global minimum.