Question 1. Getting Started

You may typeset your homework in latex or submit neatly handwritten and scanned solutions. Please make sure to start each question on a new page, as grading (with Gradescope) is much easier that way! Deliverables:

1. Submit a PDF of your writeup to assignment on Gradescope, "HW[n] Write-Up"
2. Submit all code needed to reproduce your results, "HW[n] Code".
3. Submit your test set evaluation results, "HW[n] Test Set".

After you've submitted your homework, be sure to watch out for the self-grade form.

(a) 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 with Weiran Liu and Katherine Li. I don't like the length of this homework at all. It was released late and longer than the previous one.

(b) Please copy the following statement and sign next to it. We just want to make it *extra* clear so that no one inadvertently cheats.

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.

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.

Signature: [illegible]

Question 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 $f(x) = \|x - b\|_2$. 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 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 way 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.

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

We calculate the derivative at any $\vec{x} \neq \vec{b}$

$$\begin{aligned}
\nabla_x^2 f(x) &= \nabla_x^2 \|\vec{x} - \vec{b}\|_2 \\
&= \nabla_x^2 \|\vec{x} - \vec{b}\|_2 \\
&= \nabla_x^2 \sqrt{\|\vec{x} - \vec{b}\|_2^2} \\
&= \nabla_x^2 \sqrt{(\vec{x} - \vec{b})^\top (\vec{x} - \vec{b})} \\
&= \nabla_x^2 \sqrt{\vec{x}^\top \vec{x} - 2\vec{b}^\top \vec{x} \vec{b}^\top \vec{b}} \\
&= \nabla_x \frac{1}{2} \frac{2\vec{x} - 2\vec{b}}{\sqrt{\vec{x}^\top \vec{x} - 2\vec{b}^\top \vec{x} + \vec{b}^\top \vec{b}}} \\
&= -\frac{1}{2} \times \frac{1}{2} \frac{2\sqrt{\vec{x}^\top \vec{x} - 2\vec{b}^\top \vec{x} + \vec{b}^\top \vec{b}} - (2\vec{x} - 2\vec{b})^\top \frac{2\vec{x} - 2\vec{b}}{\sqrt{\vec{x}^\top \vec{x} - 2\vec{b}^\top \vec{x} + \vec{b}^\top \vec{b}}}}{\vec{x}^\top \vec{x} - 2\vec{b}^\top \vec{x} + \vec{b}^\top \vec{b}} \\
&= -\frac{1}{4} \frac{2(\vec{x}^\top \vec{x} - 2\vec{b}^\top \vec{x} + \vec{b}^\top \vec{b}) - (2\vec{x} - 2\vec{b})^\top (2\vec{x} - 2\vec{b})}{(\vec{x}^\top \vec{x} - 2\vec{b}^\top \vec{x} + \vec{b}^\top \vec{b})^{3/2}} \\
&= -\frac{1}{4} \frac{2(\vec{x} - \vec{b})^\top (\vec{x} - \vec{b}) - 4(\vec{x} - \vec{b})^\top (\vec{x} - \vec{b})}{((\vec{x} - \vec{b})^\top (\vec{x} - \vec{b}))^{3/2}} \\
&= -\frac{1}{4} \frac{-2(\vec{x} - \vec{b})^\top (\vec{x} - \vec{b})}{((\vec{x} - \vec{b})^\top (\vec{x} - \vec{b}))^{3/2}} \\
&= \frac{1}{2} \frac{(\vec{x} - \vec{b})^\top (\vec{x} - \vec{b})}{((\vec{x} - \vec{b})^\top (\vec{x} - \vec{b}))^{3/2}} \\
&\geq 0
\end{aligned}$$

Now we need to consider the gradient at $\vec{x} = \vec{b}$. At this point, the gradient of different directions are different. Therefore, let's define it as $f(\vec{0}) = \vec{0}$ and we are good. Because the second-order gradient is non-negative, the function is a convex function. However, my friend reminds me that we should use the definition of convex function here:

$$\begin{aligned}
&\forall \vec{x}_1, \vec{x}_2, t \in [0,1] \\
&f(t\vec{x}_1 + (1-t)\vec{x}_2) \leq tf(\vec{x}_1) + (1-t)f(\vec{x}_2) \\
&\|t\vec{x}_1 + (1-t)\vec{x}_2\|_2 \leq t\|\vec{x}_1\|_2 + (1-t)\|\vec{x}_2\|_2
\end{aligned}$$

This inequality is guaranteed by triangle inequality of l2-norm.

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

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

We start at $\vec{x}_0 = [0, 0]$. **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 $\vec{b} \neq 0$?**

From the previous part we know that the global minimum can be found at $\arg\min_{\vec{x}} f(\vec{x}) = \vec{b}$.

$$\nabla_x f(x) = \frac{1}{2} \frac{2\vec{x} - 2\vec{b}}{\sqrt{\vec{x}^\top \vec{x} - 2\vec{b}^\top \vec{x} + \vec{b}^\top \vec{b}}}$$

Let's simulate this:

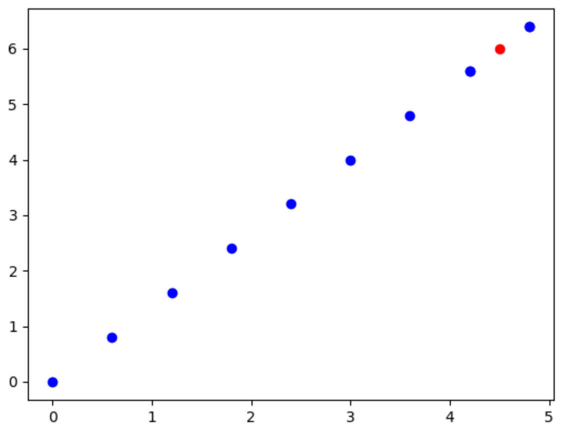
Step 1:

$$\nabla_x f(x) = \frac{1}{2} \frac{-2\begin{bmatrix} 4.5 \\ 6 \end{bmatrix}}{\sqrt{4.5 \times 4.5 + 6 \times 6}} = -\begin{bmatrix} 0.6 \\ 0.8 \end{bmatrix}$$

$$\vec{x}_1 = \vec{x}_0 + \begin{bmatrix} 0.6 \\ 0.8 \end{bmatrix} = \begin{bmatrix} 0.6 \\ 0.8 \end{bmatrix}$$

The following steps will be simulated in the given code. The result is:

```
number of step: 10
[[ 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]]
optimal point:[ 4.5 6. ]
```



As we can see from above, gradient descent will jump between [4.2, 5.6] and [4.8, 6.4] for ever and never get within 0.01 of the optimal solution. Therefore, gradient descent cannot find the optimal solution in this case

Let's prove this by setting $\vec{x}_i = [4.2, 5.6]^\top$:

$$\begin{aligned}\nabla_x f(\vec{x}_i) &= \frac{\vec{x}_i - \vec{b}}{\sqrt{\vec{x}_i^\top \vec{x}_i - 2\vec{b}^\top \vec{x}_i + \vec{b}^\top \vec{b}}} \\ &= \frac{\begin{bmatrix}4.2\\5.6\end{bmatrix} - \begin{bmatrix}4.5\\6\end{bmatrix}}{\sqrt{4.2^2 + 5.6^2 - 2\times(4.2\times 4.5 + 5.6\times 6) + 4.5^2 + 6^2}} \\ &= \begin{bmatrix}-0.6\\-0.8\end{bmatrix}\end{aligned}$$

$$\begin{aligned}\vec{x}_{i+1} &= \vec{x}_i - \nabla f(\vec{x}_i) \\ &= \begin{bmatrix}4.2\\5.6\end{bmatrix} - \begin{bmatrix}-0.6\\-0.8\end{bmatrix} = \begin{bmatrix}4.8\\6.4\end{bmatrix}\end{aligned}$$

Now we need to double check if $\vec{x}_{i+2} = [4.2, 5.6]^\top$

$$\begin{aligned}\nabla_x f(\vec{x}_{i+1}) &= \frac{\vec{x}_{i+1} - \vec{b}}{\sqrt{\vec{x}_{i+1}^\top \vec{x}_{i+1} - 2\vec{b}^\top \vec{x}_{i+1} + \vec{b}^\top \vec{b}}} \\ &= \frac{\begin{bmatrix}4.8\\6.4\end{bmatrix} - \begin{bmatrix}4.5\\6\end{bmatrix}}{\sqrt{4.8^2 + 6.4^2 - 2\times(4.8\times 4.5 + 6.4\times 6) + 4.5^2 + 6^2}} \\ &= \begin{bmatrix}0.6\\0.8\end{bmatrix}\end{aligned}$$

$$\begin{aligned}\vec{x}_{i+2} &= \vec{x}_{i+1} - \nabla f(\vec{x}_i) \\ &= \begin{bmatrix}4.8\\6.4\end{bmatrix} - \begin{bmatrix}0.6\\0.8\end{bmatrix} = \begin{bmatrix}4.2\\5.6\end{bmatrix}\end{aligned}$$

We don't know whether gradient descent will find the optimal solution or not for a general $\vec{b} \neq 0$. For example, gradient descent can find the optimal solution in one step for $\vec{b} = [1, 0]^\top$. To be more specific, gradient descent can find the optimal solution in one step for the following $\vec{b}$.

$$\vec{x}_0 - \frac{1}{2}\frac{2\vec{x}_0 - 2\vec{b}}{\sqrt{\vec{x}_0^\top \vec{x}_0 - 2\vec{b}^\top \vec{x}_0 + \vec{b}^\top \vec{b}}} = \vec{b}$$

$$\vec{x}_0 - \vec{b} = \frac{\vec{x}_0 - \vec{b}}{\sqrt{\vec{x}_0^\top \vec{x}_0 - 2\vec{b}^\top \vec{x}_0 + \vec{b}^\top \vec{b}}}$$

$$-\vec{b} = \frac{-\vec{b}}{\sqrt{\vec{b}^\top \vec{b}}}$$

$$\sqrt{\vec{b}^\top \vec{b}} = 1$$

$$\|b\|_2 = 1$$

```
""" Tools for calculating Gradient Descent for ||Ax-b||. """
import matplotlib.pyplot as plt
import numpy as np


def main():
################################################################################
# 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 = 10 # number of GD steps to take
# step_size = lambda i: 1 # step size at iteration i
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, b)

# print out the values of the x_i
print('number of step: ' + str(positions.shape[0] - 1))
print(positions)
print('optimal point:' + str(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()


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, optimal=[], error=0.01):
"""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)]
for k in range(total_step_count):
current_position = compute_update(A, b, positions[-1], k, step_size)
positions.append(current_position)
if (len(optimal) > 0) and (np.linalg.norm(current_position - optimal) <= error ** 2):
break
return np.array(positions)


main()
#print(compute_gradient(np.array([[1, 0], [0, 1]]), np.array([4.5, 6]), np.array([0, 0])))
```

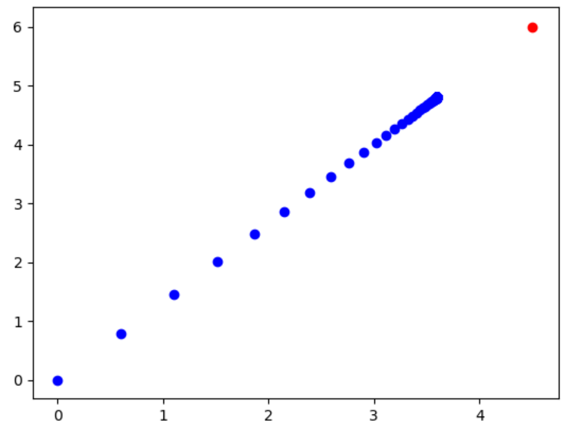
3. We are minimizing $f(\vec{x}) = \|\vec{x} - \vec{b}\|_2$, where $\vec{x} \in \mathbb{R}^2$ and $\vec{b} = [4.5, 6] \in \mathbb{R}^2$, now with a decreasing step size of $t_i = \left(\frac{5}{6}\right)^i$ at step i. That is,

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

We start at $\vec{x}_0 = [0, 0]$. **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 $\|\vec{x}_i\|_2$.) **What about general** $\vec{b} \neq \vec{0}$**?**

Again, we use the code to do a simulation and get:

```
number of step: 1000
[[ 0.  0. ]
[ 0.6 0.8 ]
[ 1.1 1.46666667]
...,
[ 3.6 4.8 ]
[ 3.6 4.8 ]
[ 3.6 4.8 ]]
optimal point:[ 4.5 6. ]
```



As we can see, the new adaptive gradient descent got stuck at $\vec{x} = [3.6, 4.8]^\top$. We take a look at the change of $\|\vec{x}_i\|_2$.

$$\|\vec{x}_{i+1}\|_2 = \|\vec{x}_i - \left(\frac{5}{6}\right)^i \nabla f(\vec{x}_i)\|_2$$
$$\leq \|\vec{x}_i\|_2 + \left(\frac{5}{6}\right)^i \|\nabla f(\vec{x}_i)\|_2$$

Let's compute the l2-norm of $\nabla f(\vec{x}_i)$:

$$\|\nabla f(\vec{x}_i)\|_2$$

$$= \left(\frac{\vec{x}^\top - \vec{b}^\top}{\sqrt{(\vec{x}-\vec{b})^\top(\vec{x}-\vec{b})}} \frac{\vec{x}-\vec{b}}{\sqrt{(\vec{x}-\vec{b})^\top(\vec{x}-\vec{b})}} \right)^{1/2}$$
$$=1$$

Substitute this into the previous triangle inequality:

$$\|\vec{x}_{i+1}\|_2 \leq \|\vec{x}_i\|_2 + \left(\frac{5}{6}\right)^i$$

Therefore, we just need to prove the following infinite series converge to a number that is smaller than $\|\vec{b}\|_2$.

$$S = 1 + \frac{5}{6} + \left(\frac{5}{6}\right)^2 + \cdots + \left(\frac{5}{6}\right)^i = 1 \times \frac{1-\left(\frac{5}{6}\right)^i}{1-\frac{5}{6}}$$

$$\lim_{i\to\infty} S = \frac{1}{1-\frac{5}{6}} = 6$$

Therefore, we know $\lim_{i\to\infty} \|\vec{x}_i\|_2 \leq 6$. However, we know $\|\vec{b}\|_2 = 7.5$. We have proved that:

$$\lim_{i\to\infty} \|\vec{x}_i\|_2 < \|\vec{b}\|_2$$

As a conclusion, the new adaptive gradient descent cannot reach the optimal solution with an error of 0.01.

For a general $\vec{b}$, this might not be true. If the norm of $\|\vec{b}\| < 5$, then it is possible for this method to work. For example, this approach can find the optimal solution if $\vec{b} = [1, 0]^\top$

```
""" Tools for calculating Gradient Descent for ||Ax-b||. """
import matplotlib.pyplot as plt
import numpy as np


def main():
################################################################################
# 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 = 1000 # number of GD steps to take
# step_size = lambda i: 1 # step size at iteration i
step_size = lambda i: np.power(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, b)

# print out the values of the x_i
print('number of step: ' + str(positions.shape[0] - 1))
print(positions)
print('optimal point:' + str(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()


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, optimal=[], error=0.01):
"""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)]
for k in range(total_step_count):
current_position = compute_update(A, b, positions[-1], k, step_size)
positions.append(current_position)
if (len(optimal) > 0) and (np.linalg.norm(current_position - optimal) <= error ** 2):
break
return np.array(positions)


main()
#print(compute_gradient(np.array([[1, 0], [0, 1]]), np.array([4.5, 6]), np.array([0, 0])))
```

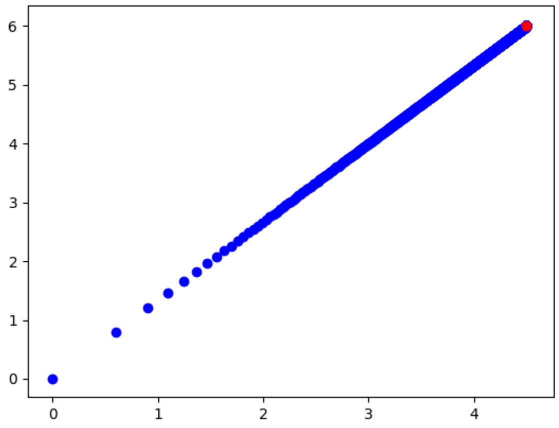
4. We are minimizing $f(x) = \|\vec{x} - \vec{b}\|_2$, where $\vec{x} \in \mathbb{R}^2$ and $\vec{b} = [4.5, 6] \in \mathbb{R}^2$, now with a decreasing step size of $t_i = \frac{1}{i+1}$ at step i. That is,

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

We start at $\vec{x}_0 = [0, 0]$. **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 $\|\vec{x}_i\|_2$, and use $\sum_{i=1}^{n} \frac{1}{i}$ is of the order $\log n$.) **What about general** $b \neq \vec{0}$**?**

Yes. Gradient descent in this case can find the optimal solution. It takes 2081 steps to do so. (See the answer below):

```
number of step: 1005
[[ 0.  0. ]
[ 0.6 0.8 ]
[ 0.9 1.2 ]
...,
[ 4.49307892 5.9907719 ]
[ 4.49367653 5.99156871]
[ 4.49427355 5.99236473]]
optimal point:[ 4.5 6. ]
```



First, given the knowledge of the function (convexity), we know that if we start from $x_0 = \vec{0}$, we will move along the line of $\vec{b}$. We take a look at the change of $\|\vec{x}_{i+1}\|_2$.

$$\begin{aligned} \|\vec{x}_{i+1}\|_2 &= \|\vec{x}_i - \frac{1}{i+1} \|\nabla f(\vec{x}_i)\|_2 \\ &\leq \|\vec{x}_i\|_2 + \frac{1}{i+1} \|\nabla f(\vec{x}_i)\|_2 \end{aligned}$$

From the previous part we know that $\|\nabla f(\vec{x}_i)\|_2 = 1$. Substitute this into the previous triangle inequality:

$$\|\vec{x}_{i+1}\|_2 \leq \|\vec{x}_i\|_2 + \frac{1}{i+1}$$

Now we look the infinite series:

$$S = \sum_{i=0}^{n} \frac{1}{i+1} = \ln(n+1)$$

Using Wikipedia page we know that:

$$\ln(n+1) \leq \sum_{i=0}^{n} \frac{1}{i+1} \leq \ln(n) + 1$$

Our goal is to find a n (number of iterations) such that $|\|\vec{x}\|_2 - \|\vec{b}\|_2| \leq \epsilon$. From above we know that:

$$\begin{aligned}
&|\|\vec{x}_{i+1}\|_2 - \|\vec{b}\|_2| \\
&\leq |\sum_{i=0}^{n} \frac{1}{i+1} - \|\vec{b}\|_2| \\
&\approx |\ln(n) - \|\vec{b}\|_2| \\
|\ln(n) - \|\vec{b}\|_2| &\approx \epsilon \\
\ln(n) &\approx \epsilon + \|\vec{b}\|_2 \\
n &\approx e^{\epsilon + \|\vec{b}\|_2} \\
n &\approx e^{\|\vec{b}\|_2}
\end{aligned}$$

Once our number of iteration exists $e^{\|\vec{b}\|_2}$, we will overshoot $\vec{b}$. The maximum overshoot distance from the optimal solution is $\frac{1}{i+1}$ because $\|\nabla f(\vec{x})\| = 1$. This is monotonically decreasing as we approach $\vec{b}$. In other words, the above infinite series turns to

$$\sum_{j=n+1}^{k} \frac{(-1)^{j+1}}{j} \approx \ln 2$$

Therefore This is true for any arbitrary $\vec{b}$ because we can always find a n for a given $\vec{b}$.

```
""" Tools for calculating Gradient Descent for ||Ax-b||. """
import matplotlib.pyplot as plt
import numpy as np


def main():
##################################################################################
# TODO(student): Input Variables
```

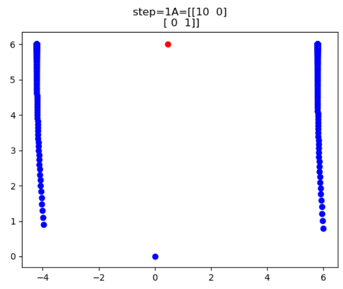
```
A = np.array([[1, 0], [0, 1]]) # do not change this until the last part
b = np.array([4.5, 6]) # b in the equation ||Ax-b||
initial_position = np.array([0, 0]) # position at iteration 0
total_step_count = 10000 # number of GD steps to take
# step_size = lambda i: 1 # step size at iteration i
step_size = 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, b)

# print out the values of the x_i
print('number of step: ' + str(positions.shape[0] - 1))
print(positions)
print('optimal point:' + str(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()


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, optimal=[], error=0.01):
"""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)]
for k in range(total_step_count):
current_position = compute_update(A, b, positions[-1], k, step_size)
positions.append(current_position)
if (len(optimal) > 0) and (np.linalg.norm(current_position - optimal) <= error ** 2):
break
return np.array(positions)


main()
#print(compute_gradient(np.array([[1, 0], [0, 1]]), np.array([4.5, 6]), np.array([4.8, 6.4])))
```

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

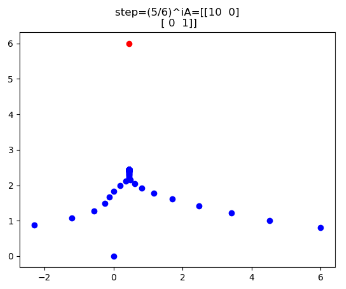
We can see that the first two methods don't work but the third one does work in these cases. This is because we can do linear transformation and get a new variable $\vec{x}^* = \mathbf{A}\vec{x}$. If $\mathbf{A}$ is invertible, we can always find a unique $\vec{x}$ given $\vec{x}^*$. And we already know the answer to $f(\vec{x}^*) = \|\vec{x}^* - \vec{b}\|_2$. On the other hand, if $\mathbf{A}$ is not invertible there are two possibilities. If $\vec{b}$ is in the column space of $\mathbf{A}$ then we can still do the reparamerization trick as if $\mathbf{A}$ were invertible. If $\vec{b}$ is not in the column space of $\mathbf{A}$ then gradient descent will not always set $\vec{x} = \vec{0}$ because $\vec{x}^*$ is always orthogonal to $\vec{b}$.

$$\mathbf{A} = \begin{bmatrix} 10 & 0 \\ 0 & 1 \end{bmatrix}$$

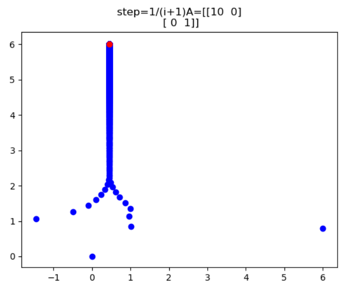
step $= 1$



step $= \left(\frac{5}{6}\right)^i$

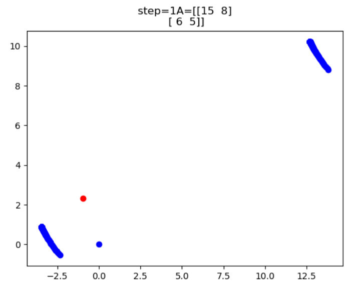


step $= \frac{1}{i+1}$

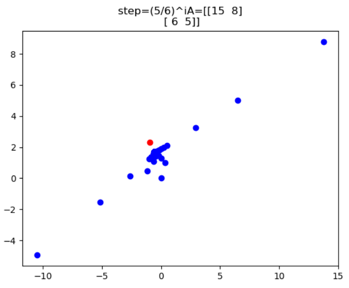


$$\mathbf{A} = \begin{bmatrix} 15 & 8 \\ 6 & 5 \end{bmatrix}$$

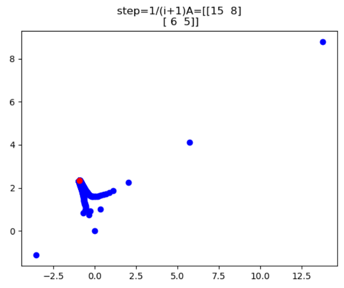
step $= 1$



step $= \left(\frac{5}{6}\right)^i$



step $= \frac{1}{i+1}$



```
""" Tools for calculating Gradient Descent for ||Ax-b||. """
import matplotlib.pyplot as plt
import numpy as np


def main():
#################################################################################
# TODO(student): Input Variables
allAs = (np.array([[10, 0], [0, 1]]), np.array([[15, 8], [6, 5]]))
#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||
total_step_count = 10000 # number of GD steps to take
# step_size = lambda i: 1 # step size at iteration i
step_names = ('step=1', 'step=(5/6)^i', 'step=1/(i+1)')
```

```
save_names = ('fixed', 'adaptive', 'reciprocal')
step_funcs = (lambda i: 1, lambda i: np.power(5/6, i), lambda i: 1/(i+1)) # step size at iteration
    i
#################################################################################

for iA, A in enumerate(allAs):
print('A='+str(A))
for i, (step_title, file_name, step_size) in enumerate(zip(step_names, save_names, step_funcs)):
initial_position = np.array([0, 0]) # position at iteration 0
# computes desired number of steps of gradient descent
positions = compute_updates(A, b, initial_position, total_step_count, step_size, b)

# print out the values of the x_i
print(step_title)
print('number of step: ' + str(positions.shape[0] - 1))
print(positions)
print('optimal point:' + str(np.dot(np.linalg.inv(A), b)))

# plot the values of the x_i
plt.figure()
plt.title(step_title + 'A='+str(A))
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()
plt.savefig('Figure_2e_' + file_name + '-' + str(iA) + '.png')
plt.close()


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, optimal=[], error=0.01):
"""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)]
for k in range(total_step_count):
current_position = compute_update(A, b, positions[-1], k, step_size)
positions.append(current_position)
if (len(optimal) > 0) and (np.linalg.norm(current_position - optimal) <= error):
break
return np.array(positions)


#main()
#print(compute_gradient(np.array([[1, 0], [0, 1]]), np.array([4.5, 6]), np.array([4.8, 6.4])))

print(np.array([[1, 1, 2], [0, 1, 2]]).shape)
```

Question 3. Convergence Rate of Gradient Descent

In the previous problem, you examined $\|\mathbf{A}\vec{x} - \vec{b}\|_2$ (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 $\frac{1}{2}\|\mathbf{A}\vec{x} - \vec{b}\|_2^2$ (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 $\vec{b} \in \mathbb{R}^n$, consider the quadratic function $f(\vec{x}) = \frac{1}{2}\|\mathbf{A}\vec{x}\|_2^2$ such that $\mathbf{A}^\top\mathbf{A}$ is positive definite.

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

$$\|\vec{u}^\top\vec{v}| \leqslant \|u\|_2\|v\|_2$$

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

1. First, consider the case $\vec{b} = \vec{0}$, and think of each $\vec{x} \in \mathbb{R}^d$ 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 $\gamma > 0$.** Use $\vec{x}_0$ to denote the initial condition of where you start gradient descent from.

$$\begin{aligned}
&\nabla_{\vec{x}}\frac{1}{2}\|\mathbf{A}\vec{x}\|_2^2\\
&= \nabla_{\vec{x}}\frac{1}{2}(\mathbf{A}\vec{x})^\top(\mathbf{A}\vec{x})\\
&= \mathbf{A}^\top\mathbf{A}\vec{x}
\end{aligned}$$

Therefore, we can write the update equation:

$$\begin{aligned}
&\vec{x}_{i+1}\\
&= \vec{x}_i - \gamma\nabla_{\vec{x}}f(\vec{x}_i)\\
&= \vec{x}_i - \gamma\mathbf{A}^\top\mathbf{A}\vec{x}_i\\
&= (\mathbf{I} - \gamma\mathbf{A}^\top\mathbf{A})\vec{x}_i
\end{aligned}$$

Because $\mathbf{A}^\top\mathbf{A}$ is positive definite, we can do eigendecomposition and get $\mathbf{A}^\top\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^\top$. Substitute into the formula above gives us:

$$\begin{aligned}
&\vec{x}_{i+1}\\
&= (\mathbf{I} - \gamma\mathbf{A}^\top\mathbf{A})\vec{x}_i\\
&= (\mathbf{I} - \gamma\mathbf{V}\mathbf{\Lambda}\mathbf{V}^\top)\vec{x}_i
\end{aligned}$$

Let's now compute $\vec{x}_{i+2}$

$$\begin{aligned}
&\vec{x}_{i+2}\\
&= (\mathbf{I} - \gamma\mathbf{V}\mathbf{\Lambda}\mathbf{V}^\top)\vec{x}_{i+1}\\
&= (\mathbf{I} - \gamma\mathbf{V}\mathbf{\Lambda}\mathbf{V}^\top)(\mathbf{I} - \gamma\mathbf{V}\mathbf{\Lambda}\mathbf{V}^\top)\vec{x}_i\\
&= (\mathbf{I} - \gamma\mathbf{V}\mathbf{\Lambda}\mathbf{V}^\top)(\mathbf{I} - \gamma\mathbf{V}\mathbf{\Lambda}\mathbf{V}^\top)\vec{x}_i\\
&= (\mathbf{I} - 2\gamma\mathbf{V}\mathbf{\Lambda}\mathbf{V}^\top + \gamma^2\mathbf{V}\mathbf{\Lambda}\mathbf{V}^\top\mathbf{V}\mathbf{\Lambda}\mathbf{V}^\top)\vec{x}_i\\
&= (\mathbf{V}^\top\mathbf{V} - \mathbf{V}2\gamma\mathbf{\Lambda}\mathbf{V}^\top + \mathbf{V}\gamma^2\mathbf{\Lambda}^2\mathbf{V}^\top)\vec{x}_i\\
&= \mathbf{V}(I - 2\gamma\mathbf{\Lambda} + \gamma^2\mathbf{\Lambda}^2)\mathbf{V}^\top\vec{x}_i
\end{aligned}$$

We can then write out the general form of the solution to $\vec{x_n}$:

$$\vec{x}_n = \mathbf{V}\sum_{i=0}^{n}(\binom{n}{i}(\gamma\mathbf{\Lambda})^i)\mathbf{V}^\top\vec{x}_0 = \mathbf{V}(\mathbf{I}-\gamma\mathbf{\Lambda})^i\mathbf{V}^\top\vec{x}_0$$

where $\mathbf{V}$ and $\mathbf{\Lambda}$ are the eigenvectors and eigenvalues of $\mathbf{A}^\top\mathbf{A}$

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

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

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 for $\vec{x}_0$ for large enough n.

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

Copy the solution from above down here:

$$\vec{x}_n = \mathbf{V}\sum_{i=0}^{n}(\binom{n}{i}(\gamma\mathbf{\Lambda})^i)\mathbf{V}^\top\vec{x}_0 = \mathbf{V}(\mathbf{I}-\gamma\mathbf{\Lambda})^i\mathbf{V}^\top\vec{x}_0$$

where $\mathbf{V}$ and $\mathbf{\Lambda}$ are the eigenvectors and eigenvalues of $\mathbf{A}^\top\mathbf{A}$

We can read off B from the formula above: $\mathbf{B} = \mathbf{V}(\mathbf{I}-\gamma\mathbf{\Lambda})^i\mathbf{V}^\top$. The eigenvalues of B are given in the diagonal matrix $(\mathbf{I}-\gamma\mathbf{\Lambda})^i$. Because we need *all* the eigenvalues of $\mathbf{B}$ to be less than or equal to 1 in absolute value, we have:

$$-\mathbf{I} \leq \mathbf{I}-\gamma\mathbf{\Lambda} \leq \mathbf{I}$$
$$0 \leq \mathbf{\Lambda} \leq \frac{2}{\gamma}\mathbf{I}$$

The above means if the eigenvalues of $A^\top A$ is bounded between $[0, \frac{2}{\gamma}]$, the state evolution of the iterations is stable.

3. We want to bound the progress of gradient descent in the general case, when $\vec{b}$ 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 $\phi(\vec{x}) = \vec{x} - \gamma \nabla f(\vec{x})$, for some constant step size $\gamma > 0$. **Show that for any $\vec{x}, \vec{x}' \in \mathbb{R}^d$,**

$$\|\phi(\vec{x}) - \phi(\vec{x}')\|_2 \leqslant \beta \|\vec{x} - \vec{x}'\|_2$$

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

$$\begin{aligned}
&\|\phi(\vec{x}) - \phi(\vec{x}')\|_2 \\
=&\|\vec{x} - \gamma\nabla f(\vec{x}) - \vec{x}' + \gamma\nabla f(\vec{x}')\|_2 \\
=&\|\vec{x} - \vec{x}' - (\gamma\nabla f(\vec{x}) - \gamma\nabla f(\vec{x}'))\|_2 \\
=&\|\vec{x} - \vec{x}' - \gamma(\nabla f(\vec{x}) - \nabla f(\vec{x}'))\|_2 \\
=&\|\vec{x} - \vec{x}' - \gamma\mathbf{A}^\top\mathbf{A}(\vec{x} - \vec{x}')\|_2 \\
=&\|(\mathbf{I} - \gamma\mathbf{A}^\top\mathbf{A})(\vec{x} - \vec{x}')\|_2 \\
\leq&\lambda_{max}(\mathbf{I} - \gamma\mathbf{A}^\top\mathbf{A})\|(\vec{x} - \vec{x}')\|_2 \\
\leq&\beta\|(\vec{x} - \vec{x}')\|_2
\end{aligned}$$

We can find a proof of $\|Ax\|_2 \leq \lambda_{max}(A)\|x\|_2$, which comes from the Rayleigh Quotient inequality $\frac{\|Ax\|_2}{\|x\|_2} \leq \lambda_{max}(A)$.

The last step is true because we just need to find the eigenvalue which has the largest deviation from 1. The largest deviation can only be found at the maximum eigenvalue or the minimum eigenvalue because of the convexity of the quadratic function above.

4. Now we give a bound for progress after k steps of gradient descent. Define

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

Show that

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

and conclude that

$$\|\vec{x}_{k+1} - \vec{x}^*\|_2 \leqslant \beta^{k+1}\|\vec{x}_0 - \vec{x}^*\|_2$$

$$\begin{aligned}
&\|\phi(\vec{x}_k) - \phi(\vec{x}^*)\|_2 \\
=&\|\vec{x}_k - \gamma\nabla f(\vec{x}) - \vec{x}_k^* + \gamma\nabla f(\vec{x}^*)\| \\
=&\|\vec{x}_k - \gamma\mathbf{A}^\top\mathbf{A}\vec{x} - \vec{x}^* + \gamma\mathbf{A}^\top\mathbf{A}\vec{x}^*\|_2 \\
=&\|\vec{x}_{k+1} - \vec{x}^*\|_2
\end{aligned}$$

The above formula is true because by definition, $\Phi\vec{x}_k = \vec{x}_{k+1}$ and the optimal point has a gradient that is zero $\nabla f(\vec{x}^*) = 0$.

$$\begin{aligned}
&\|\vec{x}_{k+1} - \vec{x}^*\|_2 \\
=&\|\phi(\vec{x}_k) - \phi(\vec{x}^*)\|_2 \\
=&\frac{\|\phi(\vec{x}_k) - \phi(\vec{x}^*)\|_2}{\|\phi(\vec{x}_{k-1}) - \phi(\vec{x}^*)\|_2}\frac{\|\phi(\vec{x}_{k-1}) - \phi(\vec{x}^*)\|_2}{\|\phi(\vec{x}_{k-2}) - \phi(\vec{x}^*)\|_2}\cdots\frac{\|\phi(\vec{x}_1) - \phi(\vec{x}^*)\|_2}{\|\phi(\vec{x}_0) - \phi(\vec{x}^*)\|_2}\frac{\|\phi(\vec{x}_0) - \phi(\vec{x}^*)\|_2}{\|\vec{x}_0 - \phi(\vec{x}^*)\|_2}\|\vec{x}_0 - \phi(\vec{x}^*)\|_2 \\
\leq&\beta^{k+1}\|\vec{x}_0 - \phi(\vec{x}^*)\|_2
\end{aligned}$$

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

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

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

The above equalities are true because we have:

$$\nabla f(\vec{x}^*) = \mathbf{A}^\top(\mathbf{A}\vec{x}^* - \vec{b}) = 0$$

6. **Show that**

$$f(\vec{x}_k) - f(\vec{x}^*) \leqslant \frac{\alpha}{2}\|\vec{x}_k - \vec{x}^*\|_2^2$$

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

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

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

Again we use Rayleigh Quotient here. We can use the conclusion in part (d) and get:

$$\begin{aligned}
&f(\vec{x}_k) - f(\vec{x}^*)\\
\leqslant&\frac{\alpha}{2}\|x_k - x\|_2^2\\
\leqslant&\frac{\alpha}{2}(\beta^k)^2\|\vec{x}_0 - \vec{x}^*\|_2^2\\
\leqslant&\frac{\alpha}{2}\beta^{2k}\|\vec{x}_0 - \vec{x}^*\|_2^2
\end{aligned}$$

7. 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 can pick γ such that β is as small as possible, as a function of $\lambda_{\min}\left(\mathbf{A}^\top\mathbf{A}\right)$, $\lambda_{\max}\left(\mathbf{A}^\top\mathbf{A}\right)$. **Write the resulting convergence rate as a function of $\kappa = \frac{\lambda_{\max}\left(\mathbf{A}^\top\mathbf{A}\right)}{\lambda_{\min}\left(\mathbf{A}^\top\mathbf{A}\right)}$, That is, show that**

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

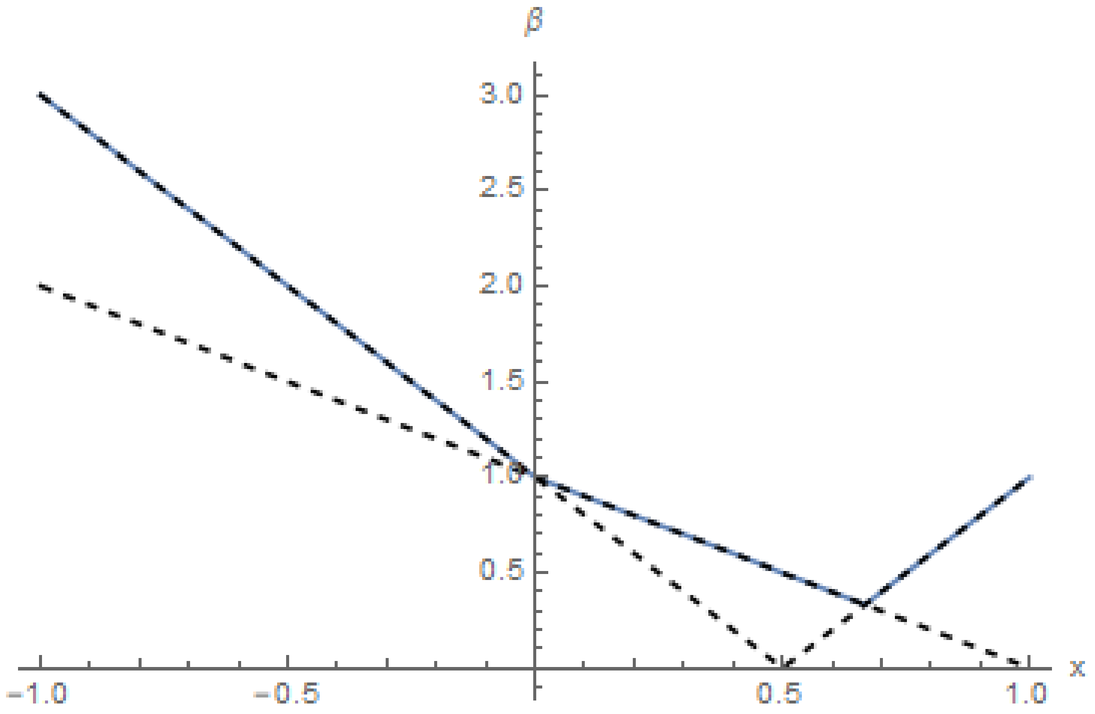
Here is what we want:

$$\begin{aligned}\beta &= \max\left\{|1-\gamma\lambda_{\max}\left(\mathbf{A}^\top\mathbf{A}\right)|, |1-\gamma\lambda_{\min}\left(\mathbf{A}^\top\mathbf{A}\right)|\right\}\\ &= \frac{\kappa-1}{\kappa+1}\\ &= \frac{\lambda_{\max}\left(\mathbf{A}^\top\mathbf{A}\right)-\lambda_{\min}\left(\mathbf{A}^\top\mathbf{A}\right)}{\lambda_{\max}\left(\mathbf{A}^\top\mathbf{A}\right)+\lambda_{\min}\left(\mathbf{A}^\top\mathbf{A}\right)}\end{aligned}$$

Let's plot a general function β:

$$\beta = \begin{cases}|ax-1| & |ax-1| \geq |bx-1|\\ |bx-1| & |ax-1| < |bx-1|\end{cases}$$

where we assume $a \geq b \geq 0$



We can read out the global minimum on the graph, which is achieved at $|ax-1| = |bx-1|, x > 0$. Therefore we should pick:

$$\begin{aligned}1-\gamma\lambda_{\max}\left(\mathbf{A}^\top\mathbf{A}\right) &= -1+\gamma\lambda_{\min}\left(\mathbf{A}^\top\mathbf{A}\right)\\ \gamma &= \frac{2}{\lambda_{\max}\left(\mathbf{A}^\top\mathbf{A}\right)+\lambda_{\min}\left(\mathbf{A}^\top\mathbf{A}\right)}\end{aligned}$$

$$\beta = 1 - \frac{2}{\lambda_{\max}\left(\mathbf{A}^\top\mathbf{A}\right) + \lambda_{\min}\left(\mathbf{A}^\top\mathbf{A}\right)}\lambda_{\max}\left(\mathbf{A}^\top\mathbf{A}\right) = \frac{\lambda_{\max}\left(\mathbf{A}^\top\mathbf{A}\right) - \lambda_{\min}\left(\mathbf{A}^\top\mathbf{A}\right)}{\lambda_{\max}\left(\mathbf{A}^\top\mathbf{A}\right) + \lambda_{\min}\left(\mathbf{A}^\top\mathbf{A}\right)}$$

Therefore, we got the desired form:

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

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

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

For this problem, all Python libraries are permitted.

1. 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

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

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

$$\begin{aligned}
&\nabla_{x_1} L(x_1, y_1) \\
=&\nabla_{x_1} - \sum_{i=1}^{7} (\sqrt{(a_i - x_1)^2 + (b_i - y_1)^2} - d_i)^2 \\
=&-\sum_{i=1}^{7} \nabla_{x_1} (\sqrt{(a_i - x_1)^2 + (b_i - y_1)^2} - d_i)^2 \\
=&-\sum_{i=1}^{7} 2(\sqrt{(a_i - x_1)^2 + (b_i - y_1)^2} - d_i) \frac{1}{2} \frac{1}{\sqrt{(a_i - x_1)^2 + (b_i - y_1)^2}} (2(a_i - x_1))(-1) \\
=&-2\sum_{i=1}^{7} (\sqrt{(a_i - x_1)^2 + (b_i - y_1)^2} - d_i) \frac{x_1 - a_i}{\sqrt{(a_i - x_1)^2 + (b_i - y_1)^2}}
\end{aligned}$$

$$\begin{aligned}
&\nabla_{y_1} L(x_1, y_1) \\
=&\nabla_{y_1} - \sum_{i=1}^{7} (\sqrt{(a_i - x_1)^2 + (b_i - y_1)^2} - d_i)^2 \\
=&-\sum_{i=1}^{7} \nabla_{y_1} (\sqrt{(a_i - x_1)^2 + (b_i - y_1)^2} - d_i)^2
\end{aligned}$$

$$
\begin{aligned}
&= -\sum_{i=1}^{7} 2(\sqrt{(a_i - x_1)^2 + (b_i - y_1)^2} - d_i)\frac{1}{2}\frac{1}{\sqrt{(a_i - x_1)^2 + (b_i - y_1)^2}}(2(b_i - y_1))(-1) \\
&= -2\sum_{i=1}^{7}(\sqrt{(a_i - x_1)^2 + (b_i - y_1)^2} - d_i)\frac{y_1 - b_i}{\sqrt{(a_i - x_1)^2 + (b_i - y_1)^2}}
\end{aligned}
$$

2. The provided code generates

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

for $\mu = [0, 0]^\top$, $\sigma_s = 100$ 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 $\vec{0}$ and starting at a random point. Which of the following step sizes is a reasonable one, $1, 0.01, 0.001$ or 0.0001?

Using lr=1 is too big and cannot find the real location; it returns a runtime warning. lr=0.01 and 0.001 are relatively good. lr=0.0001 is too slow to converge to the solution. Where we started is not very important in this simulation and random starting point performs very similarly to zero starting point.

lr = 1.0

The real object location is

$$\begin{bmatrix}44.38632327 & 33.36743274\end{bmatrix}$$

The estimated object location with zero initialization is

$$\begin{bmatrix}nan & nan\end{bmatrix}$$

The estimated object location with random initialization is

$$\begin{bmatrix}nan & nan\end{bmatrix}$$

lr = 0.01

The real object location is

$$\begin{bmatrix}44.38632327 & 33.36743274\end{bmatrix}$$

The estimated object location with zero initialization is

$$\begin{bmatrix}43.07188433 & 32.71217817\end{bmatrix}$$

The estimated object location with random initialization is

$$\begin{bmatrix}43.07188433 & 32.71217817\end{bmatrix}$$

lr = 0.001

The real object location is

$$\begin{bmatrix} 44.38632327 & 33.36743274 \end{bmatrix}$$

The estimated object location with zero initialization is

$$\begin{bmatrix} 43.07188433 & 32.71217817 \end{bmatrix}$$

The estimated object location with random initialization is

$$\begin{bmatrix} 43.07188433 & 32.71217817 \end{bmatrix}$$

lr = 0.0001

The real object location is

$$\begin{bmatrix} 44.38632327 & 33.36743274 \end{bmatrix}$$

The estimated object location with zero initialization is

$$\begin{bmatrix} 42.56743732 & 32.03587615 \end{bmatrix}$$

The estimated object location with random initialization is

$$\begin{bmatrix} 42.57881496 & 32.05165343 \end{bmatrix}$$

```
from common import *
import numpy as np


def bmatrix(a):
"""Returns a LaTeX bmatrix
Retrieved from https://stackoverflow.com/questions/17129290/numpy-2d-and-1d-array-to-latex-bmatrix
:a: numpy array
:returns: LaTeX bmatrix as a string
"""
if len(a.shape) > 2:
raise ValueError('bmatrix can at most display two dimensions')
lines = str(a).replace('[', '').replace(']', '').splitlines()
rv = [r'\[']
rv += [r'\begin{bmatrix}']
rv += [' ' + ' & '.join(l.split()) + r'\\' for l in lines]
rv += [r'\end{bmatrix}']
rv += [r'\]\par']
return '\n'.join(rv)


##########################################################################
######### 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
for ik in range(sensor_loc.shape[0]):
grad = grad + (2 * (1 - single_distance[ik] / np.linalg.norm(single_obj_loc - sensor_loc[ik, :]))
* (single_obj_loc - sensor_loc[ik, :]))
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.

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

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

Output:
obj_loc: 1 * d numpy array. The mle for the location of the object.

"""
obj_loc = initial_obj_loc
# Your code: do gradient descent
for i in range(num_iters):
obj_loc = obj_loc - (lr * compute_gradient_of_likelihood(obj_loc, sensor_loc, single_distance))
return obj_loc


if __name__ == "__main__":
########################################################################
######### MAIN ########################################################
########################################################################

# Your code: set some appropriate learning rate here 0.01, 0.001, 0.0001
all_step_sizes = [1.0, 0.01, 0.001, 0.0001]
for _, lr in enumerate(all_step_sizes):
print('\hfill \linebreak')
print('lr = ' + str(lr) + ' \par')
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 \par')
print(bmatrix(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 \par')
print(bmatrix(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 \par')
print(bmatrix(estimated_obj_loc))
```

3. (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**:

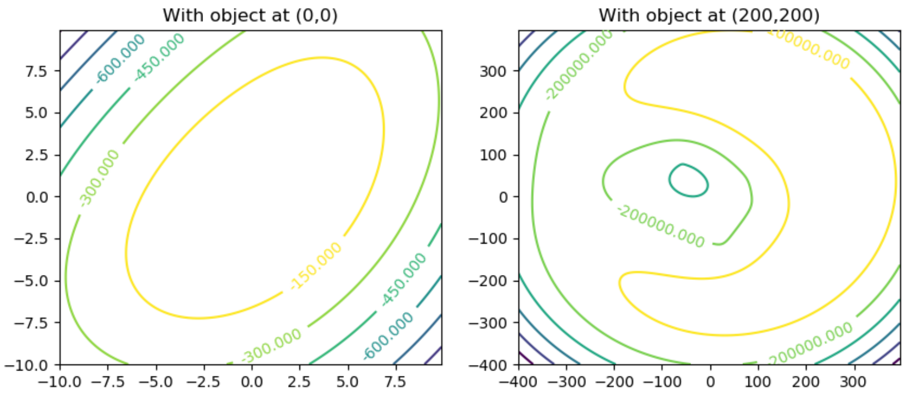
- Generate $m = 7$ sensor locations (a_i, b_i) from $(\mathbf{0}, \sigma_s^2\mathbf{I})$ (Use the same σ_s from the previous part.)
- Generate $mn = 7$ distance measurements $D_{i1} = \|(a_i, b_i) - (x_1, y_1)\| + \mathcal{N}(0,1)$.

For each data set, carry out gradient descents 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 $\mathcal{N}(\mathbf{0}, \sigma^2\mathbf{I})$, 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** $(x_1, y_1) = (100,100)$**.**
- 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?

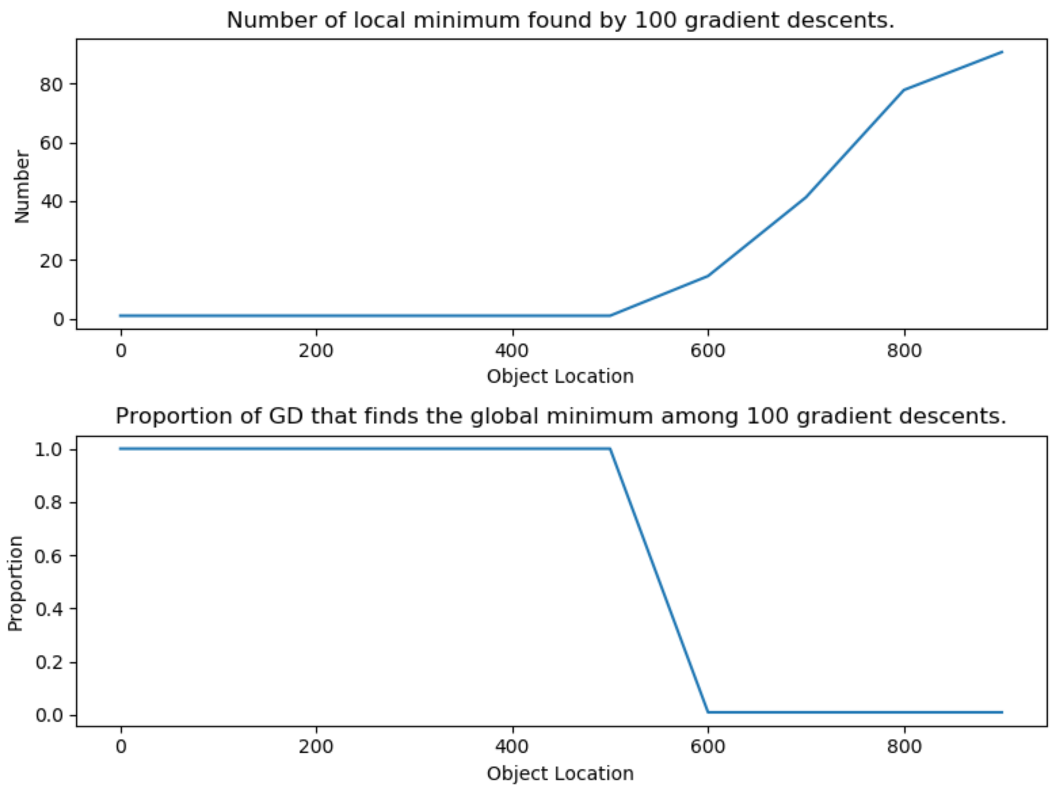
Please be aware that the code might take a while to run.

As we can see below, when the object is moved from the origin, we started to get local minimum in our log likelihood function. This might be problematic if we are not careful about where to start and how to do gradient descent.

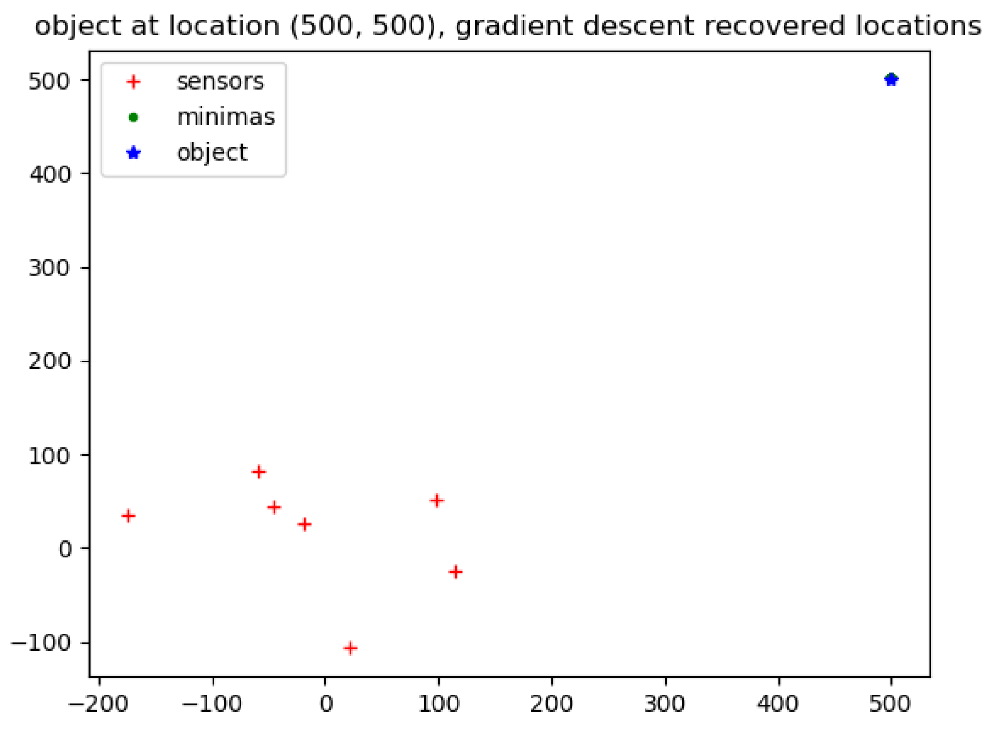


Here we show two things. The further away the object is from the origin, the more global minimum there might be. This is consistent with our observation in the change of contour plot. This is due to the fact that the further away the object, the less useful information we can get from a bunch of clustered sensors.

After a certain distance away from the origin, gradient descent cannot find the global minimum anymore. This is not surprising because there are more local attractors for gradient descents to be trapped.



I'm sorry but I don't see any pattern in my result. The gradient descent successfully found the true object location. I think we are supposed to point out that all sensors are clustered together and the object is very far away from them. This means the readings of all sensors will be very close such that they are not very informative.



```
from common import *
from part_b_starter import find_mle_by_grad_descent_part_b

########################################################################
######### 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 = 0.0
for ik in range(sensor_loc.shape[0]):
func_value -= (np.linalg.norm(obj_loc - sensor_loc[ik, :]) - distance[ik]) ** 2
return func_value

if __name__ == "__main__":
########################################################################
######### Compute the function value at local minimum for all experiments.###
########################################################################
num_sensors = 7

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

# num_data_replicates = 10
num_gd_replicates = 100
```

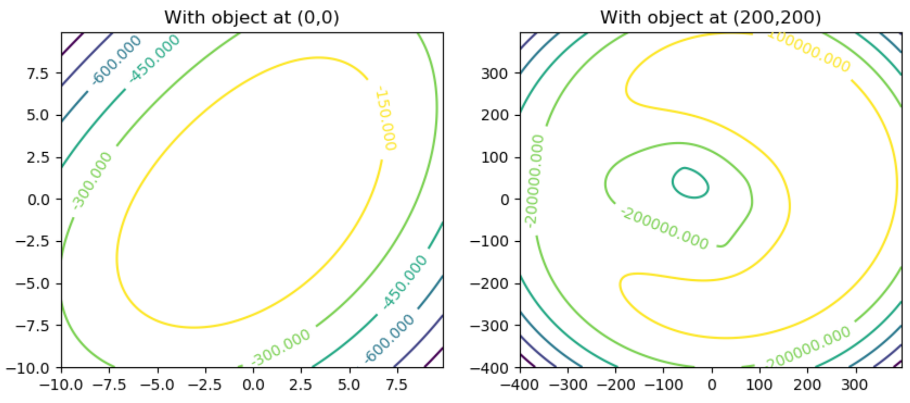
```
obj_locs = [[[i,i]] for i in np.arange(0,1000,100)]

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.

plt.subplot(212)
plt.plot(np.arange(0,1000,100), proportion_global)
plt.title('Proportion of GD that finds the global minimum among 100 gradient descents.')
plt.xlabel('Object Location')
plt.ylabel('Proportion')
fig.tight_layout()
plt.savefig('prop_obj.png')

########################################################################
######### Plots of contours. ###########################################
########################################################################
np.random.seed(0)
```

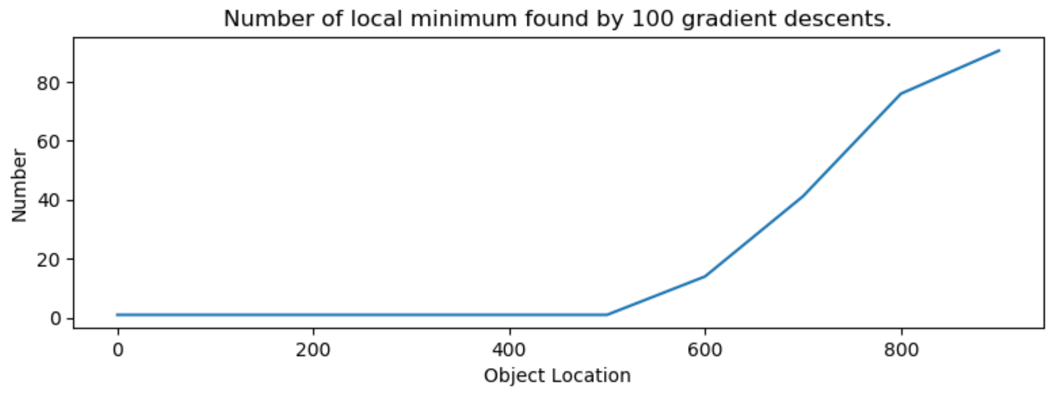
```
# sensor_loc = np.random.randn(7,2) * 10
x = np.arange(-10.0, 10.0, 0.1)
y = np.arange(-10.0, 10.0, 0.1)
X, Y = np.meshgrid(x, y)
obj_loc = [[0,0]]
obj_loc, distance = generate_data_given_location(sensor_loc,
obj_loc, k = num_sensors, d = 2)

Z =  np.array([[log_likelihood((X[i,j],Y[i,j]),
sensor_loc, distance[0]) for j in range(len(X))] \
for i in range(len(X))])


plt.figure(figsize=(10,4))
plt.subplot(121)
CS = plt.contour(X, Y, Z)
plt.clabel(CS, inline=1, fontsize=10)
plt.title('With object at (0,0)')
#plt.show()

np.random.seed(0)
# sensor_loc = np.random.randn(7,2) * 10
x = np.arange(-400,400, 4)
y = np.arange(-400,400, 4)
X, Y = np.meshgrid(x, y)
obj_loc = [[200,200]]
obj_loc, distance = generate_data_given_location(sensor_loc,
obj_loc, k = num_sensors, d = 2)

Z =  np.array([[log_likelihood((X[i,j],Y[i,j]),
sensor_loc, distance[0]) for j in range(len(X))] \
for i in range(len(X))])


# Create a simple contour plot with labels using default colors. The
# inline argument to clabel will control whether the labels are draw
# over the line segments of the contour, removing the lines beneath
# the label
#plt.figure()
plt.subplot(122)
CS = plt.contour(X, Y, Z)
plt.clabel(CS, inline=1, fontsize=10)
plt.title('With object at (200,200)')
#plt.show()
plt.savefig('likelihood_landscape.png')


########################################################################
######### Plots of Found local minimas. ###########################################
########################################################################
#sensor_loc
#minimas = np.zeros((len(obj_locs), 10, num_gd_replicates, 2))
#true_object_locs = np.zeros((len(obj_locs), 10, 2))
object_loc_i = 5
trail = 0

plt.figure()
plt.plot(sensor_loc[:, 0], sensor_loc[:, 1], 'r+', label="sensors")
plt.plot(minimas[object_loc_i, trail, :, 0], minimas[object_loc_i, trail, :, 1], 'g.',
    label="minimas")
plt.plot(true_object_locs[object_loc_i, trail, 0], true_object_locs[object_loc_i, trail, 1], 'b*',
    label="object")
plt.title('object at location (%d, %d), gradient descent recovered locations' % (object_loc_i*100,
    object_loc_i*100))
plt.legend()
plt.savefig('2D_vis.png')
```

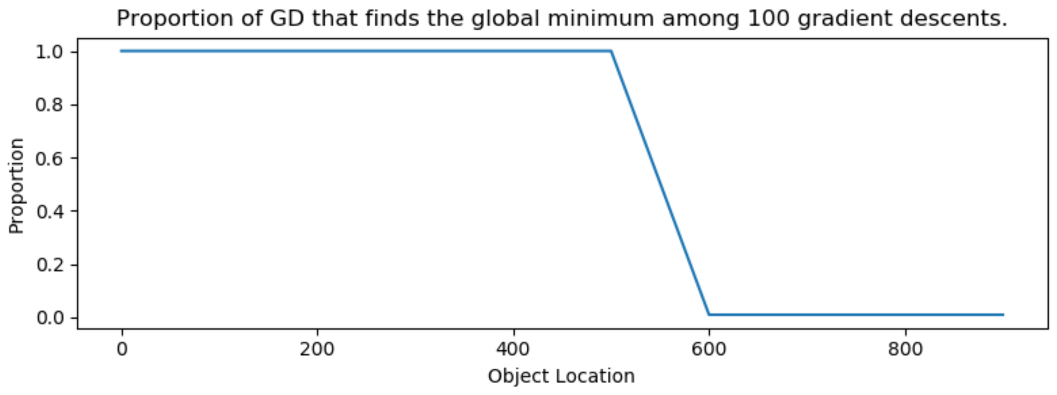
4. Repeat the previous part, except explore what happens as you reduce the variance of the measurement noise. **Comment with appropriate plots justifying your comments.**

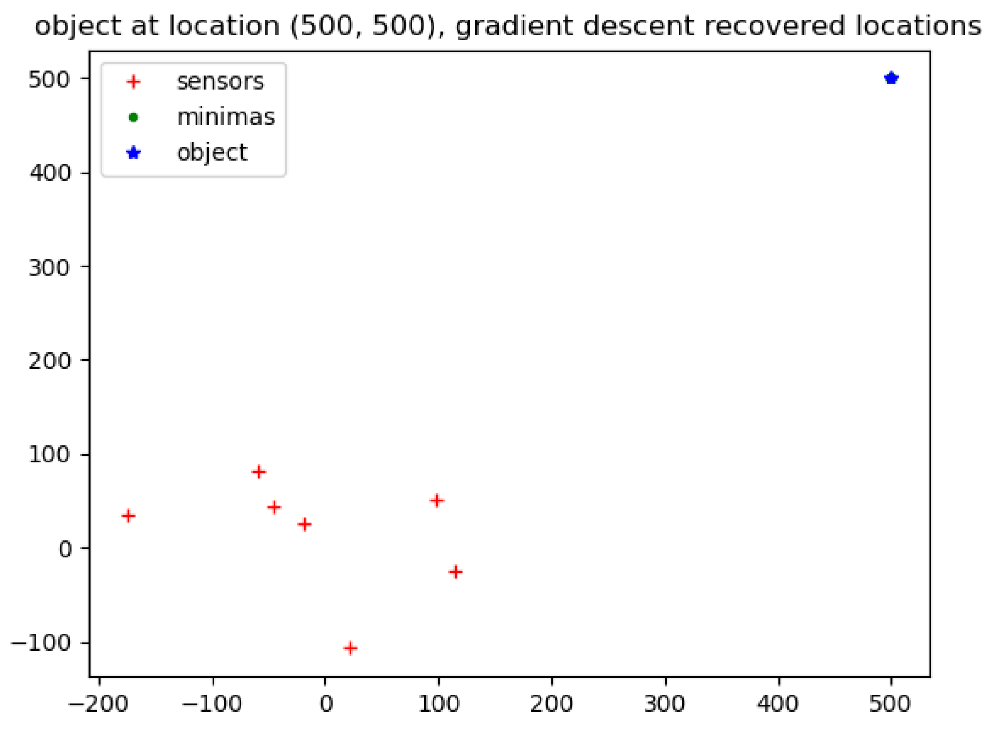
   For the sake of saving time, you can experiment with only one smaller noise, such as $\mathcal{N}(0, 0.01^2)$. You might need to change the learning rate to make the gradient descent converge.

As we can see, reducing noise doesn't really change the result that much. This is because the noise is not responsible for the generation of local minima in the previous part. Those local minima are created due to the long distance between the object and the cluster of sensors.









```
from common_d import *
from part_b_starter import find_mle_by_grad_descent_part_b

#########################################################################
######### 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 = 0.0
for ik in range(sensor_loc.shape[0]):
func_value -= (np.linalg.norm(obj_loc - sensor_loc[ik, :]) - distance[ik]) ** 2
return func_value

if __name__ == "__main__":
#########################################################################
######### Compute the function value at local minimum for all experiments.###
#########################################################################
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.

plt.subplot(212)
plt.plot(np.arange(0,1000,100), proportion_global)
plt.title('Proportion of GD that finds the global minimum among 100 gradient descents.')
plt.xlabel('Object Location')
plt.ylabel('Proportion')
fig.tight_layout()
plt.savefig('prop_obj_d.png')

########################################################################
######### Plots of contours. ###########################################
########################################################################
np.random.seed(0)
```

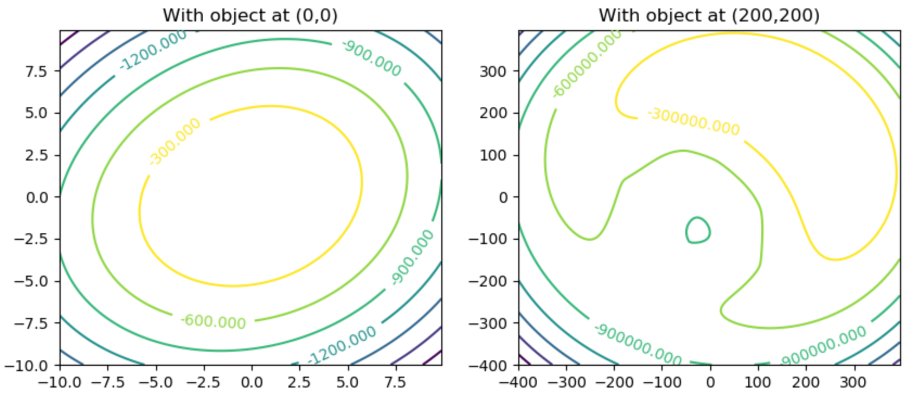
```
# sensor_loc = np.random.randn(7,2) * 10
x = np.arange(-10.0, 10.0, 0.1)
y = np.arange(-10.0, 10.0, 0.1)
X, Y = np.meshgrid(x, y)
obj_loc = [[0,0]]
obj_loc, distance = generate_data_given_location(sensor_loc,
obj_loc, k = num_sensors, d = 2)

Z = np.array([[log_likelihood((X[i,j],Y[i,j]),
sensor_loc, distance[0]) for j in range(len(X))] \
for i in range(len(X))])


plt.figure(figsize=(10,4))
plt.subplot(121)
CS = plt.contour(X, Y, Z)
plt.clabel(CS, inline=1, fontsize=10)
plt.title('With object at (0,0)')
#plt.show()

np.random.seed(0)
# sensor_loc = np.random.randn(7,2) * 10
x = np.arange(-400,400, 4)
y = np.arange(-400,400, 4)
X, Y = np.meshgrid(x, y)
obj_loc = [[200,200]]
obj_loc, distance = generate_data_given_location(sensor_loc,
obj_loc, k = num_sensors, d = 2)

Z = np.array([[log_likelihood((X[i,j],Y[i,j]),
sensor_loc, distance[0]) for j in range(len(X))] \
for i in range(len(X))])


# Create a simple contour plot with labels using default colors. The
# inline argument to clabel will control whether the labels are draw
# over the line segments of the contour, removing the lines beneath
# the label
#plt.figure()
plt.subplot(122)
CS = plt.contour(X, Y, Z)
plt.clabel(CS, inline=1, fontsize=10)
plt.title('With object at (200,200)')
#plt.show()
plt.savefig('likelihood_landscape_d.png')


########################################################################
######### Plots of Found local minimas. ###########################################
########################################################################
#sensor_loc
#minimas = np.zeros((len(obj_locs), 10, num_gd_replicates, 2))
#true_object_locs = np.zeros((len(obj_locs), 10, 2))
object_loc_i = 5
trail = 0

plt.figure()
plt.plot(sensor_loc[:, 0], sensor_loc[:, 1], 'r+', label="sensors")
plt.plot(minimas[object_loc_i, trail, :, 0], minimas[object_loc_i, trail, :, 1], 'g.',
    label="minimas")
plt.plot(true_object_locs[object_loc_i, trail, 0], true_object_locs[object_loc_i, trail, 1], 'b*',
    label="object")
plt.title('object at location (%d, %d), gradient descent recovered locations' % (object_loc_i*100,
    object_loc_i*100))
plt.legend()
plt.savefig('2D_vis_d.png')
```

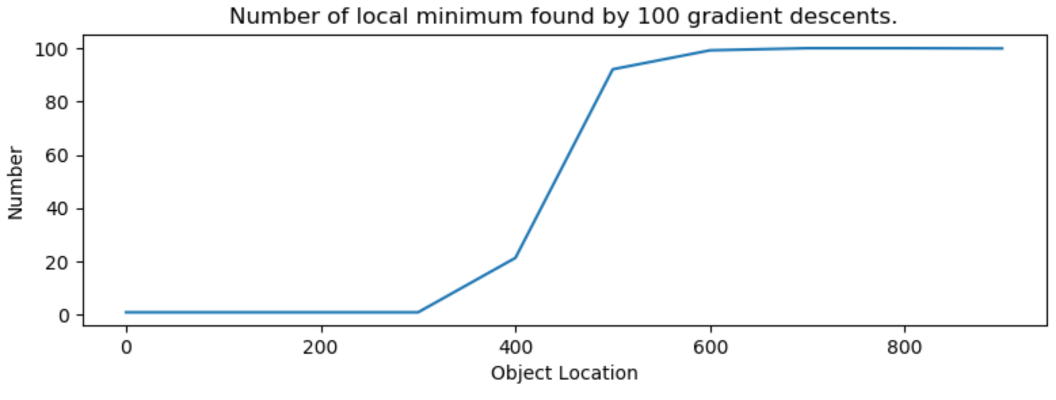
```
import numpy as np
import scipy.spatial
import matplotlib
import matplotlib.pyplot as plt
########################################################################
######### Data Generating Functions ###################################
########################################################################
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, sigma_n=0.01):
"""
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.
```

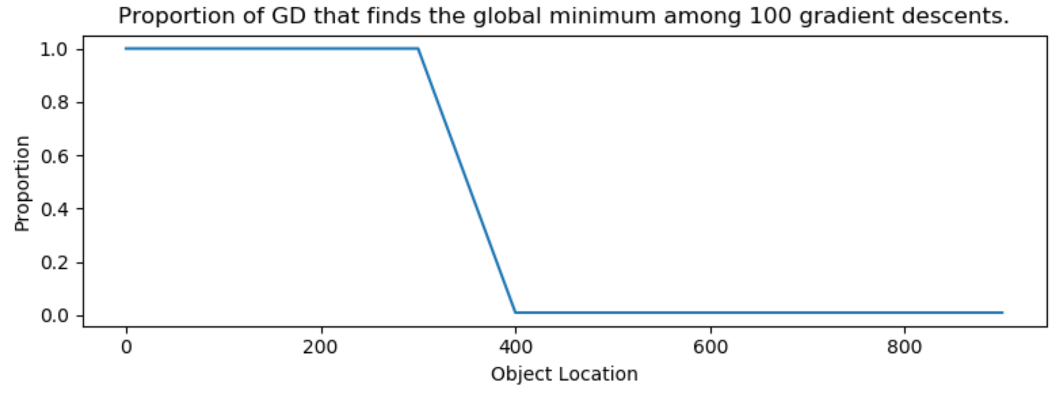
```
k: The number of sensors.
d: The spatial dimension.

Output:
distance: 1 * k numpy array. The distance between object and
the k sensors.
"""
assert k, d == sensor_loc.shape

distance = scipy.spatial.distance.cdist(obj_loc,
sensor_loc,
metric='euclidean')
distance += np.random.randn(1, k) * sigma_n
return obj_loc, distance
```

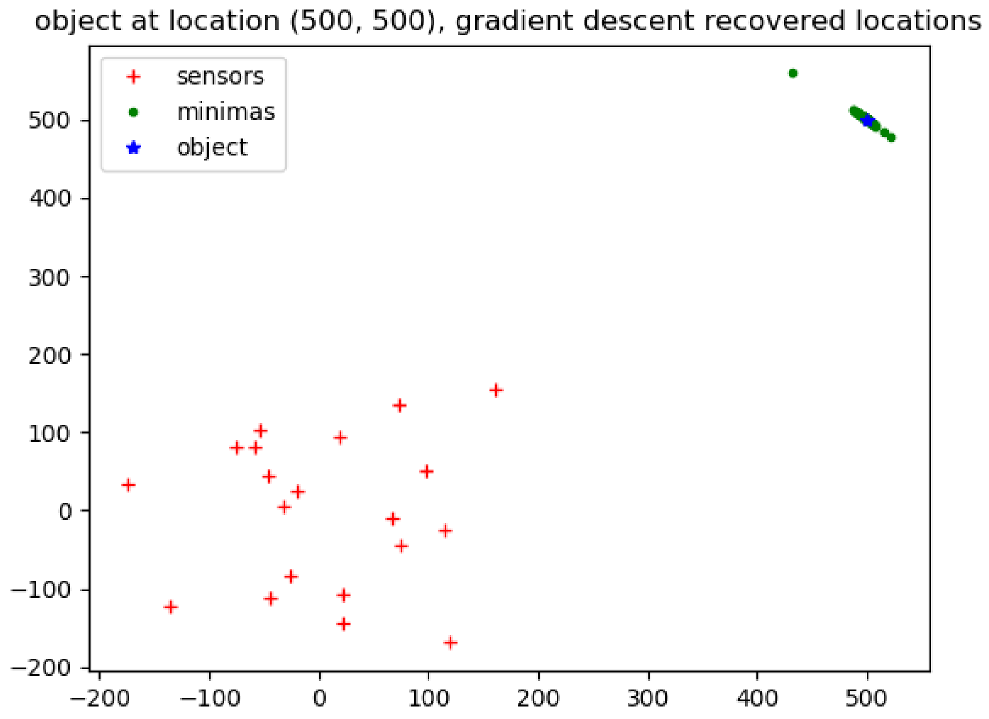
5. Repeat part (c) again, except explore what happens as you increase the number of sensors. For the sake of saving time, you can experiment with only one number of sensors, such as 20. **Comment with appropriate plots justifying your comments.**

Having more sensors lowers the threshold value for global minimum to be undetectable. This is because more sensors will generate more complicated landscape and result in more global minima. This can be seen on the graph below. We can see the predicted object locations fall on a circle of a certain radius around the origin. This is because all sensors have similar readings due to the long distances between sensors and the object and therefore . On the other hand, if the object is closer, more sensors should help as they can eliminate local minima.








object at location (500, 500), gradient descent recovered locations
sensors
minimas
object
500
400
300
200
100
0
−100
−200
−200
−100
0
100
200
300
400
500

6. 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 $i = 1, \dots, 7; j = 1, 2, \dots, 100$.

   Use the provided code to generate

   - $m = 7$ sensor locations (a_i, b_i) sampled from $\mathcal{N}(\mathbf{0}, \sigma^2\mathbf{I})$
   - $n = 100$ object locations (x_j, y_j) sampled from $\mathcal{N}(\mu, \sigma^2\mathbf{I})$ in 3 datasets: (1) Training data with $\mu = \vec{0}$, (2) Interpolating Test data with $\mu = \vec{0}$, and (3) Extrapolating Test data with $\mu = [300, 300]^\top$.
   - $mn = 700$ distance measurements $D_{ij} = \|(a_i, b_i) - (x_j, y_j)\| + \mathcal{N}(0, 1)$ 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** $(300, 300)$ **(such that we can have a better initial guess in the gradient descent).**

First we vectorize the log likelihood function: L:

$$\vec{s}_i = \begin{bmatrix} a_i \\ b_i \end{bmatrix} \quad \vec{r}_j = \begin{bmatrix} x_j \\ y_j \end{bmatrix}$$

$$\begin{aligned} &L(\vec{s}_k, \dots, \vec{s}_m) \\ &= -\sum_{j=1}^{n}\sum_{i=1}^{m}(\|\vec{s}_i - \vec{r}_j\| - D_{ij})^2 \end{aligned}$$

$$\begin{aligned} &\nabla_{\vec{s}_k} L(\vec{s}_1, \dots, \vec{s}_m) \\ =&\nabla_{\vec{s}_k}\left(-\sum_{j=1}^{n}\sum_{i=1}^{m}(\|\vec{s}_i - \vec{r}_j\| - D_{ij})^2\right) \\ =& -2\sum_{j=1}^{n}\left((1 - \frac{D_{kj}}{\|\vec{s}_k - \vec{r}_j\|})(\vec{s}_k - \vec{r}_j)\right) \end{aligned}$$

The real sensor locations are

$$\begin{bmatrix} 176.4052346 & 40.01572084 \\ 97.87379841 & 224.08931992 \\ 186.75579901 & -97.72778799 \\ 95.00884175 & -15.13572083 \\ -10.32188518 & 41.05985019 \\ 14.40435712 & 145.4273507 \\ 76.10377251 & 12.16750165 \end{bmatrix}$$

The predicted sensor locations are

$$\begin{bmatrix} 176.28223286 & 39.76550241 \\ 97.73185793 & 224.12127842 \\ 186.68461118 & -97.59207958 \\ 94.90639305 & -15.16741152 \\ -10.17511568 & 40.93691838 \\ 14.19497946 & 145.42410633 \\ 76.11863016 & 12.24807699 \end{bmatrix}$$

The MSE for Case 1 is 110281.73834969159

The MSE for Case 2 is 19498516.75994162

The MSE for Case 2 (if we knew mu is [300,300]) is 220.62874213060118

My result here shows that the estimated sensor positions are good at predicting the objects from the same distribution but not a new distribution. However, if we know the mean of the new distribution we can start from there and still find a good estimate.

This is due to the fact that our error in estimating object location come from two parts: the error in estimating sensor positions and the error in measuring object distances. If the objects are drawn from the same distribution, we can reduce the fist kind of error, which is similar to test our model on training set. If the objects are drawn from a different distribution, our model will perform worse due to the first kind of error, which is like testing our model on test set. Having a starting point at the mean almost guarantees to find the global minimum. However, my friend got different results by repeating the random starting for only 5 times instead of 10 times. Here are her results:

The MSE for Case 1 is 86.3261605932419

The MSE for Case 2 is 19225374.10947336

The MSE for Case 2 (if we knew mu is [300,300]) is 211.64034337274097

It's interesting to see that MSE is much lower for Case 1. I think it is because in the latter replicates, we somehow started at a bad point which might be caused by the random number generator of the Gaussian distribution. This starting point is so bad that it drives us to some local minima that are further away from the object positions.

Question 5. Backpropogation 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 $\vec{a}$ in the neural network is given by

$$l_i(\vec{a}) = \sigma(W_i\vec{a} + \vec{b}_i).$$

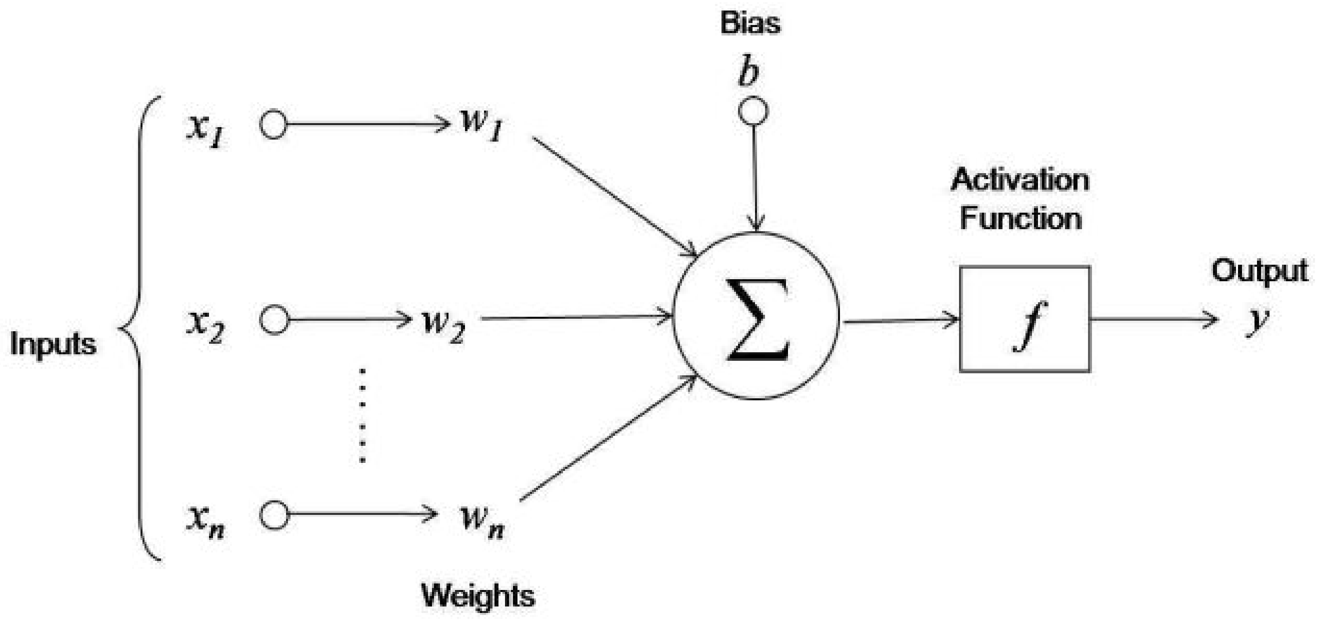
In this equation, W is a $n_{i+1} \times m_i$ matrix that maps the input $\vec{a}_i$ of dimension m_i to a vector of dimension n_{i+1}, where n_{i+1} is the size of layer $i+1$ and we have that $m_i = n_{i-1}$. The vector $\vec{b}_i$ is the bias vector added after the matrix multiplication, and σ is the nonlinear function applied element-wise to the result of the matrix multiplication and addition. Each layer is computed sequentially where the output of one layer is used as the input to the next. To compute the derivatives of the weights W_i and the biases $\vec{b}_i$ 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.

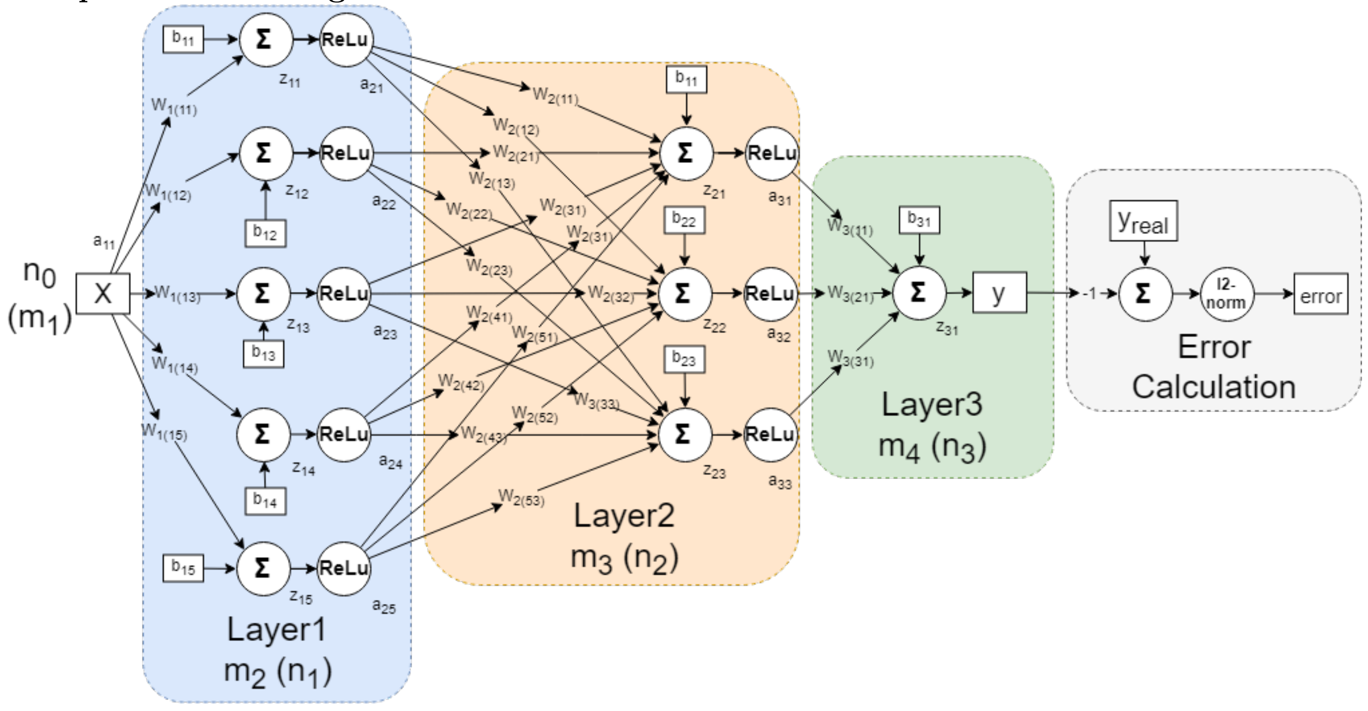
(a) **Start by drawing a small example network with three computational layers, where the last layer has a single scalar output. The first layer should have a single external input corresponding to the input x. The computational layers should have widths of 5, 3, and 1 respectively. The final "output" 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 as and W_i and b 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.

I used draw.io to draw my diagram. I will follow the following example:



I literally spent 1 hour trying to get this silly graph. But here your are, if that's what the question is asking for.



(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

$$\mathbf{MSE}(\hat{\vec{y}}) = \frac{1}{2}\sum_{i=1}^{n}||y_i - \hat{y}_i||_2^2$$

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 $\hat{\vec{y}}$. In `backprop.py` **implement the functions** `QuadraticCost.fx` **and** `QuadraticCost.dx`

$$\begin{aligned}
\nabla_{\hat{\vec{y}}}\mathbf{MSE}(\hat{\vec{y}}) &= \nabla_{\hat{\vec{y}}}\frac{1}{2}\sum_{i=1}^{n}||y_i - \hat{y}_i||_2^2 \\
&= \nabla_{\hat{\vec{y}}}\frac{1}{2}||\hat{\vec{y}} - \vec{y}||_2^2 \\
&= \frac{1}{2}2(\hat{\vec{y}} - \vec{y}) \\
&= \hat{\vec{y}} - \vec{y}
\end{aligned}$$

```
# 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 ((y - yp) ** 2) / 2

# Derivative of the cost function with respect to yp
@staticmethod
def dx(y,yp):
# TODO: PART B ##########################################################################
# Implement me
# ######################################################################################
return yp-y
```

(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

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

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

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

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

$$\nabla_z \sigma_{\textbf{linear}}(z) = \nabla_z z = 1$$

$$\nabla_z \sigma_{\textbf{ReLU}}(z) = \begin{cases} 0 & z < 0 \\ 1 & \textbf{otherwise} \end{cases}$$

$$\begin{aligned} \nabla_z \sigma_{\textbf{tanh}}(z) &= \nabla_z \frac{e^z - e^{-z}}{e^z + e^{-z}} \\ &= \frac{(e^z + e^{-z})(e^z + e^{-z}) - (e^z - e^{-z})(e^z - e^{-z})}{(e^z + e^{-z})^2} \\ &= 1 - (\frac{e^z - e^{-z}}{e^z + e^{-z}})^2 \\ &= 1 - (\sigma_{\textbf{tanh}}(z))^2 \end{aligned}$$

```
# Sigmoid function fully implemented as an example
class SigmoidActivation(object):
@staticmethod
def fx(z):
return 1 / (1 + np.exp(-z))

@staticmethod
def dx(z):
return SigmoidActivation.fx(z) * (1 - SigmoidActivation.fx(z))

# Hyperbolic tangent function
class TanhActivation(object):

# Compute tanh for each element in the input z
@staticmethod
def fx(z):
# TODO: PART C ##################################################################################
# Implement me
# ##########################################################################################
return np.tanh(z)

# Compute the derivative of the tanh function with respect to z
@staticmethod
def dx(z):
# TODO: PART C ###########################################################################
# Implement me
# ##########################################################################################
return 1 - np.tanh(z) ** 2

# Rectified linear unit
class ReLUActivation(object):
@staticmethod
def fx(z):
# TODO: PART C ###########################################################################
# Implement me
# ##########################################################################################
fz = np.zeros(z.shape)
fz[z > 0] = z[z > 0]
return fz.astype('float')

@staticmethod
def dx(z):
# TODO: PART C ###########################################################################
# Implement me
# ##########################################################################################
grad_z = np.zeros(z.shape)
grad_z[z > 0] = 1
return grad_z.astype('float')

# Linear activation
class LinearActivation(object):
@staticmethod
def fx(z):
# TODO: PART C ###########################################################################
# Implement me
# ##########################################################################################
return z

@staticmethod
def dx(z):
# TODO: PART C ###########################################################################
# Implement me
# ##########################################################################################
return np.ones(z.shape)
```

(d) We now need to compute the derivative of the cost function with respect to the weights W and the biases $\vec{b}$ 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 $\frac{\partial \textbf{MSE}}{\partial \vec{a}_{i+1}}$ is given, where $\vec{a}_{i+1}$ is the input to layer $i+1$. Write the expression for $\frac{\partial \textbf{MSE}}{\partial \vec{a}_i}$ in terms of $\frac{\partial \textbf{MSE}}{\partial \vec{a}_{i+1}}$. Then implement these derivative calcualtions in the function `Model.train`. =

$$
\begin{aligned}
&\frac{\partial \text{MSE}}{\partial \vec{a}_i} \\
&= \frac{\partial \vec{a}_{i+1}}{\partial \vec{a}_i} \frac{\partial \text{MSE}}{\partial \vec{a}_{i+1}} \\
&= \frac{\partial l_i(\vec{a}_i)}{\partial \vec{a}_i} \frac{\partial \text{MSE}}{\partial \vec{a}_{i+1}} \\
&= \frac{\partial \sigma(W_i \vec{a}_i + \vec{b}_i)}{\partial \vec{a}_i} \frac{\partial \text{MSE}}{\partial \vec{a}_{i+1}} \\
&= \frac{\partial (W_i \vec{a}_i + \vec{b}_i)}{\partial \vec{a}_i} \frac{\partial \sigma(W_i \vec{a}_i + \vec{b}_i)}{\partial (W_i \vec{a}_i + \vec{b}_i)} \frac{\partial \text{MSE}}{\partial \vec{a}_{i+1}} \\
&= W_i^\top \frac{\partial \sigma(W_i \vec{a}_i + \vec{b}_i)}{\partial (W_i \vec{a}_i + \vec{b}_i)} \frac{\partial \text{MSE}}{\partial \vec{a}_{i+1}}
\end{aligned}
$$

```
# Computes the gradients at each layer. y is the true labels, yp is the
# predicted labels, and z is a list of the intermediate values in each
# layer. Returns the gradients and the forward pass outputs (per layer).
#
# In particular, we compute dMSE/dz_i. The reasoning behind this is that
# in the update function for the optimizer, we do not give it the z values
# we compute from evaluating the network.
def compute_grad(self, x, y):
# Feed forward, computing outputs of each layer and
# intermediate outputs before the non-linearities
yp, a, z = self.evaluate(x)

# d represents (dMSE / da_i) that you derive in part (e);
#   it is inialized here to be (dMSE / dyp)
d = self.cost.dx(y.T, yp)
grad = []

# Backpropogate the error
for layer, curZ in zip(reversed(self.layers), reversed(z)):
# TODO: PART D #########################################################################
# Compute the gradient of the output of each layer with respect to the error
# grad[i] should correspond with the gradient of the output of layer i
# before the activation is applied (dMSE / dz_i); be sure values are stored
# in the correct ordering!
# #####################################################################################
temp = d * layer.dx(curZ)
grad.insert(0, temp)
d = np.dot(layer.W.T, temp)

return grad, a
```

(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.

The difference between my gradient and the numerical one is as follows:
Debugging gradients..
squared difference of layer 0: 2.7948517185902616e-10
squared difference of layer 1: 1.8434874222551898e-10
squared difference of layer 2: 1.0236215246871991e-10

```
    #### PART F ####
if DEBUG_MODEL:
print('Debugging gradients..')
# Build the model
activation = activations["ReLU"]
model = Model(x.shape[1])
model.addLayer(DenseLayer(10, activation()))
model.addLayer(DenseLayer(10, activation()))
model.addLayer(DenseLayer(1, LinearActivation()))
model.initialize(QuadraticCost())

grad, _ = model.compute_grad(x, y)
n_grad, _ = model.numerical_grad(x, y)
for i in range(len(grad)):
print('squared difference of layer %d:' % i, np.linalg.norm(grad[i] - n_grad[i]))
```

(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 $\frac{\partial \mathbf{MSE}}{\partial \vec{a}_i}$ and the outputs of each layer $\vec{a}_i$ to compute the derivates $\frac{\partial \mathbf{MSE}}{\partial W_i}$ and $\frac{\partial \mathbf{MSE}}{\partial \vec{b}_i}$. Use the learning rate η, given by `self.eta` in the function, to scale the gradients when using them to update the model parameters.

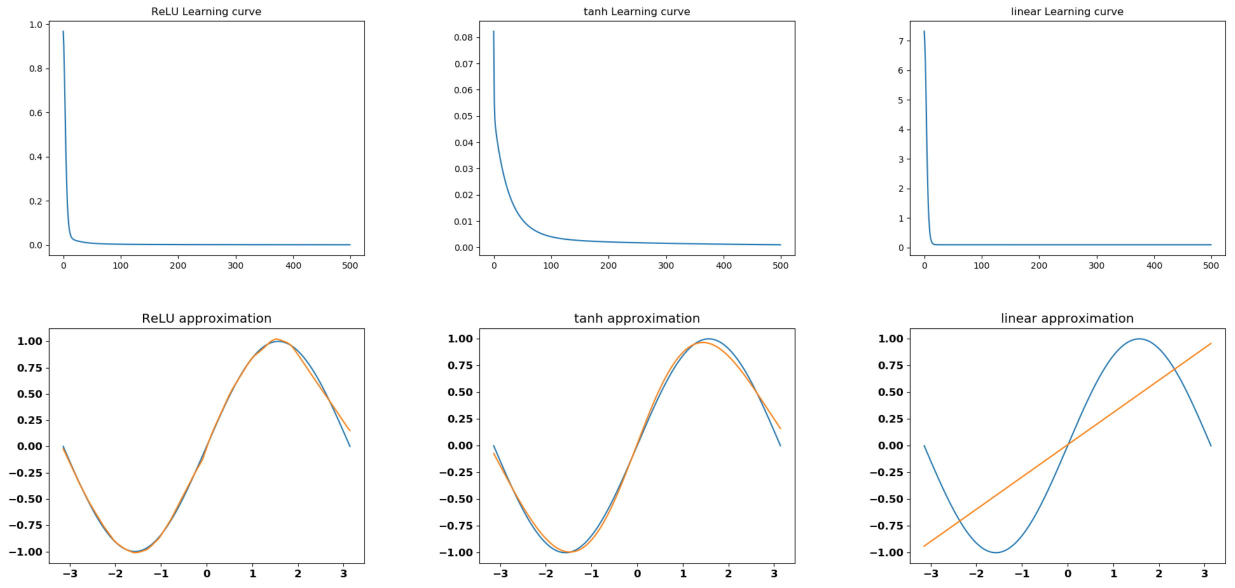
$$\begin{aligned}
&\frac{\partial \text{MSE}}{\partial W_i} \\
&= \frac{\partial \text{MSE}}{\partial \vec{z}_i}\frac{\partial \vec{z}_i}{\partial W_i} \\
&= \frac{\partial \text{MSE}}{\partial \vec{z}_i}\frac{\partial (W_i\vec{a}_i + \vec{b}_i)}{\partial W_i} \\
&= \frac{\partial \text{MSE}}{\partial \vec{z}_i}\vec{a}_i^\top
\end{aligned}$$

$$\begin{aligned}
&\frac{\partial \text{MSE}}{\partial \vec{b}_i} \\
&= \frac{\partial \text{MSE}}{\partial \vec{z}_i}\frac{\partial (W_i\vec{a}_i + \vec{b}_i)}{\partial \vec{b}_i} \\
&= \frac{\partial \text{MSE}}{\partial \vec{z}_i}
\end{aligned}$$

```
# 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 activation
# 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 training examples input
# (in parallel) to the network.
#
# It may help to think about how you would calculate the update if we input just one
# training example at a time; then compute a mean over these individual update values.
# ######################################################################################
dW = - self.eta / m * curGrad.dot(curA. T)
dBias = - self.eta / m * np.sum(curGrad, axis=1).reshape(layer.b.shape)
layer.updateWeights(dW)
layer.updateBias(dBias)
```

(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`.

```
Standard fully connected network
ReLU MSE: 0.000408666485428
linear MSE: 0.0967346963549
tanh MSE: 0.000991349026287
```



```
#### PART G ####
if BASE_MODEL:
print('\n----------------------------------------\n')
print('Standard fully connected network')
for key in names:
# Build the model
activation = activations[key]
model = Model(x.shape[1])
model.addLayer(DenseLayer(100,activation()))
model.addLayer(DenseLayer(100,activation()))
model.addLayer(DenseLayer(1,LinearActivation()))
model.initialize(QuadraticCost())

# Train the model and display the results
hist = model.train(x,y,500,GDOptimizer(eta=lr[key]))
yHat = model.predict(x)
yHats[key] = model.predict(xLin)
error = np.mean(np.square(yHat - y))/2
print(key+' MSE: '+str(error))
plt.figure()
plt.plot(hist)
plt.title(key+' Learning curve')
# plt.show()
plt.savefig('Figure_2g_'+key+'-Learning curve.png')
plt.close()
```

```
# Plot the approximations
font = {'family' : 'DejaVu Sans',
'weight' : 'bold',
'size'   : 12}
matplotlib.rc('font', **font)
y = np.sin(xLin)
for key in activations:
plt.figure()
plt.plot(xLin,y)
plt.plot(xLin,yHats[key])
plt.title(key+' approximation')
# plt.savefig(key+'-approx.png')
# plt.show()
plt.savefig('Figure_2g_'+key+'-approx.png')
plt.close()
```

(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.

We can draw the following conclusions from this simulation:

1. More hidden layers $\Rightarrow$ More accurate approximation.

2. More nodes in a layer $\nRightarrow$ More accurate approximation. (Error can increase in some cases)

3. The choice of activation function doesn't really matter because sometimes tanh performs better but sometimes ReLU does a better job.

Training with various sized network

ReLU MSE Error

Layers	5 nodes	10 nodes	25 nodes	50 nodes
1	0.01457	0.02752	0.00935	0.02050
2	0.04289	0.00637	0.00421	0.00235
3	0.06549	0.00358	0.00088	0.00047

tanh MSE Error

Layers	5 nodes	10 nodes	25 nodes	50 nodes
1	0.03041	0.01341	0.00441	0.00426
2	0.01726	0.00971	0.00406	0.00194
3	0.02124	0.00476	0.00247	0.00041

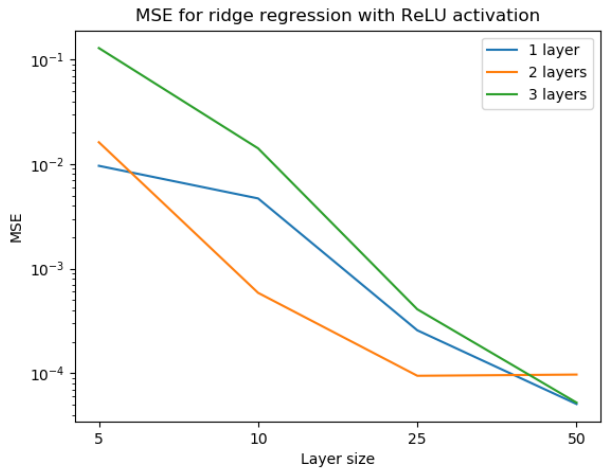
```
   # Train with different sized networks
#### PART H ####
if DIFF_SIZES:
print('\n----------------------------------------\n')
print('Training with various sized network')
names = ['ReLU', 'tanh']
sizes = [5, 10, 25, 50]
widths = [1, 2, 3]
errors = {}
y = np.sin(x)
for key in names:
error = []
for width in widths:
for size in sizes:
activation = activations[key]
model = Model(x.shape[1])
for _ in range(width):
model.addLayer(DenseLayer(size, activation()))
model.addLayer(DenseLayer(1, LinearActivation()))
model.initialize(QuadraticCost())
hist = model.train(x, y, 500, GDOptimizer(eta=lr[key]))
yHat = model.predict(x)
yHats[key] = model.predict(xLin)
e = np.mean(np.square(yHat - y)) / 2
error.append(e)
errors[key] = np.asarray(error).reshape((len(widths), len(sizes)))

# Print the results

for key in names:
```

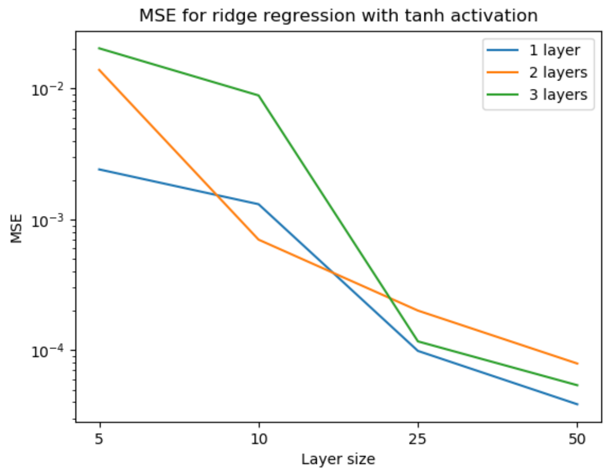
```
error = errors[key]
print(key + ' MSE Error \par')
print(r'\begin{tabu} to 0.9\textwidth { |X[c] |X[c] |X[c] |X[c] |X[c] | }')
print(r'\hline')
header = '{:^8}'
for _ in range(len(sizes)):
header += ' {:^8}'
header += ' {:^8}'
headerText = ['Layers'] + ['& ' + str(s) + ' nodes' for s in sizes] + [' \\\\']
print(header.format(*headerText))
for width, row in zip(widths, error):
text = '{:>8}'
for _ in range(len(row)):
text += ' {:<8}'
text += ' {:<8}'
rowText = [str(width)] + ['& ' + '{0:.5f}'.format(r) for r in row] + [' \\\\']
print(r'\hline')
print(text.format(*rowText))
print(r'\hline')
print(r'\end{tabu}\par')
print(r'\hfill \par')
```

(i) Implement 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.

Without updating the weights, all graphs of MSE look very similar. That is, the chosen of activation function is not important if weights don't get updated.

We see a different pattern here. More hidden layers don't always result in more accurate results; more nodes, however, lead to better results. This might be because the number of data points is more important to get a good approximation in this case.





Running ridge regression on last layer

ReLU MSE Error

Layers	5 nodes	10 nodes	25 nodes	50 nodes
1	0.00963	0.00470	0.00026	0.00005
2	0.01619	0.00059	0.00009	0.00010
3	0.12888	0.01415	0.00041	0.00005

tanh MSE Error

Layers	5 nodes	10 nodes	25 nodes	50 nodes
1	0.00241	0.00131	0.00010	0.00004
2	0.01389	0.00070	0.00020	0.00008
3	0.02032	0.00886	0.00012	0.00005

(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.

Training with SGD
ReLU MSE Error

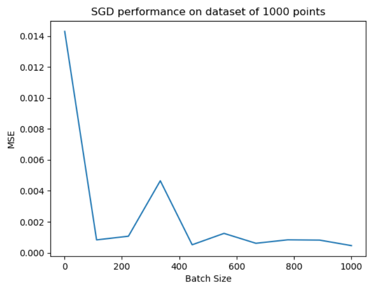
$$\begin{bmatrix} 0.01430215 & 0.00083005 & 0.00106798 & 0.00465038 & 0.00051436 & 0.0012502 \\ 0.00061091 & 0.00083514 & 0.00081652 & 0.00045935 & & \end{bmatrix}$$

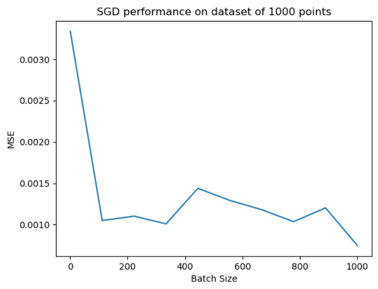
linear MSE Error

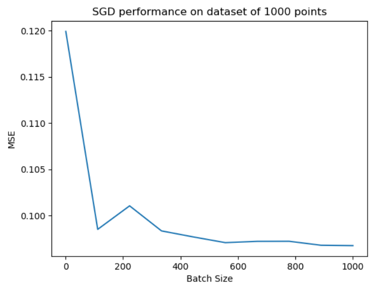
$$\begin{bmatrix} 0.11992036 & 0.09849684 & 0.10104001 & 0.09832828 & 0.09767383 & 0.09705695 \\ 0.09719791 & 0.0972049 & 0.09677514 & 0.0967347 & & \end{bmatrix}$$

tanh MSE Error

$$\begin{bmatrix} 0.00333721 & 0.00104971 & 0.00110245 & 0.00100891 & 0.00143913 & 0.00129275 \\ 0.00118075 & 0.00103572 & 0.00120268 & 0.00074603 & & \end{bmatrix}$$







As we can see, gradient descent can successfully predict the correct result if the batch size is large enough. After a certain threshold of batch size, the error is relatively low. However, this might not be stable and by chance it might perform badly sometimes.

```
# Train the network given the inputs x and the corresponding observations y
# The network should be trained for numEpochs iterations using the supplied
# optimizer
# FOR SGD only
def trainSGD(self, x, y, numEpochs, optimizer, batchsize):
# Initialize some stuff
n = x.shape[0]
assert batchsize <= n
x = x.copy()
y = y.copy()
hist = []
optimizer.initialize(self.layers)

# Run for the specified number of epochs
for epoch in range(0, numEpochs):
# Compute the gradients
sample_idxes = np.random.choice(x.shape[0], batchsize, replace=False)
x_sample = x[sample_idxes, :]
```

```
y_sample = y[sample_idxes, :]
grad, a = self.compute_grad(x_sample, y_sample)

# Update the network weights
optimizer.update(self.layers, grad, a)

# Compute the error at the end of the epoch
yh = self.predict(x)
C = self.cost.fx(y, yh)
C = np.mean(C)
hist.append(C)
return hist


#### BONUS PART J ####
if SGD:
# Test for SGD... Implement!
print('\n----------------------------------------\n')
print('Training with SGD \par')
batch_size = np.linspace(1, x.shape[0], 10, dtype=int)
for key in names:
key_errors = np.zeros_like(batch_size, dtype=float)
# Build the model
print(key + ' MSE Error \par')
print('\hfill \linebreak')
for i, batch in enumerate(batch_size):
activation = activations[key]
model = Model(x.shape[1])
model.addLayer(DenseLayer(100, activation()))
model.addLayer(DenseLayer(100, activation()))
model.addLayer(DenseLayer(1, LinearActivation()))
model.initialize(QuadraticCost())

# Train the model and display the results
hist = model.trainSGD(x, y, 500, GDOptimizer(eta=lr[key]), batch)
yHat = model.predict(x)
yHats[key] = model.predict(xLin)
error = np.mean(np.square(yHat - y)) / 2
# print(key + ' MSE: ' + str(error))
key_errors[i] = error
print(bmatrix(key_errors))

plt.figure()
plt.plot(batch_size, key_errors)
plt.title('SGD performance on dataset of '+str(x.shape[0])+' points')
plt.xlabel('Batch Size')
plt.ylabel('MSE')
plt.savefig('Figure_5j_' + key + '.png')
plt.close()
```

(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.

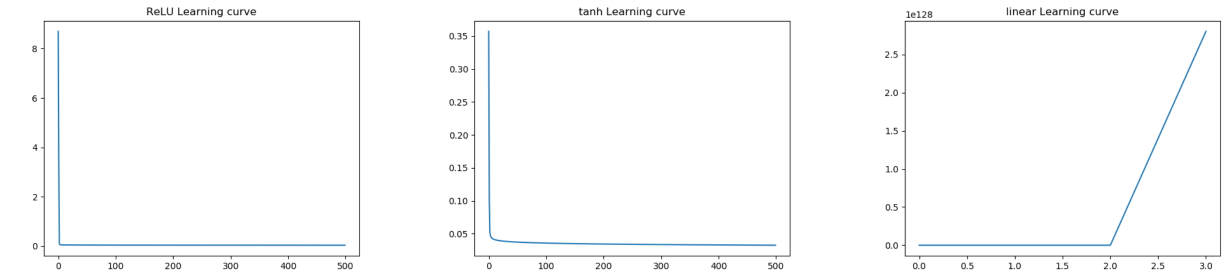
It doesn't converge if I set the learning rate to be $\frac{1}{\text{iteration}}$ for 500 iterations so I changed it to $\frac{0.1}{\text{iteration}}$ and get the following results:

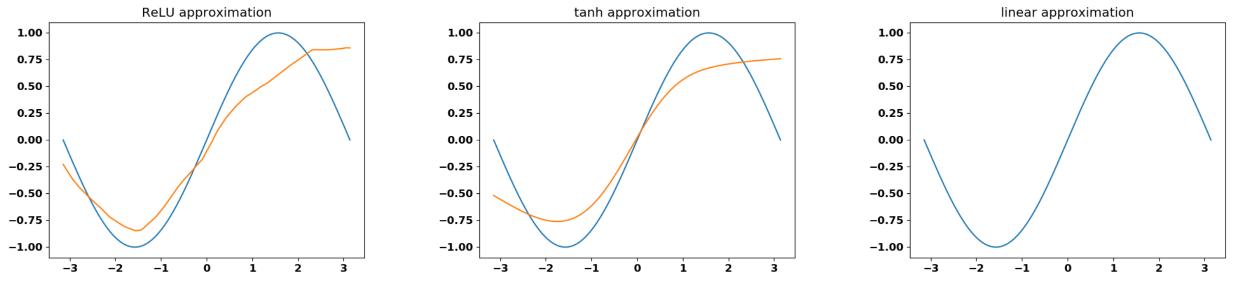
Standard fully connected network with adaptive learning rates

ReLU MSE: 0.03902823766065979

linear MSE: nan

tanh MSE: 0.03258046567801145





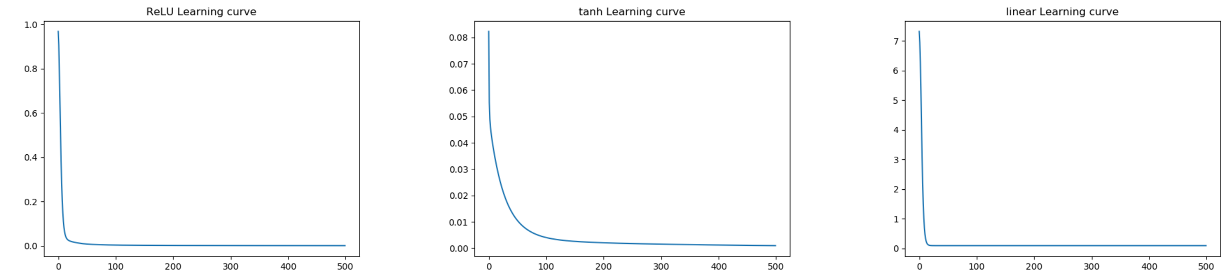
We can combine this answer to what we have in part (g):

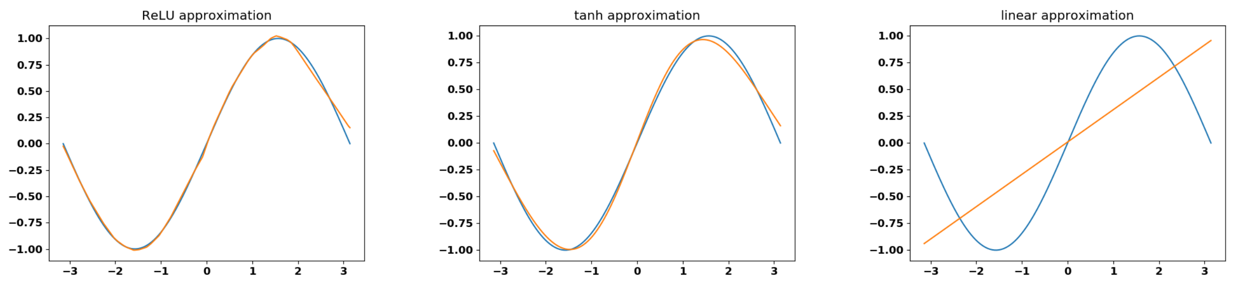
Standard fully connected network

ReLU MSE: 0.000408666485428

linear MSE: 0.0967346963549

tanh MSE: 0.000991349026287



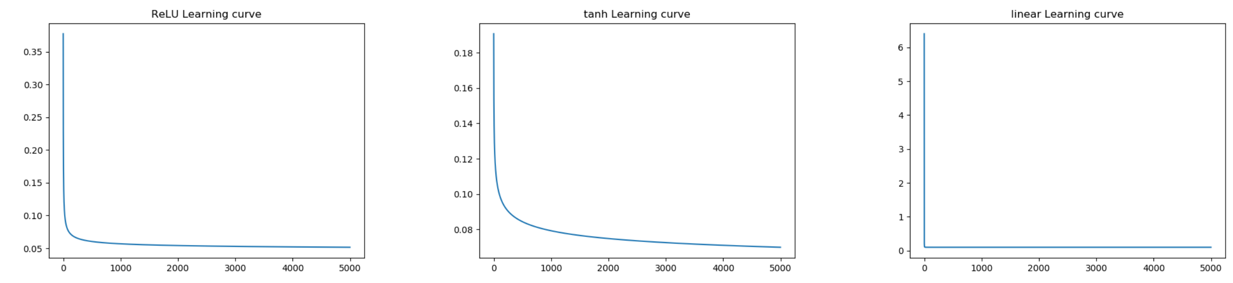


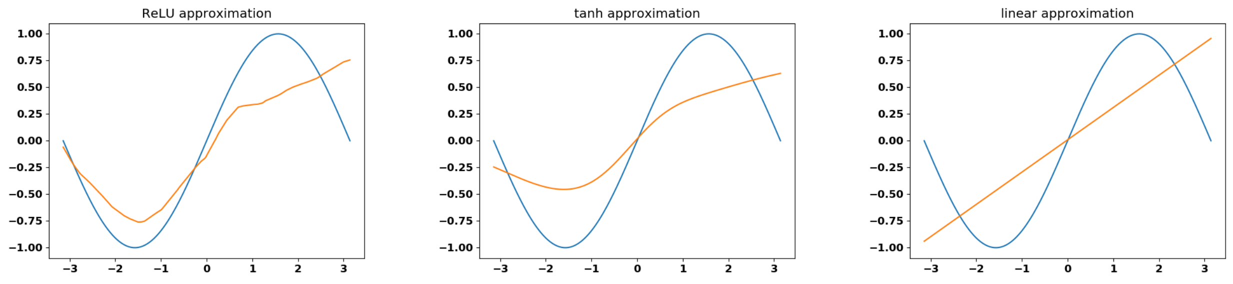
As we can see above, the adaptive learning rates actually perform worse than the fixed learning rates for the same number of iterations. In addition, it's not very stable in terms of convergence as we can see in the linear approximation case. Let's do another experiment with even smaller adaptive learning rates $\frac{0.01}{\text{iteration}}$ and run for 5000 iterations:

ReLU MSE: 0.051606301655336535

linear MSE: 0.09673478835944274

tanh MSE: 0.06992730092794452





Surprisingly, the approximation is still really bad even if I have run it for 5000 iterations this time. The adaptive learning rate that works really well in a simple function doesn't seem to perform very good in neural network.

```
# Gradient descent optimization
# The learning rate is specified by eta
class GDOptimizerAdaptive(object):
def __init__(self, eta):
self.eta = eta
self.iter = 0


def initialize(self, layers):
pass


def resetIter(self):
self.iter = 0


# 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 activation
# 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 training examples input
# (in parallel) to the network.
#
# It may help to think about how you would calculate the update if we input just one
# training example at a time; then compute a mean over these individual update values.
# #####################################################################################
#print(str(self.iter) + 'iteration: ' + str(self.eta(self.iter)))
dW = - self.eta(self.iter) / m * curGrad.dot(curA.T)
dBias = - self.eta(self.iter) / m * np.sum(curGrad, axis=1).reshape(layer.b.shape)
layer.updateWeights(dW)
layer.updateBias(dBias)
self.iter += 1


# Train the network given the inputs x and the corresponding observations y
# The network should be trained for numEpochs iterations using the supplied
# optimizer
def trainAdaptive(self, x, y, numEpochs, optimizer):

# Initialize some stuff
n = x.shape[0]
x = x.copy()
y = y.copy()
hist = []
optimizer.initialize(self.layers)

# Run for the specified number of epochs
for epoch in range(0, numEpochs):
# Compute the gradients
grad, a = self.compute_grad(x, y)

# Update the network weights
optimizer.update(self.layers, grad, a)

# Compute the error at the end of the epoch
yh = self.predict(x)
C = self.cost.fx(y, yh)
C = np.mean(C)
hist.append(C)
return hist



#### PART K ####
if DIFF_RATES:
# lr = dict(ReLU=0.02, tanh=0.02, linear=0.005)
lrFunc = lambda it: 0.01/(it+1)
print('\n----------------------------------------\n')
print('Standard fully connected network with adaptive learning rates')
for key in names:
# Build the model
```

```
activation = activations[key]
model = Model(x.shape[1])
model.addLayer(DenseLayer(100, activation()))
model.addLayer(DenseLayer(100, activation()))
model.addLayer(DenseLayer(1, LinearActivation()))
model.initialize(QuadraticCost())

# Train the model and display the results
hist = model.trainAdaptive(x, y, 5000, GDOptimizerAdaptive(lrFunc))
yHat = model.predict(x)
yHats[key] = model.predict(xLin)
error = np.mean(np.square(yHat - y)) / 2
print(key + ' MSE: ' + str(error))
plt.figure()
plt.plot(hist)
plt.title(key + ' Learning curve')
# plt.show()
plt.savefig('Figure_5k-2_' + key + '-Learning curve.png')
plt.close()

# Plot the approximations
font = {'family': 'DejaVu Sans',
'weight': 'bold',
'size': 12}
matplotlib.rc('font', **font)
y = np.sin(xLin)
for key in activations:
plt.figure()
plt.plot(xLin, y)
plt.plot(xLin, yHats[key])
plt.title(key + ' approximation')
# plt.savefig(key+'-approx.png')
# plt.show()
plt.savefig('Figure_5k-2_' + key + '-approx.png')
plt.close()
```

```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import random as random


# Gradient descent optimization
# The learning rate is specified by eta
class GDOptimizerAdaptive(object):
def __init__(self, eta):
self.eta = eta
self.iter = 0


def initialize(self, layers):
pass


def resetIter(self):
self.iter = 0


# 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 activation
# 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 training examples input
# (in parallel) to the network.
#
# It may help to think about how you would calculate the update if we input just one
# training example at a time; then compute a mean over these individual update values.
# ######################################################################################
#print(str(self.iter) + 'iteration: ' + str(self.eta(self.iter)))
dW = - self.eta(self.iter) / m * curGrad.dot(curA.T)
dBias = - self.eta(self.iter) / m * np.sum(curGrad, axis=1).reshape(layer.b.shape)
layer.updateWeights(dW)
layer.updateBias(dBias)
self.iter += 1


# 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 activation
# 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 training examples input
# (in parallel) to the network.
#
# It may help to think about how you would calculate the update if we input just one
# training example at a time; then compute a mean over these individual update values.
# ######################################################################################
dW = - self.eta / m * curGrad.dot(curA.T)
dBias = - self.eta / m * np.sum(curGrad, axis=1).reshape(layer.b.shape)
layer.updateWeights(dW)
layer.updateBias(dBias)


# 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 ((y - yp) ** 2) / 2

# Derivative of the cost function with respect to yp
@staticmethod
def dx(y, yp):
# TODO: PART B #########################################################################
# Implement me
# ######################################################################################
return yp - y


# Sigmoid function fully implemented as an example
class SigmoidActivation(object):
@staticmethod
def fx(z):
return 1 / (1 + np.exp(-z))

@staticmethod
def dx(z):
return SigmoidActivation.fx(z) * (1 - SigmoidActivation.fx(z))


# Hyperbolic tangent function
class TanhActivation(object):
# Compute tanh for each element in the input z
@staticmethod
def fx(z):
# TODO: PART C ##################################################################################
# Implement me
# ######################################################################################
return np.tanh(z)

# Compute the derivative of the tanh function with respect to z
```

```
@staticmethod
def dx(z):
# TODO: PART C ##########################################################################
# Implement me
# ######################################################################################
return 1 - np.tanh(z) ** 2


# Rectified linear unit
class ReLUActivation(object):
@staticmethod
def fx(z):
# TODO: PART C ##########################################################################
# Implement me
# ######################################################################################
fz = np.zeros(z.shape, dtype=float)
fz[z > 0] = z[z > 0]
return fz

@staticmethod
def dx(z):
# TODO: PART C ##########################################################################
# Implement me
# ######################################################################################
grad_z = np.zeros(z.shape, dtype=float)
grad_z[z > 0] = 1.0
return grad_z


# Linear activation
class LinearActivation(object):
@staticmethod
def fx(z):
# TODO: PART C ##########################################################################
# Implement me
# ######################################################################################
return z

@staticmethod
def dx(z):
# TODO: PART C ##########################################################################
# Implement me
# ######################################################################################
return np.ones(z.shape, dtype=float)


# This class represents a single hidden or output layer in the neural network
class DenseLayer(object):
# numNodes: number of hidden units in the layer
# activation: the activation function to use in this layer
def __init__(self, numNodes, activation):
self.numNodes = numNodes
self.activation = activation

def getNumNodes(self):
return self.numNodes

# Initialize the weight matrix of this layer based on the size of the matrix W
def initialize(self, fanIn, scale=1.0):
s = scale * np.sqrt(6.0 / (self.numNodes + fanIn))
self.W = np.random.normal(0, s,
(self.numNodes, fanIn))
self.b = np.random.uniform(-1, 1, (self.numNodes, 1))
```

```
# Apply the activation function of the layer on the input z
def a(self, z):
return self.activation.fx(z)

# Compute the linear part of the layer
# The input a is an n x k matrix where n is the number of samples
# and k is the dimension of the previous layer (or the input to the network)
def z(self, a):
return self.W.dot(a) + self.b # Note, this is implemented where we assume a is k x n

# Compute the derivative of the layer's activation function with respect to z
# where z is the output of the above function.
# This derivative does not contain the derivative of the matrix multiplication
# in the layer. That part is computed below in the model class.
def dx(self, z):
return self.activation.dx(z)

# Update the weights of the layer by adding dW to the weights
def updateWeights(self, dW):
self.W = self.W + dW

# Update the bias of the layer by adding db to the bias
def updateBias(self, db):
self.b = self.b + db


# This class handles stacking layers together to form the completed neural network
class Model(object):
# inputSize: the dimension of the inputs that go into the network
def __init__(self, inputSize):
self.layers = []
self.inputSize = inputSize

# Add a layer to the end of the network
def addLayer(self, layer):
self.layers.append(layer)

# Get the output size of the layer at the given index
def getLayerSize(self, index):
if index >= len(self.layers):
return self.layers[-1].getNumNodes()
elif index < 0:
return self.inputSize
else:
return self.layers[index].getNumNodes()

# Initialize the weights of all of the layers in the network and set the cost
# function to use for optimization
def initialize(self, cost, initializeLayers=True):
self.cost = cost
if initializeLayers:
for i in range(0, len(self.layers)):
if i == len(self.layers) - 1:
self.layers[i].initialize(self.getLayerSize(i - 1))
else:
self.layers[i].initialize(self.getLayerSize(i - 1))

# Compute the output of the network given some input a
# The matrix a has shape n x k where n is the number of samples and
# k is the dimension
# This function returns
# yp - the output of the network
# a - a list of inputs for each layer of the newtork where
#     a[i] is the input to layer i
#     (note this does not include the network output!)
# z - a list of values for each layer after evaluating layer.z(a) but
#     before evaluating the nonlinear function for the layer
def evaluate(self, x):
curA = x.T
a = [curA]
z = []
for layer in self.layers:
z.append(layer.z(curA))
curA = layer.a(z[-1])
a.append(curA)
yp = a.pop()
return yp, a, z

# Compute the output of the network given some input a
# The matrix a has shape n x k where n is the number of samples and
# k is the dimension
def predict(self, a):
a, _, _ = self.evaluate(a)
return a.T

# Computes the gradients at each layer. y is the true labels, yp is the
# predicted labels, and z is a list of the intermediate values in each
# layer. Returns the gradients and the forward pass outputs (per layer).
#
# In particular, we compute dMSE/dz_i. The reasoning behind this is that
# in the update function for the optimizer, we do not give it the z values
# we compute from evaluating the network.
def compute_grad(self, x, y):
# Feed forward, computing outputs of each layer and
# intermediate outputs before the non-linearities
yp, a, z = self.evaluate(x)

# d represents (dMSE / da_i) that you derive in part (e);
#   it is inialized here to be (dMSE / dyp)
d = self.cost.dx(y.T, yp)
grad = []

# Backpropogate the error
for layer, curZ in zip(reversed(self.layers), reversed(z)):
# TODO: PART D ########################################################################
# Compute the gradient of the output of each layer with respect to the error
# grad[i] should correspond with the gradient of the output of layer i
# before the activation is applied (dMSE / dz_i); be sure values are stored
# in the correct ordering!
# #####################################################################################
temp = d * layer.dx(curZ)
grad.insert(0, temp)
d = np.dot(layer.W.T, temp)

return grad, a

# Computes the gradients at each layer. y is the true labels, yp is the
# predicted labels, and z is a list of the intermediate values in each
# layer. Uses numerical derivatives to solve rather than symbolic derivatives.
# Returns the gradients and the forward pass outputs (per layer).
#
# In particular, we compute dMSE/dz_i. The reasoning behind this is that
# in the update function for the optimizer, we do not give it the z values
# we compute from evaluating the network.
def numerical_grad(self, x, y, delta=1e-4):
```

```
# computes the loss function output when starting from the ith layer
# and inputting z_i
def compute_cost_from_layer(layer_i, z_i):
cost = self.layers[layer_i].a(z_i)
for layer in self.layers[layer_i + 1:]:
cost = layer.a(layer.z(cost))
return self.cost.fx(y.T, cost)

# numerically computes the gradient of the error with respect to z_i
def compute_grad_from_layer(layer_i, inp):
mask = np.zeros(self.layers[layer_i].b.shape)
grad_z = []
# iterate to compute gradient of each variable in z_i, one at a time
for i in range(mask.shape[0]):
mask[i] = 1
delta_p_output = compute_cost_from_layer(layer_i, inp + mask * delta)
delta_n_output = compute_cost_from_layer(layer_i, inp - mask * delta)
grad_z.append((delta_p_output - delta_n_output) / (2 * delta))
mask[i] = 0

return np.vstack(grad_z)

_, a, _ = self.evaluate(x)

grad = []
i = 0
curA = x.T
for layer in self.layers:
curA = layer.z(curA)
grad.append(compute_grad_from_layer(i, curA))
curA = layer.a(curA)
i += 1

return grad, a

# Train the network given the inputs x and the corresponding observations y
# The network should be trained for numEpochs iterations using the supplied
# optimizer
def train(self, x, y, numEpochs, optimizer):

# Initialize some stuff
n = x.shape[0]
x = x.copy()
y = y.copy()
hist = []
optimizer.initialize(self.layers)

# Run for the specified number of epochs
for epoch in range(0, numEpochs):
# Compute the gradients
grad, a = self.compute_grad(x, y)

# Update the network weights
optimizer.update(self.layers, grad, a)

# Compute the error at the end of the epoch
yh = self.predict(x)
C = self.cost.fx(y, yh)
C = np.mean(C)
hist.append(C)
return hist
```

```
# Train the network given the inputs x and the corresponding observations y
# The network should be trained for numEpochs iterations using the supplied
# optimizer
# FOR SGD only
def trainSGD(self, x, y, numEpochs, optimizer, batchsize):
# Initialize some stuff
n = x.shape[0]
assert batchsize <= n
x = x.copy()
y = y.copy()
hist = []
optimizer.initialize(self.layers)

# Run for the specified number of epochs
for epoch in range(0, numEpochs):
# Compute the gradients
sample_idxes = np.random.choice(x.shape[0], batchsize, replace=False)
x_sample = x[sample_idxes, :]
y_sample = y[sample_idxes, :]
grad, a = self.compute_grad(x_sample, y_sample)

# Update the network weights
optimizer.update(self.layers, grad, a)

# Compute the error at the end of the epoch
yh = self.predict(x)
C = self.cost.fx(y, yh)
C = np.mean(C)
hist.append(C)
return hist


# Train the network given the inputs x and the corresponding observations y
# The network should be trained for numEpochs iterations using the supplied
# optimizer
def trainAdaptive(self, x, y, numEpochs, optimizer):

# Initialize some stuff
n = x.shape[0]
x = x.copy()
y = y.copy()
hist = []
optimizer.initialize(self.layers)

# Run for the specified number of epochs
for epoch in range(0, numEpochs):
# Compute the gradients
grad, a = self.compute_grad(x, y)

# Update the network weights
optimizer.update(self.layers, grad, a)

# Compute the error at the end of the epoch
yh = self.predict(x)
C = self.cost.fx(y, yh)
C = np.mean(C)
hist.append(C)
return hist


def bmatrix(a):
"""Returns a LaTeX bmatrix
Retrieved from https://stackoverflow.com/questions/17129290/numpy-2d-and-1d-array-to-latex-bmatrix
```

```
:a: numpy array
:returns: LaTeX bmatrix as a string
"""
if len(a.shape) > 2:
raise ValueError('bmatrix can at most display two dimensions')
lines = str(a).replace('[', '').replace(']', '').splitlines()
rv = [r'\[']
rv += [r'\begin{bmatrix}']
rv += [' ' + ' & '.join(l.split()) + r'\\' for l in lines]
rv += [r'\end{bmatrix}']
rv += [r'\]\par']
return '\n'.join(rv)


if __name__ == '__main__':
# switch these statements to True to run the code for the corresponding parts
# PART E
DEBUG_MODEL = False
# Part G
BASE_MODEL = False
# Part H
DIFF_SIZES = False
# Part I
RIDGE = False
# Part J
SGD = False
# Part K
DIFF_RATES = True

# Generate the training set
np.random.seed(9001)
x = np.random.uniform(-np.pi, np.pi, (1000, 1))
y = np.sin(x)
xLin = np.linspace(-np.pi, np.pi, 250).reshape((-1, 1))
yHats = {}

activations = dict(ReLU=ReLUActivation,
tanh=TanhActivation,
linear=LinearActivation)
lr = dict(ReLU=0.02, tanh=0.02, linear=0.005)
names = ['ReLU', 'linear', 'tanh']

#### PART F ####
if DEBUG_MODEL:
print('Debugging gradients..')
# Build the model
activation = activations["ReLU"]
model = Model(x.shape[1])
model.addLayer(DenseLayer(10, activation()))
model.addLayer(DenseLayer(10, activation()))
model.addLayer(DenseLayer(1, LinearActivation()))
model.initialize(QuadraticCost())

grad, _ = model.compute_grad(x, y)
n_grad, _ = model.numerical_grad(x, y)
for i in range(len(grad)):
print('squared difference of layer %d:' % i, np.linalg.norm(grad[i] - n_grad[i]))

#### PART G ####
if BASE_MODEL:
print('\n----------------------------------------\n')
print('Standard fully connected network')
for key in names:
```

```
# Build the model
activation = activations[key]
model = Model(x.shape[1])
model.addLayer(DenseLayer(100, activation()))
model.addLayer(DenseLayer(100, activation()))
model.addLayer(DenseLayer(1, LinearActivation()))
model.initialize(QuadraticCost())

# Train the model and display the results
hist = model.train(x, y, 500, GDOptimizer(eta=lr[key]))
yHat = model.predict(x)
yHats[key] = model.predict(xLin)
error = np.mean(np.square(yHat - y)) / 2
print(key + ' MSE: ' + str(error))
plt.figure()
plt.plot(hist)
plt.title(key + ' Learning curve')
# plt.show()
#plt.savefig('Figure_5g_' + key + '-Learning curve.png')
plt.close()

# Plot the approximations
font = {'family': 'DejaVu Sans',
'weight': 'bold',
'size': 12}
matplotlib.rc('font', **font)
y = np.sin(xLin)
for key in activations:
plt.figure()
plt.plot(xLin, y)
plt.plot(xLin, yHats[key])
plt.title(key + ' approximation')
# plt.savefig(key+'-approx.png')
# plt.show()
#plt.savefig('Figure_5g_' + key + '-approx.png')
plt.close()

# Train with different sized networks
#### PART H ####
if DIFF_SIZES:
print('\n----------------------------------------\n')
print('Training with various sized network \par')
names = ['ReLU', 'tanh']
sizes = [5, 10, 25, 50]
widths = [1, 2, 3]
errors = {}
y = np.sin(x)
for key in names:
error = []
for width in widths:
for size in sizes:
activation = activations[key]
model = Model(x.shape[1])
for _ in range(width):
model.addLayer(DenseLayer(size, activation()))
model.addLayer(DenseLayer(1, LinearActivation()))
model.initialize(QuadraticCost())
hist = model.train(x, y, 500, GDOptimizer(eta=lr[key]))
yHat = model.predict(x)
yHats[key] = model.predict(xLin)
e = np.mean(np.square(yHat - y)) / 2
error.append(e)
errors[key] = np.asarray(error).reshape((len(widths), len(sizes)))
```

```
# Print the results

for key in names:
error = errors[key]
print(key + ' MSE Error \par')
print(r'\begin{tabu} to 0.9\textwidth { |X[c] |X[c] |X[c] |X[c] |X[c] | }')
print(r'\hline')
header = '{:^8}'
for _ in range(len(sizes)):
header += ' {:^8}'
header += ' {:^8}'
headerText = ['Layers'] + ['& ' + str(s) + ' nodes' for s in sizes] + [' \\\\']
print(header.format(*headerText))
for width, row in zip(widths, error):
text = '{:>8}'
for _ in range(len(row)):
text += ' {:<8}'
text += ' {:<8}'
rowText = [str(width)] + ['& ' + '{0:.5f}'.format(r) for r in row] + [' \\\\']
print(r'\hline')
print(text.format(*rowText))
print(r'\hline')
print(r'\end{tabu}\par')
print(r'\hfill \par')

# Perform ridge regression on the last layer of the network
#### PART I ####
if RIDGE:
print('\n----------------------------------------\n')
print('Running ridge regression on last layer \par')
from sklearn.linear_model import Ridge

errors = {}
for key in names:
error = []
sizes = [5, 10, 25, 50]
widths = [1, 2, 3]
for width in widths:
for size in sizes:
activation = activations[key]
model = Model(x.shape[1])
for _ in range(width):
model.addLayer(DenseLayer(size, activation()))
model.initialize(QuadraticCost())
ridge = Ridge(alpha=0.1)
X = model.predict(x)
ridge.fit(X, y)
yHat = ridge.predict(X)
e = np.mean(np.square(yHat - y)) / 2
error.append(e)
errors[key] = np.asarray(error).reshape((len(widths), len(sizes)))

# Print the results
for key in names:
error = errors[key]
print(key + ' MSE Error \par')
print(r'\begin{tabu} to 0.9\textwidth { |X[c] |X[c] |X[c] |X[c] |X[c] | }')
print(r'\hline')
header = '{:^8}'
for _ in range(len(sizes)):
header += ' {:^8}'
header += ' {:^8}'
```

```
headerText = ['Layers'] + ['& ' + str(s) + ' nodes' for s in sizes] + [' \\\\']
print(header.format(*headerText))
for width, row in zip(widths, error):
text = '{:>8}'
for _ in range(len(row)):
text += ' {:<8}'
text += ' {:<8}'
rowText = [str(width)] + ['& ' + '{0:.5f}'.format(r) for r in row] + [' \\\\']
print(r'\hline')
print(text.format(*rowText))
print(r'\hline')
print(r'\end{tabu}\par')
print(r'\hfill \par')

# Plot the results
for key in names:
error = errors[key]
plt.figure()
for width, row in zip(widths, error):
layer = ' layers'
if width == 1:
layer = ' layer'
plt.semilogy(row, label=str(width) + layer)
plt.title('MSE for ridge regression with ' + key + ' activation')
plt.xticks(range(len(sizes)), sizes)
plt.xlabel('Layer size')
plt.ylabel('MSE')
plt.legend()
# plt.savefig(key+'-ridge.png')
# plt.show()
plt.savefig('Figure_5I_' + key + '-ridge.png')
plt.close()

#### BONUS PART J ####
if SGD:
# Test for SGD... Implement!
print('\n---------------------------------------\n')
print('Training with SGD \par')
batch_size = np.linspace(1, x.shape[0], 10, dtype=int)
for key in names:
key_errors = np.zeros_like(batch_size, dtype=float)
# Build the model
print(key + ' MSE Error \par')
print('\hfill \linebreak')
for i, batch in enumerate(batch_size):
activation = activations[key]
model = Model(x.shape[1])
model.addLayer(DenseLayer(100, activation()))
model.addLayer(DenseLayer(100, activation()))
model.addLayer(DenseLayer(1, LinearActivation()))
model.initialize(QuadraticCost())

# Train the model and display the results
hist = model.trainSGD(x, y, 500, GDOptimizer(eta=lr[key]), batch)
yHat = model.predict(x)
yHats[key] = model.predict(xLin)
error = np.mean(np.square(yHat - y)) / 2
# print(key + ' MSE: ' + str(error))
key_errors[i] = error
print(bmatrix(key_errors))

plt.figure()
plt.plot(batch_size, key_errors)
plt.title('SGD performance on dataset of '+str(x.shape[0])+' points')
plt.xlabel('Batch Size')
plt.ylabel('MSE')
plt.savefig('Figure_5j_' + key + '.png')
plt.close()


#### PART K ####
if DIFF_RATES:
# lr = dict(ReLU=0.02, tanh=0.02, linear=0.005)
lrFunc = lambda it: 0.01/(it+1)
print('\n-----------------------------------------\n')
print('Standard fully connected network with adaptive learning rates')
for key in names:
# Build the model
activation = activations[key]
model = Model(x.shape[1])
model.addLayer(DenseLayer(100, activation()))
model.addLayer(DenseLayer(100, activation()))
model.addLayer(DenseLayer(1, LinearActivation()))
model.initialize(QuadraticCost())

# Train the model and display the results
hist = model.trainAdaptive(x, y, 5000, GDOptimizerAdaptive(lrFunc))
yHat = model.predict(x)
yHats[key] = model.predict(xLin)
error = np.mean(np.square(yHat - y)) / 2
print(key + ' MSE: ' + str(error))
plt.figure()
plt.plot(hist)
plt.title(key + ' Learning curve')
# plt.show()
plt.savefig('Figure_5k-2_' + key + '-Learning curve.png')
plt.close()

# Plot the approximations
font = {'family': 'DejaVu Sans',
'weight': 'bold',
'size': 12}
matplotlib.rc('font', **font)
y = np.sin(xLin)
for key in activations:
plt.figure()
plt.plot(xLin, y)
plt.plot(xLin, yHats[key])
plt.title(key + ' approximation')
# plt.savefig(key+'-approx.png')
# plt.show()
plt.savefig('Figure_5k-2_' + key + '-approx.png')
plt.close()
```

Question 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 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.

How does the chain rule of matrix calculus work? What's the dimension of each component?

To answer this question, we should look at a simple example, where we compute the derivative of an arbitrary function $F(X) : \mathbb{R}^{a\times b} \to \mathbb{R}^{c\times d}$, with respect to the input is $X \in \mathbb{R}^{a\times b}$. We can draw $F(X)$ out:

$$F(X) = \begin{bmatrix} F_{11}(X) & F_{12}(X) & \dots & F_{1d}(X) \\ F_{21}(X) & F_{22}(X) & \dots & F_{2d}(X) \\ \vdots & \vdots & \vdots & \vdots \\ F_{c1}(X) & F_{c2}(X) & \dots & F_{cd}(X) \end{bmatrix}$$

$$\text{where,} \quad F_{ij} \text{ is a function of } \mathbb{R}^{a\times b} \to \mathbb{R}$$

Now we take the derivative of $F(X)$ that:

$$\frac{\partial F(X)}{\partial X} = \begin{bmatrix} \frac{\partial F(X)}{\partial X} & \frac{\partial F_{12}(X)}{\partial X} & \dots & \frac{\partial F_{1d}(X)}{\partial X} \\ \frac{\partial F_{21}(X)}{\partial X} & \frac{\partial F_{22}(X)}{\partial X} & \dots & \frac{\partial F_{2d}(X)}{\partial X} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial F_{c1}(X)}{\partial X} & \frac{\partial F_{c2}(X)}{\partial X} & \dots & \frac{\partial F_{cd}(X)}{\partial X} \end{bmatrix}$$

The derivative now becomes much larger which has a dimension of $\frac{\partial F(X)}{\partial X} \in \mathbb{R}^{(ac)\times(bd)}$. This is not quite the same as what we have seen in backprop. Let's look at more typical examples in neural networks.

Example 1:

$$F(X) = \sigma(WX + B) \text{ where } \sigma(\bullet) \text{ is a nonlinear function apply to EACH element of } X$$
$$X \in \mathbb{R}^{a\times b} \quad W \in \mathbb{R}^{c\times a} \quad B \in \mathbb{R}^{c\times b} \quad F(X) \in \mathbb{R}^{c\times b} \quad F : \mathbb{R}^{a\times b} \to \mathbb{R}^{c\times b}$$

$$\begin{aligned} \frac{\partial F(X)}{\partial X} &= \frac{\partial \sigma(WX + B)}{\partial (WX + B)} \frac{\partial (WX + B)}{\partial X} \\ &= \frac{\partial \sigma(WX + B)}{\partial (WX + B)} \end{aligned}$$

Because $\sigma(\bullet)$ keeps its input's dimensions, $\frac{\partial\sigma(\bullet)}{\partial(\bullet)}$ should be the same dimension as well. The derivatives here is performed element-wise. Now let's look the dimensions:

$$\begin{aligned}\frac{\partial\sigma(WX+B)}{\partial(WX+B)} &\in \mathbb{R}^{c\times b}\\ W^\top &\in \mathbb{R}^{a\times c}\end{aligned}$$

Therefore, the above matrix multiplication must be rearranged to be:

$$\frac{\partial F(X)}{\partial X} = W^\top \frac{\partial\sigma(WX+B)}{\partial(WX+B)} \qquad \frac{\partial F(X)}{\partial X} \in \mathbb{R}^{a\times b}$$

Let's make sure the product of the derivative and X yields the same dimension as $F(X)$:

$$\begin{aligned}\frac{\partial F(X)}{\partial X} &\in \mathbb{R}^{a\times b}\\ X &\in \mathbb{R}^{a\times b}\end{aligned}$$

We can see that their dimension don't match! Why? Recall that $\sigma(\bullet)$ keeps its input's dimensions so when we multiple the derivative and the input, we should do element-wise multiplication rather than matrix multiplication. As a summary, when we do backward propagation, we should write the derivatives in backward order as well. Example 2:

$$\begin{aligned}\frac{\partial F(W)}{\partial W} &= \frac{\partial\sigma(WX+B)}{\partial(WX+B)}\frac{\partial(WX+B)}{\partial W}\\ &= \frac{\partial\sigma(WX+B)}{\partial(WX+B)}X^\top\end{aligned}$$

$$\begin{aligned}\frac{\partial\sigma(WX+B)}{\partial(WX+B)} &\in \mathbb{R}^{c\times b}\\ X^\top &\in \mathbb{R}^{b\times a}\end{aligned}$$

As we can see here, there is no need for rearrangement any more.