Assignment (5)

Artificial Intelligence and Machine Learning (24-787 Fall 2022)

Due Date: 12/01/2022 (Thursday) @ 11:59 pm EST

In case a problem requires programming, it should be programmed in Python. In Programming, you should use plain Python language, unless otherwise stated. For example, if the intention of a Problem is familiarity with numpy library, it will be clearly noted in that problem to use numpy. Please submit your homework through Gradescope.

Submissions: There are two steps to submitting your assignment on Gradescope:

1. HW05 Writeup: Submit a combined pdf file containing the **answers to theoretical questions** as well as the **pdf form of the FILE.ipynb notebooks**.

- Convert the jupyter notebook to a pdf file. Ensure that the submitted notebooks have been run and the cell outputs are visible - **Hint:** Restart and Run All option in the Kernel menu. **Make sure all plots are visible in the pdf.**
- If an assignment has theoretical and mathematical derivation, scan your handwritten solution and make a PDF file.
- Then concatenate them all together in your favorite PDF viewer/editor. The file name (FILE)should be saved as **HW-assignmentnumber-andrew-ID.pdf**. For example for assignment 1, my FILE = HW-1-andrewid.pdf
- Submit this final PDF on Gradescope. During submission, it will prompt you to select the pages of the PDF that correspond to each question, **make sure to tag the questions correctly!**

2. HW05 Code: Submit a ZIP folder containing the FILE.ipynb notebooks for each of the programming questions. The ZIP folder containing your iPython notebook solutions should be named as HW-assignmentnumber-andrew-ID.zip

You can refer to Numpy documentation while working on this assignment. **ANY** deviations from the submission structure shown above would attract penalty to the assignment score. Please use Piazza for any questions on the assignment.

PROBLEM 1

Support Vector Machines **[30 points]**

In this problem, you'll practice to solve a classification problem using SVM. In class, we have seen how to formulate SVM as solving a constrained quadratic optimization problem. Now, you will implement an SVM in the primal form. Conveniently, the **cvxopt** module in python provides a solver for constrained quadratic optimization problem, which does essentially all of the work for you. This solver can solve arbitrary constrained quadratic optimization problems of the following form:

$$\begin{aligned} \arg\min_{\mathbf{z}} \quad & \frac{1}{2}\mathbf{z}^T\mathbf{Q}\mathbf{z} + \mathbf{p}^T\mathbf{z} \\ \text{s.t.} \quad & \mathbf{G}\mathbf{z} \leq \mathbf{h} \end{aligned} \tag{1}$$

a) (Programming problem) You are given a data file **clean_lin.txt**. The first two columns are coordinates of points, while the third column is label.

Now let's implement the following quadratic optimization problem:

$$\begin{aligned} \arg\min_{\mathbf{w},b} \quad & \frac{1}{2}||\mathbf{w}||^2 \\ \text{s.t.} \quad & y_i(\mathbf{w}^T\mathbf{x}^{(i)} + b) \geq 1, \quad (i = 1, 2, \cdots) \end{aligned} \tag{2}$$

To do this, you have to find how to turn this constrained optimization problem into the standard form shown in (1). Things you should keep in mind: which variable(s) does **z** need to represent? What do you need to construct **Q**, **p**, **G** and **h**?

Hint: **z** should be 3×1. **G** should be $n \times 3$, where n is the number of training samples.

Train the linear SVM on the data using ***cvxopt.solvers.qp(Q,p,G,h)***. Plot the decision boundary and margin boundaries. You should have a plot similar to Fig. 1

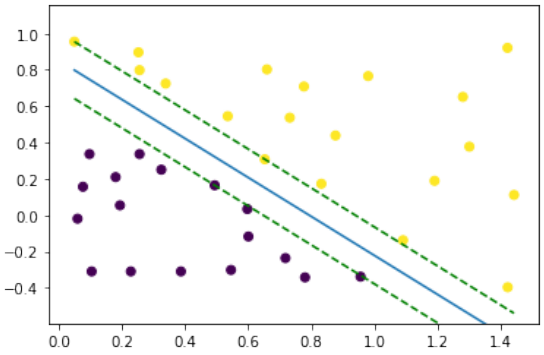

Fig. 1: Expected decision boundary for the linearly separable dataset.

b) (Programming problem) Now let's go ahead to solve a linearly non-separable case using SVM. Load the training data in **dirty_nonlin.txt**.

As discussed in the lecture, introducing slack variables for each point to have a soft-margin SVM can be used for non-separable data. The soft-margin SVM, which has following form:

$$\begin{aligned} \arg\min_{\mathbf{w},b,\xi} \quad & \frac{1}{2}||\mathbf{w}||^2 + C\sum_i \xi_i \\ \text{s.t.} \quad & y_i(\mathbf{w}^T\mathbf{x}^{(i)} + b) \geq 1 - \xi_i, \quad (i = 1, 2, \cdots) \\ & \xi_i \geq 0, \quad (i = 1, 2, \cdots) \end{aligned} \tag{3}$$

At this point, use $C = 0.05$ in your code, but keep that as a variable because you will be asked to vary C in the subsequent questions. Again, you want to think about what the terms in the standard formulation represent now.

Hint: The problem is still quadratic in the design variables. So your solution, if found, will be the global minimum. **z** should be $(n + 3) \times 1$. **G** should be $2n \times (n + 3)$. If you construct your design vector as $[w_1, w_2, b, \xi_1, \cdots, \xi_n]$, you shall see that **G** can be constructed by putting together four sub matrices (upper left $n \times 3$, lower right $n \times n$ etc.).

Finally, plot the decision boundary and margin boundaries. You should expect to have a plot similar to Fig. 2.

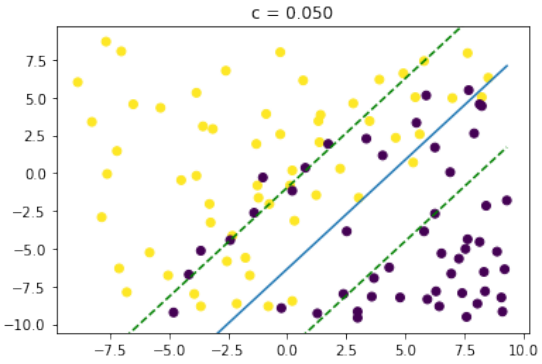

Fig. 2: Decision boundary for the linearly non-separable dataset.

c) (Programming problem) Use your code in **b)** to draw 4 plots, each corresponding to different values of $C = [0.1, 1, 100, 1000000]$. Discussion your observations of the decision margins.

(Problem 2 on next page)

PROBLEM 2

Physics-Informed Neural Networks (PINNs) **[40 points]**

In this problem, you'll get some hands-on experience on implementing physics-informed neural networks (PINNs) to solve partial differential equations (PDEs). With a PINN, our goal is to be able to input co-ordinates in time and space to a neural network, which will output the value of the PDE solution function at those specific points in time and space. To do so, we'll use a loss function that combines a loss based on physical constraints using the form of the PDE directly ($\mathcal{L}_f$) with a loss based on observed data at the boundary and initial conditions ($\mathcal{L}_\mathcal{D}$) to train a neural network. For this problem, our goal is to build a network that learns the solution of a 1-D PDE called the **Burgers' Equation**. Versions of the 1-D Burgers' equation are found in fluid mechanics, traffic flow, and acoustics. The 1-D Burgers' equation is defined as:

$$u_t = -u\frac{\partial u}{\partial x} + \nu\frac{\partial^2 u}{\partial x^2}, \quad x \in [-1, 1], \quad t \in [0, 1] \tag{4}$$

where $u(x, t)$ is the solution of the equation as a function of time t and space x, and ν is a constant. To comply with the notation used for PINNs, we can rewrite the PDE as:

$$f := u_t + uu_x - \nu u_{xx}, \quad x \in [-1, 1], \quad t \in [0, 1] \tag{5}$$

where f is the residual of the PDE solution.

Therefore, the physics loss can be written as:

$$\mathcal{L}_f := \frac{1}{N_f}\sum_{i=1}^{N_f} \|f(t_f^i, x_f^i)\|^2 \tag{6}$$

where (t_i^f, x_i^f) specify the **collocation points** where the residual $f(t, x)$ is calculated based on the predictions for u. The gradients needed to calculate the residual, u_x, u_t, and u_{xx} can be obtained as a byproduct of the neural network backpropagation process. In addition to this physics loss, the initial and/or boundary conditions are sampled as $\{(t_\mathcal{D}^i, x_\mathcal{D}^i, u^i)\}_{i=1}^{N_\mathcal{D}}$. For the points lying on the initial or boundary conditions, we use a mean squared error (MSE) loss to make sure the output of the PINN at these points satisfies the prescribed conditions.

$$\mathcal{L}_\mathcal{D} := \frac{1}{N_\mathcal{D}}\sum_{i=1}^{N_\mathcal{D}} \|u_{pred}(t_\mathcal{D}^i, x_\mathcal{D}^i) - u_{actual}^i\|^2 \tag{7}$$

The initial and boundary conditions are given by

$$\begin{aligned} u(t = 0, x < 0) &= 0 \\ u(t = 0, x >= 0) &= 1 \\ u(t, x = -1) &= 1 \\ u(t, x = 1) &= 1 \end{aligned}$$

The PDE will be solved with a mesh discretization of 256 uniformly spaced mesh elements, and 100 uniformly spaced time discretization points ($\Delta t = 0.01$). You can use the NumPy function **np.linspace()** to generate these vectors. Use $\nu = \frac{0.01}{\pi}$ for this problem. For faster training, you can complete this question using GPU access through Google Colaboratory.

Now let's see how we can implement this model. The main components of the PINN that you will encounter in this problem are as follows:

- **Neural Network**: First, we implement a neural network model to learn the solution $u(t, x)$ including several fully connected layers with nonlinear activation layers. The model gets two-dimensional input of (t, x) and predicts a scalar value for the solution u in that point.
- **Autograd**: To calculate the physics loss, we need to compute derivatives of solution in time and space as seen in Burgers' equation. In PyTorch, we can use **torch.autograd.grad** to get these derivatives. For example, the following expression calculates the gradient of u with respect to t, u_t:

```
u_t = torch.autograd.grad( u, t, grad_outputs=torch.ones_like(u),
retain_graph=True, create_graph=True)[0]
```

  Using autograd we can compute the terms in equation 5 and output f to be later used in computing loss.

- **Loss**: In the loss function, you can first compute the values of $f(t_f, x_f)$ to act as your physics-based loss based on your prediction for $u(t_f, x_f)$. Next, you can compute the MSE loss on the initial condition and boundary condition points, $u(t_\mathcal{D}, x_\mathcal{D})$ add this to the physics loss to get the total loss of the prediction. You can then backpropagate and update the model using this combined loss.
- **Optimizer**: PINNs use a nonlinear optimizer, **L-BFGS**, for improved convergence compared to gradient descent. The L-BFGS implementation has already been provided for you in the notebook.

You are provided with a template notebook which covers these different steps of implementing a PINN. Please follow these steps and implement the following parts:

a) (Programming Problem) - 5 points The first step towards implementing the PINN will be to define the initial conditions and the solution space. Following the steps in the notebook, define the initial and boundary conditions, and using **np.meshgrid()**, define 2-D arrays that store the possible combinations of x and t based on the solution domain. Plot the mesh array using the provided notebook, and the input condition. You can check your answers against the provided file **q2_data.npy**, which contains the exact solution to the Burgers' equation at the x and t co-ordinates used for this problem. Follow the subsequent steps in the notebook to reshape the data and convert it to Torch tensors.

b) (Programming Problem) - 10 points Next, we will construct a fully connected neural network to act as a PINN. Following the instructions in the notebook, complete the **FCNN** class to create a fully connected neural network in PyTorch with a tanh activation between each layer. This network should take in an input with two features, and output one feature. You may choose the number of hidden layers and the number of neurons per hidden layer.

c) (Programming Problem) - 10 points Finally, we will define the loss functions used to optimize the neural network. Following the instructions in the template notebook, complete the **net_u()** function, the **net_f()** function, and the **loss_func()** function. Train your network using the provided optimizer, your loss should be less than 1×10^{-4} at the end of the training process.

d) (Programming Problem) - 5 points Complete the **predict()** function, which predicts the solution to the PDE at a set of specified points. Report the error of the solution using the provided code snippet. Run the plotting code snippets to visualize your predictions compared to the ground truth data.

e) (Programming Problem) - 5 points Re-train your network with $\nu = 1$ and $\nu = 0.1$. Plot the predicted solution at $t = 1$ for $\nu = 1, 0.1$ and $\frac{0.01}{\pi}$. Briefly (1 sentence) describe the effect of the ν parameter on the behavior of the solution and model performance.

f) (Theoretical Problem) - 5 points Briefly (∼2 sentences) describe how PyTorch's AutoGrad functionality works, and why it is useful for implementing PINNs.