EECS 182 Deep Neural Networks Fall 2025 Anant Sahai and Gireeja Ranade Homework 2

This homework is due on Friday Sep 19 2025, at 10:59PM.

1. Optimizers as Penalized Linear Improvement with different norm penalties

In lecture, you saw the locally linear perspective of a neural network and the loss by Taylor expanding the loss around the current value of the parameters. This approximation is only very good in a near neighborhood of those values. One way to proceed with optimization is to consider the size of the neighborhood as a hyperparameter and to bound our update to stay within that neighborhood while minimizing our linear approximation to the loss. You saw in lecture that the choice of norm in defining that neighborhood also matters.

In this problem, you will work out for yourself a slightly different perspective. Instead of treating the norm as a constraint (with the size of the acceptable norm as a hyperparameter), we can do an unconstrained optimization with a weighted penalty that corresponds to the squared norm — where that weight is a hyperparameter.

At each iteration, we wish to maximize linear improvement of the objective (as defined by the dot-product between the gradient and the update) locally regularized by a penalty on the size of the update. This can be expressed (in traditional minimization form) as:

$$u = \underset{\Delta\theta}{\operatorname{argmin}} \quad \underbrace{g^T\Delta\theta}_{\text{Linear Improvement}} \quad + \quad \frac{1}{\alpha} \quad \underbrace{d(\Delta\theta)}_{\text{Distance Penalty}} \quad , \tag{1}$$

where $g = \nabla f(\theta)$ is the gradient of the loss, α is a scalar, and d is a scalar-output distance function $\mathbb{R}^{dim(\theta)} \to \mathbb{R}^+$.

(a) Let's assume *Euclidean distance* is the norm that captures our sense of relevant neighborhoods in parameter space. Then our objective can be:

$$u = \underset{\Delta\theta}{\operatorname{argmin}} \; g^T\Delta\theta \; + \frac{1}{\alpha}\|\Delta\theta\|_2^2. \tag{2}$$

What is the analytical solution for u in the above problem? What standard optimizer does this recover?

Solution: To solve for u, we can take the gradient of the objective with respect to $\Delta\theta$ and set it to zero:

$$\nabla_{\Delta\theta}\left(g^T\Delta\theta + \frac{1}{\alpha}\|\Delta\theta\|_2^2\right) = g + \frac{2}{\alpha}\Delta\theta = 0. \tag{3}$$

Then, solving for $\Delta\theta$, we get:

$$\Delta\theta = -\frac{\alpha}{2} g, \tag{4}$$

which is simply a scaled version of the negative gradient. This solution is equivalent to standard gradient descent with a step size of $\frac{\alpha}{2}$. SGD or GD can be understood as maximizing linear improvement subject to a Euclidean distance penalty over parameters.

This can also be viewed equivalently, if we split out the directions and magnitude, as:

$$\Delta\theta = -\frac{\alpha}{2}\|g\|_2 \frac{g}{\|g\|_2}, \tag{5}$$

where the $\frac{g}{\|g\|_2}$ defines the unit vector (in 2-norm) in the direction we want, and the $\frac{\alpha}{2}\|g\|_2$ tells the magnitude we want to move in that direction. Notice that this magnitude scales with the size (in 2-norm) of the gradient vector.

(b) Now, consider an alternative way of capturing local neighborhood size – the squared infinity norm over parameters. Recall that this is defined as $\|x\|_\infty = \max_i |x_i|$. Our objective is now:

$$u = \underset{\Delta\theta}{\operatorname{argmin}}\ g^T\Delta\theta\ + \frac{1}{\alpha}\|\Delta\theta\|_\infty^2. \tag{6}$$

What is the analytical solution for u in this case? Which optimizer does this correspond to?

Solution: To solve for u, we can rewrite the objective as:

$$u = \underset{\Delta\theta}{\operatorname{argmin}}\ g^T\Delta\theta + \frac{1}{\alpha}\max_i |\Delta\theta_i|^2. \tag{7}$$

Let $M = \max_i |\Delta\theta_i|$. Then, we can express the objective as:

$$u = \underset{\Delta\theta, M}{\operatorname{argmin}}\ g^T\Delta\theta + \frac{1}{\alpha}M^2, \quad \text{subject to } |\Delta\theta_i| \le M \text{ for all } i. \tag{8}$$

To minimize the objective, we want to make $g^T\Delta\theta$ as negative as possible while keeping M small. The optimal solution occurs when $|\Delta\theta_i| = M$ for all i, and $\Delta\theta_i$ has the opposite sign of g_i. Thus,

$$\Delta\theta_i = -M \cdot \text{sign}(g_i). \tag{9}$$

From this we know the solution is the sign of the gradient, scaled by a constant $-M$. To solve for M, substituting this back into the objective, we get:

$$u = -Mg^T\text{sign}(g) + \frac{1}{\alpha}M^2 = -M\|g\|_1 + \frac{1}{\alpha}M^2. \tag{10}$$

To minimize this with respect to M, we set the derivative to zero:

$$\nabla_M(-M\|g\|_1 + \frac{1}{\alpha}M^2) = -\|g\|_1 + \frac{2}{\alpha}M = 0 \tag{11}$$

$$M = \frac{\alpha}{2}\|g\|_1. \tag{12}$$

Therefore, the solution for u is:

$$u_i = -\frac{\alpha}{2}\|g\|_1 \cdot \text{sign}(g). \tag{13}$$

This is taking a step in the direction of the sign of the gradient. Intuitively, this is because the infinity norm penalizes the largest element of the update, and so the optimal solution is to make all elements of the update equal to each other in magnitude. This sign-function picks out the direction that we move to do our update. It is not the same direction as the direction of the gradient.

However, in the update (13), we also have the $\|g\|_1$ scaling. This is different from SignGD. When the gradient is small, we take a small step while when the gradent is large, we take a larger step. Interestingly, we measure the size of the gradent in 1-norm while choosing the magnitude of the motion.

So both solutions have the same basic form. A magnitude that depends on the size of the gradient times a unit-vector pointing based on the gradient. However, in this case, the magnitude is measured using the 1-norm and the unit-vector is a unit-vector in the infinity-norm.

This is a manifestation of a more general pattern where the two norms involved are dual to each other. The 2-norm just happens to be its own dual.

2. Optimizers and their convergence

Consider $\mathcal{O}$: a simplified Adam-style optimizer without weight decay that has iterates

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \alpha_t M_t \nabla f_t(\boldsymbol{\theta}_t) \tag{14}$$

where f_t is the loss at iteration t and α_t is the step size (learning rate).

Further suppose that the adaptive scaling matrix M_t is recomputed over each epoch of training and just consists of a diagonal populated by the inverses of the square roots of the mean squared value for the gradients during the epoch for that specific coordinate.

For this part, we have exactly $n = 1$ training point corresponding to the single equation

$$[1, 0.1, 0.01]\boldsymbol{\theta} = 1 \tag{15}$$

with a 3-dimensional learnable parameters $\boldsymbol{\theta}$. Suppose that we start with $\boldsymbol{\theta}_0 = \mathbf{0}$ and use squared loss $f_t(\boldsymbol{\theta}) = (1 - [1, 0.1, 0.01]\boldsymbol{\theta})^2$.

(a) **What specific vector $\boldsymbol{\theta}$ would standard vanilla SGD (i.e. (14) with $M_t = I$ and $\alpha_t = \alpha$) converge to assuming $\alpha > 0$ was small enough to give convergence?**

Solution: SGD in this case is identical to gradient descent since there is only one training point. To get convergence with a constant step size, we need the gradient itself to get to zero. That happens only when (15) is actually solved. Gradient-descent on squared-loss can only move $\boldsymbol{\theta}$ in the direction of $\begin{bmatrix} 1 \\ 0.1 \\ 0.01 \end{bmatrix}$ and given that we start at $\boldsymbol{\theta} = \mathbf{0}$, this means that we are looking for a solution of the form

$\boldsymbol{\theta} = \beta \begin{bmatrix} 1 \\ 0.1 \\ 0.01 \end{bmatrix}$. Plugging this into (15), we get

$$\beta[1, 0.1, 0.01] \begin{bmatrix} 1 \\ 0.1 \\ 0.01 \end{bmatrix} = 1 \tag{16}$$

which has the solution $\beta = \frac{1}{1.0101}$ giving us the overall answer $\boldsymbol{\theta}^* = \frac{1}{1.0101} \begin{bmatrix} 1 \\ 0.1 \\ 0.01 \end{bmatrix}$.

Notice that this is quite insensitive to the second and third coordinates, and if the test-data is like the training data in having small numbers there, the overall impact of anything there will be quite small indeed.

(b) **What specific vector $\boldsymbol{\theta}$ would the simplified version of Adam $\mathcal{O}$ converge to assuming appropriate step-sizes $\alpha_t > 0$ to give convergence?**

Solution: In this case, we observe that the premultiplication by M_t in the updates (14) will have the effect of moving $\boldsymbol{\theta}$ in the rescaled direction $\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$. Other than that, the same argument applies as the previous part. So we get:

$$\beta[1, 0.1, 0.01] \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = 1 \tag{17}$$

which has the solution $\beta = \frac{1}{1.11}$ giving us the overall answer $\boldsymbol{\theta}_a^* = \frac{1}{1.11} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$.

Note that to get this convergence, we need to use a diminishing step size schedule for this highly simplified version of Adam. A constant step size wouldn't work.

Further notice that this solution $\boldsymbol{\theta}_a^*$ is significantly more sensitive to the second and third coordinates, but the weight on the first coordinate is not that different from the previous SGD solution. Consequently, as long as the test-data is like the training data in having small numbers in the second and third coordinate, by that very nature, they won't impact the final prediction by much.

(c) Consider a learning approach that first did training input feature rescaling (so that each feature had unit second-moment), then ran SGD to convergence, and then converted the solution for the rescaled problem back to the original units. **What specific vector $\boldsymbol{\theta}$ would it give as its final solution (for use in original coordinates)?**

Solution: By rescaling the inputs explicitly, we multiply the second feature by 10 and the third feature by 100 giving us a new "normalized" problem:

$$[1, 1, 1]\widetilde{\boldsymbol{\theta}} = 1 \tag{18}$$

with initial condition $\widetilde{\boldsymbol{\theta}} = \mathbf{0}$. Immediately by symmetry (or by following the same argument above), this has the solution $\widetilde{\boldsymbol{\theta}} = \frac{1}{3}\begin{bmatrix}1\\1\\1\end{bmatrix}$. To get this behavior on the original unscaled features, we need to incorporate the scaling we did into the weights. (This should be reminiscent of considerations that we have in dropout training as well as in the use of batch normalization.) This gives us the final answer: $\boldsymbol{\theta}_r^* = \frac{1}{3}\begin{bmatrix}1\\10\\100\end{bmatrix}$

Notice that this is far more sensitive to the second and third coordinates. Even if test data had small numbers like the training data in those positions, they would significantly impact the predictions.

This illustrates the different inductive biases of these three different approaches to solving an underdetermined equation.

3. Coding Question: Initialization and Optimizers

In this question, you'll implement He Initialization and Different Optimizers. You will have the choice between two options:

Use Google Colab (Recommended). Open this url and follow the instructions in the notebook.

Use a local Conda environment. Clone `https://github.com/Berkeley-CS182/cs182fa25_public/tree/main` and refer to `hw02/code/README.md` for further instructions.

(a) What you observe in the mean of gradient norm plot above in the above plots? **Try to give an explanation.**

Solution: Random and He initialization should train as expected. When we use zero initialization, our gradients become zero as well, which makes it very difficult to train our network to obtain nonzero weights.

4. Visualizing features from local linearization of neural nets

In the first discussion, you trained a 1-hidden-layer neural network with SGD and visualized how the network fitted the function leveraging the "elbows" of the non-linear activation function ReLU. In this question, we are going to visualize the effective "features" that correspond to the local linearization of this network in the neighborhood of the parameters.

We provide you with some starter code in the course repo, or you can use Google Colab. For this question, please submit the `.pdf` export of the jupyter notebook when it is completed. In addition, answer the questions below, including plots from the notebook where relevant.

(a) **Visualize the features corresponding to $\frac{\partial}{\partial w_i^{(1)}} y(x)$ and $\frac{\partial}{\partial b_i^{(1)}} y(x)$ where $w_i^{(1)}$ are the first hidden layer's weights and the $b_i^{(1)}$ are the first hidden layer's biases.** These derivatives should be evaluated at at least both the random initialization and the final trained network. When visualizing these features, plot them as a function of the scalar input x, the same way that the notebook plots the constituent "elbow" features that are the outputs of the penultimate layer.

Solution: See notebook.

(b) During training, we can imagine that we have a generalized linear model with a feature matrix corresponding to the linearized features corresponding to each learnable parameter. We know from our analysis of gradient descent, that the singular values and singular vectors corresponding to this feature matrix are important.

Use the SVD of this feature matrix to plot both the singular values and visualize the "principle features" that correspond to the d-dimensional singular vectors multiplied by all the features corresponding to the parameters.

(HINT: Remember that the feature matrix whose SVD you are taking has n rows where each row corresponds to one training point and d columns where each column corresponds to each of the learnable features. Meanwhile, you are going to be plotting/visualizing the "principle features" as functions of x even at places where you don't have training points.)

Solution: See notebook.

(c) Augment the jupyter notebook to add a second hidden layer of the same size as the first hidden layer, fully connected to the first hidden layer. **Allow the visualization of the features corresponding to the parameters in both hidden layers, as well as the "principle features" and the singular values.**

Solution: See notebook.

5. Analyzing Distributed Training

For real-world models trained on lots of data, the training of neural networks is parallelized and accelerated by running workers on distributed resources, such as clusters of GPUs. In this question, we will explore three popular distributed training paradigms:

All-to-All Communication: Each worker maintains a copy of the model parameters (weights) and processes a subset of the training data. After each iteration, each worker communicates with every other worker and updates its local weights by averaging the gradients from all workers.

Parameter Server: A dedicated server, called the parameter server, stores the global model parameters. The workers compute gradients for a subset of the training data and send these gradients to the parameter server. The server then updates the global model parameters and sends the updated weights back to the workers.

Ring All-Reduce: Arranges n workers in a logical ring and updates the model parameters by passing messages in a circular fashion. Each worker computes gradients for a subset of the training data, splits the gradients into n equally sized chunks and sends a chunk of the gradients to their neighbors in the ring. Each worker receives the gradient chunks from its neighbors, updates its local parameters, and passes the updated gradient chunks along the ring. After $n-1$ passes, all gradient chunks have been aggregated across workers, and the aggregated chunks are passed along to all workers in the next $n-1$ steps. This is illustrated in Fig. 1.

	Number of Messages Sent	**Size of each message**
All-to-All	**Solution:** $n(n-1)$	p
Parameter Server	$2n$	**Solution:** p
Ring All-Reduce	$n(2(n-1))$	**Solution:** $\frac{p}{n}$

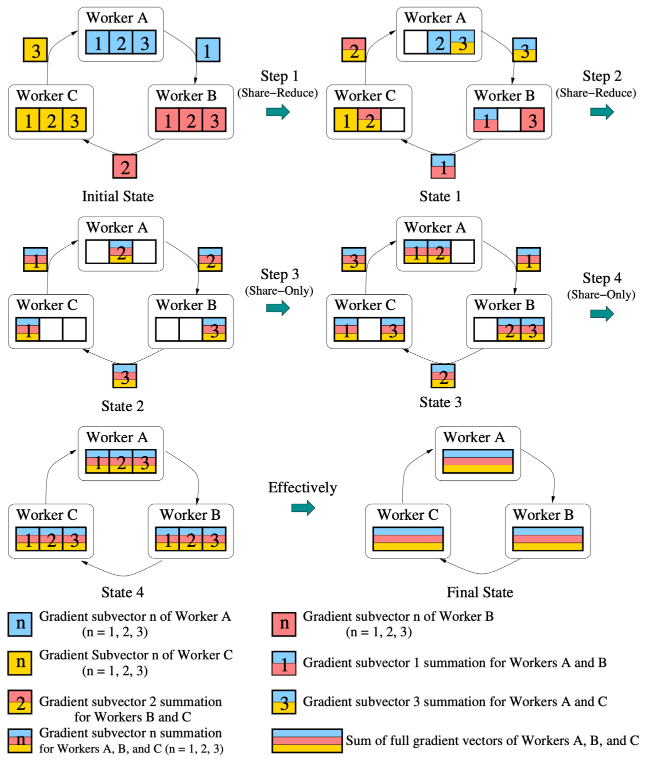
Worker A
Worker B
Worker C
Step 1 (Share–Reduce)
Step 2 (Share–Reduce)
Step 3 (Share–Only)
Step 4 (Share–Only)
Initial State
State 1
State 2
State 3
State 4
Effectively
Final State
Gradient subvector n of Worker A (n = 1, 2, 3)
Gradient subvector n of Worker B (n = 1, 2, 3)
Gradient Subvector n of Worker C (n = 1, 2, 3)
Gradient subvector 1 summation for Workers A and B
Gradient subvector 2 summation for Workers B and C
Gradient subvector 3 summation for Workers A and C
Gradient subvector n summation for Workers A, B, and C (n = 1, 2, 3)
Sum of full gradient vectors of Workers A, B, and C

Figure 1: Example of Ring All-Reduce in a 3 worker setup. Source: Mu Et. al, *GADGET: Online Resource Optimization for Scheduling Ring-All-Reduce Learning Jobs*

For each of the distributed training paradigms, fill in the total number of messages sent and the size of each message. Assume that there are n workers and the model has p parameters, with p divisible by n.

Solution: **All-to-All Communication**: Each worker (there are n of them) needs to exchange messages with every other worker (there are $n-1$ others) to share the computed gradients.

Parameter Server: Each worker communicates with the parameter server to send gradients and receive updated weights. The server needs to handle incoming and outgoing messages from all workers, but workers do not communicate directly with each other.

Notice that the total amount of network traffic now only scales as np instead of n^2p in terms of the order.

Ring All-Reduce: Each worker communicates only with its neighbors in the ring for $2(n-1)$ rounds. We multiply by n because there are n workers. But each message is much smaller since we only share $\frac{p}{n}$ parameters each time. Thus the total amount of network traffic is similar to the parameter server case.

It turns out that it is possible to make ring-all-reduce style approaches also tolerant to faults and nodes that might die or be slow.

6. Optimization Techniques for "Bad" Objective Functions

In this coding question, you will learn about three cool techniques that can help you optimize challenging objective functions that are hard to optimize with vanilla gradient descent.

For the purpose of understanding these techniques, we will focus on local minima in this problem. Note that in general, local optima are not really a practical issue on most supervised learning objectives in modern deep learning. This is because as running stochastic gradient descent with these objective functions on overparameterized (large enough) neural networks often coverge to a global optimum (there are often many optima due to symmetry in neural networks). If you are interested in some theoretical justifications, you might find this paper interesting (Gradient Descent Finds Global Minima of Deep Neural Networks).

However, for objective functions that are less standard (e.g., feedback signal from human) and in reinforcement learning, local minima and other things that manifest similarly can play a huge role. This coding question teaches you some basics on how you might be able to optimize these functions.

In particular, we will be working with the following objective function $f(x, y)$ and the goal is to find the best pair of (x, y) that maximizes the function.

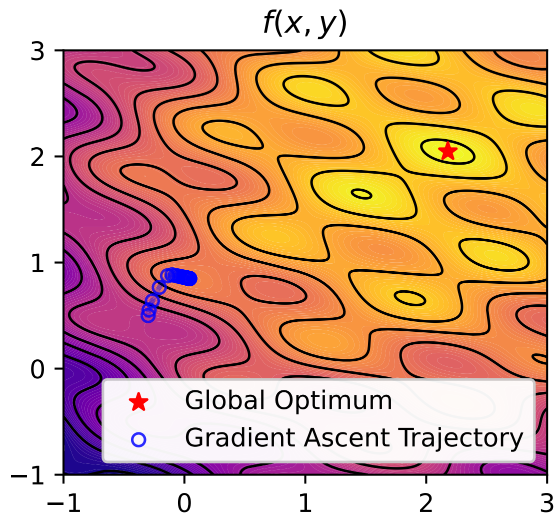

Figure 2: The objective function $f(x, y)$ has many local optima causing the naive gradient ascent approach to find a local maximum far away from the global optimum.

Implement all the TODOs in the Google Colab. **Answer the written questions below.**

(a) (Part 1) **What do you observe from the optimization trajectory of your (x, y) parameters? Where do your parameters converge to?** Try a few different initialization values and see how the convergence changes.

Solution: The parameters converge to a nearby local maximum. The observation stays the same for other initialization values unlessthe parameters are initialized near the global optimum.

(b) (Part 1) **What patterns do you observe from the basins of attraction visualization? How do they relate to the sine and cosine in our objective function?**

Solution: There is a periodically repeating pattern going diagonally. The cos and sin terms in the objective function cause the function to contain local maxima and local minima periodically which is why the basins appear in a periodic pattern. One interesting exercise is to visualize the same thing but for local minima and see how the basins of attraction shift.

(c) (Part 1) **What do you notice when the learning rate η changes? Is there a good learning rate setting that finds the global optimum better? Is there any other way that we can do to modify our gradient descent algorithm to overcome the local optimum issue?**

Solution: As the learning rate increases, the optimization trajectory gets increasingly jerky. It is quite hard to find a good learning rate. Having a learning rate too low will cause it to get trapped in a local maximum. Having a high learning rate may side step the local maximum issue, but make optimization extremely unstable. To overcome the local optimum issue, we may add gradient noise and have an annealed learning rate (starting with high learning rate and gradually lowering it).

(d) (Part 2) Take a close look at the global optimum for the original function (red star) and the global optimum for the smoothed function (blue star). **What do you observe as σ increases? Why? Can you give an example where the global optimum of the smoothed function is very far from the global optimum of the original function?**

Solution: The global optimum for the smoothed function moves further away from the global optimum of the original function when a stronger smoothing kernel is applied (bigger σ). The smoothed function value is an aggregation of the surrounding values of the original function. The global optimum in the smoothed function only tells you (approximately) which area (on average) has the highest value and may ignore the individual spikes. For example, consider a function: $f(x) = -x^2 + \mathbb{I}_{1000-\delta<x<1000+\delta}(x^2+1)$ where δ is a very small positive value. Without the indicator term $\mathbb{I}$, it is a standard quadratic function that has the global maximum to be at $x = 0$. However, with the indicator term, now the maximum of the function is at $x = 1000$ where $f(1000) = 1$. If we apply a smoothing kernel on this function, the resulting function would be nearly identical to $-x^2$ as long as δ is small enough, but the maximum happens for a completely different x value (0 vs. 1000).

(e) (Part 3a) **What do you observe when you change δ, η (learning rate) and N (the sample size for Monte-Carlo estimate)?**

Solution: As N increases and η decreases, the trajectory is less jerky and more smooth. Learning slows down when δ is too large because the local approixmation of the gradient is getting worse.

(f) (Part 3b) **Can you provide an intuitive explanation of why the gradient estimator works?** *(Hint: which direction is the gradient estimator trying to push (x, y) into? By what magnitude?)*

Solution: For each parameter sample $(\tilde{x}, \tilde{y})$, the gradient estimator is pushing (x, y) towards $(\tilde{x}, \tilde{y})$ with a magnitude proportional to how far $(\tilde{x}, \tilde{y})$ from (x, y) and the value of $f(\tilde{x}, \tilde{y})$. Intuitively, we want to move the parmeters (x, y) towards regions where $f(x, y)$ is high. This estimator matches such an intuition.

(g) (Part 3b) **What do you observe when you change η (learning rate) and N (the sample size for Monte-Carlo estimate)? How does it compare to the finite-different method?**

Solution: As N increases and η decreases, the trajectory is less jerky and more smooth. Policy gradient often finds the solution much faster with fewer samples needed (N).

(h) (Part 3c) Compare the trajectory of reparam gradient vs. policy gradient. **What do you observe? Is one more straight than the other one? Why do you think that is the case?**

Solution: Policy gradient can sometimes find the solution faster than reparameterization gradient but reparam. gradient is much more consistent (it can even make consistent progress even when $N = 1$).

(i) (Part 4) **How well do you expect naive gradient descent to perform on this quantized objective function? Explain why.**

Solution: The gradient of f is 0 on the quantized function almost everywhere, so the naive gradient descent algorithm would completely fail.

(j) (Part 4) **Can you use reparameterization gradient for this function? Explain why.**

Solution: The reparameterization gradient method would also fail for the same reason as the previous part because it uses the gradient of f as part of its estimator.

7. Homework Process and Study Group

Citing sources and collaborators are an important part of life, including being a student!
We also want to understand what resources you find helpful and how much time homework is taking, so we can change things in the future if possible.

(a) **What sources (if any) did you use as you worked through the homework?**

(b) **If you worked with someone on this homework, who did you work with?**
List names and student ID's. (In case of homework party, you can also just describe the group.)

(c) **Roughly how many total hours did you work on this homework?**

Contributors:

- Kevin Frans.
- Anant Sahai.
- Luke Jaffe.
- Kevin Li.
- Hao Liu.
- Sheng Shen.
- Andrew Ng.
- Linyuan Gong.
- Romil Bhardwaj.
- Qiyang Li.
- Ryan Scott.
- Ria Doshi.