CSC 7700/4700 Foundational AI Project 1

Learning Objective: Students will learn the fundamentals of deep learning by implementing multilayer perceptrons (MLPs) from scratch in Python, using the Numpy library for handling n-dimensional arrays. The written code will be used to train and evaluate MLPs for regression and classification applications.

Tasks: Students will write a program in Python 3.10.16 and numpy 2.2.1 for developing, training, and evaluating MLPs, with the following features (more details provided in instructions):

1. Defining an MLP with any number of hidden layers. Each hidden layer can have any number of neurons with activation functions independently configurable for each layer.
2. Vectorized forward propagation for parallel feedforward computing for a data batch of size n.
3. Vectorized backpropagation with configurable loss function.
4. Training routine with a configurable number of epochs, learning rate, momentum, etc.
5. Ability to plot training and validation loss.

Instructions:

I have provided you with a template Python file in the course Github repository, named mlp.py. This must be used as a starting point for your project. Modifications to method and class arguments are OK. Additional code descriptions and comments are provided in the template

1. Your code must have an abstract class named ActivationFunction with at least the following abstract methods:
    a. **forward(x)**, which computes and returns $\varphi(\boldsymbol{X})$.
    b. **derivative(x)**, which computes and returns $\varphi'(\boldsymbol{X})$.
2. Your code must implement the following activation functions by subclassing ActivationFunction:
    a. Sigmoid
    b. Tanh
    c. Relu
    d. **Softmax** (will have different args than the others)
    e. Linear
    f. (Grad Students Only) **Softplus**
    g. (Grad Students Only) **Mish**
3. Your code must define an abstract class named LossFunction with at least the following abstract methods:
    a. **loss(y_true, y_pred)**, which computes $l\left(\mathrm{y_{true}}^{(i)}, \mathrm{y_{pred}}^{(i)}\right)$ for each sample
    b. **derivative(y_true, y_pred)**, which computes $l'\left(\mathrm{y_{true}}^{(i)}, \mathrm{y_{pred}}^{(i)}\right)$ for each sample
4. Your code must implement the following loss functions by subclassing LossFunction:
    a. SquaredError
    b. CrossEntropy
5. Your code must define a class named Layer, with the following init arguments:
    a. **fan_in**: the number of neurons in the layer prior to this one
    b. **fan_out**: the number of neurons in this layer
    c. **activation_function**: an instance of an ActivationFunction
6. The Layer class' init method should initialize the weight matrix using Glorot uniform.
7. The Layer class must implement the following methods:
    a. **forward(h)**: computes the output of this layer given input h
        i. (5% BONUS): Implement dropout with a configurable dropout rate

   b. **backward(h,delta)**: computes and returns $(\partial_{\boldsymbol{W}}\mathcal{L}, \partial_{\vec{b}}\mathcal{L})$, given layer input h and the next layer's δ term. Also computes and stores this layer's δ term, $\delta = \partial_{\vec{o}}\mathcal{L}$.

8. Your code must define a class named MultilayerPerceptron, with the following init arguments:
   a. **layers**: A list or tuple of Layer objects in order from first to last layer
9. The MultilayerPerceptron class must implement the following methods:
   a. **forward(x)**: Forward propagates the network's input matrix, $\boldsymbol{X}$, and computes and returns the network's output, $\boldsymbol{Y}_{\text{pred}}$.
   b. **backward(loss_grad, input_data)**: Backward propagates the loss gradient through the network, computing $(\partial_{\boldsymbol{W}}\mathcal{L}, \partial_{\vec{b}}\mathcal{L})$ for each layer and storing them in lists. The method returns a Tuple of $(\partial_{\boldsymbol{W}^{(1)}}\mathcal{L}, \partial_{\boldsymbol{W}^{(2)}}\mathcal{L}, \cdots, \partial_{\boldsymbol{W}^{(N)}}\mathcal{L})$, $(\partial_{\vec{b}^{(1)}}\mathcal{L}, \partial_{\vec{b}^{(2)}}\mathcal{L}, \cdots, \partial_{\vec{b}^{(N)}}\mathcal{L})$ for N layers
   c. **train(train_x, train_y, val_x, val_y, loss_func, learning_rate, batch_size, epochs)**: Trains the model using minibatch stochastic gradient descent (mSGD) by:
      i. first batching the training data into batches of no greater than **batch_size**
      ii. looping over the training set for a defined number of **epochs**. For each loop:
         1. loop over each batch in the training set. For each batch:
            a. Compute the network's output on the batch
            b. Compute the loss gradient, $\partial_{\boldsymbol{Y}_{\text{pred}}}\mathcal{L}$, according to the provided LossFunction instance, **loss_func**
            c. Apply backpropagation to determine $(\partial_{\boldsymbol{W}}\mathcal{L}, \partial_{\vec{b}}\mathcal{L})$ for each layer
            d. Update the weights using $(\partial_{\boldsymbol{W}}\mathcal{L}, \partial_{\vec{b}}\mathcal{L})$ for each layer
         2. Compute the loss after updating weights and keep track of it. Print the training and validation loss out to the terminal for each epoch
      iii. After each epoch:
         1. compute the average loss across all batches and store this value
         2. compute the loss on the full validation set and store this value
      iv. When all epochs are finished, return the training and validation loss
      v. (5% Bonus): In addition to vanilla mSGD, implement the RMSProp optimizer. This should be enabled through an optional Boolean parameter in the train method, **rmsprop**

10. **Vehicle MPG Dataset**:
    a. Download the Vehicle MPG regression dataset from the UCI ML repo and split it into 70% training, 15% validation, and 15% testing. Feel free to copy the code in the linear regression example to handle data downloading and formatting.
    b. Design and train a multilayer perceptron using your code. Try to get the best performance that you can achieve by testing different architectures, hyperparameters, and activation functions, etc.
    c. After training, evaluate the model on the testing set and report the total loss.
    d. Create a **train_mpg.py** script that downloads and splits the dataset, instantiates your MLP design, and trains the model. The training and validation loss should be printed out at each epoch.

11. **MNIST**:
    a. Download the MNIST handwritten digits classification dataset (training and test), and split the training set into 80% training/validation. Each of the 28x28 images (matrix) can be flattened into vectors of length 784.
    b. Design and train a MLP using your code. Try to get the best performance that you can achieve by testing different architectures, hyperparameters, and activation functions, etc.

c. After training, evaluate the model on the full testing set and report overall accuracy.
d. Create a **train_mnist.py** script that downloads and splits the dataset, instantiates your MLP design, and trains the model. The training and validation loss should be printed out at each epoch.
e. **NOTE**: I don't care what resources or methods you use to download and split the data (step a). You are free to use generative AI and 3rd part libraries to help with that part.

Deliverables:

You will prepare a short report in 12 point Calibri font (1/2" margins), composed of the following sections:

Abstract: In one paragraph, describe your general approach to the project and report important results for each application, such as accuracy for MNIST and total loss for vehicle MPG.

Methodology: Half a page or less describing your approach to coding the MLP engine. One page or less describing the design of your MLPs for each application, challenges you had, and how you addressed those challenges.

Results: For both applications, use matplotlib to generate plots of the loss curves (both training and validation).

For the Vehicle MPG Regression Dataset:

- Use matplotlib to generate a plot of the loss curves (both training and validation).
- Report the total testing loss over the full testing dataset
- Select 10 different samples from testing and report the predicted MPG against the true MPG. This should be displayed as a Table

For MNIST:

- Use matplotlib to generate a plot of the loss curves (both training and validation).
- Report the accuracy on the full testing dataset.
- Select 1 for each class (0-9) from testing and show these samples (images) along with the predicted class for each.

Code Repo Link: Provide a link to the Github repository containing all of the code for the project. The train_mnist.py and train_mpg.py scripts should be at the top-level directory in the repo.

Discussion & Conclusion: One half page or less discussing the results and what you learned from the project.