Final Programming Assignment:
Building Your Neural Network with Linear Algebra

In this final project, you will implement a neural network from scratch in Python by combining the mathematical modeling techniques you have learned over the entire quarter, including linear algebra, least squares, chain rule, gradient descent, and nonlinear optimization. You will test your network on learning the classical hand-written digit classification task with the MNIST dataset. This is an individual project and everyone needs to submit their own code implementation and report.

Background Reading

Before you get started, read the following materials to ensure you have enough background knowledge for a basic machine learning application. In particular, please make sure you are familiar with the concepts of matrix-matrix multiplication, feature vector, loss function, gradient descent optimization, and the chain rule. We also highly recommend you finish TA6 before working on these programming tasks.

- Course notes
- VMLS book Chapter 13, 14
- Flower book Chapter 6
- Starter code (read through all of them, both comments and implementations)
- How to build your own Neural Network from scratch in Python
- More about forward
- More about gradient

Task Overview

You are asked to implement a data-fitting classifier that can predict the label of an input vector. Specifically, in the context of hand-written digit classification, the label is an integer number between 0 to 9, and the input vector is a vectorized 28×28 image with each element as a floating-point number ranging from 0 to 1 (telling the grayscale of the pixel). You need to solve the problem in both the least-squares way and the neural network way.

Appetizer Tasks (50 points)

Before diving into the neural network world, let's first try to solve the hand-written digit classification problems using our well-understood mathematical tool of least squares. We will use the least-squares to classify a subset of the MNIST hand-written digit dataset. The training dataset for this project has 1000 images, which are stored as a 1000×784 matrix in *train_data.txt* (You may refer to the constructor of the class *Classifier* to learn how to read a matrix from a text file). The testing dataset for this project has 200 images, which are stored as a 200×784 matrix in *test_data.txt*. The labels for the training set are stored as a 1D array (or a 1000×1 matrix) in *train_labels.txt* and the labels for the testing set are in *test_labels.txt*.

Here is a brief and high-level description of the mathematical model you may consider implementing. You will solve a least-squares problem $XW = Y$ to fit your data set, with X as a 1000×785 (with the bias term!) data matrix, W as a 785×10 model matrix, and Y is a 1000×10 true value right-hand matrix composed of 1 and -1 (or any values that distinguish the two classes). For the easiness of the implementation, you may solve the problem as ten separate least-squares problems with each column $\vec{\theta}_i$ (a 785×1 vector) as the unknowns. After calculating the values of W, you may use it to predict the class of a new image x by a simple vector-matrix multiplication $\vec{v} = \vec{x}W$ and then pick the index with the maximum value from the vector $p = argmax_i\{v_i\}$. You need to test your least-squares classifier on the test dataset and report your least-squares model's accuracy (by counting how many images are predicted correctly).

[*Hint: you may suffer from a singular X due to the large blank regions in each image. You need to add a small random number to each element in A to solve the problem to make it solvable in your least-squares solver. For example, you may add a noise ranging from* $[0, 0.001]$ *to each element in A to perturb its features.*]

Entree Tasks (100 points)

After building your least-squares approximators, your next step is to re-implement the same task by designing a neural network model to improve the learning performance. The key idea is to express the data flow of the neural network with multiple layers. Typically, you may build an architecture composed of linear layers and nonlinear layers alternatively and end with a loss function layer (i.e., the structure of the network layers is linear->nonlinear->linear->...->loss). For each layer, you need to implement two functions of *forward* propagation and *backward* propagation. You will chain each layer together and calculate the correct derivatives to optimize the loss function with gradient descent. You will find the conceptual background and the technical details for implementing the network in the course notes and the referred reading materials listed above.

The programming of our handcrafted neural network consists of the following steps:

1. *Task 1: Linear layer*

   - **Forward (Linear):**
     - Input: X (#samples, #features0)
     - Output: Y (#samples, #features1)
     - Stored data: input X (#samples, #features0)

In this function, you will forward the data from the previous layer to the next layer by performing a matrix-matrix multiplication. You need to store the input data because you will need this data in the backward step (remember the intermediate data we discussed in our simple 1D network model in class). For simplicity, we did not consider the bias in our NN model.

- **Backward (Linear):**
  - Input: $\frac{\partial loss}{\partial Y}$ (#samples, #features1).
  - Output: $\frac{\partial loss}{\partial X}$ (#samples, #features0)
  - Stored data: weight gradient matrix $\frac{\partial loss}{\partial W}$ (#features0, #features1)

  The input of the backward function is $\frac{\partial loss}{\partial Y}$, which is the partial derivative of the loss with respect to the layer's output Y. Notice that the shape of $\frac{\partial loss}{\partial Y}$ is the same as the shape of the output of the Forward function Y. In particular, we have

  $$\frac{\partial loss}{\partial X} = \frac{\partial loss}{\partial Y}\frac{\partial Y}{\partial X} = \frac{\partial loss}{\partial Y}W^T \quad \text{(note that we always store the current weight W)}$$

  $$\frac{\partial loss}{\partial W} = X^T\frac{\partial loss}{\partial Y} \quad \text{(Remember that we stored the X in the forward step so you can use it here)}$$

- **Checkpoint: numerical derivative** There is a standard way to validate the correctness of an analytical derivative in computational mathematics. You may always calculate the *numerical derivative* by following its definition of $\frac{df(x)}{dx} = \frac{f(x+\Delta x)-f(x)}{\Delta x}$. Here, a very small Δx can be used. If your analytical derivatives were implemented correctly, the two results should be consistent with each other. We can use this idea to check the implementation of each network layer before assembling them.

2. *Task 2: Nonlinear activation function*

- **Forward (ReLU):**
  - Input: X (#samples, #features).
  - Output: Y (#samples, #features)
  - Stored data: input X (#samples, #features)

  You will implement the ReLU activation function in this step. Then you need to create a matrix Y with the same size as the input (#samples, #features). The ReLU function is defined as

  $$ReLU(x) = max(0, x), \tag{5.1}$$

  which is an element-wise operation in the matrix. This means that you need to go through each element in the matrix Y and set

  $$Y(i, j) = max(0, A(i, j)).$$

  At the same time, you need to store the input data for the future usage in the backward step.

- **Backward (ReLU):**
  - Input: $\frac{\partial loss}{\partial Y}$ (#samples, #features).
  - Output: $\frac{\partial loss}{\partial X}$ (#samples, #features).

  Note that the gradient of ReLU is

$$grad(ReLU(x)) = \begin{cases} 1, \text{ if } x > 0 \\ 0, \text{ if } x \leq 0 \end{cases}$$

  Therefore, by the chain rule, we calculate each element of the output matrix as

$$\frac{\partial loss}{\partial X}(i,j) = \begin{cases} \frac{\partial loss}{\partial Y}(i,j) & , \text{ if } A(i,j) > 0 \\ 0 & , \text{ if } A(i,j) \leq 0 \end{cases}$$

  Notice that we need to use the value of $A(i,j)$ that you stored in the forward step for the if-statement in the expression.

  [*Hint: If you find your network loss converges to something else rather than 0, check the implementation of your ReLU backward function. It needs to strictly follow the math specified above. Other alternative implementations might get you pass the checkpoint test (mainly because in those tests you only have one layer), but it won't give you the correct derivatives in a chain.*]

3. *Task 3: MSE loss function*

- **Forward (Loss):**
  - Input: Y_{pred} (#samples, #features), Y_{truth} ((#samples, #features).
  - Output: *loss* (scalar)
  - Stored data: input $Y_{pred} - Y_{truth}$ (#samples, #features)

  Finally, we will calculate the loss function. First, you need to store the difference between the prediction Y_{pred} (output of the last linear layer) and the ground truth Y_{truth} as intermediate data (you will need it for the backward step). Then you can calculate the loss as

$$loss = \frac{1}{\#samples} \|Y_{pred} - Y_{truth}\|_2^2$$

- **Backward (Loss):**
  - Input: No input.
  - Output: $\frac{\partial loss}{\partial Y_{pred}}$(#samples, #features)

  It is interesting to notice that the gradient of the loss to the prediction is not depend on the loss itself!

$$\frac{\partial loss}{\partial Y_{pred}} = \frac{2}{\#samples}(Y_{pred} - Y_{truth}).$$

4. *Task 4: Network architecture* The network contains a vector of linear layers and a vector of non-linear layers. The sizes of the weights in the linear layers are specified by a vector of pairs of integers, which is the constructor's input. Each pair corresponds to the weight matrix T in a layer with the matrix dimension specified by the pair.

   - **Forward (pipeline):**
     - Input: X (#samples, #features0)
     - Output: Y_{pred} (#samples, #features1)

     You need to propagate the input X from the first layer to the last layer (before the loss function) by calling the forward function of each layer in sequence. The output of the previous forward is the input of the next forward. For example, for a network with k linear layers and $k-1$ activation layers, the data flow could be: $linear[0] \rightarrow activation[0] \rightarrow linear[1] \rightarrow activation[1] \rightarrow \ldots \rightarrow linear[k-2] \rightarrow activation[k-2] \rightarrow linear[k-1]$. Note that the first and the last layers should always be linear.

   - **Backward (pipeline):**
     - Input: $\frac{\partial loss}{\partial Y_{pred}}$ (#samples, #features).
     - Output: $\frac{\partial loss}{\partial X}$ (#samples, #features)

     You will propagate the output gradient (the one we got from the Forward method) back through the network. The traversing order is reversed in the backward propagation.

   - **Checkpoint: regression** We provide a sample implementation for your classification training function. There is no implementation requirement for this part. But you may use this checkpoint to validate the correctness of your network architecture (by observing the decreasing of your loss). At the same time, you may read the way it implements the gradient descent and make your own in your following implementation. The function you need to read and mimic is *Train_One_Epoch*, in which you will establish your gradient descent optimizer. You need first to forward the batched data to the network and to get the loss function. Then you may see backward the gradient of the loss back through the loss function and the network. Remember that we have stored the gradients for all the weights in the backward. Thus, you only need to update the weights with stored weight gradients. Do not forget to multiply the negative gradients by the learning rate! Of course, you can also play with the learning rate to see how it affects the performance of the gradient descent.

5. *Task 5: Classification* The framework is almost the same as the regression. The only difference is that you need to convert the image label from numbers 0-9 to its one-hot encoding vector. For example, the labels $[0, 3, 8]^T$ (with the dimension (3,1)) becomes

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{5.2}$$

This step is slightly different from the encoding vector you created in the least-squares code (in which case you used -1 instead of 0).

6. *Evaluation* After finishing everything, we are ready to test the performance of your neural network on completing the regression and classification challenges. Run the test functions in both classes and observe the performance gain. If everything is implemented correctly, you should observe the decrease of loss and the increase of accuracy in the output. Compare the results with the least-squares model. Play with the different parameters in the network to improve its performance. We also provide a visualization function for you to visualize the predicted results. Correct predictions will be marked in green and wrong predictions are in red.

Dessert Task (Extra Credits, up to 10 points)

Finish the following task(s) to receive extra credits:

1. Implement the bias in the neural network model for classification. (3 pts)
2. Design a new network architecture (e.g., $[1linear] \rightarrow [2nonlinears] \rightarrow [1linear] \rightarrow [2nonlinears] \rightarrow$ ...) and evaluate its performance. (4 pts)
3. Design a new nonlinear activation function and its derivative and use it to replace ReLU in the current architecture. Run the same learning tests and evaluate the performance by comparing the results with the standard implementation. You need to report the comparison in your final report (5 pts)
4. Employ the network to solve a new regression or classification problem. You need to collect or generate data for a new data-fitting problem and report the learning model's performance. Designing a new analytical function for regression does not count for a new problem. (5 pts)
5. The accuracy of your network ranks Top 1 in the class. (5 pts)

Submission

You are expected to submit your jupyter notebook code along with a technical report. The report needs to include the following three aspects. (1) Briefly explain your code implementation and experiment setup (e.g., network architectures, parameters, and training batch size and step size, etc.). (2) Report the network's performance by plotting the loss convergence rate and testing accuracy. (3) Visualize the failure cases and briefly discuss the potential reasons for the failed predictions. If you implement any new features for extra credits, discuss them in details in the report too.