TAMU CSCE 625 (Spring 2026)
Assignment #3

1 Starting Point

Scripts: From the GitHub repository, you will find:

- Script `dnn_misc.py`, which you will be amending by adding code for questions in Section 3.
- Script `dnn_cnn_2.py`, which you will be amending by adding code for questions in Section 3.
- Various other Python scripts: `dnn_mlp.py`, `dnn_mlp_nonlinear.py`, `dnn_cnn.py`, `hw5_dnn_check.py`, `dnn_im2col.py`, and `data_loader.py`, which you are not allowed to modify.
- Various scripts: `q33.sh`, `q34.sh`, `q35.sh`, `q36.sh`, `q37.sh`, `q38.sh`, and `q310.sh`; you will use these to generate output files.

Data: You will use `mnist_subset.json`.

Submission: Please submit a single `.zip` file named `Assignment_3_firstname_lastname.zip` (e.g., `Assignment_3_Cheng_Zhang.zip`) to Canvas. The following will constitute your submission:

- The two Python scripts `dnn_misc.py` and `dnn_cnn_2.py`, amended with the code you added for Section 3.
- Seven `.json` files, which will be the output of the seven scripts above. We reserve the right to run your code to regenerate these files, but you are expected to include them.

  ```
  MLP_lr0.01_m0.0_w0.0_d0.0.json
  MLP_lr0.01_m0.0_w0.0_d0.5.json
  MLP_lr0.01_m0.0_w0.0_d0.95.json
  LR_lr0.01_m0.0_w0.0_d0.0.json
  CNN_lr0.01_m0.0_w0.0_d0.5.json
  CNN_lr0.01_m0.9_w0.0_d0.5.json
  CNN2_lr0.001_m0.9_w0.0_d0.5.json
  ```

- a `collaboration.txt` that lists with whom you have discussed the homework.

  Collaboration: You may discuss your homework with your classmates. However, you need to write your solutions and submit them separately. In your submission, you need to list with whom you have discussed the homework in a `.txt` file `collaboration.txt`. Please list each classmate's name and NetID as a row in the .txt file (e.g., Cheng Zhang, chzhang). That is, if you discussed the homework with two classmates, your `.txt` file will have two rows. If you did not discuss your homework with your classmates, just write "no discussion" in `collaboration.txt`. Please consult the syllabus for what is and is not an acceptable collaboration.

2 Introduction

2.1 Dataset

We will use mnist_subset (images of handwritten digits from 0 to 9). The dataset is stored in a JSON-formatted file mnist_subset.json. You can access its training, validation, and test splits using the keys 'train', 'valid', and 'test', respectively. For example, suppose we load mnist_subset.json to the variable x. Then, $x['train']$ refers to the training set of mnist_subset. This set is a list with two elements: $x['train'][0]$ containing the features of size N (samples) $\times D$ (dimension of features), and $x['train'][1]$ containing the corresponding labels of size N.

2.2 Tasks

You will implement neural networks (Section. 3). Specifically, you will:

- Finish the implementation of all Python functions in our template codes.
- Run your code by calling the specified scripts to generate output files.
- Submit (1) all *.py files, and (2) all *.json files that you have amended or created.

In the next subsection, we will provide a **high-level** checklist of what you need to do. You are not responsible for loading/pre-processing data; we have done that for you. For specific instructions, please refer to the text in Section. 3, as well as the corresponding Python scripts.

2.2.1 Neural networks

Preparation: Read Section 3 as well as dnn_mlp.py and dnn_cnn.py.

Coding: First, in dnn_misc.py, finish implementing:

- forward and backward functions in class linear_layer
- forward and backward functions in class relu
- backward function in class dropout (before that, please read forward function)

Refer to dnn_misc.py and Section 3 for more information.

Second, in dnn_cnn.py, finish implementing the main function. There are five TODO items. Refer to dnn_cnn.py and Section 3 for more information.

Running your code: Run the scripts q33.sh, q34.sh, q35.sh, q36.sh, q37.sh, q38.sh, and q310.sh after you finish your implementation. This will generate, respectively:

```
MLP_lr0.01_m0.0_w0.0_d0.0.json
MLP_lr0.01_m0.0_w0.0_d0.5.json
MLP_lr0.01_m0.0_w0.0_d0.95.json
LR_lr0.01_m0.0_w0.0_d0.0.json
CNN_lr0.01_m0.0_w0.0_d0.5.json
CNN_lr0.01_m0.9_w0.0_d0.5.json
CNN2_lr0.001_m0.9_w0.0_d0.5.json
```

What to submit: Submit dnn_misc.py, dnn_cnn_2.py, and the above seven .json files.

2.3 Cautions

Please do not import packages that are not listed in the provided code. Follow the instructions in each section strictly to code up your solutions. **Do not change the output format. Do not modify the code unless we instruct you to do so.** A homework solution that does not match the provided setup, such as format, name, initialization, etc., **will not** be graded. It is your responsibility to **make sure that your code runs with the provided commands and scripts.**

3 Neural networks: Multi-Layer Perceptrons (MLPs) and Convolutional Neural Networks (CNNs)

3.1 Background

In recent years, neural networks have been one of the most powerful machine learning models. Many toolboxes/platforms (e.g., TensorFlow, PyTorch) are publicly available for efficiently constructing and training neural networks. The core idea of these toolboxes is to treat a neural network as a combination of *data transformation (or mathematical operation) modules.*

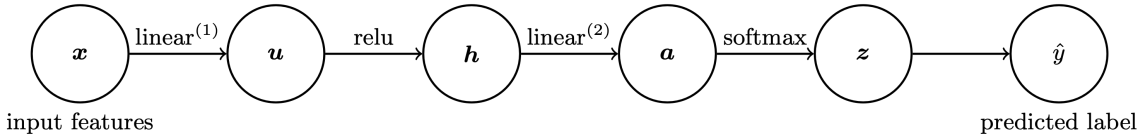

Figure 1: A diagram of a multi-layer perceptron (MLP). The edges mean mathematical operations (modules), and the circles mean variables. The term **relu** stands for rectified linear units.

For example, in Figure 1 we provide a diagram of a multi-layer perceptron (MLP) for a K-class classification problem. The edges correspond to modules, and the circles correspond to variables. Let $(\boldsymbol{x} \in \mathbb{R}^D, y \in \{1, 2, \ldots, K\})$ be a labeled instance. Such an MLP performs the following computations:

$$\begin{aligned}
\textbf{input features:} \quad & \boldsymbol{x} \in \mathbb{R}^D && (1)\\
\textbf{linear}^{(1)}: \quad & \boldsymbol{u} = \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}, \quad \boldsymbol{W}^{(1)} \in \mathbb{R}^{M\times D}, \quad \boldsymbol{b}^{(1)} \in \mathbb{R}^M && (2)\\
\textbf{relu:} \quad & \boldsymbol{h} = \max\{0, \boldsymbol{u}\} = \begin{bmatrix} \max\{0, u[1]\} \\ \vdots \\ \max\{0, u[M]\} \end{bmatrix} && (3)\\
\textbf{linear}^{(2)}: \quad & \boldsymbol{a} = \boldsymbol{W}^{(2)}\boldsymbol{h} + \boldsymbol{b}^{(2)}, \quad \boldsymbol{W}^{(2)} \in \mathbb{R}^{K\times M}, \quad \boldsymbol{b}^{(2)} \in \mathbb{R}^K && (4)\\
\textbf{softmax:} \quad & \boldsymbol{z} = \begin{bmatrix} \frac{e^{a[1]}}{\sum_k e^{a[k]}} \\ \vdots \\ \frac{e^{a[K]}}{\sum_k e^{a[k]}} \end{bmatrix} && (5)\\
\textbf{predicted label:} \quad & \hat{y} = \arg\max_k z[k]. && (6)
\end{aligned}$$

For a K-class classification problem, one popular loss function for training is the cross-entropy loss (i.e., to learn $\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}, \boldsymbol{b}^{(1)}, \boldsymbol{b}^{(2)}$).

$$\ell = -\sum_k \mathbf{1}[y == k] \log z[k], \tag{7}$$

$$\text{where} \quad \mathbf{1}[\text{True}] = 1; \text{otherwise}, 0. \tag{8}$$

For ease of notation, let us define the one-hot (i.e., 1-of-K) encoding:

$$\boldsymbol{y} \in \mathbb{R}^K \quad \text{and} \quad y[k] = \begin{cases} 1, & \text{if } y = k, \\ 0, & \text{otherwise.} \end{cases} \tag{9}$$

so that

$$\ell = -\sum_k y[k] \log z[k] = -\boldsymbol{y}^T \begin{bmatrix} \log z[1] \\ \vdots \\ \log z[K] \end{bmatrix} = -\boldsymbol{y}^T \log \boldsymbol{z}. \tag{10}$$

We can then perform error-backpropagation, a way to compute partial derivatives (or gradients) w.r.t. the parameters of a neural network, and use gradient-based optimization to learn the parameters.

3.2 Modules

Now we will provide more information on modules for this assignment. Each module has its own parameters (but note that a module may have no parameters). Moreover, each module can perform a *forward pass* and a *backward pass*. The forward pass performs the computation of the module, given the input to the module. The backward pass computes the partial derivatives of the loss function with respect to the input and parameters, given the partial derivatives of the loss function with respect to the output of the module. Consider a module `<module-name>`. Let `<module-name>.forward` and `<module-name>.backward` be its forward and backward passes, respectively.

For example, the linear module may be defined as follows.

$$\text{Forward pass:} \quad \boldsymbol{u} = \text{linear}^{(1)}.\text{forward}(\boldsymbol{x}) = \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}, \qquad (11)$$

$$\text{where} \quad \boldsymbol{W}^{(1)}\text{and } \boldsymbol{b}^{(1)} \text{ are its parameters.}$$

$$\text{Backward pass:} \quad \left[\frac{\partial \ell}{\partial \boldsymbol{x}}, \frac{\partial \ell}{\partial \boldsymbol{W}^{(1)}}, \frac{\partial \ell}{\partial \boldsymbol{b}^{(1)}}\right] = \text{linear}^{(1)}.\text{backward}(\boldsymbol{x}, \frac{\partial \ell}{\partial \boldsymbol{u}}). \qquad (12)$$

Let us assume that we have implemented all the desired modules. Then, getting $\hat{\boldsymbol{y}}$ for $\boldsymbol{x}$ is equivalent to running the forward pass of each module in order, given $\boldsymbol{x}$. All the intermediate variables (i.e., $\boldsymbol{u}, \boldsymbol{h}$, etc.) will be computed along the forward pass. Similarly, computing the partial derivatives of the loss function with respect to the parameters is equivalent to running the backward pass of each module in reverse order, given $\frac{\partial \ell}{\partial \boldsymbol{z}}$.

In this question, we provide a Python environment based on the idea of modules. Every module is defined as a class, so you can create multiple modules of the same functionality by creating multiple object instances of the same class. Your work is to finish the implementation of several modules, where these modules are elements of a multi-layer perceptron (MLP) or a convolutional neural network (CNN). We will apply these models to the 10-class classification problem in MNIST. We will train the models using stochastic gradient descent with mini-batch and explore how different hyperparameters of optimizers and regularization techniques affect training and validation accuracies over training epochs. For a deeper understanding, check out the seminal work of Yann LeCun et al., “Gradient-based learning applied to document recognition,” 1998.

We give a specific example below. Suppose that, at iteration t, you sample a mini-batch of N examples $\{(\boldsymbol{x}_i \in \mathbb{R}^D, \boldsymbol{y}_i \in \mathbb{R}^K)\}_{i=1}^N$ from the training set ($K = 10$). Then, the loss of such a mini-batch given by Figure 1 is:

$$\begin{aligned}
\ell_{mb} &= \frac{1}{N}\sum_{i=1}^{N} \ell(\text{softmax.forward}(\text{linear}^{(2)}.\text{forward}(\text{relu.forward}(\text{linear}^{(1)}.\text{forward}(\boldsymbol{x}_i)))), \boldsymbol{y}_i) && (13)\\
&= \frac{1}{N}\sum_{i=1}^{N} \ell(\text{softmax.forward}(\text{linear}^{(2)}.\text{forward}(\text{relu.forward}(\boldsymbol{u}_i))), \boldsymbol{y}_i) && (14)\\
&= \dots && (15)\\
&= \frac{1}{N}\sum_{i=1}^{N} \ell(\text{softmax.forward}(\boldsymbol{a}_i), \boldsymbol{y}_i) && (16)\\
&= \frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} y_i[k] \log z_i[k]. && (17)
\end{aligned}$$

That is, in the forward pass, we can perform the computation of a certain module to all the N input examples, and then pass the N output examples to the next module. This is the same case for the backward pass. For example, according to Figure 1, given the partial derivatives of the loss with respect to $\{\boldsymbol{a}_i\}_{i=1}^N$,

$$\frac{\partial \ell_{mb}}{\partial \{\boldsymbol{a}_i\}_{i=1}^N} = \begin{bmatrix} \left(\frac{\partial \ell_{mb}}{\partial \boldsymbol{a}_1}\right)^T \\ \left(\frac{\partial \ell_{mb}}{\partial \boldsymbol{a}_2}\right)^T \\ \vdots \\ \left(\frac{\partial \ell_{mb}}{\partial \boldsymbol{a}_{N-1}}\right)^T \\ \left(\frac{\partial \ell_{mb}}{\partial \boldsymbol{a}_N}\right)^T \end{bmatrix} \in \mathbb{R}^{N\times K}, \tag{18}$$

$\texttt{linear}^{(2)}\texttt{.backward}$ will compute $\frac{\partial \ell_{mb}}{\partial \{\boldsymbol{h}_i\}_{i=1}^N}$ and pass it back to `relu.backward`.

We note that, $\mathbb{R}^{N\times K}$ means that $\frac{\partial \ell_{mb}}{\partial \{\boldsymbol{a}_i\}_{i=1}^N}$ is represented in the row-wise fashion for Python code.

3.3 Preparation

Q3.1 Please read through `dnn_mlp.py` and `dnn_cnn.py`. Both files will use modules defined in `dnn_misc.py` (which you will modify). Your work is to understand how modules are created, how they are linked to perform the forward and backward passes, and how parameters are updated based on gradients (and momentum). The architectures of the MLP and CNN defined in `dnn_mlp.py` and `dnn_cnn.py` are shown in Fig. 2 and Fig. 3, respectively.

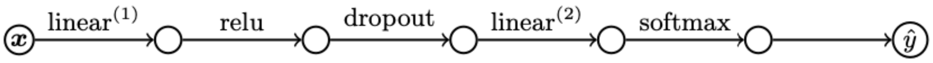

Figure 2: The diagram of the MLP implemented in dnn mlp.py. The circles mean variables and the edges mean modules.

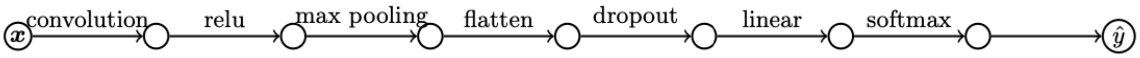

Figure 3: The diagram of the CNN implemented in `dnn_cnn.py`. The circles correspond to variables and edges correspond to modules. Note that the input to CNN may not be a vector (e.g., in `dnn_cnn.py` it is an image, which can be represented as a 3-dimensional tensor). The flattened layer is to reshape its input into a vector.

What to submit: Nothing.

3.4 Coding: Modules

[50 points: 10+10+10+10+10]

Q3.2 You will modify `dnn_misc.py`. This script defines all modules that you will need to construct the MLP and CNN in `dnn_mlp.py` and `dnn_cnn.py`, respectively. You have three tasks.

1. Finish the implementation of `forward` and `backward` functions in `class linear_layer`. Please follow Equation 2 for the forward pass and derive the partial derivatives accordingly.
2. Finish the implementation of `forward` and `backward` functions in `class relu`. Please follow Equation 3 for the forward pass and derive the partial derivatives accordingly.
3. Finish the implementation of the `backward` function in `class dropout`. We define the forward and the backward passes of dropout as follows.

Forward pass:

$$\mathbf{s} = \text{dropout.forward}(\mathbf{q} \in \mathbb{R}^J) = \frac{1}{1-r} \times \begin{bmatrix} \mathbf{1}[p[1] \geq r] \times q[1] \\ \vdots \\ \mathbf{1}[p[J] \geq r] \times q[J] \end{bmatrix}, \tag{19}$$

$$\text{where } p[j] \text{is sampled uniformly from} [0,1), \forall j \in \{1, \ldots, J\},$$
$$\text{and } r \in (0,1) \text{ is a pre-defined scalar named dropout rate.} \tag{20}$$

Backward pass:

$$\frac{\partial \ell}{\partial \boldsymbol{q}} = \text{dropout.backward}(\boldsymbol{q}, \frac{\partial \ell}{\partial \boldsymbol{s}}) = \frac{1}{1-r} \times \begin{bmatrix} \mathbf{1}[p[1] \geq r] \times \frac{\partial \ell}{\partial s[1]} \\ \vdots \\ \mathbf{1}[p[J] \geq r] \times \frac{\partial \ell}{\partial s[J]} \end{bmatrix}. \tag{21}$$

Note that $p[j]$, $j \in \{1, \ldots, J\}$ and r are not learned, so we do not need to compute the derivatives with respect to them. Moreover, $p[j]$, $j \in \{1, \ldots, J\}$ are re-sampled every forward pass (for every data instance) and are kept for the following backward pass. The dropout rate r is set to 0 during testing.

Detailed descriptions/instructions about each pass (i.e., what to compute and what to return) are included in `dnn_misc.py`. Please read carefully.

Note that in this script, we do `import numpy as np`. Thus, to call a function `XX` from numpy, please use `np.XX`.

What to do and submit: Finish the implementation of 5 functions specified above in `dnn_misc.py`. Submit your completed `dnn_misc.py`. We do provide a checking code `assignment3_dnn_check.py` to check your implementation. Just simply run: `python3 assignment3_dnn_check.py`.

3.5 Testing `dnn_misc.py` with Multi-Layer Perceptron (MLP)

[16 points: 4+4+4+4]

Q3.3 *What to do and submit:* Run script `q33.sh`. It will output `MLP_lr0.01_m0.0_w0.0_d0.0.json`. *What it does:* `q33.sh` will run `python3 dnn_mlp.py` with learning rate 0.01, no momentum, no weight decay, and dropout rate 0.0. The output file stores the training and validation accuracies over 30 training epochs.

Q3.4 *What to do and submit:* Run script `q34.sh`. It will output `MLP_lr0.01_m0.0_w0.0_d0.5.json`. *What it does:* `q34.sh` will run `python3 dnn_mlp.py --dropout_rate 0.5` with learning rate 0.01, no momentum, no weight decay, and dropout rate 0.5. The output file stores the training and validation accuracies over 30 training epochs.

Q3.5 *What to do and submit:* Run script `q35.sh`. It will output `MLP_lr0.01_m0.0_w0.0_d0.95.json`. *What it does:* `q35.sh` will run `python3 dnn_mlp.py --dropout_rate 0.95` with learning rate 0.01, no momentum, no weight decay, and dropout rate 0.95. The output file stores the training and validation accuracies over 30 training epochs.

You will observe that the model in Q3.4 will give better validation accuracy (at epoch 30) compared to Q3.3. Specifically, dropout is widely used to prevent overfitting. However, if we use a too-large dropout rate (like the one in Q3.5), the validation accuracy (together with the training accuracy) will be relatively lower, essentially underfitting the training data.

Q3.6 *What to do and submit:* Run script `q36.sh`. It will output `LR_lr0.01_m0.0_w0.0_d0.0.json`. *What it does:* `q36.sh` will run `python3 dnn_mlp_nononlinear.py` with learning rate 0.01, no momentum, no weight decay, and dropout rate 0.0. The output file stores the training and validation accuracies over 30 training epochs.

The network has the same structure as the one in Q3.3, except that we remove the relu (nonlinear) layer. You will see that the validation accuracies drop significantly (the gap is around 0.03).

3.6 Testing `dnn_misc.py` with convolutional neural networks (CNN)

[8 points: 4+4]

Q3.7 *What to do and submit:* Run script `q37.sh`. It will output `CNN_lr0.01_m0.0_w0.0_d0.5.json`. *What it does:* `q37.sh` will run `python3 dnn_cnn.py` with learning rate 0.01, no momentum, no weight decay, and dropout rate 0.5. The output file stores the training and validation accuracies over 30 training epochs.

Q3.8 *What to do and submit:* Run script `q38.sh`. It will output `CNN_lr0.01_m0.9_w0.0_d0.5.json`. *What it does:* `q38.sh` will run `python3 dnn_cnn.py --alpha 0.9` with learning rate 0.01, momentum 0.9, no weight decay, and dropout rate 0.5. The output file stores the training and validation accuracies over 30 training epochs.

You will see that Q3.8 will lead to faster convergence than Q3.7 (i.e., the training/validation accuracies will be higher than 0.94 after 1 epoch). That is, using momentum will lead to more stable updates of the parameters.

3.7 Coding: Building a deeper architecture

[22 points]

Q3.9 The CNN architecture in `dnn_cnn.py` has only one convolutional layer. In this question, you are going to construct a two-convolutional-layer CNN (see Figure 4) using the modules you implemented in Q3.2. Please modify the `main` function in `dnn_cnn_2.py.`. The code in `dnn_cnn_2.py.` is similar to that in `dnn_cnn.py`, except that there are several parts marked as `TODO`. You need to fill in your code so as to construct the CNN in Figure 4.

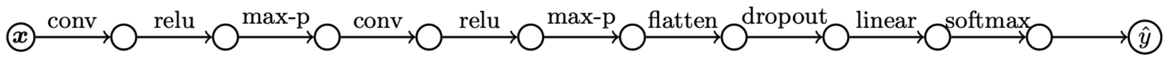

Figure 4: The diagram of the CNN you are going to implement in `dnn_cnn_2.py.` The term conv stands for convolution; max-p stands for max pooling. The circles correspond to variables and the edges correspond to modules. Note that the input to CNN may not be a vector (e.g., in `dnn_cnn_2.py.` it is an image, which can be represented as a 3-dimensional tensor). The flattened layer is to reshape its input into a vector.

What to do and submit: Finish the implementation of the `main` function in `dnn_cnn_2.py` (search for `TODO` in `main`). Submit your completed `dnn_cnn_2.py`.

3.8 Testing `dnn_cnn_2.py`

[4 points]

Q3.10 *What to do and submit:* Run script `q310.sh`. It will output `CNN2_lr0.001_m0.9_w0.0_d0.5.json`. *What it does:* `q310.sh` will run `python3 dnn_cnn_2.py --alpha 0.9` with learning rate 0.01, momentum 0.9, no weight decay, and dropout rate 0.5. The output file stores the training and validation accuracies over 30 training epochs.

You will see that you can achieve slightly higher validation accuracies than those in Q3.8.