ECE421: Introduction to Machine Learning — Fall 2024

Assignment 2: Gradient Descent, Multiclass Logistic Regression, and K-Means

Due Date: Friday, October 18, 11:59 PM

Objectives

In this assignment, you will implement your own (small) version of `PyTorch` library in the `myTorch.py` file.

You will be implementing the following algorithms using `Python` and functions in the `NumPy` library only:

- four versions of stochastic gradient descent, namely, **SGD**, **Heavy-ball Momentum**, **Nestrov Momentum**, and **ADAM**,
- **Multiclass Logistic Regression**, and
- **K-Means** (Bonus)

You will also be asked to answer several questions related to your implementations.

To avoid any potential installation issue, you are encouraged to develop your solution using Google Colab notebooks.

Requirements

In your implementations, please use the function prototype provided (*i.e.* name of the function, inputs and outputs) in the detailed instructions presented in the starter-code and the remainder of this document. We will be testing your code using a test function that which evokes the provided function prototype. If our testing file is unable to recognize the function prototype you have implemented, you can lose significant portion of your marks. In the assignment folder, the following files are included in the `starter_code` folder:

- `myTorch.py`
- `util.py`
- `test_A2.py`

These files contain the test function and an outline of the functions that you will be implementing. You also need to submit a separate `PA2_qa.pdf` file that answer questions related to your implementations.

1 Gradient Descent

In the `myTorch.py` file, you will complete the `Optimizer` class implementation, which will be used in the `MultiClassLogisticRegression` class to train a logistic regression model. We implemented the constructor method, *i.e.*, the `__init__` method, of the `Optimizer` class for you in the starter code.[1]

1.1 `Optimizer.sgd` method

In this part, you should implement the `Optimizer.sgd` method. Specifically, you will be implementing the function listed below. For more instructions and hints, see the comments in the `myTorch.py` file.

- `sgd(self, gradient)`
  Function implementation considerations: This function computes the update vector that will be used by gradient descent algorithm (*i.e.*, $-\eta \nabla_{\underline{x}} f(\underline{x}_t)$, where η denotes the learning rate). Useful attributes for implementing this function is:

[1] The constructor is executed at the time of object creation and sets value to the object's attributes.

– `self.lr`: the learning rate.

For evaluation, we provide you the test functions `q1()`, `q2()`, `q3()`, and `q4()` in `tests_A2.py` file. Go over these test functions and the function `test_sgd` in the `tests_A2.py` file to better understand what these functions do and **answer the following question(s)** after running these tests. Write and save your answers in a separate `PA2_qa.pdf` file. Remember to submit this file together with your code.

1.1.a) The test function `q1()` runs your SGD implementation with four different learning rates to find $w^\star = \underset{w}{\arg\min} f(w) = 10w^2 + 20w + 10$. The starting point used by this test is $w_0 = 5$ and the maximum number of iteration is set to 20. This function reports the value of $w^\star$ together with some other information on your screen. `q1()` should show slow convergence for learning rates 0.001 and 0.005, and faster convergence with learning rates 0.01 and 0.05.

1.1.a.i) Describe the termination criteria used in the `test_sgd` function in the `tests_A2.py` file. (1 marks)

1.1.a.ii) Include the figures generated by `q1()` in your `PA2_qa.pdf` file. (1 marks)

1.1.a.iii) With learning rate $\eta = 0.05$, what would be the value of w_1, *i.e.*, after one iteration of SGD update. Show your mathematical process. If you implemented SGD correctly, the figures generated by `q1()` should verify your w_1. (1 marks)

1.1.b) The test function `q2()` runs your SGD implementation with four different learning rates to find $w^\star = \underset{w}{\arg\min} f(w) = 22w^2 + 44w + 22$. The starting point used by this test is $w_0 = 5$, and the maximum number of iteration is set to 35. This function reports the value of $w^\star$ together with some other information on your screen. With a correct implementation of SGD, `q2()` must show that SGD stopped after 35 iteration without meeting the convergence criterion with $\eta = 0.001$ and $\eta = 0.005$. However, with $\eta = 0.01$, you should be able to find the optimal solution with smaller number of iterations.

1.2.b.i) Include the figures generated by `q2()` in your `PA2_qa.pdf` file. (1 marks)

1.2.b.ii) When $\eta = 0.05$, SGD would fail to converge to the optimal solution. What causes such behavior? (1 marks)

1.1.c) The test function `q3()` runs your SGD implementation with four different learning rates to find $w^\star = \underset{w}{\arg\min} f(w) = 10w^2 + 10\sin(\pi w)$. The starting point used by this test is $w_0 = 5$, and the maximum number of iteration is set to 2000. This function reports the value of $w^\star$ together with some other information on your screen. With a correct implementation of SGD, `q3()` must show that SGD fails to converge to the global optimum point with these four learning rate values.

1.2.c.i) Include the figures generated by `q3()` in your `PA2_qa.pdf` file. (1 marks)

1.2.c.ii) In 1-2 sentences describe the behavior of SGD in `q3()` when $\eta = 0.001, 0.005$, and 0.01. Explain why SGD fails to find the global optimum point? (1 marks)

1.2.c.iii) In 1-2 sentences describe the behavior of SGD in `q3()` when $\eta = 0.05$. (1 marks)

1.1.d) The test function `q4()` runs your SGD implementation with four different learning rates to find $\underline{w}^\star = \underset{\underline{w}}{\arg\min} f(\underline{w}) = 2w_1^2 + 0.2w_2^2$. The starting point used by this test is $\underline{w}_0 = (3, 3)$, and the maximum number of iteration is set to 500. This function reports the value of $w^\star$ together with some other information on your screen.

1.2.d.i) Include the figures generated by `q4()` in your `PA2_qa.pdf` file. (1 marks)

1.2.d.ii) In 1-2 sentences describe the behavior of SGD in `q3()` when $\eta = 0.005$ and 0.01. How is this behavior related to the stretched nature of the function $f(\underline{w})$? (1 marks)

1.2.d.iii) In 1-2 sentences describe the behavior of SGD in `q3()` when $\eta = 0.03$. (1 marks)

The following is the mark breakdown for Part 1.1:

(i) Test file successfully runs the test `q1()`, `q2()`, and `q3()`: 3 marks

(ii) Test file successfully runs the test q4(): 2 marks

(iii) Outputs of all four tests are close to the expected output: 4 marks

(iv) Code content is organized well and annotated with comments: 1 marks

(v) Questions are answered correctly: 10 marks

1.2 `Optimizer.heavyball_momentum` and `Optimizer.nestrov_momentum` methods

In this part, you should implement the `Optimizer.heavyball_momentum` method. Note that the implementations of `Optimizer.heavyball_momentum` and `Optimizer.nestrov_momentum` are identical. This is due to the fact that the only difference between these two variants of SGD is in their input gradient vector. Specifically, you will be implementing the function listed below. For more instructions and hints, see the comments in the `myTorch.py` file.

- `heavyball_momentum(self, gradient)`
  Function implementation considerations: This function computes the update vector that will be used by gradient descent with heavy ball momentum (*i.e.*, $-\eta\nabla_{\underline{x}} f(\underline{x}_t) + \gamma v_{t-1}$, where η denotes the learning rate, γ denotes the momentum parameter, and v_{t-1} denotes the previous update vector). Useful attributes for implementing this function is:
  - `self.lr` and `self.gama`: the learning rate and the momentum parameter.
  - `self.v`: this attribute can be used to record the last momentum (*i.e.*, update) vector.

For evaluation, we provide you the test functions q5(), q6(), q7(), and q8() in `tests_A2.py` file. Go over these test functions and the function `test_momentum` in the `tests_A2.py` file to better understand what these functions do and **answer the following question(s)** after running these tests. Write and save your answers in a separate `PA2_qa.pdf` file. Remember to submit this file together with your code.

1.2.a) The test function q5() runs your Heavy-ball Momentum implementation with four different learning rates to find $w^\star = \arg\min_{w} f(w) = 10w^2 + 10\sin(\pi w)$. The starting point used by this test is $w_0 = 5$ and the maximum number of iteration is set to 2000. This function reports the value of $w^\star$ together with some other information on your screen.

1.2.a.i) Include the figures generated by q5() in your `PA2_qa.pdf` file. (1 marks)

1.2.a.ii) In 1-2 sentences, compare the performance of SGD with and without heavy-ball momentum by comparing the outcome of tests q3() and q5() (2 marks)

1.2.b) The test function q6() runs your Heavy-ball Momentum implementation with four different learning rates to find $\underline{w}^\star = \arg\min_{\underline{w}} f(\underline{w}) = 2w_1^2 + 0.2w_2^2$. The starting point used by this test is $\underline{w}_0 = (3, 3)$, and the maximum number of iteration is set to 500. This function reports the value of $w^\star$ together with some other information on your screen.

1.2.b.i) Include the figures generated by q4() in your `PA2_qa.pdf` file. (1 marks)

1.2.c) The test function q7() runs the Nestrov Momentum implementation with four different learning rates to find $w^\star = \arg\min_{w} f(w) = 10w^2 + 10\sin(\pi w)$. The starting point used by this test is $w_0 = 5$ and the maximum number of iteration is set to 2000. This function reports the value of $w^\star$ together with some other information on your screen.

1.2.c.i) Include the figures generated by q5() in your `PA2_qa.pdf` file. (1 marks)

[Note: You do not need to implement `Optimizer.nestrov_momentum`. We had already taken care of that method. Check out the `test_momentum` function in the `tests_A2.py` file to see how we use different gradient vectors for gradient descent with Heavy-ball and Nestrov Momentum.**]**

1.2.d) The test function `q8()` runs your Nestrov Momentum implementation with four different learning rates to find $\underline{w}^{\star} = \arg\min_{\underline{w}} f(\underline{w}) = 2w_1^2 + 0.2w_2^2$. The starting point used by this test is $\underline{w}_0 = (3, 3)$, and the maximum number of iteration is set to 500. This function reports the value of $w^{\star}$ together with some other information on your screen.

1.2.d.i) Include the figures generated by `q4()` in your `PA2_qa.pdf` file. (1 marks)

1.2.d.ii) In 1-2 sentences, compare the performance of Nestrov Momentum with the heavy-ball momentum by comparing the outcome of tests `q5()` and `q6()` with that of `q7()` and `q8()`. (1 marks)

The following is the mark breakdown for Part 1.2:

(i) Test file successfully runs the test `q5()`, `q6()`, `q7()`, and `q8()`: 4 marks

(ii) Outputs of all four tests are close to the expected output: 4 marks

(iii) Code content is organized well and annotated with comments: 1 marks

(iv) Questions are answered correctly: 7 marks

1.3 `Optimizer.adam` method

In this part, you will be implementing the fourth variation of gradient descent by implementing the `Optimizer.adam` method in the `myTorch.py` file. Adam is a variant of stochastic optimization that only requires first-order gradients with little memory requirement. Below is an excerpt from the paper "Adam: A Method for Stochastic Optimization," by Diederik P. Kingma and Jimmy Ba.

> [Adam] computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients; the name Adam is derived from adaptive moment estimation. [Adam] is designed to combine the advantages of two recently popular methods: AdaGrad , which works well with sparse gradients, and RMSProp, which works well in on-line and non-stationary settings.
>
> Some of Adam's advantages are that the magnitudes of parameter updates are invariant to rescaling of the gradient, its stepsizes are approximately bounded by the stepsize hyperparameter, it does not require a stationary objective, it works with sparse gradients, and it naturally performs a form of step size annealing.

Like the other three methods, `Optimizer.adam` should return the update vector by which you increment the model's weights. So, you should implement the following pseudo-code:

Listing 1: `Optimizer.adam` method. $\underline{a} \odot \underline{a}$ indicates the elementwise square of the vector $\underline{a}$. With β_v^t and β_m^t we denote β_v and β_m to the power t.

```
procedure sgd($\nabla_{\underline{w}} f(\underline{w}_t)$)
    $\underline{m}_{t+1} \leftarrow (1-\beta_m)\nabla_{\underline{w}} f(\underline{w}_t) + \beta_m \underline{m}_t$
    $\underline{v}_{t+1} \leftarrow (1-\beta_v)\nabla_{\underline{w}} f(\underline{w}_t) \odot \nabla_{\underline{w}} f(\underline{w}_t) + \beta_v \underline{v}_t$
    $\hat{\underline{m}}_{t+1} \leftarrow \frac{\underline{m}_{t+1}}{1-\beta_m^t}$
    $\hat{\underline{v}}_{t+1} \leftarrow \frac{\underline{v}_{t+1}}{1-\beta_v^t}$
    $t \leftarrow t+1$
    update $= -\frac{\eta \hat{\underline{m}}_{t+1}}{\sqrt{\hat{\underline{v}}_{t+1}}+\epsilon}$ \\ $\eta$ is the learning rate. The division and the square root are element-wise.
    return update
```

you will be implementing the function listed below. For more instructions and hints, see the comments in the `myTorch.py` file.

- `adam(self, gradient)`
  Function implementation considerations: This function computes the update vector that will be used by Adam. Useful attributes for implementing this function is:
  - `self.lr`, `self.beta_m`, `self.beta_v`, and `self.epsilon`

- `self.m`, `self.v`, and `self.t`: these attributes can be used to record the last first and second moment of the gradient and time step `t`.

Furthermore, useful functions in `NumPy` for implementing this method are:

- `square`: to find the element-wise square of a vector
- `sqrt`: to find the element-wise square root of a vector

For evaluation, we provide you the test functions `q9()`, `q10()`, and `q11()` in `tests_A2.py` file. Go over these test functions and the function `test_sgd` in the `tests_A2.py` file to better understand what these functions do and **answer the following question(s)** after running these tests. Write and save your answers in a separate `PA2_qa.pdf` file. Remember to submit this file together with your code.

1.3.a) The test function `q9()` runs your Adam implementation with four different learning rates to find $w^\star = \arg\min_{w} f(w) = 2w_1^2 + 0.2w_2^2$. The starting point used by this test is $\underline{w}_0 = (3, 3)$, and the maximum number of iteration is set to 500. This function reports the value of $w^\star$ together with some other information on your screen.

1.3.a.i) Include the figures generated by `q9()` in your `PA2_qa.pdf` file. (1 marks)

1.3.a.ii) In 1-2 sentences, compare the performance of adam with momentum method (heavy-ball or Nestrov) (2 marks)

1.3.b) The test function `q10()` runs your Adam implementation with four different learning rates to find $\underline{w}^\star = \arg\min_{\underline{w}} f(\underline{w}) = 1000(2w_1^2 + 0.2w_2^2)$. This function is the scaled version of the function used in test `q9()`. The starting point used by this test is $\underline{w}_0 = (3, 3)$, and the maximum number of iteration is set to 500. This function reports the value of $w^\star$ together with some other information on your screen.

1.3.b.i) Include the figures generated by `q10()` in your `PA2_qa.pdf` file. (1 marks)

1.3.b.ii) Based on the outcome of `q9()` and `q10()`, describe the advantage of Adam in 1-2 sentence. (2 marks) [**HINT:** run `q11()` to see what could be the impact of scaling the function (or gradients) on the other optimization method such as gradient descent with Nestrov Momentum. You don't need to report the output of `q11()` in your report. Also, note that `q11()` would most often result in error. Don't worry. That is intentional. Try to understand why this happens.]

The following is the mark breakdown for Part 1.3:

(i) Your code successfully passes the test `q9()`, `q10()`: 4 marks

(ii) Outputs of tests `q9()` and `q10()` are close to the expected output: 4 marks

(iii) Code content is organized well and annotated with comments: 1 marks

(iv) Questions are answered correctly: 6 marks

2 Multiclass Logistic Regression

In the `myTorch.py` file, you will complete the `MultiClassLogisticRegression` class implementation.

2.1 Implementing the Learning Model

In this part, you will be implementing the functions listed below. For more instructions and hints, see the comments in the `myTorch.py` file.

- `add_bias(self, X)`
  Function implementation considerations: This function inserts a column of 1's to X. useful functions in `NumPy` for implementing this method are:
  - `insert`: Inserts values along the given axis before the given indices.

  For evaluation, we provide you the test function `q12()`.

- `unique_classes_(self, y)`
  Function implementation considerations: This function returns a list that contains the unique elements in `y`. useful functions in `NumPy` for implementing this method are:
  - `unique`: Finds the unique elements of an array.

- `class_labels_(self, classes)`
  Function implementation considerations: This function returns a dictionary with elements of the list `classes` as its keys and a unique integer from 0 to the total number of classed as their values. For instance, if `classes = ['blue', 'red', 'yellow']`, then

  ```
  class_labels_(classes) = {'blue': 0, 'red': 1, 'yellow': 2}
  ```

- `one_hot(self, y)`
  Function implementation considerations: This function returns the one-hot encoded version of `y`. Useful attributes for implementing this method are:
  - `self.class_labels`: A dictionary with each class as its keys and a unique integer from 0 to the total number of classed as their values.

  For evaluation, we provide you the test function `q13()`.

- `softmax(self, z)`
  Function implementation considerations: This function is the softmax function, which converts each row of the input matrix `z` into a probability distribution. Thus, if `z`$\in \mathbb{R}^{n\times c}$,then `softmax(z)` returns a matrix in $\mathbb{R}^{n\times c}$, where each element is non-negative and each row of the returned matrix should sum to 1. Standard `NumPy` functions like `exp()`, `sum`. etc can be used.

  For evaluation, we provide you the test function `q14()`.

- `predict_with_X_aug_(self, X)`
  Function implementation considerations: This function returns the predicted probability distribution for each datapoint in `X`, *i.e.*, each row of `X`, based on the model's weight parameter. Note that the input to this function must be an augmented input matrix. Assuming that input $X \in \mathbb{R}^{M\times(d+1)}$ and model's weight, i.e., `self.weights` is in $\mathbb{R}^{c\times(d+1)}$,[2] `predict_with_X_aug_(X)` returns a matrix in $\mathbb{R}^{M\times c}$, where each row of it is a valid probability distribution.

  For evaluation, we provide you the test function `q15()`.

[2]each row corresponds to the weight parameter of a class

- `predict(self, X)`
  Function implementation considerations: This function returns the predicted probability distribution for each datapoint in `X`, *i.e.*, each row of `X`, based on the model's weight parameter. Note that the input to this function is not an augmented input. Assuming that input $X \in \mathbb{R}^{M\times(d)}$ and model's weight, *i.e.*, `self.weights` is in $\mathbb{R}^{c\times(d+1)}$, `predict(X)` returns a matrix in $\mathbb{R}^{M\times c}$, where each row of it is a valid probability distribution. Useful methods for implementing this method are:

  – `self.add_bias`

  – `self.predict_with_X_aug_`

  For evaluation, we provide you the test function `q16()`.

- `predict_classes(self, X)`
  Function implementation considerations: This function returns the predicted class for each datapoint in `X`, *i.e.*, each row of `X`, based on the model's weight parameter. Note that the input to this function is not an augmented input. Assuming that input $X \in \mathbb{R}^{M\times(d)}$ and model's weight, *i.e.*, `self.weights` is in $\mathbb{R}^{c\times(d+1)}$, `predict_classes(X)` returns an `numpy ndarray` with M elements, where each element denotes the predicted class for one of the datapoints. The predicted class, is the class with highest predicted probability. Useful methods for implementing this method are:

  – `self.predict`

  – `argmax` form `NumPy`

  For evaluation, we provide you the test function `q17()`.

- `score(self, X, y)`
  Function implementation considerations: This function returns the ratio of the datapoints in X that are correctly classified by your model, *i.e.*, the predicted class is derived by `predict_classes` function matches the true class specified in `y`. Note that `X` is not augmented. Useful methods for implementing this method are:

  – `self.predict_classes`

  For evaluation, we provide you the test function `q18()`.

- `evaluate_(self, X_aug, y_one_hot_encoded)`
  Function implementation considerations: This function returns the ratio of the datapoints that are correctly classified by your model. Note that the input data batch to `evaluate_` is augmented and the true labels are one-hot-encoded. Useful methods for implementing this method are:

  – `self.predict_with_X_aug_`

  – `argmax` form `NumPy`

  For evaluation, we provide you the test function `q19()`.

- `cross_entropy(self, y_one_hot_encoded, probs)`
  Function implementation considerations: This function returns the cross entropy error given the one-hot-encoded version of the true labels and the predicted probabilities. Therefore, for a batch of M datapoitns, `y_one_hot_encoded` is a M by c matrix and `probs` is a M by c matrix, where each row contains the predicted probability distribution for a datapoint in the batch.

  For evaluation, we provide you the test function `q20()`.

- `compute_grad(self, X, y_one_hot_encoded, w)`
  Function implementation considerations: Given an augmented batch of data `X` in $\mathbb{R}^{M\times(d+1)}$, the one-hot-encoded true labels of the data batch, and the weight parameters `w` in $\mathbb{R}^{c\times(d+1)}$ where c denote the number of classes, this function returns the gradients of E_{in} at `w`. Therefore, it returns a matrix in $\mathbb{R}^{c\times(d+1)}$.

  For evaluation, we provide you the test function `q21()`.

The following is the mark breakdown for Part 2:

(i) Your code successfully passes the test `q12()` to `q20()`: 9 marks

(ii) Your code successfully passes the test `q21()`: 2 marks

(iii) Code content is organized well and annotated with comments: 1 marks

2.2 Implementing the Learning Algorithm

In this part, you will be implementing the functions listed below.

- `fit`
  Function implementation considerations: This function fits the logistic regression model to the input dataset. For more instructions and hints, see the comments in the `myTorch.py` file. Useful functions in `NumPy` for implementing this method are:
  - `random.choice`
  - `abs`
  - `max`

For evaluation, we provide you the test functions `q22()` and `q23()` in `tests_A2.py` file. Go over these test functions and the function `train_model` in the `util.py` file to better understand what these functions do and **answer the following question(s)** after running these tests. Write and save your answers in a separate `PA2_qa.pdf` file. Remember to submit this file together with your code.

2.2.a The test function `q22()` runs your implementation on the Iris dataset.

2.2.a.i Include the figures generated by `q22()` in your `PA2_qa.pdf` file. (2 marks)

2.2.a.ii In 1-2 sentences, compare the performance of the four variants of gradient descent on this dataset (2 marks)

2.2.a.iii In 1-2 sentences, explain how is it possible that the loss derived by the Adam optimizer is smaller than that of Heavy-ball Momentum, but the evaluation score of Adam is equal to the evaluation score of the heavy-ball momentum. (2 marks)

2.2.b The test function `q23()` runs your implementation on the digits dataset.

2.2.b.i Include the figures generated by `q23()` in your `PA2_qa.pdf` file. (2 marks)

The following is the mark breakdown for Part 2:

(i) Outputs of tests `q22()` and `q23()` are close to the expected output: 2 marks

(ii) Code content is organized well and annotated with comments: 1 mark

(iii) Questions are answered correctly: 8 marks

3 K-Means Clustering (Bonus)

Implement the kmeans function in the `myTorch.py` file. You should initialize your k cluster centers to random elements of examples.

After a few iterations of k-means, your centers will be very dense vectors. In order for your code to run efficiently and to obtain full credit, you will need to precompute certain dot products for squared distance calculation. You might find `generateClusteringExamples` in `util.py` useful for testing your code.
The following is the mark breakdown for Part 3:

(i) Your implementation performs well on the hidden tests (you do not have access to the hidden tests): 1.5 marks

4 Discussion

Please answer the following short questions so we can improve future assignments.

4.a How much time did you spend on each part of this assignment? (1 mark)

4.b Any additional feedback? (optional)

5 Turning It In

You need to submit your version of the following files:

- `myTorch.py`
- `PA2_qa.pdf` that answer questions related to the implementations.
- The cover file with your name and student ID filled.

Please pack them into a single folder, compress into a .zip file and name it as PA2.zip.