CS 763 Assignment 4

Due Date: 19 March, 23:00

March 5, 2018

Problem 1

The task of this assignment is to implement a Vanilla RNN for a "many-to-one" classification task. Bellow, you will find the details of tasks to be completed for this assignment:

1. **Creating the RNN Layer**: Create a file **RNN.lua**. Here, define a torch class **RNN** having the following state variables:

   (a) **W**, which is a matrix of size ((input dimension + hidden dimension) × hidden dimension)

   (b) **B**, which is a matrix of size (hidden dimension × 1)

   (c) **gradW** being the same size as **W**

   (d) **gradB** also being the same size as **B** and

   The **RNN** class should also have the following member functions:

   (a) **__init(*(input dimension), (hidden dimension)*)** which should allocate the desired memory for the state variables and appropriately initialize them.

   (b) **forward(*input*)** computes and returns the *thought vector* after the last time step and also saves it in the state variable **output**.

   (c) **backward(*input, gradOutput*)** unrolls the RNN and performs back propagation through time, computes and updates the state variables **gradInput**, **gradW** and **gradB** and also returns **gradInput**. Note that the loss is computed at the end of the sequence, which then needs to be back-propagated through time by unrolling the RNN.

2. **Creating the Framework**: Create a file called **Model.lua**. It should define a class called **Model** which will provide the framework for adding layers to the 'model' and would allow us to forward and back propagate through the various layers contained in that model. The class must contain the following state variables and functions:

(a) **nLayers**: number of layers in the model.

(b) **H**: RNN size, *i.e.* hidden dimension.

(c) **V**: vocabulary size.

(d) **D**: Word vector size.

(e) **isTrain**: (optional) which should be set to *true* when the model is training and *false* otherwise. This is useful for modules where the forward passes differ between training and testing (e.g. Dropout, etc.). This is only required if you implement such a module.

The member functions are:

(a) **forward(*input*)**: ***input*** is a *Torch* tensor and the returned output is also in the form of a *Torch* tensor. Note that the returned output corresponds to the output of the last layer contained in this model. **Inputs should always be considered as batches.**

(b) **backward(*input*, *gradOutput*)**: Sequentially calls the backward function for the layers contained in the model (to finally compute the gradient of the loss with respect to the parameters of the different layers contained in the model) using the chain rule.

The **Model** can have other variables and functions in addition to the ones mentioned above if required.

3. **The Loss-function**: Create a file **Criterion.lua**. The file should contain the class **Criterion** implementing the *cross-entropy* loss function. The class has no state variables and should contain the following member functions:

   - **forward(*input*, *target*)** which takes an ***input*** of size (batchsize) $\times$ (number of classes) and ***target*** which a 1D tensor of size (batchsize). This function computes the average cross-entropy loss over the batch.
   - **backward(*input*, *target*)** computes and returns the gradient of the Loss with respect to the ***input*** to this layer. More Info.

4. **Implementation Tips:** Note that as it is a "many-to-one" classification task, your loss will only be calculated at the final layer after the last time step. Your prediction will also be taken from the final layer after that last time step.
   You may have to use truncated back propagation through time if sequences are long as, otherwise, your gradients might vanish
   This framework serves only as a rough skeleton. There are caveats in implementing RNNs. So you may have to incorporate additional modules or additional parameters in specific modules.

5. **Training**: For training you will be provided with a sequences of varying lengths. The training data contains around 1200 such sequences. maximum length of sequence can be 2948. The total vocabulary of these sequences is 153 i.e. there are total 153 unique numbers in the sequence. Every sequence can be classified into two classes. You will have

to create an encoding for the input and you can start with the easiest one i.e. the one-hot encoding. RNNs are computationally complex and architecturally not every thing can be parallelized. Therefore, start with only a single layer and chose hidden dimension in the ranges of 64, 128, or 256. Ideally, these settings should get you quite good results.

Evaluation

You are encouraged to upload your prediction results to the Kaggle website to get a real time update about the performance of your model on the hold-out test set. Here you can also compare your model's performance as compared to that of others in our class. The site will be up soon and you will be notified on moodle.

1. Implement a script 'trainModel.lua' which takes arguments:

   - -modelName (model name), should create a folder with 'model name' and save the model in that folder.
   - -data /path/to/train/data.bin, location of the training data. It is the same data that is provided in the assignment but stored in the specified location.
   - -target /path/to/target/labels.bin, location of the target labels.

   The script trains a model having the performance of the **bestModel** on the data provided. The training time should not exceed 12 hours.

2. Finally, implement a script 'testModel.lua' which takes args '-modelName (modelName)' and '-data path/to/test.bin'. The file test.bin has same dimension as 'train.bin'. The script should load the model saved in '(model name)' folder and run it on the test data and save the predictions as 1D tensor named 'testPrediction.bin' in PWD. We will evaluate **bestModel** and the model saved in part (3).

The folder structure for the assignment should contain,

- **src**: Folder containing all the codes.
- **bestModel**: Folder for saving files of best model.
- **trainModel.lua**: For training and saving the trained model.
- **testModel.lua**: For testing model accuracy.

Marking Scheme

Implementing the framework is compulsory. The input to such a framework is to be considered as batches. All implementation should be optimized with vectorization considering the input to be a batch. For instance if the batch size is 100, then implementations which loop over the input tensor 100 times for computing output and gradients would be heavily penalized.

The credits for different objectives are as follows,

1. Correctly implementing RNN layer class and integrating with framework: [**30 marks**]

2. Correctly implementing Model and integrating with framework: [**20 marks**]

3. Accuracy of the best model: 0-50 marks. If a model gets accuracy of more than 85%, the full 50 marks will be awarded. [**50 marks**]