EECS 182 Deep Neural Networks
Fall 2023 Anant Sahai

Homework 0

This homework is due on Tuesday, Aug 29, 2023, at 10:59PM.

1. Gradient Descent Doesn't Go Nuts with Ill-Conditioning

Consider a linear regression problem with n training points and d features. When $n = d$, the feature matrix $F \in \mathbb{R}^{n \times n}$ has some maximum singular value α and an extremely tiny minimum singular value. We have noisy observations $\mathbf{y} = F\mathbf{w}^* + \boldsymbol{\epsilon}$. If we compute $\hat{\mathbf{w}}_{inv} = F^{-1}\mathbf{y}$, then due to the tiny singular value of F and the presence of noise we observe that $\|\hat{\mathbf{w}}_{inv} - \mathbf{w}^*\|_2 = 10^{10}$.

Suppose instead of inverting the matrix we decide to use gradient descent instead. We run k iterations of gradient descent to minimize the loss $\ell(w) = \frac{1}{2}\|\mathbf{y} - F\mathbf{w}\|^2$ starting from $\mathbf{w}_0 = \mathbf{0}$. We use a learning rate η which is *small enough* that gradient descent cannot possibly diverge for the given problem. (**This is important. You will need to use this.**)

The gradient-descent update for $t > 0$ is:

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \eta \left(F^\top (F\mathbf{w}_{t-1} - \mathbf{y}) \right).$$

We are interested in the error $\|\mathbf{w}_k - \mathbf{w}^*\|_2^2$. We want to show that in the worst case, this error can grow at most linearly with iterations k and in particular $\|\mathbf{w}_k - \mathbf{w}^*\|_2 \leq k\eta\alpha\|\mathbf{y}\|_2 + \|\mathbf{w}^*\|_2$.

i.e. The error cannot go "nuts," at least not very fast.

For the purposes of the homework, you only have to prove the key idea, since the rest follows by applying induction and the triangle inequality.

Show that for $t > 0$, $\|\mathbf{w}_t\|_2 \leq \|\mathbf{w}_{t-1}\|_2 + \eta\alpha\|\mathbf{y}\|_2$.

(HINT: What do you know about $(I - \eta F^\top F)$ if gradient descent cannot diverge? What are its eigenvalues like? Use this fact.)

Solution: We have,

$$\begin{aligned}
\|w_t\|_2 &= \|w_{t-1} - \eta(F^\top(Fw_{t-1} - y))\|_2 \\
&= \|(I - \eta F^\top F)w_{t-1} + \eta F^\top y\|_2 \\
&\leq \|(I - \eta F^\top F)w_{t-1}\|_2 + \|\eta F^\top y\|_2 \\
&\leq \sigma_{\max}(I - \eta F^\top F)\|w_{t-1}\|_2 + \eta\sigma_{max}(F)\|y\|_2 \\
&\leq \|w_{t-1}\|_2 + \eta\alpha\|y\|_2,
\end{aligned}$$

where we used the fact that η is chosen so that gradient descent does not diverge so the maximum eigenvalue of $(I - F^\top F)$ cannot have an absolute value greater than 1. But $I - F^\top F$ is a real symmetric matrix and the spectral theorem tells us that the singular values are therefore just the absolute values of the eigenvalues. This means that the maximum singular value also must have absolute value less than 1.

2. Regularization from the Augmentation Perspective

Assume $\mathbf{w}$ is a d-dimensional Gaussian random vector $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \Sigma)$ and Σ is symmetric positive-definite. Our model for how the $\{y_i\}$ training data is generated is

$$y = \mathbf{w}^\top \mathbf{x} + Z, \quad Z \sim \mathcal{N}(0, 1), \tag{1}$$

where the noise variables Z are independent of $\mathbf{w}$ and iid across training samples. Notice that all the training $\{y_i\}$ and the parameters $\mathbf{w}$ are jointly normal/Gaussian random variables conditioned on the training inputs $\{\mathbf{x}_i\}$. Let us define the standard data matrix and measurement vector:

$$X = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_n^T \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}.$$

In this model, the MAP estimate of $\mathbf{w}$ is given by the Tikhonov regularization counterpart of ridge regression:

$$\widehat{\mathbf{w}} = (X^\top X + \Sigma^{-1})^{-1} X^\top \mathbf{y}, \tag{2}$$

In this question, we explore Tikhonov regularization from the data augmentation perspective.

Define the matrix Γ as a $d \times d$ matrix that satisfies $\Gamma^\top \Gamma = \Sigma^{-1}$. Consider the following augmented design matrix (data) $\hat{X}$ and augmented measurement vector $\hat{y}$:

$$\hat{X} = \begin{bmatrix} X \\ \Gamma \end{bmatrix} \in \mathbb{R}^{(n+d)\times d}, \quad \text{and} \quad \hat{\mathbf{y}} = \begin{bmatrix} \mathbf{y} \\ \mathbf{0}_d \end{bmatrix} \in \mathbb{R}^{n+d},$$

where $\mathbf{0}_d$ is the zero vector in $\mathbb{R}^d$. **Show that the ordinary least squares problem**

$$\operatorname*{argmin}_{\mathbf{w}} \|\hat{\mathbf{y}} - \hat{X}\mathbf{w}\|_2^2$$

has the same solution as (2).

(HINT: Feel free to just use the formula you know for the OLS solution. You don't have to rederive that. This problem is not intended to be hard or time consuming.)

Solution: The solution to $\min_{\mathbf{w}} \|\hat{\mathbf{y}} - \hat{X}\mathbf{w}\|_2^2$ can be solved using the OLS formula as:

$$\begin{aligned} \left(\hat{X}^\top \hat{X}\right)^{-1} \hat{X}\hat{\mathbf{y}} &= \left(\begin{bmatrix} X^\top \Gamma^\top \end{bmatrix} \begin{bmatrix} X \\ \Gamma \end{bmatrix} \right)^{-1} \begin{bmatrix} X^\top \Gamma^\top \end{bmatrix} \begin{bmatrix} \mathbf{y} \\ \mathbf{0}_k \end{bmatrix} \\ &= (X^\top X + \Gamma^\top \Gamma)^{-1} X^\top \mathbf{y} + (X^\top X + \Gamma^\top \Gamma)^{-1} \Gamma^\top \mathbf{0} \\ &= (X^\top X + \Sigma^{-1})^{-1} X^\top \mathbf{y}. \end{aligned}$$

This exactly agrees with (2) and so we are done.

3. Vector Calculus Review

Let $\mathbf{x}, \mathbf{c} \in \mathbb{R}^n$ and $A \in \mathbb{R}^{n \times n}$. For the following parts, before taking any derivatives, identify what the derivative looks like (is it a scalar, vector, or matrix?) and how we calculate each term in the derivative. Then carefully solve for an arbitrary entry of the derivative, then stack/arrange all of them to get the final

result. Note that the convention we will use going forward is that vector derivatives of a scalar (with respect to a column vector) are expressed as a row vector, i.e. $\frac{\partial f}{\partial \mathbf{x}} = [\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, ..., \frac{\partial f}{\partial x_n}]$ since a row acting on a column gives a scalar. You may have seen alternative conventions before, but the important thing is that you need to understand the types of objects and how they map to the shapes of the multidimensional arrays we use to represent those types.

(a) Show $\frac{\partial}{\partial \mathbf{x}}(\mathbf{x}^T\mathbf{c}) = \mathbf{c}^T$

Solution: This is a vector derivative of a scalar quantity, so our result will be a row vector. Looking at the i-th entry, $\frac{\partial}{\partial x_i}(\mathbf{x}^T\mathbf{c}) = \frac{\partial}{\partial x_i}(\sum_j c_j x_j) = c_i$. Stacking all the entries into a row vector, we get $\mathbf{c}^T$.

(b) Show $\frac{\partial}{\partial \mathbf{x}}||\mathbf{x}||_2^2 = 2\mathbf{x}^T$

Solution: This is a vector derivative of a scalar quantity, so our result will be a row vector. Looking at the i-th entry, $\frac{\partial}{\partial x_i}(||\mathbf{x}||_2^2) = \frac{\partial}{\partial x_i}(\sum_j x_j^2) = 2x_i$. Stack all the entries into a row to get $2\mathbf{x}^T$.

(c) Show $\frac{\partial}{\partial \mathbf{x}}(A\mathbf{x}) = A$

Solution: This is a vector derivative of a vector quantity, so the result will be a matrix. Let $\mathbf{f} = A\mathbf{x}$. Note that $f_i = \sum_k A_{ik}x_k$

Looking at the (i, j)-th entry of our matrix, $\frac{\partial f_i}{\partial x_j} = \frac{\partial}{\partial x_j}(\sum_k A_{ik}x_k) = A_{ij}$. Arranging all of these in a matrix will recover A.

(d) Show $\frac{\partial}{\partial \mathbf{x}}(\mathbf{x}^T A\mathbf{x}) = \mathbf{x}^T(A + A^T)$

Solution: This is a vector derivative of a scalar quantity, so our result will be a vector. Before taking any derivatives, we can write $\mathbf{x}^T A\mathbf{x} = \sum_i \sum_j A_{ij}x_i x_j$. Taking the derivative with respect to an arbitrary x_k and focusing on just the terms involving x_k (as the derivative of the other terms wrt x_k is zero), we can write

$$\frac{\partial}{\partial x_k}(\mathbf{x}^T A\mathbf{x}) = \frac{\partial}{\partial x_k}((\sum_{j\neq k} A_{kj}x_k x_j) + (\sum_{i\neq k} A_{ik}x_i x_k) + A_{kk}x_k^2)$$

$$= (\sum_{j\neq k} A_{kj}x_j) + (\sum_{i\neq k} A_{ik}x_i) + 2A_{kk}x_k$$

$$= (\sum_{j} A_{kj}x_j) + (\sum_{i} A_{ik}x_i) = \sum_i (A_{ki}x_i + A_{ik}x_i)$$

$$= \mathbf{x}^T(\text{kth row of A} + \text{kth col of A}) = \mathbf{x}^T(A_k + A_k^T)$$

Stacking all the results in a row vector, we get $\frac{\partial}{\partial \mathbf{x}}(\mathbf{x}^T A\mathbf{x}) = \mathbf{x}^T(A + A^T)$ as desired.

(e) Under what condition is the previous derivative equal to $2\mathbf{x}^T A$?

Solution: We want $(A + A^T) = 2A$. This is true if and only if $A = A^T$, ie. the matrix A is symmetric.

4. ReLU Elbow Update under SGD

In this question we will explore the behavior of the ReLU nonlinearity with Stochastic Gradient Descent (SGD) updates. The hope is that this problem should help you build a more intuitive understanding for how SGD works and how it iteratively adjusts the learned function.

We want to model a 1D function $y = f(x)$ using a 1-hidden layer network with ReLU activations and no biases in the linear output layer. Mathematically, our network is

$$\hat{f}(x) = \mathbf{W}^{(2)}\Phi\left(\mathbf{W}^{(1)}x + \mathbf{b}\right)$$

where $x, y \in \mathbb{R}$, $\mathbf{b} \in \mathbb{R}^d$, $\mathbf{W}^{(1)} \in \mathbb{R}^{d\times 1}$, and $\mathbf{W}^{(2)} \in \mathbb{R}^{1\times d}$. We define our loss function to be the squared error,

$$\ell\left(x, y, \mathbf{W}^{(1)}, \mathbf{b}, \mathbf{W}^{(2)}\right) = \frac{1}{2}\left\|\hat{f}(x) - y\right\|_2^2.$$

For the purposes of this problem, we define the gradient of a ReLU at 0 to be 0.

(a) Let's start by examining the behavior of a single ReLU with a linear function of x as the input,

$$\phi(x) = \begin{cases} wx + b, & wx + b > 0 \\ 0, & \text{else} \end{cases}.$$

Notice that the slope of $\phi(x)$ is w in the non-zero domain.
We define a loss function $\ell(x, y, \phi) = \frac{1}{2}\|\phi(x) - y\|_2^2$. **Find the following:**

(i) The location of the 'elbow' e of the function, where it transitions from 0 to something else.

(ii) The derivative of the loss w.r.t. $\phi(x)$, namely $\frac{d\ell}{d\phi}$

(iii) The partial derivative of the loss w.r.t. w, namely $\frac{\partial \ell}{\partial w}$

(iv) The partial derivative of the loss w.r.t. b, namely $\frac{\partial \ell}{\partial b}$

Solution:

(i)

$$e = -\frac{b}{w}$$

(ii)

$$\frac{d\ell}{d\phi} = \phi(x) - y$$

(iii)

$$\frac{\partial \ell}{\partial w} = \frac{\partial \ell}{\partial \phi}\frac{\partial \phi}{\partial w} = \begin{cases} (\phi(x) - y)x, & wx + b > 0 \\ 0, & \text{else} \end{cases}$$

(iv)

$$\frac{\partial \ell}{\partial b} = \frac{\partial \ell}{\partial \phi}\frac{\partial \phi}{\partial b} = \begin{cases} (\phi(x) - y), & wx + b > 0 \\ 0, & \text{else} \end{cases}$$

(b) Now suppose we have some training point (x, y) such that $\phi(x) - y = 1$. In other words, the prediction $\phi(x)$ is 1 unit above the target y — we are too high and are trying to pull the function downward.

Describe what happpens to the slope and elbow of $\phi(x)$ when we perform gradient descent in the following cases:

(i) $\phi(x) = 0$**.**

(ii) $w > 0$**,** $x > 0$**, and** $\phi(x) > 0$**. It is fine to check the behavior of the elbow numerically in this case.**

(iii) $w > 0$**,** $x < 0$**, and** $\phi(x) > 0$**.**

(iv) $w < 0$**,** $x > 0$**, and** $\phi(x) > 0$**. It is fine to check the behavior of the elbow numerically in this case.**

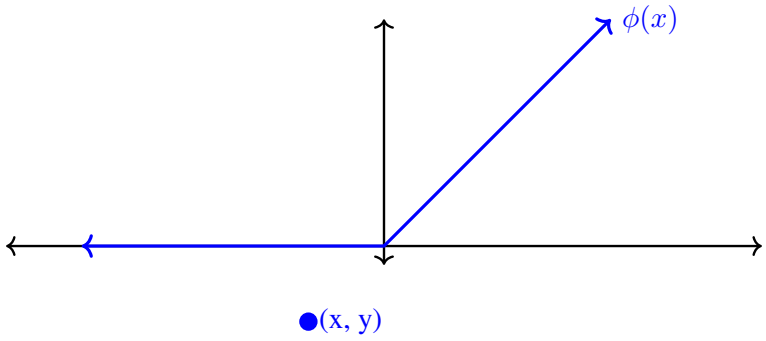
Additionally, draw and label $\phi(x)$**, the elbow, and the qualitative changes to the slope and elbow after a gradient update to** w **and** b**. You should label the elbow location and a candidate** (x, y) **pair.** Remember that the update for some parameter vector $\mathbf{p}$ and loss ℓ under SGD is

$$\mathbf{p}' = \mathbf{p} - \lambda \nabla_{\mathbf{p}}(\ell),\ \lambda > 0.$$

Solution: For each of the cases the change in the slope is determined by $\frac{\partial \ell}{\partial w}$ and the new elbow is located at

$$e' = \frac{-(b - \Delta b)}{w - \Delta w} = \frac{-b + \lambda \frac{\partial \ell}{\partial b}}{w - \lambda \frac{\partial \ell}{\partial w}}$$

(i) No changes since the derivatives are 0.



(ii) The slope gets shallower but the elbow can move left or right depending on the step size.

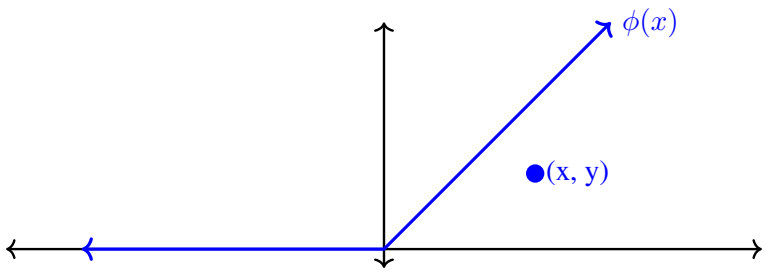
$$\frac{\partial \ell}{\partial w} = 1x > 0 \implies w > w - \lambda x$$

$$\frac{\partial \ell}{\partial b} = 1 \implies b > b - \lambda$$

Thus, the elbow moves from $-\frac{b}{w}$ to $-\frac{b-\lambda}{w-\lambda x}$. The elbow moves right if and only if

$$\begin{aligned} -\frac{b}{w} &< \frac{b - \lambda}{w - \lambda x} \\ \Rightarrow \frac{\lambda(bx - w)}{w(w - \lambda x)} &< 0 \\ \Rightarrow \frac{b - \frac{w}{x}}{\frac{w}{x} - \lambda} &< 0 \\ \Rightarrow (b - \frac{w}{x})(\lambda - \frac{w}{x}) &> 0. \end{aligned}$$

In other words, if the bias b and the step size λ are on the same side of $\frac{w}{x}$, then the elbow moves right, and left otherwise.
Several numerical checks show the motion of the elbow.

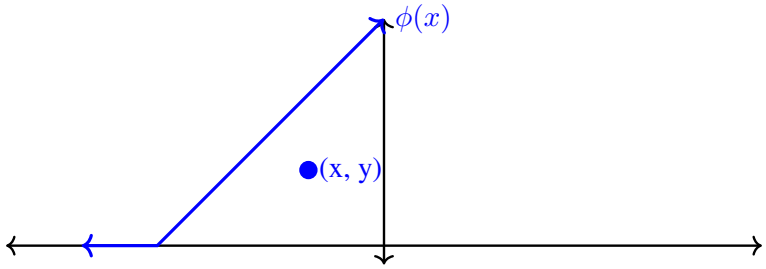


(iii) The slope gets steeper and the elbow moves right. Using the fact that $w, b > 0$ and $e < 0$ for this configuration,

$$\frac{\partial \ell}{\partial w} = 1x < 0 \implies |w| < |w - \lambda x|$$

$$\frac{\partial \ell}{\partial b} = 1 \implies |b| > |b - \lambda|$$

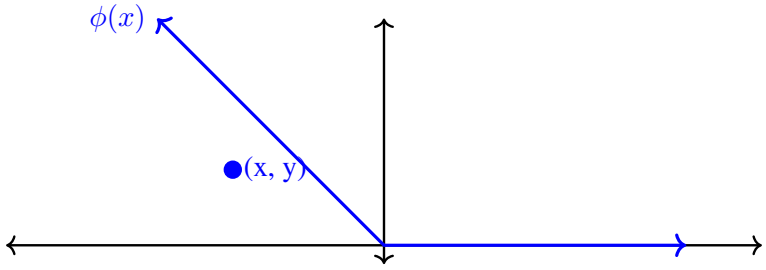
$$\therefore e' > e$$



(iv) The slope gets steeper and the elbow moves left. Since $w, x < 0$ for this configuration,

$$\frac{\partial \ell}{\partial w} = 1x > 0 \implies |w| < |w - \lambda x|$$

Several numerical checks show the motion of the elbow.



Believe it or not, even when the slope moves in the opposite direction you expect it to the error still decreases. You can try this yourself with a learning rate of 0.1, $x = 1$, $y = 1$, $w = -2$, and $b = 4$.

We encourage you to check the behavior (numerically is probably the easiest method) for these cases when $\phi(x) - y < 0$ and see what changes and what stays the same.

You may give yourself full credit for self-grades if you checked the behavior for a single value in each case rather than in general.

(c) Now we return to the full network function $\hat{f}(x)$. **Derive the location e_i of the elbow of the i'th elementwise ReLU activation.**

Solution:

$$\mathbf{W}_i^{(1)} x + \mathbf{b}_i = 0$$
$$x = -\frac{\mathbf{b}_i}{\mathbf{W}_i^{(1)}}$$

(d) **Derive the new elbow location e_i' of the i'th elementwise ReLU activation after one stochastic gradient update with learning rate λ.**

Solution: The new location after an SGD update is

$$e_i' = \frac{-\mathbf{b}_i + \lambda \frac{\partial \ell}{\partial \mathbf{b}_i}}{\mathbf{W}_i^{(1)} - \lambda \frac{\partial \ell}{\partial \mathbf{W}_i^{(1)}}}.$$

We have

$$\frac{\partial \ell}{\partial \mathbf{W}^{(1)}} = x \left(\mathbf{W}^{(2)} \Phi \left(\mathbf{W}^{(1)} x + \mathbf{b} \right) - y \right) \mathbf{W}^{(2)} \frac{\partial \Phi}{\partial \left(\mathbf{W}^{(1)} x + \mathbf{b} \right)}$$

For the ReLU activation,

$$\frac{\partial \Phi}{\partial \left(\mathbf{W}^{(1)} x + \mathbf{b} \right)} = \text{diag}\Big(\Big)(\mathbb{1} \left(\mathbf{W}^{(1)} x + \mathbf{b} > 0 \right)\Big)$$

where $\mathbb{1}(\cdot)$ is an indicator variable for whether each element of the contents is true or false. Since we are only interested in the i'th index of this gradient, we can simplify $\mathbf{W}^{(2)} \frac{\partial \Phi}{\partial (\cdot)}$ to

$$\begin{cases} \mathbf{W}_i^{(2)}, & \left(\mathbf{W}^{(1)} x + \mathbf{b} \right)_i > 0 \\ 0, & \text{else} \end{cases}$$

Therefore,

$$\frac{\partial \ell}{\partial \mathbf{W}_i^{(1)}} = \begin{cases} x \left(\mathbf{W}^{(2)} \Phi \left(\mathbf{W}^{(1)} x + \mathbf{b} \right) - y \right) \mathbf{W}_i^{(2)}, & \mathbf{W}_i^{(1)} x + \mathbf{b}_i > 0 \\ 0, & \text{else} \end{cases}$$

$$\frac{\partial \ell}{\partial \mathbf{b}_i} = \left(\mathbf{W}^{(2)} \Phi \left(\mathbf{W}^{(1)} x + \mathbf{b} \right) - y \right) \mathbf{W}_i^{(2)} \frac{\partial \Phi_i}{\partial \left(\mathbf{W}^{(1)} x + \mathbf{b} \right)}$$

$$= \begin{cases} \left(\mathbf{W}^{(2)} \Phi \left(\mathbf{W}^{(1)} x + \mathbf{b} \right) - y \right) \mathbf{W}_i^{(2)}, & \mathbf{W}_i^{(1)} x + \mathbf{b}_i > 0 \\ 0, & \text{else} \end{cases}$$

Putting everything together,

$$e_i' = \begin{cases} \frac{-\mathbf{b}_i + \lambda \left(\mathbf{W}^{(2)} \Phi \left(\mathbf{W}^{(1)} x + \mathbf{b} \right) - y \right) \mathbf{W}_i^{(2)}}{\mathbf{W}_i^{(1)} - \lambda x \left(\mathbf{W}^{(2)} \Phi \left(\mathbf{W}^{(1)} x + \mathbf{b} \right) - y \right) \mathbf{W}_i^{(2)}}, & \mathbf{W}_i^{(1)} x + \mathbf{b}_i > 0 \\ e_i, & \text{else} \end{cases}$$

$$= \begin{cases} \frac{-\mathbf{b}_i + \lambda\left(\hat{f}(x) - y\right)\mathbf{W}_i^{(2)}}{\mathbf{W}_i^{(1)} - \lambda x\left(\hat{f}(x) - y\right)\mathbf{W}_i^{(2)}}, & \mathbf{W}_i^{(1)} x + \mathbf{b}_i > 0 \\ e_i, & \text{else} \end{cases}$$

We can still predict the change in each component ReLU function $\Phi_i(x)$ as long as we know $\hat{f}(x) - y$, $\mathbf{W}_i^{(1)}$, $\mathbf{b}_i$, and $\mathbf{W}_i^{(2)}$.

5. Using PyTorch to Learn the Color Organ

One of the greatest recent developments in easy-to-use software packages is the easy availability of automatic differentiation. Although the underlying technology had been well established for over four decades (originally developed for control and scientific modeling applications in the context of differential equations), today packages like PyTorch expose this power to us in an easy to use way. This means that we as human engineers no longer have to worry about manually computing derivatives for any purpose other than taking exams, doing proofs, and basically learning material. In practical applications, the computer can do it for us without any bugs. As students whose careers will span the next four decades, we want you to consider this as a part of your engineering inheritance so that you can use it freely without thinking of it as being any more special than a hash table.

This problem and the accompanying Jupyter Notebook `color_organ_learning.ipynb` will show you how this power can be used to learn the value for the resistors in the color organ simply by having examples of where you want the LEDs to be on and off. This will build on the use of PyTorch that you will have seen in discussion.

(NOTE: A "color organ" is a fun hardware lab exercise where students build an analog circuit where different LEDs light up depending on whether a particular tone is a low frequency, high frequency, etc. This shows the intimate connection between filters and classifiers. Students in 16B often have to build such a circuit and manually tune it so that it recognizes the right kind of tones.)

(a) Let's start with an example low pass filter where, given a desired transfer function, we can determine a resistor value manually for our predicted transfer function such that the transfer functions match. In the interactive plot, we show a transfer function of a desired low pass filter (orange dotted line); we want to design our circuit (given a fixed capacitor value) such that predicted transfer function (blue solid line) is equivalent. For the following problems, we provide a function `evaluate_lp_circuit` that evaluates the transfer function magnitude given a resistor value. Note that these functions use `torch` functions instead of `numpy` as we will typically use `torch` tensors instead of `numpy` arrays in this notebook for training. **Use the slider to find a resistor value such that the predicted and desired transfer functions match and report the resistor value.**

The use of `torch` tensors instead of `numpy` arrays allows the package to do the bookkeeping behind the scenes that allows derivatives to be easily calculated. This will be important in later parts.

Solution: We see that the functions match well using a resistor value around 200 Ohms (in fact, the desired transfer function is generated with a 199 Ohm resistor.)

(b) Now, suppose that instead of seeing the entire transfer function that we want to match, we are only given some data about which frequencies lie in the pass band (i.e., which frequencies cause the LED on our color organ to be lit). Again, we can determine a resistor value manually. In the interactive plot, we show a transfer function of a low pass filter with its corresponding cutoff frequency. The table shows the desired behavior (red bars denote that the LED is on for a given frequency while black bars denote that the LED is off); we want to design the circuit such that the LEDs light up in the same way.

Use the slider to find a resistor value and corresponding cutoff frequency such that the predicted lights match the desired lights and report the corresponding resistor value and cutoff frequency.

Solution: Any resistor value between 180 Ohms and 228 Ohms results in the correct LEDs lighting up.

(c) Assume that we are able to query the desired transfer function directly (i.e., we can play a frequency and record the magnitude of the output). Can we *learn* the resistor value in the low pass circuit directly from this data (instead of manually designing the circuit as you did in the first part)?

The code for this part creates a model of the low pass filter in PyTorch, generates training data, and then trains the circuit using mean squared error loss until convergence (i.e., loss or gradients are very small) or the maximum number of training steps are reached. Here, we use the `torch.autograd.grad` function to automatically find the derivative of the loss with respect to the input (the resistor value of the low pass circuit). All we need to do is input the transfer function and define the loss!

The plots show how the transfer function of the learned circuit evolved during training, the loss surface, the derivative with respect to each training point at each iteration, and the total gradient at each training iteration. Note that the learned transfer function and resistor value change more slowly when the gradient is small, and more quickly when the gradient is large, but if we continue to iterate, we will converge to a local minimum in the loss surface. Try initializing the circuit with different resistor values (there is an optional argument for the circuit class constructor; if you leave it blank it will be initialized to a random value between 0 and 1000). **How long does the circuit take to converge with different initializations? Does it converge to the same value you found in the previous parts?** Try changing the learning rate (this parameter controls how far we step at each iteration in the direction of the negative gradient). **What values of `lr` cause training to diverge? What values cause the circuit to converge quickly?**

Solution: The resistor value converges to 200 Ohms from an initialization at 500 Ohms in about 45,000 to 50,000 iterations with a learning rate of 200. For initializations between 1-200 Ohms, the resistor value converges within 20,000 steps (with the same learning rate) and for initializations above 200 Ohms, it takes longer to converge the further we start from the optimal value (but within 100,000 steps).

For extremely large learning rates (e.g., greater than 2×10^6), training can either diverge (i.e., result in negative resistance values or overflow) or oscillate. Even for smaller learning rates, if the initialization is too close to zero, then it can diverge. Learning rates around 10^5 give convergence within about 200 iterations (assuming that the initialization is greater than 5 Ohms).

(d) Now, using the same loss function as in the previous part, let's try to learn the resistor value using only the binary data we have (LED on the color organ being on or off). **Change the code to use binary data instead of the transfer function magnitudes. What is the learned resistor value?** (Hints: Some potentially useful constants are defined at the start of the notebook. The loss function must take in floats (not boolean values).)

Solution: See Jupyter Notebook. The resistor value converges to about 330 Ohms, which does not match the 200 Ohms we had found previously.

(e) What happened in the previous part? We did not converge to the same resistor value as we did in the previous part. Why? Because in trying to fit to the binary data, we can see that the positive and negative samples in our training data are pulling the resistor value in opposite directions (bottom left plot) and the final solution we end up at is where this "tug of war" situation achieves a balance. This balance need not end up where we would like it to. Can we fix this problem by adjusting the loss function? **Adjust the loss function such that the negative samples (where the LEDs should be off) are demanding an output other than 0 in a way that helps get the balance to end up where you**

want it. Can you find a loss function that yields the same resistor value as when training with the full transfer function?

Solution: See Jupyter Notebook.

(f) Let's use what we learned in the previous parts to also learn our high pass filter from binary data. **Input the high pass filter transfer function into the high pass circuit module (use the same `loss_fn` as you found in the previous part) and fill in the code that updates the value of the resistor at each training step** (Hint: use the low pass filter as an example). **What is the learned resistor value?**

Solution: See Jupyter Notebook. The learned resistor value is 33 Ohms.

(g) Now let's extend the problem to a circuit with multiple parameters and learn both resistors for a band pass filter. **Input the band pass filter transfer function into the band pass circuit module.** (Hint: you can use the functions previously defined for high and low pass circuits). **Then complete the code for updating both resistors (note that `torch.autograd.grad` returns a tuple of gradients corresponding to each input). What are the learned resistor values?** What happens if the initial resistor values are far from the solution? **Try training with initial resistor values of 900 Ohms each. Does the circuit converge within the maximum number of training steps? How much longer does it take to converge? How large are the gradients and what does the loss surface look like when the resistor values are very far from the correct solution?**

Solution: See Jupyter Notebook. For initializations far from the optimal values, it takes longer to converge to the optimal resistor values of 40 Ohms and 160 Ohms because the loss surface is flatter (and thus the gradients are smaller).

(h) Now that we have tried learning the resistor values for a band pass filter directly from binary data, let's explore a different parametrization of our filter: the Bode Plot. Let's again start by trying to learn cutoff frequencies from samples of a transfer function using two ReLU functions. **Run the code. Does the resulting Bode plot match what you would draw given the underlying transfer function data? Do the cutoff frequencies match those corresponding to the resistor values that were found in the previous part?**

Solution: See Jupyter Notebook. The resulting cutoff frequencies are very close to the expected cutoffs of 1000 Hz (high pass) and 4000 Hz (low pass). This implies that the Bode plot actually is the piecewise linear approximation that minimizes the mean squared error with respect to the true transfer function!

(i) Let's put all of these together to try and learn a color organ circuit from a low pass, high pass, and band pass circuits. Here, we train with a vector of size `(3, n)` which is *one-hot encoded*, meaning that for each of the n datapoints, one of the three values is 1 and the rest are 0. This encoding corresponds to our LEDs being on or off for one and only one of the three filters at a given frequency. **Train the color organ circuit and verify that the learned resistor values match those from the previous parts. Try initializing the resistors to different values; does it take longer or shorter to learn the entire color organ circuit than a single one of the filters (low pass, high pass, or band pass)?**

Solution: See Jupyter Notebook. Learning all of the resistors for the color organ takes longer than learning one of the individual filters.

The final part of the notebook visualizes the computation graph that PyTorch is using to compute the derivatives of each transfer function with respect to the resistor values. See if you can match each operation in the graph to the transfer functions for each filter. Hopefully, this graph gives you an idea of how PyTorch can determine the partial derivatives that you have been using throughout the notebook. Congratulations, you now know how to use the considerable power of PyTorch to automatically differentiate arbitrary functions and find the corresponding local minima of these functions via gradient descent!

6. 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? Write it down here where you'll need to remember it for the self-grade form.**

Contributors:

- Yaodong Yu.
- Anant Sahai.
- Saagar Sanghavi.
- Josh Sanz.
- Michael Danielczuk.
- Kumar Krishna Agrawal.

