EECS 189 Introduction to Machine Learning
Fall 2020 HW8

This homework is due **Wednesday, October 28 at 11:59 p.m.**

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 the appropriate assignment on Gradescope. 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.
3. If there is a test set, submit your test set evaluation results.

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 inadvertently cheats.

I certify that all solutions are entirely in my words and that I have not looked at another student's solutions nor have I looked at any online solutions to any of these problems. I have credited all external sources in this write up.

2 Towards understanding Few-shot Learning by Predicting Auxiliary Labels

Make sure to submit the code you write in this problem to "HW8 Code" on Gradescope.

This problem helps us understand the "latent space" view of what learning is about. Here, we explore things in the simplest possible setting, but the idea is far more generally applicable. It is the foundation of ideas like "self-supervision" and "few-shot learning." The practical problem is that in many real world settings, there is not enough labeled data. Or more precisely, there is not enough data that has been labeled with the labels that you care about. How can we hope to get machine learning to succeed in these cases? At the most basic level, the only patterns that we can ever hope to learn are those that are simple enough relative to the amount of data that we have.

We have seen in earlier homeworks that having lots of features or a high dimensional input can be regularizing. However, even in those cases, the examples we gave of where learning succeeds still had some inner structure to those inputs that was approximately low-dimensional. We know that approaches like least-squares and ridge-regression only really care about the singular-value structure of the input feature matrix — they don't care if things have been rotated. And so, if there are only a few big singular values, then those are what define the dimensionality of the input data. If the true pattern that explains the target labels aligns well to those important directions, then learning can succeed. This was made completely explicit with PCA: an approach that allows us to discover those directions and then just use them.

If we ask ourselves what is the story behind PCA, the most natural story involves thinking about a latent space. There is a true low-dimensional space where the truly important (physically significant) parameters are. All the inputs and features that we measure or compute are noisy projections of that underlying latent space.

In a way, PCA is a way of picking out the "loud directions" in the input data. But should "loudness" always be synonymous with "likely to be important" when we do machine learning? Clearly not. We saw in the last homework how assuming sparsity could help us even if the inputs didn't have any explicitly loud directions in them. If we knew that the true pattern is only going to depend on a few of the features, then we could use sparsity-seeking approaches (like OMP, LASSO, or even simple thresholding/pruning to zero) to learn the pattern.

This problem explores a different way of getting at the underlying latent space when simply looking at the loud directions is not enough. The key insight is philosophical: if something is actually real, then it should influence the world in multiple ways. No task is an isolated island — almost certainly, there are going to be other tasks that are related in the sense of depending on the same underlying (physically real) variables. We should be able to use auxiliary labels for presumably related tasks to help us set the stage to do well on our task. How? By using those to figure out what the important directions are in feature space.

Suppose our observed data inputs $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_n]^\top \in \mathbb{R}^{n\times d}$ are generated by hidden data points from the latent space,

$$\mathbf{L} = [\boldsymbol{\ell}_1, \cdots, \boldsymbol{\ell}_n]^\top \in \mathbb{R}^{n\times r}, \quad \text{and} \quad \boldsymbol{\ell} \sim \mathcal{N}(\mathbf{0}, \sigma_\ell^2 \mathbf{I}_{r\times r}).$$

In math, the data matrix can be defined as $\mathbf{X} = \mathbf{L}\mathbf{V}_{ux}^\top$,

$$\mathbf{x}_i = \mathbf{V}_{ux}\boldsymbol{\ell}_i + \boldsymbol{\epsilon}_i, \quad \boldsymbol{\epsilon}_i \sim \mathcal{N}(\mathbf{0}, \sigma_x^2 \mathbf{I}_{d\times d}), \quad i \in [n],$$

where $\boldsymbol{\ell}_i$ is the vector of latent variables associated with the i-th training sample, $\boldsymbol{\ell}_i \in \mathbb{R}^r$, and $\mathbf{V}_{ux} \in \mathbb{R}^{d\times r}$. We assume orthonormality of these vectors that relate the important latent space to the observed inputs, namely $\mathbf{V}_{ux}^\top \mathbf{V}_{ux} = \mathbf{I}_{r\times r}$. Suppose that the underlying true output $\widetilde{\mathbf{y}} = [\widetilde{y}_1, \cdots, \widetilde{y}_n]^\top$ is generated from the latent space:

$$\widetilde{y}_i = \boldsymbol{\ell}_i^\top \boldsymbol{\theta}, \quad i \in [n],$$

where $\boldsymbol{\theta} \in \mathbb{R}^r$ is the ground truth weight vector. The actual observed vector $\mathbf{y}$ might be a noisy version of the underlying true outputs.

Besides the output $\mathbf{y}$, we can also observe k sets of auxiliary labels, $\{\mathbf{z}^1, \cdots, \mathbf{z}^k\}$, where $\mathbf{z}^j = [z_1^j, \cdots, z_n^j]^\top$. For these too, the underlying noise-free versions come from the same latent space:

$$\widetilde{z}_i^j = \boldsymbol{\ell}_i^\top \mathbf{q}_j, \quad i \in [n], \quad j \in [k].$$

This question helps us see the utility of such auxiliary labels for machine learning, even though we do not have access to the auxiliary labels at test time.

(a) If you look at the back-story of the training points above, notice that what we have is n i.i.d. samples of a random d-dimensional vector $\mathbf{x}$ which is distributed according to $\mathbf{x} = \mathbf{V}_{ux}\boldsymbol{\ell} + \boldsymbol{\epsilon}_x$ where $\boldsymbol{\ell} \sim \boldsymbol{\ell} \sim \mathcal{N}(\mathbf{0}, \sigma_\ell^2 \mathbf{I}_{r\times r})$ and the independent $\boldsymbol{\epsilon}_x \sim \mathcal{N}(\mathbf{0}, \sigma_x^2 \mathbf{I}_{d\times d})$. Similarly, the auxiliary k-dimensional labels $\mathbf{z}$ are distributed according to $\mathbf{z} = \mathbf{Q}^\top \boldsymbol{\ell} + \boldsymbol{\epsilon}_z$ where $\boldsymbol{\ell}$ is the same latent vector that helped make $\mathbf{x}$ and $\boldsymbol{\epsilon}_z$ is independent of the other random variables and distributed $\boldsymbol{\epsilon}_z \sim \mathcal{N}(\mathbf{0}, \sigma_z^2 \mathbf{I}_{k\times k})$.

Here, suppose that we had the expectation of $\mathbf{x}\mathbf{z}^\top$, i.e., $\mathbb{E}[\mathbf{x}\mathbf{z}^\top]$. This is a matrix with d rows and k columns. We can compute its full SVD:

$$\widetilde{\mathbf{U}}\widetilde{\boldsymbol{\Sigma}}\widetilde{\mathbf{V}}^\top = \text{SVD}\left(\mathbb{E}\left[\mathbf{x}\mathbf{z}^\top\right]\right).$$

Prove that we can use this to recover $\mathbf{V}_{ux}$ up to rotation,

$$\widehat{\mathbf{T}} = \mathbf{V}_{ux}\mathbf{R},$$

where $\mathbf{R} \in \mathbb{R}^{r\times r}$ and $\mathbf{R}^\top\mathbf{R} = \mathbf{R}\mathbf{R}^\top = \mathbf{I}_{r\times r}$. **Note that here, you need to clearly tell us what matrix $\widehat{\mathbf{T}}$ you are going to use to estimate $\mathbf{V}_{ux}$.**

(Hint: do you think that $\widetilde{U}$ might be involved? $\widetilde{V}$? What are the sizes of these matrices? What size do you need the answer to be?)

(Note: The linear mapping cannot be uniquely determined, because the latent distribution is isotropic, you could always rotate the transformation matrix $\mathbf{V}_{ux}$ and it will give you the same results.)

(b) In reality, we don't actually have access to this expectation $\mathbb{E}[\mathbf{x}\mathbf{z}^\top]$. Instead, we only have a finite number n of training samples to approximate the expectation.

Consider the following matrix $\frac{1}{n}\mathbf{X}^\top\mathbf{Z}$, where

$$\begin{aligned}
&\frac{1}{n}\mathbf{X}^\top\mathbf{Z} \\
&= \frac{1}{n}\mathbf{X}^\top \underbrace{\left[\mathbf{z}^1, \cdots, \mathbf{z}^k\right]}_{\mathbf{Z}} \\
&= \frac{1}{n}\left[\sum_{i=1}^n \mathbf{x}_i z_i^1, \cdots, \sum_{i=1}^n \mathbf{x}_i z_i^k\right]
\end{aligned}$$

$$
\begin{aligned}
&= \frac{1}{n}\left[\sum_{i=1}^{n} \mathbf{V}_{ux}\boldsymbol{\ell}_i(\mathbf{q}_1^\top \boldsymbol{\ell}_i) + \sum_{i=1}^{n} \boldsymbol{\epsilon}_i(\mathbf{q}_1^\top \boldsymbol{\ell}_i), \cdots, \sum_{i=1}^{n} \mathbf{V}_{ux}\boldsymbol{\ell}_i(\mathbf{q}_k^\top \boldsymbol{\ell}_i) + \sum_{i=1}^{n} \boldsymbol{\epsilon}_i(\mathbf{q}_k^\top \boldsymbol{\ell}_i)\right] \\
&= \mathbf{V}_{ux} \underbrace{\frac{1}{n}\left(\sum_{i=1}^{n} \boldsymbol{\ell}_i\boldsymbol{\ell}_i^\top\right)}_{\hat{\boldsymbol{\Sigma}}_n} \underbrace{\left[\mathbf{q}_1, \cdots, \mathbf{q}_k\right]}_{\mathbf{Q}} + \underbrace{\frac{1}{n}\left(\sum_{i=1}^{n} \boldsymbol{\epsilon}_i\boldsymbol{\ell}_i^\top\right)}_{\hat{\mathbf{E}}_n} \left[\mathbf{q}_1, \cdots, \mathbf{q}_k\right] \\
&= \mathbf{V}_{ux}\hat{\boldsymbol{\Sigma}}_n\mathbf{Q} + \hat{\mathbf{E}}_n\mathbf{Q}.
\end{aligned}
$$

What kind of random variables are inside the matrices $\hat{\boldsymbol{\Sigma}}_n$ and $\hat{\mathbf{E}}_n$? Give their means and the dependence of their variances on n. Are they individually going to converge to anything as n grows?

(Note: Here we assume $\boldsymbol{\ell}$ is drawn from an isotropic gaussian distribution, in practice, you may first want to zero-center the data and then perform some appropriate whitening to the data if you want to suppress any "loudness" that might exist in $\mathbf{x}$ *alone.)*

(c) Now, as a reality check, we are going to see how the ideally recovered $\widehat{\mathbf{T}}$ would work if we used it. How can do this reality check? Well, we can just assert that $\widehat{\mathbf{T}} = \mathbf{V}_{ux}\mathbf{R}$ as above and try to solve the following ridge regression problem that forces us to learn parameters that go through what we think that the latents are:

$$\min_{\boldsymbol{\theta}} \|\mathbf{y} - \mathbf{X}\widehat{\mathbf{T}}\boldsymbol{\theta}\|_2^2 + \lambda\|\widehat{\mathbf{T}}\boldsymbol{\theta}\|_2^2.$$

Prove that when $\boldsymbol{\epsilon}_i = \mathbf{0}$, $i \in [n]$ (the no observation noise case), then the solution $\widetilde{\boldsymbol{\theta}}$ to the above problem satisfies

$$\mathbf{X}\widehat{\mathbf{T}}\widetilde{\boldsymbol{\theta}} = \mathbf{L}\underbrace{(\mathbf{L}^\top\mathbf{L} + \lambda\mathbf{I})^{-1}\mathbf{L}^\top\mathbf{y}}_{\hat{\boldsymbol{\theta}}} = \mathbf{L}\hat{\boldsymbol{\theta}},$$

where $\hat{\boldsymbol{\theta}}$ is the solution to $\min_{\boldsymbol{\theta}}\left\{\|\mathbf{y} - \mathbf{L}\boldsymbol{\theta}\|_2^2 + \lambda\|\boldsymbol{\theta}\|_2^2\right\}$**.** In other words, it is as though we had been working directly with the latents themselves.

(Note: Although we can only recover the matrix $\mathbf{V}_{ux}$ up to a rotation matrix, it is equivalent to the case that we have access to the exact $\mathbf{V}_{ux}$ w.r.t. the predictions.)

(d) Next, let us suppose that there are many latent variables but only a smaller subset of them are relevant to both the true labels y as well as the auxiliary labels z^j. We rewrite the latent data matrix as

$$\mathbf{L} = [\mathbf{T}|\mathbf{F}] \in \mathbb{R}^{n\times r}, \quad \text{and} \quad \mathbf{T} \in \mathbb{R}^{n\times t}, \mathbf{F} \in \mathbb{R}^{n\times f},$$

where $\mathbf{T} = [\mathbf{t}_1, \cdots, \mathbf{t}_n]^\top$, $\mathbf{F} = [\mathbf{f}_t, \cdots, \mathbf{f}_n]^\top$, and $r = t + f$. The first t features can be viewed as the "truly relevant" features and so we can define $\mathbf{t}_i$ to be the t-dimensional vector that just has those in it. Assume the outputs (including the auxiliary outputs) are generated as

$$y_i = \mathbf{t}_i^\top\boldsymbol{\theta} \quad i \in [n].$$

and

$$z_i^j = \mathbf{t}_i^\top\mathbf{q}_j + \epsilon_i^j, \quad \epsilon_i^j \sim \mathcal{N}(0, \sigma_z^2), \quad i \in [n], \quad j \in [k].$$

So all the outputs are only related to the first t features in the latent space. From what you've done above, you should see that the SVD strategy should give you access to a transformation that gives you access to those t latent dimensions. This can be exploited to improve our ability to learn $\mathbf{y}$.

Complete the coding part related to Part (a)-(b) and compare its performance with: (1). k-PCA OLS; (2) (realistic best baseline) OLS when $\mathbf{V}_{ux}$ is known in the Jupyter notebook. Note that for visualization purposes, we fix $t = 2$.

1) Based on the previous parts, implement functions to solve truncated SVD and compute $\hat{\theta}$ in the jupyter notebook.

2) Visualize the recovered latent space on the code in the jupyter notebook and *include the plot in your solution***. Compare to the ground-truth latent variable and that recovered by PCA, what do you observe?**

3) Change k, the dimension of auxiliary labels Z. Plot the performance of different baseline methods and *include the plot in your solution***. Comment on how the performance of different methods changes as k increases.**

3 Backpropagation Algorithm for Neural Networks

Make sure to submit the code you write in this problem to "HW8 Code" on Gradescope.

In this problem, we will be implementing the backpropagation algorithm to train a neural network to classify the difference between two handwritten digits (specifically the digits 3 and 9).

Note that we need to install the library mnist for loading data, which has already been provided in the notebook.

The notebook might take a while to run, so please start early.
Our implementation takes about 20 minutes to run all parts in total.

To establish notation for this problem, we define:

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

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

In the jupyter notebook, 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 your nonlinear classifier. The code currently trains a network with one hidden layer with 4 nodes.

(a) **Start by drawing a small example network with one hidden layer, where the last layer has a single scalar output. The first layer should have 784 inputs corresponding to the input image x which consists of a 28×28 pixels. The computational layers should have widths of 16, 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. We will be using a simple least squares classifier. We will regress all the positive classes to 1, and the negative classes to -1. The sign of the predicted value will be the predicted class label. This function is used to assign an error for each prediction made by the network during training.

The error we actually care about is the misclassification error (MCE) which will be:

$$\text{MCE}(\hat{\mathbf{y}}) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{I}\{\text{sign}(y_i) \neq \text{sign}(\hat{y}_i)\}\ .$$

However this function is hard to to optimize so the implementation will be optimizing the mean squared error cost (MSE) function, which is given by

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

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 corresponding part of the notebook, 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:**

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

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

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

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

(d) We have implemented the forward propagation part of the network for you (see `Model.evaluate` in the notebook). 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 $\frac{\partial \mathrm{MSE}}{\partial \mathbf{a}_{i+1}}$ is given, where $\mathbf{a}_{i+1}$ is the input to layer $i+1$. Write the expression for $\frac{\partial \mathrm{MSE}}{\partial \mathbf{a}_i}$ in terms of $\frac{\partial \mathrm{MSE}}{\partial \mathbf{a}_{i+1}}$. Then implement these derivative calcualtions in the function `Model.compute_grad`.** Recall, $\mathbf{a}_{i+1}$ is given by

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

(e) One method to shortcut training is to randomly initialize the weights of the neural network and only train the weights of the last layer. This effectively treats the hidden layers of the neural network as random feature mappings. **Use ridge regression to optimize the weights of the final layer. Compare the resulting approximation mean-squared-error values and misclassification error values for the 8 cases above (2 nonlinearities times 4 widths). Comment on what you see.**

(f) We use gradients to update the model parameters using batched stochastic gradient descent. **Implement the function `GDOptimizer.update` to update the parameters in each layer of the network.** You will need to use the derivatives $\frac{\partial \mathrm{MSE}}{\partial \mathbf{z}_i}$ and the outputs of each layer $\mathbf{a}_i$ to compute the derivatives $\frac{\partial \mathrm{MSE}}{\partial \mathbf{W}_i}$ and $\frac{\partial \mathrm{MSE}}{\partial \mathbf{b}_i}$. Use the learning rate η, given by `self.eta` in the function, to scale the gradients when using them to update the model parameters. **Choose several different batch sizes and number of training epochs. Report the final error on the training set given the various batch sizes and training epochs**

(g) Let's now explore how the number of hidden nodes per layer affects the approximation. **Train a models using the tanh and the ReLU activation functions with** $2, 4, 8, 16$**, and** 32 **hidden nodes per layer (width)**. 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.**

(h) **Optional. Not for credit:** Currently the code classifies between the digits 3 and 9, modify the variables `N0` and `N1` in the code to classify between other pairs of digits. Do your results change if you pick different digits? Do you need more or less hidden nodes to reach the same accuracy? Do you need more or less training epochs?

4 Entropy, KL Divergences, and Cross-Entropy

So far, we have mostly considered so called *Euclidean* spaces, the most prominent example is the vector space $\mathbb{R}^d$ with inner product $\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^\top \mathbf{y}$ and norm $\|\mathbf{x}\|_2^2 = \mathbf{x}^\top \mathbf{x}$. For example in linear regression, we had the loss function

$$f(\boldsymbol{\theta}) = \|\mathbf{X}^\top \boldsymbol{\theta} - \mathbf{y}\|_2^2 = \sum_{i=1}^{n} (\mathbf{x}_i^\top \boldsymbol{\theta} - y_i)^2$$

which uses precisely the Euclidean norm squared to measure the error.

In this problem we will implicitly consider a different geometric structure, which defines a metric on probability distributions. In more advanced courses, this can be developed further to show how probability distributions are naturally imbued with a curved non-Euclidean intrinsic geometry. Here, our goals are more modest — we just want you to better understand the relationship between probability distributions, entropy, KL Divergence, and cross-entropy.

Let $\mathbf{p}$ and $\mathbf{q}$ be two probability distributions, i.e. $p_i \geq 0$, $q_i \geq 0$, $\sum_i p_i = 1$ and $\sum_i q_i = 1$, then we define the Kullback-Leibler divergence

$$\mathrm{KL}(\mathbf{p}, \mathbf{q}) = \sum_i p_i \ln \frac{p_i}{q_i}$$

which is the “distance“ to $\mathbf{p}$ from $\mathbf{q}$. We have $\mathrm{KL}(\mathbf{p}, \mathbf{p}) = 0$ and $\mathrm{KL}(\mathbf{p}, \mathbf{q}) \geq 0$ (the latter by Jensen's inequality) as would be expected from a distance metric. However, $\mathrm{KL}(\mathbf{p}, \mathbf{q}) \neq \mathrm{KL}(\mathbf{q}, \mathbf{p})$ since the KL divergence is not symmetric.

(a) *Entropy motivation:* Let $X_1, X_2, \ldots, X_n$ be independent identically distributed random variables taking values in a finite set $\{0, 1, \ldots, m\}$, i.e. $p_j = P(X_i = j)$ for $j \in \{0, 1, \ldots, m\}$. The *empirical number of occurrences* is then a random vector that we can denote $\mathbf{F}^{(n)}$ where $F_j^{(n)}$ is the number of variables X_i that happen to take a value equal to j.

Intuitively, we can consider coin tosses with $j = 0$ corresponding to heads and $j = 1$ corresponding to tails. Say we do an experiment with $n = 100$ coin tosses, then $F_0^{(100)}$ is the number of heads that came up and $F_1^{(100)}$ is the number of tails.

Recall that the number of configurations of $X_1, X_2, \ldots, X_n$ that have $f^{(n)}$ as their empirical type is $\binom{n}{f_0^{(n)}, f_1^{(n)}, \ldots, f_m^{(n)}}$. Further notice that dividing the empirical type by n yeilds an empirical probability distribution.

Show using the crudest form of Stirling's approximation ($\ell! \approx (\frac{\ell}{e})^\ell$) that this number of configurations $\binom{n}{f_0^{(n)}, f_1^{(n)}, \ldots, f_m^{(n)}}$ is approximately equal to $\exp\big(nH(f^{(n)}/n)\big)$ where the entropy H of a probability distribution is defined as $H(\mathbf{p}) = \sum_{j=0}^{m} p_j \ln \frac{1}{p_j}$.

Note: The multinomial coefficient

$$\binom{n}{f_0^{(n)}, f_1^{(n)}, \ldots, f_m^{(n)}} = \frac{n!}{f_0^{(n)}!\, f_1^{(n)}! \cdots f_m^{(n)}!}$$

is the number of ways to put n interchangeable objects into m boxes so that box j has $f_j^{(n)}$ objects in it. There are that many distinct "n-length strings" of X realizations whose empirical counts match $\mathbf{f}^{(n)}$.

Then use the attached Jupyter notebook to plot your approximation, and verify that it looks reasonable.

(b) *KL divergence motivation:* Recall that the probability of seeing a particular empirical type is given by:

$$P(\mathbf{F}^{(n)} = \mathbf{f}^{(n)}) = \binom{n}{f_0^{(n)}, f_1^{(n)}, \ldots, f_m^{(n)}} \prod_{j=0}^{m} p_j^{f_j^{(n)}}.$$

Consider the limit of large n and a sequence of empirical types so that $\frac{1}{n}\mathbf{f}^{(n)} \to \mathbf{f}$ for $n \to \infty$, where $\mathbf{f}$ is some distribution of interest.

Use Stirling's approximation to show that

$$\lim_{n\to\infty} \frac{1}{n} \ln P(\mathbf{F}^{(n)} = \mathbf{f}^{(n)}) = -\,\mathrm{KL}(\mathbf{f}, \mathbf{p})$$

Intuitively this means that the larger $\mathrm{KL}(\mathbf{f}, \mathbf{p})$ is, the easier it is to conclude $\mathbf{f} \neq \mathbf{p}$ from empirical data since the chance that we would get confused in that way is decaying exponentially. Note also that the empirical distribution is the first argument of the KL divergence and the true model is the second argument of the KL divergence — we are going from the true model to the empirical one.

After you obtain a result, use the attached Jupyter notebook to see this convergence with a simulated model, and experiment with varying both $\mathbf{f}$ and $\mathbf{p}$.

(c) **Show that for probability distributions $p(\mathbf{x}, y)$ and $q_\theta(\mathbf{x}, y) = q_\theta(y \mid \mathbf{x})q(\mathbf{x})$ with $\mathbf{x}$ from some discrete set $\mathcal{X}$ and y from some discrete set $\mathcal{Y}$ we have**

$$\mathrm{KL}(p, q_\theta) = c - \sum_{\mathbf{x}\in\mathcal{X}} \sum_{y\in\mathcal{Y}} p(\mathbf{x}, y) \ln q_\theta(y \mid \mathbf{x}) \tag{1}$$

for some constant c independent of θ.

This is one of the motivations for looking at the log of the probabilities as a loss function. From the point of view of learning θ, this is functioning the same as using the KL divergence. The next part asks you to make this explicit.

(d) In logistic regression we predict labels $y_i = +1$ or $y_i = -1$ from features $\mathbf{x}_i$ using the transition probability model

$$q_\theta(y_i \mid \mathbf{x}_i) = \frac{1}{1 + e^{-y_i \boldsymbol{\theta}^\top \mathbf{x}_i}}. \tag{2}$$

We now show that the cross-entropy loss you have seen in lectures can be formulated as minimizing the KL distance to the empirical probabilities from the probabilities induced by the model q_θ.

For convenience, we assume that all the feature $\mathbf{x}_i$ are distinct — no two training points are identical.

Use (c) to show that with the empirical distribution

$$p(\mathbf{x}, y) = \begin{cases} \frac{1}{n} & \text{if } \mathbf{x} = \mathbf{x}_i \text{ and } y = y_i \text{ for some } i = 1, 2, \ldots, n \\ 0 & \text{otherwise} \end{cases}$$

we get

$$\min_{\theta} \mathrm{KL}(p, q_{\theta}) = \min_{\theta} -\frac{1}{n} \sum_i \ln q_{\theta}(y_i \mid \mathbf{x}_i),$$

which is the cross entropy loss derived in lectures.

5 SGD in the overparametrized regime

This is a problem that helps you understand why we cared so much about the properties of minimum-norm solutions in the context of machine learning. The standard way of training neural networks in practice is stochastic gradient descent (and variants thereof). We need to understand how it behaves. Here, we just try to minimize squared error. (A similar story holds for logistic loss.)

Consider the standard least-squares problem:

$$\min_{\mathbf{w}} L(\mathbf{w}; \mathbf{X}, \mathbf{y}), \text{ where } L(\mathbf{w}; \mathbf{X}, y) := \frac{1}{2}\|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2.$$

Here $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_n]^\top$ is a $n \times d$ matrix of features and $\mathbf{y}$ is an n-dimensional vector of labels. Say $d > n$. For a single training sample $(\mathbf{x}_i, y_i)$ let $f_i(\mathbf{w}) := \frac{1}{2}(\mathbf{x}_i^\top \mathbf{w} - y_i)^2$. A *stochastic gradient* is $\nabla_{\mathbf{w}} f_i(\mathbf{w}) = (\frac{d}{d\mathbf{w}} f_i(\mathbf{w}))^\top$ where i is uniformly sampled from $\{1, \ldots, n\}$. By the properties of vector calculus, recall that the derivative of a scalar function of a vector is represented by a row vector, and the gradient is the transpose of that row vector so that we have a regular column vector.

(a) **Show that if there exists a minimizer with loss $L(\mathrm{w}; \mathbf{X}, \mathbf{y}) = 0$, then there is an infinite number of such minimizers.**

(HINT: Remember the shape of $\mathbf{X}$ *here.)*

(b) **Prove that any linear combination of stochastic gradients must lie in the row span of X.** That is, it must be a linear combination of the $\{\mathbf{x}_i\}$.

(HINT: Here, you need to actually take the derivative and see what it is.)

(c) Suppose that rows of $\mathbf{X}$ are linearly independent, and that stochastic gradient descent with constant step size $\eta > 0$ initialized at $\mathbf{w}_0 = \mathbf{0}$ converges to something when we view the sequence of vectors $\mathbf{w}_t$ as an infinite sequence. **Show that it converges to the minimum norm interpolating solution $\mathbf{w} = \mathbf{X}^\dagger \mathbf{y}$.** Here $\mathbf{X}^\dagger$ is the Moore-Penrose Pseudoinverse. You can feel free to assume that each training point is used infinitely often by SGD.

(Hint: Remember, what does it mean for a sequence to converge? What does that imply about the individual gradients? What does that mean about the quantity $\|\mathbf{X}\mathbf{w}_t - \mathbf{y}\|^2$? Finally, this part comes after the previous part for a reason.)

(d) Suppose you initialize SGD at $\mathbf{w} = \mathbf{w}_0 \neq 0$, and SGD still converges. **What will it converge to in this case?** Express your answer in terms of $\mathbf{X}, \mathbf{y}$ and $\mathbf{w}_0$.

(e) **Do the tasks from the associated Jupyter notebook and report the results.**

6 SGD (Interpolation)

This is a problem about the convergence of SGD. It can seem a bit hairy, but we've tried to simplify things so that you can understand and there is code at the end that will help you visualize what is going on. If you want, start at the end and run the code first. Then you can appreciate the phenomenon, and can work through the arguments here.

Consider we are minimizing the following optimization problem:

$$\min_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} \underbrace{(y_i - \mathbf{x}_i^\top \mathbf{w})^2}_{\mathcal{L}_i(\mathbf{w})}, \tag{3}$$

where $\mathbf{x}_i \in \mathbb{R}^d$ and the data points $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ are generated as follows,

$$y_i = \mathbf{x}_i^\top \mathbf{w}^\star, \tag{4}$$

which implies that

$$\mathcal{L}(\mathbf{w}^\star) = 0, \quad \text{and} \quad \mathcal{L}_i(\mathbf{w}^\star) = 0,\ i \in [n].$$

Here we apply stochastic gradient descent for optimizing (3),

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta\left(\frac{1}{|\mathcal{S}_t|}\sum_{i\in|\mathcal{S}_t|} \nabla\mathcal{L}_i(\mathbf{w}_t)\right), \quad t = 0, \cdots T, \tag{5}$$

where $\mathcal{S}_t$ is uniformly sampled from $\{1, \cdots, n\}$, $\mathcal{S}_t \subset \{1, \cdots, n\}$ and $|\mathcal{S}_t| = B$, and $\eta > 0$ is the step size.

Denote $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_n]^\top \in \mathbb{R}^{n\times d}$ and $\mathbf{y} = [y_1, \cdots, y_n]^\top$. Then minimizing (3) is equivalent to minimizing:

$$\min_{\mathbf{w}} \frac{1}{n}\|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2.$$

Denote the largest and smallest eigenvalue of $\frac{1}{n}\mathbf{X}^\top\mathbf{X}$ by $\lambda_{\max}$ and $\lambda_{\min}$, and denote

$$\beta = \max_{i\in[n]} \|\mathbf{x}_i\|_2^2.$$

(a) **Prove the following inequality holds for all $\mathbf{w}'$ and $\mathbf{w}$:**

$$\mathcal{L}(\mathbf{w}') \leq \mathcal{L}(\mathbf{w}) + \langle\nabla\mathcal{L}(\mathbf{w}), \mathbf{w}' - \mathbf{w}\rangle + \lambda_{\max}\|\mathbf{w} - \mathbf{w}'\|_2^2.$$

We will start you out with the first few lines of this proof. You need to finish it. You'll see some familiar tricks:

$$
\begin{aligned}
&\mathcal{L}(\mathbf{w}') \\
=&\frac{1}{n}\|\mathbf{X}\mathbf{w}' - \mathbf{y}\|_2^2 \\
=&\frac{1}{n}\|\mathbf{X}(\mathbf{w}' - \mathbf{w} + \mathbf{w}) - \mathbf{y}\|_2^2 \\
=&\frac{1}{n}\|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \frac{2}{n}\langle \mathbf{X}(\mathbf{w}' - \mathbf{w}), \mathbf{X}\mathbf{w} - \mathbf{y}\rangle + \frac{1}{n}\|\mathbf{X}(\mathbf{w} - \mathbf{w}')\|_2^2 \\
=&\cdots
\end{aligned}
$$

(Hint: expand stuff out and remember what the gradient is here.)

(b) Assume $n > d$, $\lambda_{\min} > 0$, and the data points are generated according to (4), **prove the following inequalities hold for all w:**

$$4\lambda_{\min}(\mathcal{L}(\mathbf{w}) - \mathcal{L}(\mathbf{w}^\star)) \le \|\nabla\mathcal{L}(\mathbf{w})\|_2^2 \le 4\lambda_{\max}(\mathcal{L}(\mathbf{w}) - \mathcal{L}(\mathbf{w}^\star)),$$

$$\|\nabla\mathcal{L}_i(\mathbf{w})\|_2^2 \le 4\beta(\mathcal{L}_i(\mathbf{w}) - \mathcal{L}_i(\mathbf{w}^\star)), \quad \text{for } i \in [n].$$

To start you out, we first write down the expression for $\|\nabla\mathcal{L}(\mathbf{w})\|_2^2$,

$$
\begin{aligned}
\|\nabla\mathcal{L}(\mathbf{w})\|_2^2 &= \frac{4}{n^2}\langle \mathbf{X}^\top(\mathbf{X}\mathbf{w} - \mathbf{y}), \mathbf{X}^\top(\mathbf{X}\mathbf{w} - \mathbf{y})\rangle \\
&= \frac{4}{n^2}(\mathbf{X}\mathbf{w} - \mathbf{y})^\top\mathbf{X}\mathbf{X}^\top(\mathbf{X}\mathbf{w} - \mathbf{y}).
\end{aligned}
$$

To complete this, you need to think about what $\mathbf{X}\mathbf{w} - \mathbf{y}$ can be. There are always two choices $\mathbf{0}$ or something else....

(Hint: Think about the possible nullspace of $\mathbf{X}$*. Does it matter?)*

(c) Assume $n > d$, $\lambda_{\min} > 0$, and the data points are generated according to (4), prove that by performing SGD with constant step size η, when η satisfies

$$\eta \le \frac{2\lambda_{\min}B}{2(\lambda_{\max}\beta + \lambda_{\min}\lambda_{\max}(B-1))},$$

the iterates of SGD in (5) satisify the following exponential decay in expectation of the losses:

$$\mathbb{E}\left[\mathcal{L}(\mathbf{w}_{t+1})\right] \le (1 - \eta\lambda_{\min})\,\mathbb{E}\left[\mathcal{L}(\mathbf{w}_t)\right],$$

where the expectation is taken w.r.t. the randomness in sampling the mini-batch.

Spiritually, think back to the review problem that you did way back in HW0 that was about gradient descent for ordinary least squares. There, you saw exactly this kind of exponential convergence. Here, we are doing SGD and while $\mathbb{E}\left[\nabla\mathcal{L}_i(\mathbf{w}_t)\right] = \nabla\mathcal{L}(\mathbf{w}_t)$ since a training point

i is chosen uniformly at random across the data set, that isn't quite enough. We need to make sure that we don't jump around too much due to the randomness in the choice of that training point. This is why B (the size of the minibatch) is here as well as β (the maximum norm squared of a training point).

To bound the size of the fluctuation that can happen from the randomness in the choice of the mini-batch, we need to get a handle on the expectation of the squared-norm of the stochastic gradient. How different is this than the squared-norm of the actual gradient?

Here, the following trick comes in handy that is a consequence of all the mini-batch being chosen uniformly at random from the possible B-sized subsets of the training data.

$$\mathbb{E}\left[\|\frac{1}{|\mathcal{S}_t|}\sum_{i\in\mathcal{S}_t}\nabla\mathcal{L}_i(\mathbf{w}_t)\|_2^2\right] = \frac{1}{B^2}\mathbb{E}\left[\sum_{i\in\mathcal{S}_t}\|\nabla\mathcal{L}_i(\mathbf{w}_t)\|_2^2\right] + \frac{B(B-1)}{B^2}\|\nabla\mathcal{L}(\mathbf{w}_t)\|_2^2$$

What this does is tell us that we can't be that far away. How do we use this above fact? Well, we know how we have to start. We need to use the first part of this problem:

$$\begin{aligned}\mathcal{L}(\mathbf{w}_{t+1}) &\le \mathcal{L}(\mathbf{w}_t) + \langle\nabla\mathcal{L}(\mathbf{w}_t), \mathbf{w}_{t+1}-\mathbf{w}_t\rangle + \lambda_{\max}\|\mathbf{w}_{t+1}-\mathbf{w}_t\|_2^2\\ &= \mathcal{L}(\mathbf{w}_t) - \eta\langle\nabla\mathcal{L}(\mathbf{w}_t), \frac{1}{|\mathcal{S}_t|}\sum_{i\in|\mathcal{S}_t|}\nabla\mathcal{L}_i(\mathbf{w}_t)\rangle + \eta^2\lambda_{\max}\|\frac{1}{|\mathcal{S}_t|}\sum_{i\in|\mathcal{S}_t|}\nabla\mathcal{L}_i(\mathbf{w}_t)\|_2^2,\end{aligned}$$

taking the expectation w.r.t. the randomness in sampling this mini-batch and rearranging the terms,

$$\begin{aligned}&\mathbb{E}[\mathcal{L}(\mathbf{w}_t) - \mathcal{L}(\mathbf{w}_{t+1})]\\ &\ge \eta\left\langle\nabla\mathcal{L}(\mathbf{w}_t), \mathbb{E}\left[\frac{1}{|\mathcal{S}_t|}\sum_{i\in\mathcal{S}_t}\nabla\mathcal{L}_i(\mathbf{w}_t)\right]\right\rangle - \eta^2\lambda_{\max}\mathbb{E}\left[\|\frac{1}{|\mathcal{S}_t|}\sum_{i\in\mathcal{S}_t}\nabla\mathcal{L}_i(\mathbf{w}_t)\|_2^2\right]\\ &= \cdots\\ &\vdots\ \vdots\\ &\ge \eta\lambda_{\min}\mathcal{L}(\mathbf{w}_t).\end{aligned}$$

You need to take it from here and use the ideas above to fill in the $\cdots$ lines of gaps. What can you do to those expectations? Then you need to combine terms to pull together things that depend on $\|\nabla\mathcal{L}(\mathbf{w}_t\|^2$. Once you've done that, you can apply one of the bounds from above. (Which direction does your inequality need to go?)

To complete the proof once you reach the end of the derivation above, you just take expectations of both sides.

(d) **(Completely Optional — Just to see if you understand the details. Read this part however. It is important.)** Consider $n < d$ and denote the smallest nonzero singular value of $\frac{1}{\sqrt{n}}\mathbf{X}$ by $\sigma_{\min} > 0$, and let $\tilde{\lambda}_{\min} = \sigma^2_{\min}$. The data points are still generated in a noise-free manner according to (4). It turns out that the inequalities in Part (b) still hold by replacing $\lambda_{\min}$ with $\tilde{\lambda}_{\min}$:

$$4\tilde{\lambda}_{\min}(\mathcal{L}(\mathbf{w}) - \mathcal{L}(\mathbf{w}^\star)) \le \|\nabla\mathcal{L}(\mathbf{w})\|_2^2 \le 4\lambda_{\max}(\mathcal{L}(\mathbf{w}) - \mathcal{L}(\mathbf{w}^\star)),$$
$$\|\nabla\mathcal{L}_i(\mathbf{w})\|_2^2 \le 4\beta(\mathcal{L}_i(\mathbf{w}) - \mathcal{L}_i(\mathbf{w}^\star)), \quad \text{for } i \in [n].$$

If you want to practice, **Verify the above proof still works.** The consequence of that proof working is that by performing SGD with constant step size η, when η satisfies

$$\eta \le \frac{2\tilde{\lambda}_{\min}B}{2(\lambda_{\max}\beta + \tilde{\lambda}_{\min}\lambda_{\max}(B-1))},$$

the iterates of SGD in (5) also have an exponential decay in expectation:

$$\mathbb{E}\left[\mathcal{L}(\mathbf{w}_{t+1})\right] \le \left(1 - \eta\tilde{\lambda}_{\min}\right)\mathbb{E}\left[\mathcal{L}(\mathbf{w}_t)\right],$$

where the expectation is taken w.r.t. the randomness in sampling the mini-batch. This is because none of the arguments that you used in the earlier part actually cared about what the relationship was between n and d.

This has an important consequence. When data is noiseless and you are doing least-squares, you get exponential convergence even if you do SGD instead of gradient descent. It doesn't matter if you have more data than features or more features than data.

You might wonder whether there is a difference between this fact holding in expectation and convergence holding with probability 1. Think about it. When we have an exponential decay in an expectation, even Markov's inequality will tell us that this convergence has to happen with probability 1. Suppose there was a finite probability of divergence. Then there has to be some finite non-zero level that is crossed infinitely often. Markov's inequality will rule that out. So, SGD with a constant step size (that is sufficiently small) always converges and does so exponentially fast!

(e) Now we can enjoy the payoff of the work above. The noise-free case seems insanely special. What if we have some noise or model mismatch? Adding regularization (the right way) does the trick for us.

Suppose $n < d$ and the data points are generated with noise in them as follows:

$$y_i = \mathbf{x}_i^\top \mathbf{w}^\star + \epsilon_i, \quad \epsilon \sim \mathcal{N}(0, \sigma). \tag{6}$$

Recall the augmented feature perspective on ridge regression:

$$\begin{aligned} &\min_{\boldsymbol{\theta}} \ \|\boldsymbol{\theta}\|_2^2 \\ &\text{s.t. } \hat{\mathbf{X}}\boldsymbol{\theta} = \mathbf{y}, \quad \boldsymbol{\theta} = \begin{bmatrix}\mathbf{w}\\ \boldsymbol{\xi}\end{bmatrix} \in \mathbb{R}^{d+n}, \end{aligned} \tag{7}$$

where the augmented data matrix $\hat{\mathbf{X}}$ is defined as

$$\hat{\mathbf{X}} = \left[\mathbf{X} \mid \sqrt{\lambda}\mathbf{I}\right] \in \mathbb{R}^{n\times(d+n)},$$

where λ is the regularization in ridge regression. Denote the smallest singular value of $\frac{1}{\sqrt{n}}\hat{\mathbf{X}}$ by $\sigma_{\min} > 0$, and let $\tilde{\lambda}_{\min} = \sigma_{\min}^2$, and let

$$\beta = \max_{i \in [n]} \|\hat{\mathbf{x}}_i\|_2^2,$$

where $\hat{\mathbf{x}}_i$ is the i-th row of matrix $\hat{\mathbf{X}}$. Denote the optimal solution to (7) by

$$\widehat{\boldsymbol{\theta}} = \begin{bmatrix} \widehat{\mathbf{w}} \\ \widehat{\boldsymbol{\xi}}, \end{bmatrix}.$$

Consider the case of initializing $\boldsymbol{\theta}_0 = \mathbf{0}$ for SGD and having a learning rate η that satisfies

$$\eta \leq \frac{2\tilde{\lambda}_{\min} B}{2(\lambda_{\max}\beta + \tilde{\lambda}_{\min}\lambda_{\max}(B-1))}.$$

Consider the iterates of SGD in (5) for the following problem

$$\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \underbrace{(y_i - \hat{\mathbf{x}}_i^\top \boldsymbol{\theta})^2}_{\mathcal{L}_i(\boldsymbol{\theta})},$$

and **argue based on earlier parts and earlier problems that they satisify the following inequality which shows exponential decay in expected loss for the augmented problem.**

$$\mathbb{E}\left[\mathcal{L}(\boldsymbol{\theta}_{t+1})\right] \leq \left(1 - \eta\tilde{\lambda}_{\min}\right)\mathbb{E}\left[\mathcal{L}(\boldsymbol{\theta}_t)\right],$$

where the expectation is taken w.r.t. the randomness in sampling the mini-batch. **Based on the previous HW problem, what do you think is the implication for what problem is being solved by $\mathbf{w}_t$ in the limit ($\boldsymbol{\theta}_t = \begin{bmatrix} \mathbf{w}_t \\ \boldsymbol{\xi}_t, \end{bmatrix}$) as t gets large?**

(f) The previous part has shown you that done properly, you can get SGD to converge exponentially for ridge regression. Check out the Jupyter part of this problem, run (5) for different settings as described in Part (c), (d), and (e). Report what you observed in terms of the convergence rate of (5).

One of the lessons that you will observe from the code is that the implementation details matter. If you do ridge regression and just treat it as an optimization problem, you won't just be able to use SGD and get exponential convergence with a constant step size. (You would have to adjust the step sizes to make them smaller, but this would slow down your convergence considerably.) But if you intelligently use the feature-aumentation perspective on ridge regression, you'll get exponential convergence.

This is why it is vital for people in EECS to really understand machine learning at the level of detail that we are teaching you. Because in the real world, even if you are a practicing machine learning engineer, if you are working on cutting-edge systems, you need to understand how to implement what you want to do so that it works fast. Equivalent formulations mathematically need not be equivalent from the point of view of implementation – this is one dramatic example of a case when they are not. Take EE227C and beyond if you want to understand these things more deeply.

7 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.

Contributors:

- Alexander Tsigler
- Anant Sahai
- Ashwin Pananjady
- Kailas Vodrahalli
- Peter Wang
- Philipp Moritz
- Stephen Bailey
- Stephen Tu
- Vaishaal Shankar
- Yaodong Yu