COL 341: Assignment 2

Notes:

- This assignment has two parts - Neural Network and Convolutional Neural Networks.
- You are advised to use vector operations (wherever possible) for best performance as the evaluation will be timed.
- You should use Python for all your programming solutions.
- Your assignments will be auto-graded, make sure you test your programs before submitting. We will use your code to train the model on training data and predict on test set.
- Input/output format, submission format and other details are included. Your programs should be modular enough to accept specified parameters.
- You should submit work of your own. You should cite the source, if you choose to use any external resource. You will be awarded D or F grade or DISCO in case of plagiarism.

1. **Neural Networks (100 points, Due date: 9:00 PM Tuesday, 5th October, 2021)**
   In this problem, we'll train neural networks to classify a binary class (**Toy**) and multi-class (**Devanagri handwritten characters**) dataset.

   (a) (25 points) Write a program to implement a general neural network architecture. Implement the back-propagation algorithm from first principles to train the network. You should train the network using Mini-Batch Gradient Descent. Your implementation should be generic enough so that it can work with different architectures (any specified learning rate, activation function, batch size etc.). Assume a fully connected architecture i.e., each unit in a hidden layer is connected to every unit in the next layer. Have an option for an adaptive learning rate $\eta_t = \frac{\eta_0}{\sqrt{t}}$. Use Cross Entropy Loss (CE) as the loss function. Use **sigmoid** as the activation function for the units in **intermediate layers**. Use **softmax** as the activation function for the units in **output layer**. Your implementation should also work for **tanh** and **relu** activation functions. The details of the CE loss function are given below:

$$CE = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{k} y_{ij}\log(\hat{y}_{ij}) \tag{1}$$

$$\hat{y}_{ij} = \frac{\exp z_{ij}}{\sum_{j'=1}^{k}\exp z_{ij'}} \tag{2}$$

   Here i indexes samples while j indexes class labels. Here y_i is a one-hot vector where only one value (corresponding to true class index) is non-zero for sample i. n is the number of samples in the batch. k is the number of labels. $\hat{y}_{ij} \in (0,1) : \sum_j \hat{y}_{ij} = 1 \,\forall\, i,j$ is the prediction of a sample. z_{ij} is the value being input into j^{th} perceptron of softmax layer for i^{th} sample.

   Use Your Implementation to train neural network on Toy training dataset (first column correspond to classes) with predefined parameters (corresponding to fixed learning rate) and predict on the publicly available Toy testing dataset (first column filled with -1, labels included separately). Before starting to implement, have a look at the evaluation criteria, submission instructions and coding guidelines for this part.

- Fixed Learning Rate

$$w(t) = w(t-1) - \eta_0 \nabla_w L(w; X_{b(t-1):bt}, y_{b(t-1):bt}) \tag{3}$$

Here t indicates epoch number while w denotes model weights. b indicates batch size while X, y denotes data and labels respectively. η_0, L represents fixed learning rate and loss function respectively. $\nabla_w L$ signifies gradient of the loss function with respect to model weights.

- The way to initialise weights
We will use a standard way to initialise weights in all parts of this problem. Let w_{ij}^l be the weight input to the layer l where $i \in 0...m$ and $j \in 1...n$. m and n here are the number of neurons in layers numbered $l-1$ and l respectively. Note that $\{w_{0j}\colon\ j \in 1...n\}$ corresponds to the bias vector input to layer l. Each $w_{ij}^l \sim \mathcal{N}(0,1) * \sqrt{\frac{2}{(m+1+n)}}$ where $\mathcal{N}(0,1)$ is the standard normal distribution. This is also known as xavier initialisation. You can read about it. You must use numpy.random.normal to initialise the weights and w_{ij}^l for any l must be initialised using one and only one call to numpy.random.normal. You must use numpy.random.seed to set the seed. The value of seed must be taken as input from the user. Also the code to set the seed should only run once for entire training phase. Weights when initialised must have the data type float32 (use numpy.float32() to do this). The datatype can turn to a higher one while running but initially it must be float32.

(b) (25 points) Modify neural network architecture made in part a) to cater for multi-class dataset (Davanagri handwritten characters). You have been provided image files data corresponding to train and test sets respectively. The training dataset has 8-bit 32x32 greyscale images corresponding to to 46 Devanagari characters (last column in both training and test data is for labels). Use Mean Squared Loss(MSE) as the loss function. Use **tanh** as the activation function for the units in **intermediate layers** and **output layer**.

$$MSE = \frac{1}{2n} \sum_{i=1}^{n} \sum_{j=1}^{k} (y_{ij} - \hat{y}_{ij})^2 \tag{4}$$

Here i indexes samples while j indexes class labels. Here y_i is a one-hot vector where only one value (corresponding to true class index) is non-zero for sample i. n is the number of samples in the batch. k is the number of labels. The value of $\hat{y}_{ij}$ depends on the activation function used in the output layer.

Use Your Implementation to train a neural network on the given training dataset with predefined parameters(corresponding to adaptive learning rate) and predict on the publicly testing dataset.

- Adaptive Learning Rate

$$w(t) = w(t-1) - \frac{\eta_0}{\sqrt{t}} \nabla_w L(w; X_{b(t-1):bt}, y_{b(t-1):bt}) \tag{5}$$

Here t indicates epoch number while w denotes model weights. b indicates batch size while X, y denotes data and labels respectively. η_0, L represents seed value and loss function respectively. $\nabla_w L$ signifies gradient of the loss function with respect to model weights.

(c) (12.5 points) Use Your Implementation in part a to train a neural network on the given multiclass dataset with predefined parameters. The algorithm that you used for gradient updates in part a and part b(mini-batch gradient descent) can have many variants, each giving an edge over the conventional SGD algotrithm. For this part you are required to go through these variants here . You are required to implement 5 of these which are:

- Momentum
- Nesterov
- RMSprop
- Adam
- Nadam

There are a lot of additional hyper parameters you will come across while you are implementing these. These hyperparameters are mostly kept constant(standard values are available on the web). You may want to do the same to prevent an exponential blow-up of hyperparameters. After implementation, use the different forms of gradient descent and experiment which one converges faster. Try to tweak different parameters keeping the following parameters same.

- architecture(specified in coding guidelines)
- softmax in the output layer
- Cross-entropy loss

Report the best architecture you get(the best is the one that reduces the loss to a minimum in least number of iterations). Also mention in the report how did you arrive at these parameters.

(d) (12.5 points) Use Your Implementation to train a neural network on the given Devanagri handwritten characters dataset with predefined parameters. Experiment with different types of architectures. Vary the number of hidden layers. Try different number of units in each layer, different loss and activation functions, gradient descent variants etc. What happens as we increase the number of units or the number of hidden layers? Comment on your observation and submit your best performing architecture.

Note: Use holdout method to find the best architecture. Use the public test data as the validation set. Use the prediction on the validation set to determine which architecture to use.

- **A note of caution:** your report must clearly specify how optimal parameters were reached in part (c) and (d). Any cutting corners will be treated as disciplinary misconduct and will be dealt with accordingly.

Evaluation:

- For part-a and part-b, you can get 0 (error), partial marks (code runs fine but weights are incorrect within some predefined threshold) and full (works as expected). We will check obtained weights and predictions for grading these parts. Make sure to initialise the weights in the same way as mentioned. Any deviations would lead to incorrect results.
- For part-c, marks will be given based on loss value after training till a specified time on private architectures. The lesser the value, higher the marks you get.
- For part-d, marks will be given based on accuracy on private test data-set.
- For part-c and part-d, there will be relative marking
- For part-c and part-d marking will be done for code as well as report.

Submission Instructions:

Neural Network

Submit your code in 6 executable python files called neural_a.py, neural_b.py, neural_c.py, neural_c1.py, neural_d.py, neural_d1.py

The file name should corresponds with part [a,b,c,d] of the assignment.
The parameters are dependant on the mode:

- *python neural_a.py input_path output_path param.txt*
  Here your code must read the input training and test files for the toy dataset(filenames will remain same as provided) from the *input_path*(a directory), initialise the parameters of the network to what is provided in the *param.txt* file(we will provide the absolute path to *param.txt* including the filename) and write weights and predictions to *output_path*(also a directory).
  param.txt will contain 8 lines specifying epochs, batch size, a list specifying the architecture([100,50,10] implies 2 hidden layers with 100 and 50 neurons and 10 neurons in the output layer), learning rate type(0 for fixed and 1 for adaptive), learning rate value, activation function(0 for log sigmoid, 1 for tanh, 2 for relu), loss function(0 for CE and 1 for MSE), seed value for the numpy.random.normal used(some whole number). The order will be the same as here.

weights must be written to the output_path for each layer in form of numpy arrays in *w_l.npy* file where l is the index(starting from 1) of the layers 1 to output layer(example: for architecture[100,50,10], the weight files will be *w_1.npy*, *w_2.npy*, *w_3.npy*, with the last one containing weights in the last layer). The weight files format must be the same as specified in "The way to initialise weights". the predictions must be a 1-D numpy array written to the *output_path* as *predictions.npy* file.

- *python neural_b.py input_path output_path param.txt*
  Same as for part a except the *input_path* will now contain files for devnagari dataset.

- *python neural_c.py input_path output_path param.txt*
  Here your code must read the input training file for the Devnagari dataset(filename will remain same as provided) from the *input_path*(a directory). Here the file *param.txt*, to which the complete absolute path will be provided, will contain 1 line which will be a list specifying the architecture(just the way it is specified in *param.txt* for part a)). Your code must not exceed a time limit of 300 seconds. Within this time limit you need to run the specified model with the best parameters you found and write the weight files after training to the *output_path* in the same format as specified in submission instructions for part a). You must also write a text file to the *output_path* with the name *my_params.txt* specifying each of the following in a new line for your best parameters.
  number of epochs
  batch size
  learn rate type(0 for fixed 1 for adaptive)
  learn rate value(initial value in case of adaptive)
  activation function(0 for log sigmoid, 1 for tanh, 2 for relu)
  optimizer type(0 for vanilla SGD, 1 for momentum, 2 for nesterov, 3 for RMSprop and 4 for adam, 5 for nadam)
  seed value for the numpy.random.normal

- *python neural_c1.py input_path output_path*
  Here your code must read the input training file for the Devnagari dataset(filename will remain same as provided) from the *input_path*(a directory). Your code must now run the two public architectures(see coding guidelines) across all configurations of your parameter search space, and produce all plots/tables to the *output_path* that you mention in your report. Plots must be produced in .png format and tables must be produced in .csv format.

- *python neural_d.py input_path output_path*
  Here your code must read the input training file for the Devnagari dataset(filename will remain same as provided) from the *input_path*(a directory). Your code must run the best architecture along with best parameters you concluded. It must produce the weight files to *output_path* in the same way as specified for part a).
  You must also write a text file to the *output_path* with the name *my_params.txt* specifying each of the following in a new line for your best parameters.
  number of epochs
  batch size
  learn rate type(0 for fixed 1 for adaptive)
  learn rate value(initial value in case of adaptive)
  activation function(0 for log sigmoid, 1 for tanh, 2 for relu)
  loss function (0 for CE an 1 for MSE)
  optimizer type(0 for vanilla SGD, 1 for momentum, 2 for nesterov, 3 for RMSprop and 4 for adam, 5 for nadam)
  architecture(list specifying architecture, [100,50,10] implies 2 hidden layers with 100 and 50 neurons and 10 neurons in the output layer)
  seed value for the numpy.random.normal

- *python neural_d1.py input_path output_path*
  Here your code must read the input training and test files for the Devnagari dataset(filenames will

remain same as provided) from the *input_path*(a directory). Your code must now run all configurations of your architecture and parameter search space, and produce all plots/tables to the *output_path* that you mention in your report. Plots must be produced in .png format and tables must be produced in .csv format.

- *Part (e)*
  Here you have to submit a pdf file with all details of parameters for parts c) and of best architectures and parameters d). Report results and observations of all variations experimented with irrespective of whether they lead to increase in accuracy. Report must also contain how the optimal parameters were reached. Plots/tables must be included wherever necessary.

Coding Guidelines:

(a) For parts a) and b)

- Don't shuffle/change order of the data files.
- The data should be scaled to fit 0 to 1 using 1/255 as the scaling factor. No other transformation must be done to the data before feeding to the network.
- Don't apply early stopping criterion. Run for full specified number of iterations/epochs.
- Be careful to code Mini-batch Gradient Descent to be closely consistent with the theoretical framework taught in class.
- Make sure you do not use any random function other than numpy.random.normal. Also make sure the usage is as mentioned in the question.
- Make sure there are no other libraries used other than numpy, math, time, pandas and sys. Use of any other library is prohibited.
- In case of cross entropy loss, last layer activation function is assumed to be softmax unless otherwise stated.
- For you to check your implementation, we have provided the weight files and checker script on the given link. The weight files namely *ac_w_l.npy* are weights after 5 epochs of toy and multiclass datasets. The ones named *ac_w_l_iter.npy* are after 5 iterations of the same datasets. You can use the checker code to make sure the weights agree with the evaluation weights.
- Your code must not print anything on the screen.

(b) For parts c) and d)

- Your code must not print anything on the screen.
- For part c) the architectures for public evaluation have been provided on the same link as weight checker script. You have to use only these architectures for searching parameters.
- You can use all libraries used for part a) and b). Additionally you can use matplotlib for all plots.
- Use matplotlib.use('Agg') for saving plots without display as the evaluation will be on remote PCs.You can also make sure you never use plt.show().
- Design code with emphasis on Vector Operations and minimal use of loops to ensure run-time efficiency.

(c) *neural_a.py, neural_b.py, neural_c.py,neural_d.py* must finish execution within **5 minutes** each, For *neural_c1.py,neural_d1.py*, the time limit is **TBD**

(d) Keep checking piazza for announcements regarding the assignment.

(e) Failure to comply with coding guidelines and submission instructions will lead to penalties.

Extra Readings (highly recommended for part (a,b)):

(a) Backpropogation for different loss functions

(b) Gradient descent algorithms

2. **Convolutional Neural Networks (Score: 100 Points, Due date: 11^{th} Oct. 2021, 9PM)** This part of the assignment is to get you started with Convolutional Neural Networks (CNN) and deep learning. You are going to experiment and get your hands dirty with deep learning framework **PyTorch**. A2.2 has been divided into three parts, in which you will perform image classification task. (a) Implement a simple CNN architecture and evaluate performance on Devanagari dataset, same as that of A2.1 and compare the results of both the models. (b) A slightly more complex architecture to be evaluated on CIFAR-10 dataset. (c) Competitive part on CIFAR-10. All the parts have to be implemented in **PyTorch**.

   You are allowed to use from the following modules. Make sure your scripts do not include any other modules. Python3.7, glob, os, collections, PyTorch, torchvision, numpy, pandas, skimage, scipy, scikit-learn, matplotlib, OpenCV (4.2 or higher)

   (a) **(30 Points) Devanagari Character Classification:** You are going to use the same dataset as the Neural Network Part with **46 classes** of Devanagari characters. Implement the following CNN architecture to perform the classification.

   Dataset:

   - **Train Data**: Download. Each data row contains 1025 columns. First 1024 values (index: 0-1023) correspond to the pixel value of (32×32) grayscale image (only 1 channel) of a Devanagari character. 1025^{th} value is the class label (0-45). Train dataset has been shuffled already. Make sure that you **do not shuffle** it.
   - **Public Test Data**: Download. Description same as Train Data.
   - **Private Test Data**: TBD

   Model Architecture:

   i. **CONV1** (2D Convolution Layer) in_channels = 1, out_channels = 32, kernel=3×3, stride = 1.
   ii. **BN1** 2D Batch Normalization Layer
   iii. **RELU** ReLU Non-Linearity
   iv. **POOL1** (MaxPool Layer) kernel_size=2×2, stride=2.
   v. **CONV2** (2D Convolution Layer) in_channels = 32, out_channels = 64, kernel=3×3, stride = 1.
   vi. **BN1** 2D Batch Normalization Layer
   vii. **RELU** ReLU Non-Linearity
   viii. **POOL2** (MaxPool Layer) kernel_size=2×2, stride=2.
   ix. **CONV3** (2D Convolution Layer) in_channels = 64, out_channels = 256, kernel=3×3, stride = 2.
   x. **BN1** 2D Batch Normalization Layer
   xi. **RELU** ReLU Non-Linearity
   xii. **POOL3** (MaxPool Layer) kernel_size=2×2, stride=1.
   xiii. **CONV4** (2D Convolution Layer) in_channels = 256, out_channels = 512, kernel=3×3, stride = 2.
   xiv. **RELU** ReLU Non-Linearity
   xv. **FC1** (Fully Connected Layer) output = 256
   xvi. **RELU** ReLU Non-Linearity
   xvii. **DROPOUT** Dropout layer with $p = 0.2$
   xviii. **FC2** (Fully Connected Layer) output = 46

   Instructions:
   CNN Specifications:

   - Your model has been designed to be compatible with the input size of $[C = 1, H = 32, W = 32]$.
   - Read more about ReLU, Batch Normalization and Dropouts online in order to understand their use and benefits.
   - Use **Cross Entropy Loss** `(torch.nn.CrossEntropyLoss())` function and **Adam Optimizer** `torch.optim.Adam()` with fixed learning rate `lr=1e-4`.

- Train your model for **8 epochs**.
- keep a track of average training loss for each epoch and test accuracy after each epoch. You will be asked to export these 8 loss and 8 accuracy values in a separate file during submission for the evaluation purpose.

Data Loader: You have been provided with a custom dataloader boiler-plate code for Devanagari to get you started quickly. Link TBD.

Report:

- Report public test accuracy and other observations.
- Line plot to show training loss on the y-axis and the epoch on the x-axis.
- Calculate the accuracy of public test data at the end of each epoch and plot a line graph with accuracy on y-axis and number of epochs on the x-axis.
- Compare the accuracy with the Neural Network and argue which one performs better in terms of accuracy, and training time. Comment on these observations.

Extra Readings:

- Getting started with the PyTorch
- PyTorch tutorial on YouTube
- Dropout
- Batch Normalization

(b) **(20 Points) CIFAR-10 Classification** Performing a **10 class** image classification by using the below defined architecture.

Dataset:

- **Train Data**: TBD.
- **Public Test Data**: TBD.
- **Private Test Data**: TBD

Model Architecture:

i. To be provided. It will be similar to that of part (a).

CNN Specifications:

- Your model has been designed to be compatible with the input size of $[C = 1, H = 32, W = 32]$.
- Use **Cross Entropy Loss** `(torch.nn.CrossEntropyLoss())` function and **Adam Optimizer** `torch.optim.Adam()` with fixed learning rate `lr=TBD`.
- Train your model for **N epochs**.
- keep a track of average training loss for each epoch and test accuracy after each epoch. You will be asked to export these N loss and N accuracy values in a separate file during submission for the evaluation purpose. To be provided. (N=TBD)

Data Loader: Link to custom data loader. TBD

Report: Same as Part(a)

(c) **(50 Points) CIFAR-10 Competition:** Here you have to come up with your own architecture for CIFAR-10 classification. You are free to experiment with all type of loss functions, optimizers and learning rates. This is a competitive part in which you will be evaluated over a *private test dataset*. Your model should have less than **5 million** parameters (including both trainable and non-trainable).

Submission Guidelines: TBD

Coding Guidelines Same as Q1.

Grading Scheme: TBD