CS 189 Introduction to Machine Learning
Spring 2018 HW7

1 Getting Started

Read through this page carefully. You may typeset your homework in latex or submit neatly handwritten/scanned solutions. Please start each question on a new page. Deliverables:

1. Submit a PDF of your writeup, **with an appendix for your code**, to assignment on Gradescope, "HW7 Write-Up". If there are graphs, include those graphs in the correct sections. Do not simply reference your appendix.
2. If there is code, submit all code needed to reproduce your results, "HW7 Code".
3. If there is a test set, submit your test set evaluation results, "HW7 Test Set".

After you've submitted your homework, 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?

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

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

2 Step Size in Gradient Descent

By this point in the class, we know that gradient descent is a powerful tool for moving towards local minima of general functions. We also know that local minima of convex functions are global minima. In this problem, we will look at the convex function $f(x) = ||x - b||_2$. 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 the rate at which the different step sizes decrease and draw some conclusions about the rate of convergence. Notice that we want to make sure the we get to some local minimum and we want to do it as quickly as possible.

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

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

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

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

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

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

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

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

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

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

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

(e) 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.

3 Convergence Rate of Gradient Descent

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Show that

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

and conclude that

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

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

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

(f) **Show that**

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

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

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

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

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

4 Sensors, Objects, and Localization

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

First, the setup. Let us say there are m sensors and n objects located in a two dimensional plane. The m sensors are located at the points $(a_1, b_1), \ldots, (a_m, b_m)$. The n objects are located at the points $(x_1, y_1), \ldots, (x_n, y_n)$. We have measurements for the distances between the sensors and the

objects: D_{ij} is the measured distance from sensor i to object j. The distance measurement has noise in it. Specifically, we model

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

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

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

For this problem, all Python libraries are permitted.

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

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

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

(b) The provided code generates

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Use the provided code to generate

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

- Use the same σ as before, i.e. $\sigma = 100$.

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

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

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

5 Backpropagation Algorithm for Neural Networks

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

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

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

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

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

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

(a) **Start by drawing a small example network with three computational layers, where the last layer has a single scalar output.** The first layer should have a single external input

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

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

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

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

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

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

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

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

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

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

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

(d) We have implemented the forward propagation part of the network for you (see `Model.evaluate` in the code). We now need to compute the derivative of the cost function with respect to the

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

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

(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.**

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

(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`.

(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.**

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

(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.**

(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/it-

eration. 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.**

6 Your Own Question

Write your own question, and provide a thorough solution.

Writing your own problems is a very important way to really learn the material. The famous "Bloom's Taxonomy" that lists the levels of learning is: Remember, Understand, Apply, Analyze, Evaluate, and Create. Using what you know to create is the top-level. We rarely ask you any HW questions about the lowest level of straight-up remembering, expecting you to be able to do that yourself. (e.g. make yourself flashcards) But we don't want the same to be true about the highest level.

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

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