Problem Set

Deadline: Wednesday, Feb. 10, at 11:59pm.

Submission: You need to submit three files through MarkUs[1]:

- Your answers to the questions, as a PDF file `pset_writeup.pdf`. While we encourage LaTeX, A scan is OK as long as it's readable (see note about neatness points).
- The completed code files `q2.py` and `q3.py`.

Neatness Points: Five points will be given for neatness. You will receive these points as long as we don't have a hard time reading your solutions or understanding the structure of your code.

Late Submission: 5% of the marks will be deducted for each day late, up to a maximum of 6 days. After that, no submissions will be accepted.

Homeworks are individual work.

1. **Gradient Descent with Momentum. [50 points]** *Requires only Lecture 1.*

   In lecture, we derived the closed-form dynamics of one optimization algorithm, namely gradient descent. In this problem, you'll analyze the dynamics of another optimization algorithm frequently used to train neural nets: gradient descent with momentum. The idea is that rather than directly updaing the parameters (i.e. position) using the gradient, you instead maintain a velocity vector, and update the velocity opposite the gradient as if it were a force. The update rule is:

$$\begin{aligned}\mathbf{v}^{(k+1)} &\leftarrow \beta \mathbf{v}^{(k)} - \alpha \nabla \mathcal{J}(\mathbf{w}^{(k)}) \\ \mathbf{w}^{(k+1)} &\leftarrow \mathbf{w}^{(k)} + \mathbf{v}^{(k+1)}\end{aligned}$$

   We'll denote the initial parameters as $\mathbf{w}^{(0)}$, and the initial velocity is $\mathbf{v}^{(0)} = \mathbf{0}$. The two hyperparameters are α and β. Here, α is a learning rate, just like in gradient descent. Also, β is a damping parameter which you can think of as viscosity. Sensible values of β are between 0 and 1, with smaller values corresponding to more damping; $\beta = 0$ corresponds to ordinary gradient descent, while $\beta = 1$ is an undamped system. (So physically speaking, ordinary gradient descent resembles a particle moving through a thick fluid.)

   Your job is to derive the closed form dynamics for a convex quadratic cost function:

$$\mathcal{J}(\mathbf{w}) = \tfrac{1}{2}\mathbf{w}^\top \mathbf{H}\mathbf{w},$$

   where $\mathbf{H}$ is symmetric and PSD.

   (a) **[10 points]** By reasoning about invariance, argue that we may assume WLOG that $\mathbf{H}$ is a diagonal matrix. *(See NNTD Chapter 1 for discussion of what's required to make assumptions WLOG. You will probably need to make an inductive argument.)*

[1] `https://markus.teach.cs.toronto.edu/csc2541-2021-01/`

(b) **[5 points]** For diagonal $\mathbf{H}$ with diagonal entries h_j, each coordinate j evolves independently as:

$$\begin{aligned} v_j^{(k+1)} &\leftarrow \beta v_j^{(k)} - \alpha \frac{\partial \mathcal{J}}{\partial w_j} \\ &= \beta v_j^{(k)} - \alpha h_j w_j^{(k)} \\ w_j^{(k+1)} &\leftarrow w_j^{(k)} + v_j^{(k+1)} \end{aligned}$$

From here on out, we'll drop the j subscripts to avoid clutter (and you may do so in your answer as well).

To help analyze the dynamics, we'll first write this as a linear recurrence: $\mathbf{z}^{(k+1)} = \mathbf{B}\mathbf{z}^{(k)}$, where $\mathbf{z}^{(k)} = (w^{(k)}\ v^{(k)})^\top$. Find the matrix $\mathbf{B}$.

(c) **[5 points]** Find the eigenvalues of $\mathbf{B}$. (You can do this by hand or using a symbolic algebra package.)

Hint: For this part and later parts, you may find it helpful to give some of your answers in terms of $\gamma = 1 + \beta - \alpha h$.

(d) **[5 points]** There is a certain threshold T (depending on α and β) such that directions with curvature $h > T$ oscillate while directions with curvature $0 < h < T$ approach 0 monotonically. The former are known as *underdamped*, and the latter are known as *overdamped*. If $h = T$, this is known as *critically damped*. Find T.

Note: this is a good place to check your work by making sure your solution agrees with the result of actually running the algorithm.

(e) **[10 points]** Now suppose we're given that minimum and maximum curvature are $h_{\min}$ and $h_{\max}$, respectively. Assume the hyperparameters are chosen such that $\alpha \leq h_{\max}^{-1}$ and $0 \leq \beta \leq 1$. In this part, you will determine if the algorithm converges, and if so, the rate of convergence. (The rate of convergence is $-\log|\max_j \rho(\mathbf{B}_j)|$, where $\rho(\mathbf{B}_j)$ is the *spectral radius* of $\mathbf{B}_j$, i.e. the maximum norm of any eigenvalue.)

i. Determine the spectral radius for a given h. (You will need to separately consider the underdamped and overdamped cases.)

ii. Show that the spectral radius is a monotonically decreasing function of h for $h_{\min} \leq h \leq h_{\max}$.

Note: This implies the rate of convergence is determined by the behavior in the minimum curvature direction. Note that the rate implicitly depends on the high curvature directions as well since we imposed a constraint on α based on $h_{\max}$.

iii. We already assumed $0 \leq \alpha \leq h_{\max}^{-1}$ and $0 \leq \beta \leq 1$. Determine what additional conditions (if any) are necessary for convergence, and give a formula for the rate of convergence.

iv. To help us check the correctness of your formula, please evaluate it explicitly for $(h_{\min} = h_{\max} = 0.3, \alpha = 0.2, \beta = 0.9)$ and $(h_{\min} = h_{\max} = 0.3, \alpha = 0.2, \beta = 0.5)$. Give your answers to three significant digits.

Hint: it is possible to check your answer by plotting the iterates on a log plot and making sure the slope is consistent with your answer.

(f) **[5 points]** Suppose $h_{\min} = 10^{-3}$ and $h_{\max} = 1$. If we run ordinary gradient descent, the weights converge at a rate of approximate $\kappa^{-1} = 10^{-3}$, where $\kappa = h_{\max}/h_{\min}$.

Make a surface plot of the rate of convergence as a function of α and β. Based on your plot, exhibit a pair of values (α, β) such that gradient descent with momentum converges significantly faster than gradient descent. You don't need to justify how you arrived at these values.

(g) **[10 points]** *(This part can be solved independently of the rest of this question.)* In lecture, we showed that gradient descent applied to a linear regression problem

$$\mathcal{J}(\mathbf{w}) = \frac{1}{2N}\|\mathbf{\Phi w} - \mathbf{t}\|^2$$

converges to the minimum norm solution

$$\mathbf{w}^{(\infty)} = \mathbf{\Phi}^\dagger \mathbf{t},$$

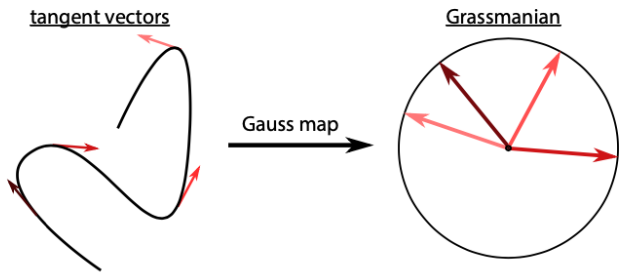
assuming $\mathbf{w}^{(0)} = \mathbf{0}$. (For simplicity, we are assuming there is no bias parameter.) Does gradient descent with momentum also converge to the minimum norm solution (assuming it converges)? Justify your answer.

2. **Computing the Grassmannian Length. [25 points]** *You can solve this problem based on Lecture 2, but Lecture 3 may provide some useful context.*

One of the recurring questions in this course will be: how nonlinear is a given neural network? One way to investigate this question is to take a straight line in input space (or weight space), and measure the complexity of its image is in output space. The Grassmannian length of a curve is one such measure of its complexity. Your job is to write a function to numerically compute the Grassmannian length of a curve.

We'll represent the curve as a mapping $\mathbf{x} : \mathbb{R} \to \mathbb{R}^n$, so that the position is represented by $\mathbf{x}(\theta)$, where θ is a scalar-valued parameter in $[\theta_{\min}, \theta_{\max}]$ which you can think of as representing "time". We assume the mapping $\mathbf{x}$ is continuously differentiable. The *velocity* at a given point is defined as $\mathbf{v}(\theta) = \mathbf{x}'(\theta)$. (The prime notation represents differentiation.) We assume $\mathbf{v}(\theta) \neq \mathbf{0}$ everywhere along the curve.

The *Gauss map* takes a point on the curve and maps it to its velocity vector, normalized to be a unit vector, i.e. $\mathcal{G}\mathbf{x}(\theta) = \hat{\mathbf{v}}(\theta) = \mathbf{v}(\theta)/\|\mathbf{v}(\theta)\|$. (Here we are using operator notation, i.e. $\mathcal{G}$ is an operator applied to the curve. This takes precedence over function application, so $\mathcal{G}\mathbf{x}(\theta)$ denotes the curve $\mathcal{G}\mathbf{x}$ evaluated at θ.) Notice that $\mathcal{G}$ maps the curve to the unit sphere in $\mathbb{R}^n$. This mapping is illustrated in the following figure[2]:



Hopefully in calculus class you learned how to measure the length of a curve:

$$\mathcal{L}(\mathbf{x}) = \int_{\theta_{\min}}^{\theta_{\max}} \|\mathbf{v}(\theta)\| \, \mathrm{d}\theta.$$

[2]The figure is taken from Poole et al., "Exponential expressivity in deep neural networks through transient chaos."

The *Grassmannian length* $\mathcal{L}_{\mathcal{G}}(\mathbf{x})$ of a curve $\mathbf{x}$ is simply the length of its Gauss map, i.e. $\mathcal{L}_{\mathcal{G}}(\mathbf{x}) = \mathcal{L}(\mathcal{G}\mathbf{x})$.

(a) **[5 points]** First we'll check that the definition is reasonable. One sanity check is that our measure of complexity should be invariant to rescaling of the output space. Show that for $\mathbf{y}(\theta) \triangleq \alpha\mathbf{x}(\theta)$, then $\mathcal{L}_{\mathcal{G}}(\mathbf{y}) = \mathcal{L}_{\mathcal{G}}(\mathbf{x})$. (You only need to provide an informal argument, not a formal proof.)

(b) **[5 points]** We'd also like our definition to be invariant to reparameterization of the curve. Let $f : \mathbb{R} \to \mathbb{R}$ be a monotonically increasing function mapping $[t_{\min}, t_{\max}]$ to $[\theta_{\min}, \theta_{\max}]$. Define the reparameterized curve $\tilde{\mathbf{x}}(t) \triangleq \mathbf{x}(f(t))$. Show that $\mathcal{L}_{\mathcal{G}}(\tilde{\mathbf{x}}) = \mathcal{L}_{\mathcal{G}}(\mathbf{x})$. (You only need to provide an informal argument, not a formal proof.)

(c) **[10 points]** Write a `JAX` function `grassmannian_length` which takes in a function representing a curve and computes its Grassmannian length. Starter code is available as `q2.py`. Your code will probably include two calls to `jax.jvp` and one call to `scipy.integrate.quad`.

Include your completed `q2.py` with your MarkUs submission. Additionally, to convince us of the correctness of your implementation, include the outputs of `compute_lengths` in your main writeup PDF.

Hint: we recommend decomposing your solution into a function that takes a curve and returns its Gauss map, and another function that computes the length of a curve. A sanity check of your implementation is that the Grassmannian length of a circle is 2π regardless of the radius.

(d) **[5 points]** Your implementation from part (c) can be applied to a neural net as long as the activation functions are continuously differentiable. Normally in deep learning, we ignore differentiability requirements and compute gradients of (say) ReLU networks, and everything works fine. But in this case, applying your implementation to a ReLU network would give nonsensical results. Briefly explain what would go wrong.

3. **Path Energy and Geodesics. [20 points]** *This problem requires up through Lecture 3.*

In Euclidean space, the shortest path between any two points $\mathbf{x}_0$ and $\mathbf{x}_1$ is a straight line, $\gamma(t) = (1-t)\mathbf{x}_0 + t\mathbf{x}_1$. A closely related fact is that the straight line is the path which minimizes the following *path energy* functional:

$$\mathcal{E}(\gamma) = \int_0^1 \|\gamma'(t)\|^2 \, \mathrm{d}t.$$

One way to understand this functional is to imagine a path with a finite number of small hops, and adding up the squared norms of each of the hops. The path energy is the continuous limit of this (normalized such that the limit is nonzero):

$$\mathcal{E}(\gamma) = \lim_{K\to\infty} K \sum_{k=0}^{K-1} \|\gamma(\tfrac{k+1}{K}) - \gamma(\tfrac{k}{K})\|^2.$$

It's possible to define path energies using a metric other than the Euclidean one. Minimizing these energies us a generalization of straight lines known as *geodesics*.

Consider the space of univariate Gaussian distributions, parameterized by $\boldsymbol{\theta} = (\mu, \sigma)$, where μ is the mean and σ is the standard deviation. We can define the path energy using the Fisher-Rao metric:

$$\mathcal{E}(\gamma) = \int_0^1 \gamma'(t)^\top \mathbf{F}_{\gamma(t)} \gamma'(t) \, \mathrm{d}t,$$

where $\mathbf{F}_{\boldsymbol{\theta}}$ denotes the Fisher information matrix at $\boldsymbol{\theta}$. Analogously to the Euclidean case, this can be seen as the continuous limit of a sum of KL divergences over finite hops:

$$\mathcal{E}(\gamma) = \lim_{K \to \infty} K \sum_{k=0}^{K-1} \mathrm{D}_{\mathrm{KL}} \left(p(\cdot\,; \gamma(\tfrac{k+1}{K})) \,\|\, p(\cdot\,; \gamma(\tfrac{k}{K})) \right).$$

Recall that Gaussians are a kind of exponential family, so there two canonical parameterizations: the natural parameters $\boldsymbol{\eta} = (h, \lambda)$,

$$h = \lambda \mu$$
$$\lambda = \sigma^{-2}$$

and the moments $\boldsymbol{\xi} = (\mu, s)$,

$$s = -\tfrac{1}{2}(\mu^2 + \sigma^2).$$

You don't need to prove this, but it turns out the path energy is the same no matter which coordinate system you compute it in.

We'll consider three paths in the space of Gaussians:

- The *linear path*, which is a straight line in (μ, σ) space. I.e., $\boldsymbol{\theta}(t) = (1-t)\boldsymbol{\theta}_0 + t\boldsymbol{\theta}_1$.
- The *geometric averages path*, which linearly averages the natural parameters: $\boldsymbol{\eta}(t) = (1-t)\boldsymbol{\eta}_0 + t\boldsymbol{\eta}_1$. (The name comes from the fact that this path is equivalent to taking geometric averages of the initial and final PDFs.)
- The *moment averages path*, which linearly averages the moments: $\boldsymbol{\xi}(t) = (1-t)\boldsymbol{\xi}_0 + t\boldsymbol{\xi}_1$.

Your job is to compute the path energies of these paths and to approximate the geodesic. The starter code is given in `q3.py`.

(a) [**5 points**] Write a function `path_energy` which takes in a path (given as a function mapping t to $\boldsymbol{\theta}$) and computes the energy of that path in the Fisher-Rao metric. I recommend using `scipy.integrate.quad`.

   i. Make sure your implementation is included in `q3.py`.

   ii. Compute the energy of the linear path, the geometric averages path, and the moment averages path between $\boldsymbol{\theta}_0 = (0, 1)$ and $\boldsymbol{\theta}_1 = (5, 2)$ by calling the provided function `compute_path_energies`. Give the output in your report.
   Hint 1: Remember that your code will run much faster if you JIT-compile it using `jax.jit`.
   Hint 2: To help you check your answer, two of the paths turn out to have the same energy. This is a deep and surprising fact. While it can be proved in only a few lines, I haven't yet found a good intuitive explanation.

(b) [**10 points**] The geodesic between two points $\boldsymbol{\theta}_0$ and $\boldsymbol{\theta}_1$ is the path connecting those points which minimizes the energy functional. In this question, you'll approximately solve for the geodesic by optimizing over a restricted class of paths, namely *linear splines*. Linear splines are piecewise linear paths, and the boundaries between linear segments are known as *knots*.

Write a function `optimize_spline_path` which numerically optimizes the knot locations to minimize the path energy. You should use at least 5 knots other than the endpoints, equally spaced (though larger numbers of knots may take longer to optimize). Initialize the knots so that the path is the same as the linear one. You will probably want to use `scipy.optimize.minimize` to do the optimization.

Hint: Unfortunately, you can't compute the gradients with respect to the knot locations just by calling `jax.grad` on your energy function, since JAX can't differentiate through SciPy routines. So you will need to write your own function to compute the gradient. Fortunately, there is a way to apply autodiff to the rest of the model, so that, e.g., you don't need to manually compute derivatives of the Fisher information. You will probably want to use `scipy.integrate.quad_vec`.

Don't forget to JIT-compile your code for efficiency. Solving this optimization problem to high accuracy can be very time consuming. For this question, we are only interested in the qualitative behavior, so you can set high error tolerances for the integration routines and limit the SciPy optimizer to 10 BFGS iterations.

i. Make sure your implementation is included in `q3.py`.
ii. Plot the linear path, the geometric averages path, the moment averages path, and your optimized spline path, by calling the provided function `plot_paths`. Include the output in your report.

(c) [**5 points**] Consider the shape of your optimized spline path and how it relates to the linear one. Give an intuitive explanation for why the spline path has a smaller energy. (We're looking for an explanation in terms of the path itself, not just the fact that you're optimizing over a set that includes the linear path.)

Hint: consider the above interpretation of the path energy as the limit of a sum of KL divergences.