Homework 7:

Instructions

Please submit a .qmd file along with a rendered pdf to the Brightspace page for this assignment. You may use whatever language you like within your qmd file, I recommend python, julia, or R.

Problem 1: Comparing Optimization Algorithms

Consider the function $f(x, y) = (1 - x^2) + 100(y - x^2)^2$, which has a global minimum at $x = 1$, $y = 1$. For this problem, you are going to explore using different optimization algorithms to find the global minimum, to gain an intuitive understanding of their different strengths and weaknesses.

(a) Make a contour plot of this function. You should observe a that the contour lines are "banana-shaped" around the global minimum point, which lies in a deep valley. In technical terms, we would say that the gradient of this function is strongly anisotropic, a fact that can cause slow or no convergence for optimization algorithms.

(b) In the code chunk below I have python code for three different optimization algorithms,(1) **stochastic gradient descent**; (2) **stochastic gradient descent with momentum**, and (3) **ADAM (ADAptive Moment Estimation)**. Starting at the initial point $x = -4$, $y = -2$, use each algorithm to find the minimum of the function f given above. Start with a learning rate of $\kappa = 10^{-4}$ for all three algorithms, and run the algorithm for 10^5 timesteps. For this problem, maintain the noise level at 0. Plot the trajectories of each algorithm and the log base 10 of the error rate as a function of the time step. What do you notice about the performance of the difference algorithms, both in terms of convergence speed and ultimate accuracy?

(c) Perform the same experiment for the learning rate $\kappa = 10^{-3}$, only comparing ADAM and gradient descent with momentum. You will likely observe that one of the methods does not converge, keep the same range of values for your trajectory/contour plot as you did in (b). Which method worked better with $\kappa = 10^3$?

(d) Now perform a comparison between ADAM with $\kappa = 10^{-2}$ against gradient descent with momentum using $\kappa = 10^{-4}$. What are the trade-offs between the two methods for these values of the learning rate?

Problem 2: Escaping a Saddle

One of the challenges of optimization of functions of many variables (which we sometimes call "high-dimensional" even if they output a single number/are scalar-valued) is that the estimate of the optimum can become trapped in a local minimum or spend a large amount of time near a saddle point. It is generally not possible to design a method that can always escape a local minimum, but fortunately many machine learning researchers believe that escaping from saddle points is a more relevant issue in real world problems because saddle points are much more common for high-dimensional functions. One of the advantages of stochastic gradient descent over regular gradient descent relates to avoiding saddle points. When training neural networks, the stochasticity of stochastic gradient descent originates from the fact that the algorithm takes steps without based on an estimation of the gradient of the loss function from a mini-batch of data, which induces noise in the gradient. To demonstrate this for a function of a few variables, we have included an explicit noise term in the the solver code that we specified above. For this problem we are going to the gradient descent algorithm and gradient descent with momentum, and explore how they behave when the trajectory is near a saddle-point.

We will use the function $f(x, y) = -x^2 + y^2$. This function has a saddle point at $x = 0$ and $y = 0$. This point is also unbounded below, so no algorithm we use will converge to a finite global minimum. You should think of a saddle point as point on a surface which is locally flat, which isn't the top of a hill or the bottom of a valley. This means that there are some directions from the saddle point where the function will eventually increase, and some directions where it will eventually decrease, just like a saddle which curves up both fore and aft and curves down to the left and right (an alternative would be to imagine a Pringles chip instead).

(a) Either write code to use the gradient descent solvers on the saddle function or adapt the code for the gradient of the function in part (a). Set the noise equal to 0 and the initial point of the trajectory to $x = -2$, $y = 0$. Use gradient descent to minimize the function, using $N = 10^5$ time steps. Plot the log base 10 of the norm of (x, y) over time and show that it converges to the saddle point.

In the real world it would be very unlikely to have the exact starting value $y = 0$, which is what causes the the solver to get stuck, and had y been slightly perturbed from 0 the trajectory would have escaped the saddle point. However in real problems the time to escape can still be quite large and the number of saddle points encountered during learning may also be quite large, so this can impact the overall learning rate.

(b) Perform additional experiments, with values of the noise parameter of 0.1, 0.5, and 1.5. Again plot the trajectories and the log base 10 of the norm of (x, y) over time. For these

problems exit the minimization routine when the absolute value of $|y| \geq 1$, so that you do not have trajectories blowing up to infinity.

Problem 3: Shallow Nets and MNIST

For this exercise, we will work on one of the standard model problems in Machine Learning, classifying handwritten digits. We will use an adaptation of the neural network code from your reading assignment to `pytorch`, which is one of the leading frameworks for training neural networks. `pytorch` is fairly flexible, you can use it with the CPU on your personal computer, with GPUs, and even on computing clusters. If you have trouble getting `pytorch` to work on your own computer I recommend trying in on google colab, or alternatively I can provide you with similar code written in pure numpy (or you are welcome to develop your own implementation).

First, you should acquire the MNIST dataset. This can be downloaded automatically using `pytorch` via the following code chunk:

```
# Here is some code that automatically downlaods the MNIST data. Technically it will also rea
# data in if you have already downloaded and the path points to the folder where you have the
# There will be 4 binary files which together contain the testing and training examples and t
# for the testing and training examples.

from torchvision import datasets, transforms



# Load MNIST


# transform defines a function which takes an image file, converts the analog bits into float
# numbers (it's a literal image file in the data), and then flattens the file. Each image is
# so at the end we get a 784x1 vector

transform = transforms.Compose([transforms.ToTensor(), transforms.Lambda(lambda x: x.view(-1)

# The first line downloads the entire MNIST dataset to the data directory (or whereever you w
# If the data is already there, this won't download it. THis downloads both the training and
# the transform keyword applies the transform defined above, the train dataset has 60,000 exa
# the test dataset has 10,000 examples. The train and test data is loaded in the variables.

train_dataset = datasets.MNIST('data/', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST('data/', train=False, transform=transform)
```

We are going to train a simple neural network to classify the MNIST images. The neural network has an input layer of 784 neurons (one for each pixel), a 30 neuron hidden layer, and a 10 neuron output layer. The outputs of the neural network will add to 1 and represent a probabilistic prediction of which digit the input is. We will initially use sigmoidal neurons to process the inputs. In the code chunks below we have code which implements a neural network to solve this classification problem:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

# Apologies for the deviation into object oriented programming. We are defining a neural netw
# dense layers and contains a routine to
```

Problem 4: Deep Nets: Overcoming Gradients