Preferred Networks Intern Screening 2019 Coding Task for Machine Learning / Mathematics Fields

Changelog

- 2019-04-19 : Initial version

Notice

- Please use one of the following programming languages:
  - C, C++, Python, Ruby, Go, Java, Rust
- Please use standard libraries of your chosen language. Optionally, you may also use the libraries specified below.
  - Multi-dimensional array libraries (ex. Python: NumPy, C++: Eigen) (Automatic differentiation libraries nor deep learning framework are not permitted.)
  - Visualization libraries (ex. Python: matplotlib)
- Please do the task yourself. Do not share or discuss this coding task with anyone, including other applicants.Do not upload your solution and/or problem description to public repository of GitHub during the screening period. If we find evidence of leakage, the applicant will be disqualified. If an applicant allows another applicant to copy answers, both applicants will be disqualified.
- We expect this task to take up to two days. You can submit your completed work without solving all of the problems. Please do your best without neglecting your coursework.

Things to submit

Please submit the following. The name of files and directories is also specified.

- Source code
  - Please create a directory called `src`. Put your code in this directory.
- A text file that explains how to build and run your code
  - Please name the file `README.txt`, `README.md`, or similar.
  - Please specify the version of the language that you used. For example, Python 3.7, C++14, etc.
  - You may write a short explanation of your code as auxiliary material.
- Predicted labels for test data in problem 4
  - Please put the file with name `prediction.txt`. The file should be put at the root directory.
- A report for problems 3,4
  - Please name the file `report.pdf`, `report.txt`, `report.md`, `report.doc`, or something like that.
  - The report should have either 1 or 2 pages in A4 format for problems 3 and 4, including figures and tables.

Evaluation

It is desirable that your submission satisfies the following. (These are not mandatory. Your submission does not need to satisfy all of them if you are too busy.)

- The source code is readable to other programmers.
- There is an appropriate amount of unit tests and/or verification code to ensure the source code is correct.
- It is easy to reproduce the experimental results.
- The submitted report is concise and easy to follow.

During the interview after the first screening process, we may ask some questions on your source code.

How to submit

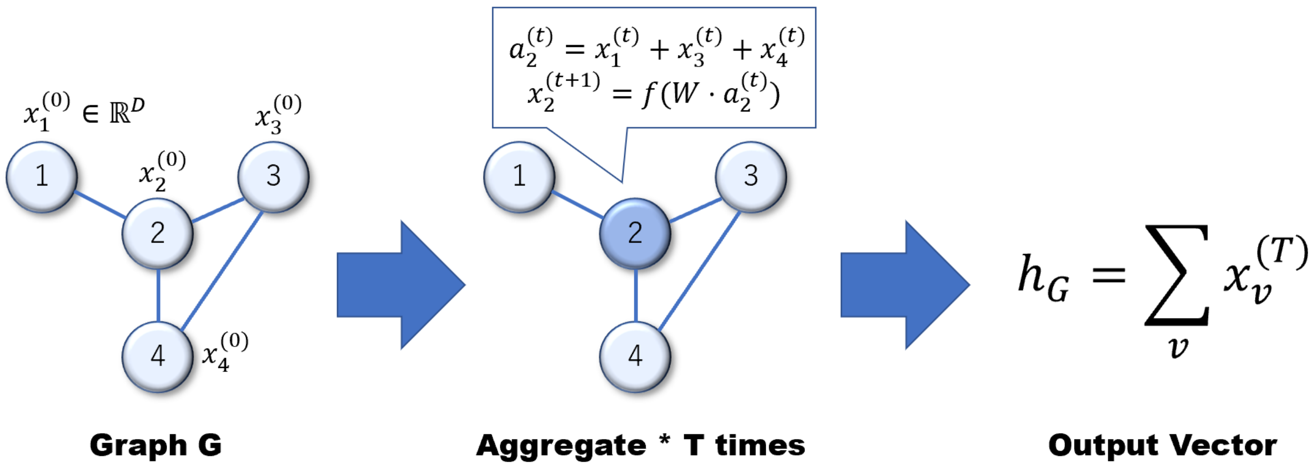
- Create a zip archive with all of the submission materials and submit it on this form. Due date is Tuesday, May 7th, 2019, 12:00 JST.

Inquiry

- If you have any questions on this problem description, please contact us at intern2019-admin@preferred.jp. An updated version will be shared with all applicants if any change is made. Note that we cannot comment on the approach or give hints to the solution.

Task description

Graph neural network (GNN) is a model of a neural network that deals with graph data. In this task, we aim to implement graph neural network from scratch and train a binary classifier for a given dataset.


$x_1^{(0)} \in \mathbb{R}^D$
$x_2^{(0)}$
$x_3^{(0)}$
$x_4^{(0)}$
1
2
3
4
$a_2^{(t)} = x_1^{(t)} + x_3^{(t)} + x_4^{(t)}$
$x_2^{(t+1)} = f(W \cdot a_2^{(t)})$
$h_G = \sum_v x_v^{(T)}$
Graph G
Aggregate * T times
Output Vector

Suppose that we are given a graph $G = (V, E)$, where V is a set of vertices and E is a set of edges. For simplicity, we deal with undirected graphs containing 10~15 vertices throughout the entire task. In the computation of GNN model, we first assign a feature vector $x_v^{(0)} \in \mathbb{R}^D$ to each node $v \in V$. Here, D is a fixed hyperparameter representing the dimension of the feature vector. In GNN, we repeat performing the operation of **aggregation** defined below. By this operation, a set of feature vectors $\{x_v^{(t)}\}_{v \in V}$ at step t is updated to $\{x_v^{(t+1)}\}_{v \in V}$.

- (Aggregation-1) $a_v^{(t)} := \sum_{w:(w,v)\in E} x_w^{(t)}$.
- (Aggregation-2) $x_v^{(t+1)} := f(W \cdot a_v^{(t)})$.

Here, W is a $D \times D$ matrix which is a parameter of GNN, and f is a nonlinear function. We need to train the entries of W. In this task, we assume that the nonlinear function f is the element-wise application of ReLU. Note that we use the common parameter W in each step of the aggregation.

After performing the aggregation several steps, we obtain a feature vector of the graph h_G by summing the feature vectors of nodes.

- (READOUT) $h_G := \sum_{v\in V} x_v^{(T)}$.

We can use this feature vector h_G to train classification problem for graph data.

Note that there exist other various kinds of GNNs in both research and practical domains, where more complex models and problem configuration are considered.

Problem 1

Implement the model of GNN described above, assuming that parameter W is fixed to some matrix.

- Concretely speaking, please write a function or class that computes h_G for a given graph G, implementing (Aggregation-1), (Aggregation-2), and (READOUT). You may use an arbitrary way to represent the graph G in the program. For example, the use of an adjacency matrix is one possible way.
- You may use an arbitrary initial feature vector $x_v^{(0)}$. For example, you may use a vector where the only first element is one and the other elements are zeros.

Please write test code that justify the correctness of the computation result. You may use arbitrary hyperparameters D and T for writing tests.

Problem 2

Let us consider solving the binary classification problem of graph data using the GNN model. As the model of our predictor, we first compute a feature vector h_G and then we compute a weighted sum of h_G. Next, we apply Sigmoid function to obtain a probability value $p \in (0,1)$. The predicted label $\hat{y}$ is defined to be 1 if $p > 1/2$; otherwise, $\hat{y}$ is 0.

- $s := A \cdot h_G + b$.
- $p := \mathrm{Sigmoid}(s) := 1/(1+\exp(-s))$.
- $\hat{y} = \mathbf{1}[p > 1/2]$.

Here, $A \in \mathbb{R}^D$ and $b \in \mathbb{R}$ are the parameters of our predictor. In what follows, we denote the parameter set of our predictor $\{W, A, b\}$ by θ.

For a correct label $y \in \{0,1\}$ of the graph G, the binary cross-entropy loss of the prediction is defined by the below.

- $L := L(G, y; \theta) := -y \log p - (1-y)\log(1-p)$.

(NOTE: If you compute the loss function simply using this definition, the intermediate computation result may cause overflow. Please avoid overflow by appropriately implementing the loss function. For example, you may be able to avoid overflow by using the following equation and approximating $\log(1+\exp(s)) \simeq s$ when s is sufficiently large.)

- $L = y\log(1+\exp(-s)) + (1-y)\log(1+\exp(s))$.

In the training of GNN, we optimize the parameter set θ so that the loss L is as minimized as possible. We use the gradient method for optimization. That is, we compute the gradients $\partial L/\partial\theta = \{\partial L/\partial W, \partial L/\partial A\}$ and we update the parameters along the direction of these gradients. We repeat this update until convergence. In this coding task, we compute the gradient by numerical differentiation for simplicity.

First, please implement a program that computes loss L for a given graph G and a label y. You may extend the implementation in problem 1.

Next, please implement a program that computes numerical differentiation of loss L. This can be computed as follows.

- Let us denote the i-th entry of parameter θ by θ_i. We approximate the gradient $\partial L/\partial\theta_i$ by $(L(G, y; \theta + \epsilon\delta_i) - L(G, y; \theta))/\epsilon$. Here, δ_i is a vector where only the i-th entry is one and others are zeros, and $\epsilon > 0$ is a small value.

After computing the gradient $\partial L/\partial\theta$, update the parameter θ by the following update rule.

- $\theta := \theta - \alpha\partial L/\partial\theta$.

Please confirm that loss L decreases when you perform this update sufficiently many times. You need to arbitrarily construct input graph G containing around 10 vertices. The label y can be also fixed arbitrarily.

Here, $\alpha > 0$ is a hyperparameter called learning rate.

For implementation, you may use arbitrary hyperparameters. You may use a set of hyperparameters listed in the "Hint of Hyperparameters" section described later.

With appropriate implementation, we have confirmed that you can train a parameter set θ so that the loss $L(G, y; \theta)$ becomes less than 0.01 in many cases.

Problem 3

Let us train the predictor implemented in the previous problem. There is a training dataset consisting of a set of pairs of a graph and a label (G, y).

The dataset for training is located at `datasets/train` directory. In this directory, there are two kinds of files: `{id}_graph.txt` and `{id}_label.txt`, which are a graph and a label data respectively. The graph is formatted as adjacency matrix as follows. Here, you can assume that the given matrix is symmetric.

```
n   # Number of nodes in the graph
a_11 a_12 ... a_1n  # a_{ij}=1 if there exists an edge between node i and j,
a_21 a_22 ... a_2n  # otherwise a_{ij}=0.
...
a_n1 a_n2 ... a_nn
```

For training the predictor, please implement Stochastic Gradient Descent (SGD) algorithm. This is an iterative algorithm that repeats the following procedure.

- (Sampling) Sample B data from the dataset uniformly at random. $B \geq 1$ is a hyperparameter. We call the sampled B data mini-batch.
- (Gradient calculation) For each data in the mini-batch, calculate the gradients of parameters and then calculate their average. That is, For mini-batch data $(G^1, y^1), \ldots, (G^B, y^B)$, calculate $\Delta\theta := \frac{1}{B}\sum_{j=1}^{B} \partial L(G^j, y^j)/\partial\theta$.
- (Update-SGD) Update θ to $\theta := \theta - \alpha\Delta\theta$. Here, $\alpha > 0$ is learning rate.

The sampling of mini-batch should be without replacement. The cycle where we sample all data in the dataset is called an epoch. After one epoch, we start sampling from the beginning again.

Additionally, please implement Momentum SGD algorithm, a variant of SGD. This algorithm can be obtained by replacing the update part of SGD with the following rule.

- (Update-Momentum SGD 1) Let w be the amount of change in the previous iteration. At the beginning of the algorithm, w is a zero vector.
- (Update-Momentum SGD 2) Update $\theta := \theta - \alpha\Delta\theta + \eta w$. Here, $\eta > 0$ is a hyperparameter called momentum. After this update, w is set to $w := -\alpha\Delta\theta + \eta w$.

Please implement SGD and Momentum SGD respectively. Next, split the dataset in `datasets/train` to training dataset and validation dataset. Train the predictor using training dataset for several epochs. Observe how the average loss and average accuracy for training dataset and validation dataset, respectively, change as the training proceeds. Describe the results in a report. Note that you cannot use the validation dataset for optimizing parameters.

Hint of Hyperparameters

Just for your information, we describe one example set of hyperparameters below. In problem 3, we verified that we can achieve an average accuracy of about 60% on the validation dataset using these hyperparameters. Note that these hyperparameters are just one example, and you may use other hyperparameters in your implementation.

- Number of steps in aggregation $T = 2$.
- Dimension of feature vectors $D = 8$.
- Learning rate $\alpha = 0.0001$, Momentum $\eta = 0.9$.
- Parameter W and A are initialized by gaussian distribution with mean 0 and standard deviation 0.4. Parameter b is initialized to 0.
- Perturbation for numerical differentiation $\epsilon = 0.001$.
- Number of epochs: 10~100. (NOTE: Required epochs until convergence depends on initialized values of parameters.)

Problem 4

Please perform additional experiments with one of the themes below. Summarize the method and your experimental results in the report.

- Implement Adam [1] optimizer.
- Modify the original definition of GNN model. For example, you may replace ReLU function with another function. You may use a two-layer perceptron (e.g., [2]) in the aggregation step, and so on.
- As a preprocessing step for the dataset, please perform some modification to the original graph data. Observe how the modification changes the performance of training. Here, you may introduce a new edge type and modify the aggregation step. One such way is to introduce one new node (we call super-node here) and connect the super-node and other nodes with newly defined edge type.

There is a test dataset in `datasets/test` directory. This is the same format with `datasets/train` except that there are no label data files. Perform binary classification using the trained predictor. Submit the prediction results in a single file formatted as follows. One line should correspond to one prediction.

```
<Predicted label 0_graph.txt>
<Predicted label 1_graph.txt>
<Predicted label 2_graph.txt>
...
```

References

[1] Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." ICLR 2014.

[2] Xu, Keyulu, et al. "How Powerful are Graph Neural Networks?." ICLR 2019.