CS 419 Assignment

This assignment has 4 questions, for a total of 45 points.
Due Date: $11:59\text{PM}, 30^{th}$ March

Instructions

- All submissions will be auto-graded. So follow the submission instructions carefully.
- Do not import packages into the files which haven't been imported already.

1. Discrete Cosine Transform 5 points

Discrete Cosine Transform (DCT) is a widely used transformation technique in signal processing and data compression. For this question, we will use the 2-Dimensional DCT-II. Consider an image X of size $N_1 \times N_2$. The DCT of this image is also a matrix Y of size $N_1 \times N_2$ given by the formula

$$Y_{k_1,k_2} = 4\sum_{n_1=0}^{N_1-1}\sum_{n_2=0}^{N_2-1} X_{n_1,n_2}\cos\left(\frac{\pi}{N_1}(n_1+\frac{1}{2})k_1\right)\cos\left(\frac{\pi}{N_2}(n_2+\frac{1}{2})k_2\right)$$

The computation of Y can be succinctly captured by a series of matrix multiplications, resulting in a much faster implementation when coded in python. Implement the vectorized version of 2-D DCT in the function `vectorized_dct` in `vectorize.py`

2. Set Retrieval 10 points

Information retrieval can be defined as the task of obtaining the most relevant resources according to some requirements from a collection of resources. Document ranking is an important problem in this domain. We are given a collection of documents and a query. We need to order the documents according to their relevance to their query, i.e. the document most relevant to the query must be a the top and the document least relevant must be at the bottom. In this problem, each query is represented as a list of vectors $v_i \in R^w$. The length of each list is m. Each vector in the list can be considered the representation of a word in the query.

We are also given a collection of k documents where each lists n vectors $a_j \in R^w$. Again, each vector in the list can be considered the representation of a word in the document. The document at the i^{th} position has $docID = i, i = 0, 1, 2, 3 \ldots k-1$ Your task is to return a list of docIDs ordered according to relevance to the query. Relevance can be computed in two different ways, according to which the documents should be ranked.

(a) (5 points) The relevance R(d,q) between a document d and a query q is calculated as follows - For a vector v_i in the query, compute $\max_{a_j} v_i^T a_j$ over the vectors a_j in the document. Let the value be s_i. This denotes the maximum similarity a word in the query has with a word in the document. Now, sum all these values of s_i for each vector v_i in the query. This gives the relevance score R(d,q).

Complete the function `relevance_one` in `vectorize.py`

(b) (5 points) The relevance R(d,q) between a document d and a query q can also be calculated as follows

$$score(v_i, a_j) = -\sum_{k=1}^{w} RELU(v_i - a_j)_k$$

Note that you will have to compute the pairwise scores between all query vectors and document embeddings and add them to obtain the relevance score.

Complete the function `relevance_two` in `vectorize.py`

Note that each function accepts a set of queries and documents and should return the ranked docIDs. Additionally, they MUST be vectorised.

3. Classification Loss Functions 15 points

You have been given a synthetic dataset for classification. The dataset contains 2000 data points, each of which has 80 features. The shape of the dataset is 2000*81, where the last column indicates the output label (since it's a binary classification problem, the output is 0/1).
Keep dataset3.csv in the same folder as Q3.py
Complete the following functions:

- `def __init__(self, args, input_dim):`
  This is the constructor. It takes as input the args, `input_dim` variables. Use the args variable to specify all the hyperparameters of the model and the neural layers at your discretion.
- `def forward(self, data):`
  This implements the forward pass of the algorithm. Input data is batched data of shape `BATCH_SIZE * num_features` Returns predictions of shape `BATCH_SIZE * 1`
- def loss(self, pred, label):
  This implements the loss function. The inputs, pred and label, are each tensors of shape `BATCH_SIZE*1`. Pred contains model prediction outputs for each of the batched inputs. The label contains 0/1 values only. The function returns a single loss value. We will be implementing different variations of the loss function, as described below.
- `def evaluate(loader, model):`
  This implements the evaluation function for binary classification. Input loader is of `torch.data_utils.DataLoader` type. It is an ordered pair where the first item is the feature matrix of shape `BATCH_SIZE * num_features`, and the second item is the label tensor of shape `BATCH_SIZE * 1`. The function returns an evaluation score based on the following definition: it will be the ratio of the number of samples that are correctly classified/the total number of sample points in that batch.

We will be implementing the three loss functions as specified next. During executions, we need to specify the type of loss function using the args variable `model_type`.

Negative Log Likelihood (NLL)

The Binary Cross Entropy Function is defined as follows for a set of N data points:

$$L_{CE} = -\frac{1}{N}\sum_{i=1}^{N} t_i \log(p_i) + (1-t_i)\log(1-p_i)$$

Here, for the ith data point: t_i is the true label (0 for class 0 and 1 for class 1) and p_i is the predicted probability of the point belonging to class 1.

When the observation belongs to class 1, the first part of the formula becomes active, the second part vanishes, and vice versa in the case observation's actual class is 0. This is how we calculate the binary cross-entropy.

The probability scores for the forward pass can be computed using a Sigmoid function as follows:

$$S(z) = \frac{1}{1+e^{-z}} = \frac{e^z}{e^z+1} = 1 - S(-z)$$

z is the score of the item x as given by the neural model.

The sigmoid function outputs a $S(z) \in [0,1]$ and indicates the probability of how close to a class the item belongs (in the case of binary classification). Therefore, having a threshold of 0.5, the binary classification output $Class(x)$ can be formulated as

$$Class(x) = \begin{cases} 1 & \text{if } S(z) > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

Your model will output a sigmoid score for each input. Subsequently, these real-valued predictions will be converted to binary labels using $Class(x)$ function. Finally, the **accuracy** is computed using the binary predictions and binary labels and is defined as $\frac{\text{no. of label matches}}{\text{total no. of items in both labels}}$

SVM Loss

For ith data point, the Hinge Loss Function is given by:

$$L_{HL}(x_i, y_i) = \begin{cases} 1 - y_i(w^T x_i + b) & \text{if } 1 - y_i(w^T x_i + b)) > 0 \\ 0 & \text{otherwise} \end{cases}$$

where, y_i is the truth label for the corresponding input instance x_i and parameter w.
Here: $w, x_i \in \mathbb{R}^n$, $b \in \mathbb{R}$, $y_i \in \{-1, 1\}$

Note: you will have to change the output labels as (1,-1) rather than (1,0) as given in the dataset before calculating the hinge loss.

Here, L_{HL} can take any float value depending on the sign of y_i and $w^T x_i$. If both signs are the same (indicating correct class prediction), $L_{HL} = 0$. And if the signs are opposite (indicating misclassification), the value of L_{HL} increases. In other words, it finds the classification boundary that guarantees the maximum margin between the data points of the different classes.

Hence during evaluation, you need to calculate the number of items on each side of the classification boundary w.r.t to the actual classes they belong to. The formula is given as follows:

$$Class(x_i) = \begin{cases} 1 & \text{if } y_i(w^T x_i + b) \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

As before, the final **accuracy** is computed using the binary predictions of $Class(x)$ and binary labels and is defined as $\frac{\text{no. of label matches}}{\text{total no. of items in both labels}}$

Ranking Loss

The Ranking Loss is defined as follows:

$$L(P, N) = \sum_{p \in P, n \in N} max(0, n - p)$$

Here, P denotes the model predictions for the set of positive items labelled 1. N denotes the model predictions for the set of negative items labelled 0. We want to impose the constraint that the positive scores should be greater than the negative scores. The above functions enforce this.

Your model will output a score for each input. During Evaluation, we will count the number of times positive items are scored higher than negative ones. The higher the count, the better the model. The counting formula is specified below:

$$L(P, N) = \sum_{p \in P, n \in N} \mathbb{1}[p > n]$$

The final **evaluation** score is the output of the counting formula.

4. Regression 15 points

Consider a regression problem in which we are given a dataset $D = \{(x_i, y_i)\}_{i=1}^{N}$. The goal of linear regression is to find ω such that $f(x) = \omega^T x$ models y. We know the analytical solution minimising the L2 loss in linear regression to be

$$\omega^* = (X^T X)^{-1} X^T y$$

In this problem, we consider the gradient descent method of solving regression. We will consider different cases, one where the model overfits the dataset and one where the model underfits the dataset and will identify methods to tackle these situations.

(a) (5 points) You have been given a dataset in the dataset4.csv file. Each row in the file represents one data sample in the dataset. There are 1000 data samples, each with ten features and one label. First, split the dataset into the train and validation sets. You will use the validation split later to implement Early stopping. Next,

Complete the following functions:

- `def init_weights(self, xdim):`
  Complete this function to initialize the parameters of the model. You can initialize $\omega, b = 0$. Be sure to create the parameters in the specified shape, or the code will assert.
- `def forward(self, batch_x):`
  This implements the forward pass of the algorithm. Input data is batched data of shape `BATCH_SIZE × num_features`. Returns predictions of shape `(BATCH_SIZE, )`
- `def backward(self, batch_x, batch_y, batch_yhat):`
  This function implements one gradient update to the parameters of the model. The update equations to be implemented are:

$$\omega^{new} = \omega - \eta \frac{\partial L(y, \hat{y})}{\partial \omega}$$

$$b^{new} = \omega - \eta \frac{\partial L(y, \hat{y})}{\partial b}$$

  Where η is a learning rate that you play with and $L(\cdot, \cdot)$ is an appropriate loss function imposing L2 regularisation. This function should return ω^{new}, b^{new}.
- `def loss(self, y, y_hat):`
  This implements the loss function. The inputs y and `y_hat` are each tensors of shape`(BATCH_SIZE, )`.
- `def score(self, y, y_hat):`
  While loss measures how worse the model is doing, score measures how well your model performs. It is a metric that is higher the better that shall be tracked to perform early stopping. One popular scoring function for regression tasks is -ve of loss.

(b) (5 points) We typically train machine learning models across epochs. One epoch is a single pass over all the samples in the data. Each such pass over the data should ideally happen in random order. It is well-known that this randomness helps in better and sometimes faster convergence. So in this question, you will implement mini-batching so that each successive call returns a batch of instances. You will make use of yield in python to do this task.

- `def minibatch(trn_X, trn_y, size):`
  The training loop calls this function in each epoch. This should return yield a batch (x, y) examples of size as given in the size argument.

Additionally, while training our models, we must choose the number of training epochs to use. Too few training epochs will result in underfitting. On the other hand, too many training epochs will result in overfitting. Today we are implementing Early Stopping according to the following rules:

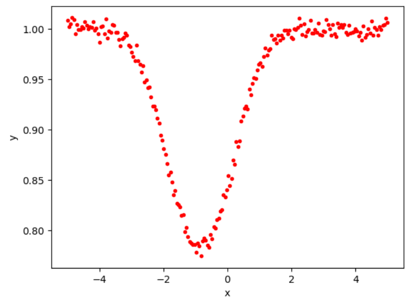
- At each epoch, we will be tracking the validation scores. The validation score will be the accuracy for classification tasks and the negative mean squared error for regression tasks.

- We remember the performance from the latest XX epochs (XX is set using the 'patience' parameter). Suppose the improvement in validation score does not exceed a certain delta D (D is set using the 'delta' parameter) before XX epochs are up. In that case, we stop training and roll back to the best model in the patience window.

Complete the following functions:

- `check(self,curr_score,model,epoch)`:
  Takes the current validation score of the model, model object at the current state of training and epoch number of the current training process as input and returns `self.should_stop_now`, which is a bool flag to decide whether training should early stop at this epoch. Achieve this in the following way
  1. Uses `self.best_score` to keep track of the best validation score observed till now (while training). If the current score exceeds the best score, then the current model is stored as the best model
  2. Uses `self.num_bad_epochs` to keep track of the number of training epochs in which no improvement has been observed. If the number of such epochs exceeds patience, stop training.

(c) (5 points) Observe the following plot of x vs y. The data that produces it is given in `data.npy` inside the Q4 directory.



This data is not suitable for performing linear regression, even after the addition of features. Instead, it is known that the data comes from a function of the following form with some noise:

$$y = f(x) = w_4 \cdot \tanh\big(w_1 x + w_2 x^2 + w_3\big) + w_5$$

Along with it are given two files: `nonlinear_regression.py` and `print_result.py`. The first file implements fitting a function of the form given above to the data given. You have to complete the code wherever comments indicate it. For purposes of verification, the code produces 11 plots. If you've implemented everything correctly, these should show the predictions gradually moving towards the data. These plots are for you, and you need not include them in your final submission - but we won't penalise you if you do. Make sure that running `print_result.py` outputs two lists. These are the reproducible results you've obtained. We should obtain the same results on running your code as well.

NOTE: You need not perform any optimization. The fitting (optimizing) code is handled inside the train function. Notice the `nn.Parameter` declarations in the constructor for NLModule - these register declared variables as parameters in `nn.Module` - from which NLModule is derived. Once forward is implemented, the dependence of the output of the Model on these parameters and the `loss_function` are used to find the optimal parameter values.

Submission Instructions

The final directory structure should look like this

```
<ROLL_NUMBER>_A1
| - - - - Q1_Q2
| - - - - -|- - - - vectorize.py
| - - - - Q3
| - - - - -|- - - - Q3.py
| - - - - Q4
| - - - - -|- - - - assignment.py
| - - - - -|- - - - linear_bestValModel.pkl
| - - - - -|- - - - nonlinear_regression.py
| - - - - -|- - - - print_result.py
| - - - - -|- - - - losses.npy
| - - - - -|- - - - weights.npy
```

Replace `ROLL_NUMBER` with your own roll number. If your Roll number has alphabets, they should be in *"small"* letters. Submit a tar file `<ROLL_NUMBER>_A1.tar.gz` on moodle.