CS 763 Assignment 3

Due Date: 21 February, 23:00

February 9, 2018

Problem

The purpose of the assignment is to implement a Convolution Neural Network (ConvNet) from the scratch and train it for image classification. If you are not convinced and are wondering- *"Why do we have to write the backward pass when frameworks in the real world, such as Torch7, compute them for you automatically and efficiently?"*, please read this blog post, I share the same opinion.

The implementation tasks for the assignment are divided into two parts:

1. Implementing the forward and backward propagation of different type of Layers (e.g. Linear, ReLU, etc.) and
2. Creating a framework for adding and interconnecting outputs and gradients of adjacent Layers using the chain rule. 0.0.

To train such a network a loss function (also called *criterion*) also needs to be defined. For implementation, *Lua* and *Torch7* are to be used. For parsing command line arguments the package *xlua* is to be used. To visualize an image, the *image* package is to be used. The *nn*, *optim* or any other package other than the *torch*, *image*, *xlua* and *math* package **cannot** be used for any of the tasks in this assignment. All tensors should be of type *double*.

All the relevant data can be found on this link. The tar.gz file in the directory contains the files you'll need to verify your implementation (discussed later). The **Train** and **Test** folders contain data specific to the overall task of object classification.

Bellow you will find the details of tasks required for this assignment.

1. **Creating the Linear Layer**: Create a file **Linear.lua**. Here, define a torch class **Linear** having the state variables:
   (a) **output** which is a matrix of size (batch size $\times$ number of output neurons)
   (b) **W** which is a matrix of size (number of output neurons $\times$ number of input neurons)

(c) **B** which is a matrix of size (number of output neurons $\times$ 1)

(d) **gradW** being the same size is **W**

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

(f) **gradInput** which has to be the same size as input.

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

(a) **__init(*(no. of input neurons), (no. output neurons*)**) which should allocate the desired memory for the state variables and appropriately initialize them.

(b) **forward**(***input***) computes and returns the output of the layer and also saves it in the state variable **output**.

(c) **backward**(***input, gradOutput***), computes and updates the state variables **gradInput**, **gradW** and **gradB** and also returns **gradInput**.

2. **Creating the ReLU Layer**: Create a file **ReLU.lua**. It should contain a class **ReLU** which has the following state variables:

   (a) **output** which is the same size as input and

   (b) **gradInput** which is also of the the same size as input.

   The **ReLU** class should implement the following member functions,

   - **__init**() which may be empty
   - **forward**(***input***) that computes the output of the layer and also updates the state variable **output** and
   - **backward**(***input, gradOutput***) that computes and returns the gradient of the Loss with respect to the input to this layer, updates the corresponding state variable **gradInput** and also returns it.

3. **Creating the Framework**: Create a file called **Model.lua**. It should define a class called **Model** which will provide the framework for adding different 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) **Layers** which is a list (Lua table) storing the objects of the different layers added to the model.

   (b) **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*)**: here ***input*** is a *Torch* tensor and this function returns the output also in form of a *Torch* tensor. Note the output that it returns is the output of the last Layer contained in this model. **Inputs should be always 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.

(c) **dispGradParam()**: sequentially print the parameters of the network with the Layer closer to output displayed first. The output format is a 2D matrix for each Layer with space separated elements.

(d) **clearGradParam()**: makes the gradients of the parameters to 0 for every Layer and is required before back-propagation of every batch.

(e) **addLayer(*Layer class object*)** is used to add an object of type *Layer* to the **Layers** table.

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

4. **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.

5. **Training**: For training your network, we provide a dataset for image classification which has six classification categories. The objects in the images have been shot from a variety of viewpoints, illumination and noises in order to facilitate a generic learnt model. The dataset can be downloaded from our Kaggle competition website (once it is up). Alternatively, it can be pulled from this link. You will find the directories *Train* and *Test* which contain the training images, training labels and test images, respectively. The training data is in the shape: (number of instances) $\times$ (height) $\times$ (width). As you may have guessed by now, the images are greyscale with height and width of 108 pixels each. The data consists of approx. 30,000 images in total. The labels for the training set is contained in *labels* which is a 1D tensor. The '.bin' file can be loaded using 'torch.load'. You may want to use cross-validation to choose the best model.

6. Functions for saving a trained model to as a binary file(s) and loading a model from a binary file. You are free to design their way of saving and loading models. All files that are required to be saved should be saved inside the folder named **bestModel**.

7. **Bonus:** As discussed in the class, introducing Momentum into vanilla Gradient Descent hastens the convergence and assists the model in overriding local minima. Implement momentum term in your framework.

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. You need to implement a *Lua* script *checkModel.lua*. The script takes following arguments:

   (a) -config which is the /path/to/modelConfig.txt

   (b) -i which is the /path/to/input.bin

   (c) -ig which is the /path/to/gradInput.bin

   (d) -o which is the /path/to/output.bin

   (e) -ow which is the /path/to/gradWeight.bin

   (f) -ob which is the /path/to/gradB.bin and

   (g) -og which is the /path/to/gradOutput.bin

   The file *modelConfig.txt* has the following format:

   (No. layers)
   (Layer description)
   (Layer description)
   ⋮
   (Layer weights path)
   (Layer bias path)

   (Layer description) varies for the two mandatory layers as:

   - Linear Layer: 'linear' (i/p nodes) (o/p nodes)
   - ReLU layer: 'relu'

   The (Layer weights path) is a path to a Lua table saved as a '.bin' file. The table has as many entries as there are Linear layers in the network and each entry contains the 2D weight tensor of that layer. The (layer bias path) line has similar information for bias.

   An example of 'modelConfig.txt' is included in the assignment folder. The arguments (a),(b) and (c) are inputs to the script.

   The 'input.bin' is a 4D torch tensor. The 'gradOutput.bin' contains gradients with respect to the output of the model. These are randomly chosen values to test the implementation and not actually calculated against a loss. For this evaluation, the batch-size is to be taken as the number of data points provided in the sample 'input.bin'.

The arguments (d),(e),(f) and (g) are to be saved by the script. The 'gradWeight.bin' and 'gradB.bin' are lua tables having same format as the tables containing sample W and B. The 'output.bin' is the tensor containing output of the model. Each entry should have its corresponding **gradW** and **gradB** tensor. The 'gradInput.bin' is gradient of the Loss with respect to the 'input.data' to the model. 'torch.save' should be used to save the bin files. (Remember to reset the gradient values to zero before you back propagate).

2. You also need to implement a script 'checkCriterion.lua' taking the following arguments:

   (a) -i /path/to/input.bin

   (b) -t /path/to/target.bin

   (c) -og /path/to/gradInput.bin

   The arguments (a) and (b) are inputs and (c) is the output. The 'input.bin' contains a 2D tensor. The 'target.bin' contains a 1D tensor having same first dimension as that 'input.bin'. The number of classes is same as that of the second dimension of 'input.bin' and indexed from 1. The script should compute the average loss for given input and target and print in console. The script should also save 'gradInput.bin' which contains gradient of loss w.r.t input.

3. 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.

4. Finally, implement a script 'testModel.lua' which takes args '-modelName (modelName)' and '-data path/to/test.bin'. The file test.bin is a 4D tensor with 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.
- **checkModel.lua**: For evaluating model implementation.
- **checkCriterion.lua**: For evaluating criterion implementation.
- **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 the Linear Algebra 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 Linear layer class and integrating with framework: 25 marks
2. Correctly implementing ReLU layer class and integrating with framework: 10 marks
3. Correctly implementing multiple Linear and ReLU layers and integrating with framework: 15 marks
4. Correctly implementing Criterion class: 10 marks
5. Accuracy of the best model: 0-40 marks. If a model gets accuracy of more than 70%, the full 40 marks will be awarded. (In order to achieve high accuracy one may need to implement other types of Layer classes such as Convolution, BatchNorm, Dropout etc. and integrate with the framework.)
6. Bonus: 10 marks for implementing Momentum.