Homework 11

Due: Thursday, April 17, 2025 at 12:00pm **(Noon)**

Programming Assignment

Introduction

In the previous assignment, we implemented a feed forward neural network using stochastic gradient descent consisting mainly of Numpy functions. In the assignment, we will instead be using Pytorch, a deep learning library that is often used for building deep neural networks. Utilizing Pytorch, we can build neural networks models more efficiently and conveniently compared to the previous assignment as you don't need to implement the details of operators and backpropagation can be done automatically.

This assignment is to familiarize yourself with Pytorch and contains two parts. First, you will re-implement HW10 with Pytorch, which contains a single layer neural network and a two-layer neural network that will predict the quality of a wine (scored out of 10) given various attributes of the wine (for example, acidity, alcohol content). You will compare the performance of both models on the UCI Wine Dataset, which you previously used in HW2 and HW10. Second, you will build a convolutional neural network and train it on the MNIST dataset. The book sections relevant to this assignment are 20.0, 20.1, 20.2, 20.3, 20.6.

Pytorch

For this assignment, we can use functions and classes provided by the Pytorch library to create our model more conveniently. For example, `torch.nn.Linear()` is a function that applies a linear transformation to the input data. This is equivalent to:

$$h(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle + b.$$

With this function, we eliminate the need to implement operator details, such as using transposing and matmul functions in python. In addition, with the tools of Pytorch we are able to implement more complex neural networks, such as a transformers network or simply adding an additional layer to our neural network. A quick starter to pytorch can be found here.

Setup

- `Preprocessing`: Once again, you will be using the UCI Wine Dataset, which contains information about various attributes of a wine and its corresponding quality rating (out of 10). However, in order to use this data with Pytorch, we must first preprocess our data, i.e., retrieving, converting, and reformatting the data into a form that can be used with Pytorch. For your convenience, we have provided you with `WineDataset` and most of `MNISTDataset`. You only need to complete the `__getitem__()` function of `MNISTDataset`.

- `Building the model`: To define a neural network in PyTorch, we create a class that inherits from `nn.Module`. We define the layers of the network in the `__init__()` function and specify how data will pass through the network in the `forward()` function, which will be called automatically when you pass data into the model. torch.nn contains the layers we will need.

  - `Linear Layers`: torch.nn.Linear defines a linear layer. A linear layer (or fully connected layer) applies a linear transformation to the input data. With Pytorch, you don't need to implement the matrix multiplication by hand, and Pytorch takes care of it in a simple function call. Also, we don't need to calculate the gradients and write backpropagation by hand. Instead, we would want

to use the out-of-the-box optimization algorithms provided by Pytorch, such as Adam, SGD, etc. Refer to the `Optimizing parameters` section below. Besides, because the composition of multiple linear layers is the same as applying one linear layer, we need some non-linear activation functions between them to increase the expressiveness of our deep models.

  - `2D Convolution Layers`: torch.nn.Conv2d defines a 2D convolution layer. A 2D convolution applies 2D filters to the input data (usually an image). The input data has shape $[N, C, H, W]$, where N is the batch size, C, H and W are the number of channels, height and width of feature maps, respectively. You need to define properties of kernels by `kernel_size`, `stride`, and `padding`. Here is a great post about the convolution operation.
  - `Activation Layers`: Rather than implementing non-linear activation functions, such as ReLU and Sigmoid activation functions, ourselves, we can utilize the activation functions provided by the Pytorch library. Using these Pytorch functions, we can apply these activation functions to our model's data. This allows us to use and test with multiple complex activation functions more easily that may improve the accuracy of our results.

- `Optimizing parameters`: To train deep models, we need loss functions and optimizers. In a single training loop, the model makes predictions on a batch of the training dataset, and backpropagates the prediction error to adjust the model's parameters. Therefore, you need loss functions to calculate the prediction error, and optimizers to backpropagate the error.

  - `Loss functions`: In the torch.nn library, there are a variety of loss functions for different tasks. In this assignment, you will use torch.nn.MSELoss to calculate the L2 loss for our wine quality prediction task, and use torch.nn.CrossEntropyLoss to calculate the cross entropy loss for the image classification task on MNIST. You may refer to the *Examples* section of CrossEntropyLoss documentation to learn how to use loss functions and optimizer together. There are basically 4 steps: initialize a loss object, empty all previous gradients, calculate the loss for the current epoch, and backpropagate the loss with the optimizer.

    * `More on cross entropy losses:` If you look at the loss functions link above, you will notice that there are 4 entropy-related losses: CrossEntropyLoss, NLLLoss, BCELoss, and BCEWithLogitsLoss. So what are the differences and which should you choose for different tasks?
      The BCELoss and BCEWithLogitsLoss are two forms of binary cross entropy and you may use them in binary classification tasks. The major difference between them is that when using BCELoss, your input values are expected to be probabilities, i.e., be the output of the sigmoid activation. However, the input values of BCEWithLogitsLoss are logits (the sigmoid activation will be called automatically within BCEWithLogitsLoss).
      CrossEntropyLoss and NLLLoss are two losses that calculate the cross entropy loss for multi-classification tasks. Similar to BCEWithLogitsLoss, the input values of CrossEntropyLoss are expected to be logits. However, NLLLoss is a little tricky because it's usually paired with the torch.nn.LogSoftmax layer. For more details, you may refer to the *Examples* section of NLLLoss documentation.

  - `Optimizers`: torch.optim is a package that implements a variety of different optimization algorithms that we have used previously, such as the stochastic gradient descent algorithm. To use an optimizer, you want to initialize an optimizer instance, which holds the trainable parameters and updates the parameters based on the computed gradients from the optimizing algorithm. For each epoch, you would want to take an optimization step, which will update your parameters. Hint: Pytorch implements this for you already with the step() function. Look here if you want more information.

- `Suggested architecture and hyperparameters:`

  We recommend creating your own model at first. If you are struggling and cannot get desired results, you can refer to our architecture.

  - `OneLayerNN`
    - `Optimizer:` SGD with learning rate 0.01.
    - `Batch size:` 64
    - `Number of training epochs:` 25
    - `Model:`
      - Simply a linear layer, no activation required.
  - `TwoLayerNN`
    - `Optimizer:` SGD with learning rate 0.01.
    - `Batch size:` 64
    - `Number of training epochs:` 25
    - `Model:`
      - Linear layer with hidden size of 32.
      - Sigmoid activation.
      - Another linear layer, no activation required.
  - `CNN`
    - `Optimizer:` SGD with learning rate 0.01.
    - `Batch size:` 64
    - `Number of training epochs:` 20
    - `Model:`
      - Conv2d layer, out_channels=16, kernel_size=3, stride=1.
      - ReLU activation.
      - Conv2d layer, out_channels=32, kernel_size=3, stride=1.
      - ReLU activation.
      - Flatten layer.
      - Linear layer, no activation required.

Training Neural Networks

The primary objective of training a neural network is to find a set of parameters that minimize the loss of our network, which in this assignment, is L2 loss for wine and cross entropy loss for MNIST. Using Pytorch, updating the parameters that minimize the loss is completed automatically.

Visualization

In `utils.py` we have provided several helper functions to visualize your models using matplotlib, a useful Python library for plotting graphs. `visualize_loss()` and `visualize_accuracy()` visualizes how your loss and accuracy change per batch. These graphs will have the batch id on its x-axis and losses or accuracy values on its y-axis. You should use this during your project report to compare the loss of different functions and parameters. We also provide another `visualize_image()` function to visualize samples from the MNIST dataset, annotated with groud truth labels and your predicted categories. `visualize_misclassified_image()` is a similar function but it only plots misclassified images. `visualize_confusion_matrix()` plots the confusion matrix of your CNN model. Hint: Uncomment the lines that call these functions.

Stencil Code & Data

UCI Wine

The `data/wine.txt` file contains 11 attributes of wine and its corresponding quality rating (out of 10) in the first column.

8x8 Hand-Written Digits

In the `data/digits.csv` file, each row is an observation of a 8 x 8 hand-written digit (0 - 9), containing a label in the first column and 8 x 8 = 64 features (pixel values) in the rest of columns.

Data Format

We have written almost all the preprocessing code for you. In Pytorch, a dataset class inherits `torch.utils.data.Dataset` and overrides the `__getitem__()` method. You also need to initialize a `torch.utils.data.Dataloader` instance for batching purposes. In this assignment, suppose `dataloader_train` is an instance of the `Dataloader` class, you can get a batch of data by

```
for X, Y in dataloader_train:
   # Your code
```

Here `X` and `Y` are Pytorch tensors. Similar to Numpy tensors, you can use `X.shape` to get the shape of a tensor. `X` is a batch of features and Y is a batch of labels.

- For the wine dataset, `X.shape` is $[N, 11]$ and `Y.shape` is $[N, 1]$, where N is the batch size.
- For the MNIST dataset, `X.shape` is $[N, 1, 8, 8]$, where 1 means each image has 1 channel (because we are using gray images), and the last two dimensions are height and width, respectively. `Y.shape` is $[N]$.

Stencil Code

You can find and download the assignment here: HW11 on Github.

We have provided the following stencil code:

- `main.py` is the entry point of program which will read in the dataset, run the models, print and visualize the results. You will tune hyperparameters, initialize optimizer and loss instances in this file.
- `models.py` contains `OneLayerNN`, `TwoLayerNN`, and `CNN` models which you will be implementing. You also need to implement the functions for training, testing, and calculating the number of correct predictions.
- `utils.py` is responsible for creating datasets and dataloaders. You can also find visualization functions here. In this file, you only need to complete the `__getitem__()` method of `MNISTDataset`.

We recommend implementing `utils.py` first. Then you can implement `OneLayerNN`, `train()`, `test()` in `models.py` and `test_linear_nn()` in `main.py`. For `train()` and `test()`, you don't need to implement the code within `if correct_num_func:` blocks for now. After you have successfully trained `OneLayerNN`, you can move on to `TwoLayerNN`. Finally you will implement `CNN`, `correct_predict_num()`, and the remaining part of `train()` and `test()` in `models.py`, as well as `test_cnn()` in `main.py`.

You should not need to modify any code to `main.py` outside of the TODOs. If you do for debugging or other purposes, please make sure all of your additions are commented out in the final handin. All the functions you need to fill in reside in `models.py, mains.py,` and `utils.py`, marked by `TODO`s. To run the

program, you need to uncomment `test_xx()` function calls in the `main()` function of `main.py` to train and test the model. Then run `python main.py` in a terminal with the course environment set up.

As a hint, please uncomment both `test_linear_nn(nn_type='one_layer')` and `test_linear_nn(nn_type='two_layer')` when testing `TwoLayerNN` on your machine. This is because we use 0 as the random seed, and if you only run `test_linear_nn(nn_type='two_layer')`, you will get different initialized parameters and the shuffling of the training set will be different. Consequently, you may get a higher loss. But as long as your implementation is correct, because the autograder runs `TwoLayerNN` after `OneLayerNN`, your code will still pass the tests on the autograder.

If you're curious and would like to read about the datasets, you can find more information about the Wine Dataset **here**, and for more information of MNIST dataset **here** but it is strongly recommended that you use the versions that we've provided in the course directory to maintain consistent formatting.

Written Report

Guiding Questions

(a). Comment on your hyperparameter choices. These include the learning rate and the number of epochs for training. (6 points)

(b). For the convolutional neural network, try different settings and discuss your findings. You may try different optimizers, batch size, number of layers, kernel size, etc. (6 points)

(c). Zhang et al. find that deep neural networks are powerful enough to easily fit random labels. Here is the conference talk given by Zhang in ICLR2017. In this paper, the authors question the measure of deep model complexity by only considering properties of the model itself. We have provided functions for you to reimplement one experiment mentioned in the paper, which is fitting randomized training labels. We will do this experiment with our CNN model and the MNIST dataset.

In `test_cnn()` of `main.py`, set `shuffle_train_label = True`. Because it takes longer for the model to fit the random label, you also need to increase `num_epoch`. If you are using the recommended CNN architecture, you can set `num_epoch = 280`.

Based on the training and testing accuracy, answer the following questions.

- What is the likely difference between decision boundaries of datasets with shuffled training labels and true training labels? Why do you expect this? (6 points)
- In the bias-complexity tradeoff lecture, we learned that when the model complexity increases, the generalization ability tends to decrease. However, in this experiment with shuffled training labels, the model structure remains the same, but we notice a huge divergence between model performance on the training set and test set. Do you think the number of parameters is the only metric for measuring a deep model's complexity? If not, what other factors may contribute to model complexity? Will the 'complexity' in the bias-complexity tradeoff theory change when the dataset (or task) changes? Justify your answer. (6 points)

Grading

Loss and Accuracy Targets

We are expecting the following testing metrics for each of the models:

- `OneLayerNN`: Test loss ≤ 0.70 on Wine. Otherwise, you may receive partial credits if loss ≤ 0.80.
- `TwoLayerNN`: Test loss ≤ 0.60 on Wine. Otherwise, you may receive partial credits if loss ≤ 0.70.
- `CNN`: Test accuracy ≥ 0.95 on MNIST. Otherwise, you may receive partial credits if accuracy ≥ 0.85.

As always, we will be grading your code based on correctness and not based on whether or not you meet these targets.

Hyperparameters

To verify the correctness of your implementation, check that your model satisfies the above training loss benchmarks. Feel free to fine-tune the values of hyperparameters. We will use the hyperparameter values that you choose when testing your model on the wine datasets and the MNIST datasets. And we strongly suggest that you first verify the correctness of your implementation before modifying hyperparameters.

Breakdown

MNISTDataset	10%
OneLayerNN	12%
TwoLayerNN	16%
CNN	38%
Report	24%
Total	100%

Handing in

You will hand in both the written assignment and the coding portion on gradescope, separately.

1. Submit your hw11 github repo containing all your source code and your project report named **report.pdf** on gradescope under "Homework 11 Code". **report.pdf** should live in the root directory of your code folder; the autograder will check for the existence of this file and inform you if it is not found. For questions, please consult the download/submission guide.

If you have questions on how to set up or use Gradescope, ask on Edstem! For this assignment, you should have written answers for Problems 1 and 2.

Obligatory Note on Academic Integrity

Plagiarism—don't do it.

As outlined in the Brown Academic Code, attempting to pass off another's work as your own can result in failing the assignment, failing this course, or even dismissal or expulsion from Brown. More than that, you will be missing out on the goal of your education, which is the cultivation of your own mind, thoughts, and abilities. Please review this course's collaboration policy and, if you have any questions, please contact a member of the course staff.