CSCI567 2017 Homework Assignment 2

Due date and time Oct. 8th (Sunday), 11:59pm

Starting point Your repository will have now a directory 'homework2/'. Please do not change the name of this repository or the names of any files we have added to it. Please perform a `git pull` to retrieve these files. You will find within it:

- python script `logistic_prog.py`, which you will be amending by adding code for questions in Sect. 4.
- python script `dnn_misc.py`, which you will be amending by adding code for questions in Sect. 5.
- python script `dnn_cnn_2.py`, which you will be amending by adding code for questions in Sect. 5.
- Various other python scripts: `dnn_mlp.py`, `dnn_mlp_nononlinear.py`, `dnn_cnn.py`, and `dnn_im2col.py`, which you are **not allowed** to modify.
- Various scripts: `q43.sh`, `q53.sh`, `q54.sh`, `q55.sh`, `q56.sh`, `q57.sh`, `q58.sh`, `q510.sh`; you will use these to generate output files.
- Dataset: **mnist_subset.json**

Submission Instructions The following will constitute your submission:

- The three python scripts above, amended with the code you added for Sect. 4 and Sect. 5. Be sure to commit your changes!
- A PDF report named `firstname_lastname_USCID.pdf`, which contains your solutions for questions in Sect. 1 and Sect. 2. For your written report, your answers must be typeset with LaTeX and generated as a PDF file. No handwritten submission will be permitted. There are many free integrated LaTeX editors that are convenient to use, please Google search them and choose the one(s) you like the most. This `http://www.andy-roberts.net/writing/latex` seems like a good tutorial.
- Eight .json files, which will be the output of the eight scripts above. We reserve the right to run your code to regenerate these files, but you are expected to include them.
`logistic_res.json`
`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
```

Collaboration You may discuss with your classmates. However, you need to write your own solutions and submit separately. Also in your written report, you need to list with whom you have discussed for each problem. Please consult the syllabus for what is and is not acceptable collaboration.

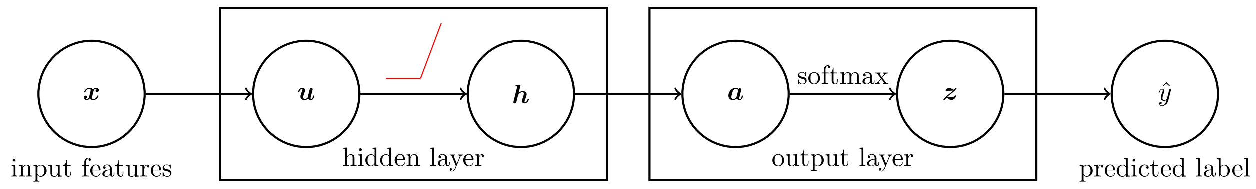

Figure 1: A diagram of a 1 hidden-layer multi-layer perceptron (MLP). *The edges mean mathematical operations, and the circles mean variables.* Generally we call the combination of a linear (or affine) operation and a nonlinear operation (like element-wise sigmoid or the rectified linear unit (relu) operation as in eq. (3)) as a hidden layer.

Algorithmic component

1 Neural networks (error-backpropagation, initialization, and nonlinearity)

[Recommended maximum time spent: 1 hour]

In the lecture (see lec8.pdf), we have talked about error-backpropagation, a way to compute partial derivatives (or gradients) w.r.t the parameters of a neural network. We have also mentioned that optimization is challenging and nonlinearity is important for neural networks. In this question, you are going to (Q1.1) practice error-backpropagation, (Q1.2) investigate how initialization affects optimization, and (Q1.3) the importance of nonlinearity.

Specifically, you are given the following 1-hidden layer multi-layer perceptron (MLP) for a K-class classification problem (see Fig. 1 for illustration and details), and ($\boldsymbol{x} \in \mathbb{R}^D, y \in \{1, 2, \cdots, K\}$) is a labeled instance,

$$\boldsymbol{x} \in \mathbb{R}^D \tag{1}$$

$$\boldsymbol{u} = \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)} \qquad , \boldsymbol{W}^{(1)} \in \mathbb{R}^{M\times D} \text{ and } \boldsymbol{b}^{(1)} \in \mathbb{R}^{M} \tag{2}$$

$$\boldsymbol{h} = \max\{0, \boldsymbol{u}\} = \begin{bmatrix} \max\{0, u_1\} \\ \vdots \\ \max\{0, u_M\} \end{bmatrix} \tag{3}$$

$$\boldsymbol{a} = \boldsymbol{W}^{(2)}\boldsymbol{h} + \boldsymbol{b}^{(2)} \qquad , \boldsymbol{W}^{(2)} \in \mathbb{R}^{K\times M} \text{ and } \boldsymbol{b}^{(2)} \in \mathbb{R}^{K} \tag{4}$$

$$\boldsymbol{z} = \begin{bmatrix} \frac{e^{a_1}}{\sum_k e^{a_k}} \\ \vdots \\ \frac{e^{a_K}}{\sum_k e^{a_k}} \end{bmatrix} \tag{5}$$

$$\hat{y} = \arg\max_k z_k. \tag{6}$$

For K-class classification problem, one popular loss function for training is the cross-entropy loss,

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

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

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

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

so that

$$l = -\sum_k y_k \log z_k = -\boldsymbol{y}^T \begin{bmatrix} \log z_1 \\ \vdots \\ \log z_K \end{bmatrix} = -\boldsymbol{y}^T \log \boldsymbol{z}. \tag{10}$$

Q1.1 Assume that you have computed $\boldsymbol{u}$, $\boldsymbol{h}$, $\boldsymbol{a}$, $\boldsymbol{z}$, given ($\boldsymbol{x}$, $\boldsymbol{y}$). Please first express $\frac{\partial l}{\partial \boldsymbol{u}}$ in terms of $\frac{\partial l}{\partial \boldsymbol{a}}$, $\boldsymbol{u}$, $\boldsymbol{h}$, and $\boldsymbol{W}^{(2)}$.

$$\frac{\partial l}{\partial \boldsymbol{u}} = ?$$

Then express $\frac{\partial l}{\partial \boldsymbol{a}}$ in terms of $\boldsymbol{z}$ and $\boldsymbol{y}$.

$$\frac{\partial l}{\partial \boldsymbol{a}} = ?$$

Finally, compute $\frac{\partial l}{\partial \boldsymbol{W}^{(1)}}$ and $\frac{\partial l}{\partial \boldsymbol{b}^{(1)}}$ in terms of $\frac{\partial l}{\partial \boldsymbol{u}}$ and $\boldsymbol{x}$. Compute $\frac{\partial l}{\partial \boldsymbol{W}^{(2)}}$ in terms of $\frac{\partial l}{\partial \boldsymbol{a}}$ and $\boldsymbol{h}$.

$$\frac{\partial l}{\partial \boldsymbol{W}^{(1)}} = ?$$
$$\frac{\partial l}{\partial \boldsymbol{b}^{(1)}} = ?$$
$$\frac{\partial l}{\partial \boldsymbol{W}^{(2)}} = ?$$

You only need to write down the final answers of the above 5 question marks. You are encouraged to use matrix/vector forms to simplify your answers. Note that $\max\{0, u\}$ is not differentiable w.r.t. u at $u = 0$. Please note that

$$\frac{\partial \max\{0, u\}}{\partial u} = \begin{cases} 1, & \text{if } u > 0, \\ 0, & \text{if } u \leq 0, \end{cases} \tag{11}$$

which stands for the Heaviside step function. You can use

$$\frac{\partial \max\{0, \boldsymbol{u}\}}{\partial \boldsymbol{u}} = H(\boldsymbol{u}) \tag{12}$$

in your derivation of $\dfrac{\partial l}{\partial \boldsymbol{u}}$.

You can also use $\cdot *$ to represent element-wise product between two vectors or matrices. For example,

$$\boldsymbol{v} \cdot * \boldsymbol{c} = \begin{bmatrix} v_1 \times c_1 \\ \vdots \\ v_I \times c_I \end{bmatrix} \in \mathbb{R}^I, \text{ where } \boldsymbol{v} \in \mathbb{R}^I \text{ and } \boldsymbol{c} \in \mathbb{R}^I. \tag{13}$$

Also note that the partial derivatives of the loss function w.r.t. the variables (e.g., a scalar, a vector, or a matrix) will have the same shape as the variables.

What to submit: No more than 5 lines of derivation for each of the 5 partial derivatives.

Q1.2 Suppose we initialize $\boldsymbol{W}^{(1)}$, $\boldsymbol{W}^{(2)}$, $\boldsymbol{b}^{(1)}$ with zero matrices/vectors (i.e., matrices and vectors with all elements set to 0), please first verify that $\dfrac{\partial l}{\partial \boldsymbol{W}^{(1)}}$, $\dfrac{\partial l}{\partial \boldsymbol{W}^{(2)}}$, $\dfrac{\partial l}{\partial \boldsymbol{b}^{(1)}}$ are all zero matrices/vectors, irrespective of $\boldsymbol{x}$, $\boldsymbol{y}$ and the initialization of $\boldsymbol{b}^{(2)}$.

Now if we perform stochastic gradient descent for learning the neural network using a training set $\{(\boldsymbol{x}_i \in \mathbb{R}^D, \boldsymbol{y}_i \in \mathbb{R}^K)\}_{i=1}^N$, please explain with a concise mathematical statement in one sentence why no learning will happen on $\boldsymbol{W}^{(1)}$, $\boldsymbol{W}^{(2)}$, $\boldsymbol{b}^{(1)}$ (i.e., they will not change no matter how many iterations are run). Note that this will still be the case even with weight decay and momentum if the initial velocity vectors/matrices are set to zero.

What to submit: No submission for the verification question. Your concise mathematical statement in one sentence for the explanation question.

Q1.3 As mentioned in the lecture (see lec8.pdf), nonlinearity is very important for neural networks. With nonlinearity (e.g., eq. (3)), the neural network shown in Fig. 1 can bee seen as a nonlinear basis function $\boldsymbol{\phi}$ (i.e., $\boldsymbol{\phi}(\boldsymbol{x}) = \boldsymbol{h}$) followed by a linear classifier f (i.e., $f(\boldsymbol{h}) = \hat{y}$).

Please show that, by removing the nonlinear operation in eq. (3) and setting eq. (4) to be $\boldsymbol{a} = \boldsymbol{W}^{(2)}\boldsymbol{u} + \boldsymbol{b}^{(2)}$, the resulting network is essentially a linear classifier. More specifically, you can now represent $\boldsymbol{a}$ as $\boldsymbol{U}\boldsymbol{x} + \boldsymbol{v}$, where $\boldsymbol{U} \in \mathbb{R}^{K \times D}$ and $\boldsymbol{v} \in \mathbb{R}^K$. Please write down the representation of $\boldsymbol{U}$ and $\boldsymbol{v}$ using $\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}, \boldsymbol{b}^{(1)}$, and $\boldsymbol{b}^{(2)}$

$$\begin{aligned} \boldsymbol{U} &= ? \\ \boldsymbol{v} &= ? \end{aligned}$$

What to submit: No more than 2 lines of derivation for each of the question mark.

2 Kernel methods

[Recommended maximum time spent: 1 hour]

In the lecture (see lec10.pdf) , we have seen the "kernelization" of regularized least squares problem. The "kernelization" process depends on an important observation: the optimal model parameter can be expressed as a linear combination of the transformed features. You are now to prove a more general case.

Consider a convex loss function in the form $\ell(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}), y)$, where $\boldsymbol{\phi}(\boldsymbol{x}) \in \mathbb{R}^M$ is a nonlinear feature mapping, and y is a label or a continuous response value.

Now solve the regularized loss minimization problem on a training set $\mathcal{D} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_N, y_N)\}$,

$$\min_{w} \sum_n \ell(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n), y_n) + \frac{\lambda}{2}||\boldsymbol{w}||_2^2 \tag{14}$$

Q2.1 Show that the optimal solution of $\boldsymbol{w}$ can be represented as a linear combination of the training samples. You can assume $\ell(s, y)$ is differentiable w.r.t. s, i.e. during derivation, you can use the derivative $\frac{\partial \ell(s,y)}{\partial s}$ and assume it is a known quantity.

What to submit: Your fewer than 10 line derivation and optimal solution of $\boldsymbol{w}$.

Q2.2 Assume the combination coefficient is α_n for $n = 1, \ldots, N$. Rewrite loss function Eqn. 14 in terms of α_n and kernel function value $K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$.

What to submit: Your objective function in terms of α and K.

Q2.3 After you obtain the general formulation for Q2.1 and Q2.2, please plug in three different loss functions we have seen so far, and examine what you get.

square loss:

$$\ell(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}), y) = \frac{1}{2}\|y - \boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x})\|_2^2 \qquad y \in \mathbb{R} \tag{15}$$

cross entropy loss:

$$\ell(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}), y) = -\{y \log[\sigma(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}))] + (1-y)\log[1 - \sigma(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}))]\} \qquad y \in \{0, 1\} \tag{16}$$

perceptron loss:

$$\ell(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}), y) = \max(-y\boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}), 0) \qquad y \in \{-1, 1\} \tag{17}$$

What to submit: Nothing.

Programming component

3 High-level descriptions

3.1 Dataset

We will use **mnist_subset** (images of handwritten digits from 0 to 9). This is the same subset of the full MNIST that we used for Homework 1. As before, the dataset is stored in a JSON-formated 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.

3.2 Tasks

You will be asked to implement 10-way classification using multinomial logistic regression (Sect. 4) and neural networks (Sect. 5). Specifically, you will

- finish the implementation of all python functions in our template code.
- run your code by calling the specified scripts to generate output files.
- add, commit, and push (1) all `*.py` files, and (2) all `*.json` files that you have created.

In the next two subsections, we will provide a **high-level** checklist of what you need to do. Furthermore, as in the previous homework, you are not responsible for loading/pre-processing data; we have done that for you. For specific instructions, please refer to text in Sect. 4 and Sect. 5, as well as corresponding python scripts.

3.2.1 Multi-class Classification

Coding In `logistic_prog.py`, finish implementing the following functions: `logistic_train_ovr`, `logistic_test_ovr`, `logistic_mul_train`, and `logistic_mul_test`. Refer to `logistic_prog.py` and Sect. 4 for more information.
Running your code Run the script `q43.sh` after you finish your implementation. This will output `logistic_res.json`.
What to submit Submit both `logistic_prog.py` and `logistic_res.json`.

3.2.2 Neural networks

Preparation Read `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 Sect. 5 for more information.

Second, in `dnn_cnn_2.py`, finish implementing the `main` function. There are three TODO items. Refer to `dnn_cnn_2.py` and Sect. 5 for more information.

Running your code Run the scripts `q53.sh`, `q54.sh`, `q55.sh`, `q56.sh`, `q57.sh`, `q58.sh`, `q510.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 seven `.json` files that are generated from the scripts.

3.3 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, initializations, etc., **will not** be graded. It is your responsibility to **make sure that your code runs with the provided commands and scripts on the VM**. Finally, make sure that you **git add, commit, and push all the required files**, including your code and generated output files.

3.4 Advice

We are extensively using softmax and sigmoid function in this homework. To avoid numerical issues such as overflow and underflow caused by *numpy.exp*() and *numpy.log*(), please use the following implementations:

- Let $\boldsymbol{x}$ be a input vector to the softmax function. Use $\tilde{\boldsymbol{x}} = \boldsymbol{x} - \max(\boldsymbol{x})$ instead of using $\boldsymbol{x}$ directly for the softmax function f, i.e. $f(\tilde{\boldsymbol{x}}_i) = \frac{\exp(\tilde{\boldsymbol{x}}_i)}{\sum_{j=1}^{D}\exp(\tilde{\boldsymbol{x}}_j)}$
- If you are using *numpy.log*(), make sure the input to the log function is positive. Also, there may be chances that one of the outputs of softmax, e.g. $f(\tilde{\boldsymbol{x}}_i)$, is extremely small but you need the value $\ln(f(\tilde{\boldsymbol{x}}_i))$, you can convert the computation into $\tilde{\boldsymbol{x}}_i - \ln(\sum_{j=1}^{D}\exp(\tilde{\boldsymbol{x}}_j))$.

We have implemented and run the code ourselves without any problems, so if you follow the instructions and settings provided in the python files, you should not encounter overflow or underflow.

4 Multi-class Classification

You will modify 4 python functions in `logistic_prog.py`. First, you will implement two functions that train and test a one-versus-rest multi-class classification model. Second, you will implement two functions that train and test a multinomial logistic regression model. Finally, you will run the command that train and test the two models using your implemented functions, and our code will automatically store your results to `logistic_res.json`.

Coding: One-versus-rest

Q4.1 Implement the code to solve the multi-class classification task with the one-versus-rest strategy. That is, train 10 binary logistic regression models following the setting provided in class: for each class $C_k, k = 1, \cdots, 10$, we create a binary classification problem as follows:

- Re-label training samples with label C_k as positive (namely 1)
- Re-label other samples as negative (namely 0)

We wrote functions to load, relabel, and sample the data for you, so you are not responsible for doing it.

Training Finish the implementation of the function `logistic_train_ovr(Xtrain, ytrain, w, b, step_size, max_iterations)`. As in the previous homework, we have pre-defined the hyper-parameters and initializations in the template code. Moreover, you will use the **AVERAGE** of gradients from all training samples to update the parameters.

Testing Finish the implementation of the function `logistic_test_ovr(Xtest, w_l, b_l)`. This function should return the predicted probability, i.e., the value output by logistic function without thresholding, instead of the 0/1 label. Formally, for each test data point $\boldsymbol{x}_i$, we get its final prediction by $\hat{y}_i = \text{argmax}_{k\in\{1,\cdots,10\}} f_k(\boldsymbol{x}_i)$, where $\hat{y}_i$ is the predicted label and $f_k(\boldsymbol{x}_i)$ is the predicted probability by the k^{th} logistic regression model f_k. Then, you compute the classification accuracy as follows:

$$Acc = \frac{\sum_{i=1}^{N_{test}} \mathbf{1}\{\hat{y}_i == y_i\}}{N_{test}}, \tag{18}$$

where y_i is the ground-truth label of $\boldsymbol{x}_i$ and N_{test} is the total number of test data instances.

What to do and submit: Your `logistic_prog.py` with completed `logistic_train_ovr` and `logistic_test_ovr`.

Coding: Multinomial logistic regression

Q4.2 Implement the multinomial logistic regression, training a 10-way classifier (with the softmax function) on **mnist_subset** dataset.

Training Finish the implementation of the function `logistic_mul_train(Xtrain, ytrain, w, b, step_size, max_iterations)`. Again, we have pre-defined the hyper-parameters and initializations in the template code. Moreover, you will use the **AVERAGE** of gradients from all training samples to update the parameters.

Testing Finish the implementation of the function `logistic_mul_test(Xtest, w_l, b_l)` For each test data point x_i, compute $\hat{y} = \text{argmax}_{k \in \{1,\cdots,10\}} p(y = k|\boldsymbol{x})$, where $p(y = k|\boldsymbol{x})$ is the predicted probability by the multinomial logistic regression. Then, compute the accuracy following Eqn. 18.

What to do and submit: Your `logistic_prog.py` with completed `logistic_mul_train` and `logistic_mul_test`.

Training and getting generated output files from both one-versus-rest and multinomial logistic regression models

Q4.3 *What to do and submit:* run script `q43.sh`. It will generate `logistic_res.json`. Add, commit, and push both `logistic_prog.py` and `logistic_res.json` before the due date. *What it does:* `q43.sh` will run `python3 logistic_prog.py`. This will train your models (for both Q4.1 and Q4.2 above) and test the trained models (for both Q4.1 and Q4.2 above). The output file stores accuracies of both models.

5 Neural networks: multi-layer perceptrons (MLPs) and convolutional neural networks (CNNs)

In recent years, neural networks have been one of the most powerful machine learning models. Many toolboxes/platforms (e.g., TensorFlow, PyTorch, Torch, Theano, MXNet, Caffe, CNTK) 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 modules*. For example, in Fig. 2, the edges correspond to module names of the same neural network shown in Fig. 1 and Sect. 1.

Now we will provide more information on modules for this homework. 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 w.r.t. the input and parameters, given the partial derivatives of the loss function w.r.t. 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.

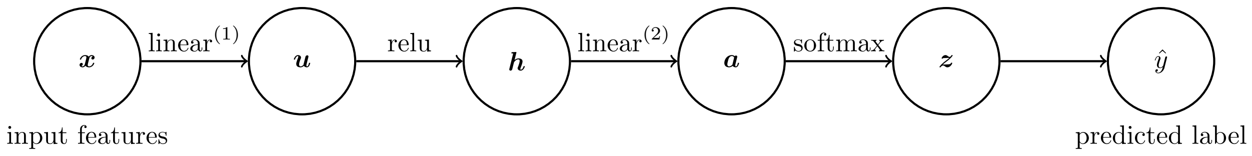
$$\begin{tikzcd} \boldsymbol{x} \arrow[r, "\text{linear}^{(1)}"] & \boldsymbol{u} \arrow[r, "\text{relu}"] & \boldsymbol{h} \arrow[r, "\text{linear}^{(2)}"] & \boldsymbol{a} \arrow[r, "\text{softmax}"] & \boldsymbol{z} \arrow[r] & \hat{y} \end{tikzcd}$$

Figure 2: A diagram of a 1-hidden layer multi-layer perceptron (MLP), with modules indicated on the edges. The circles correspond to variables. The *rectangles* shown in Fig. 1 are removed for clearness. The term *relu* stands for rectified linear units.

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

$$\text{forward pass:} \qquad \boldsymbol{u} = \text{linear}^{(1)}.\text{forward}(\boldsymbol{x}) = \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}, \tag{19}$$
$$\text{where } \boldsymbol{W}^{(1)} \text{ and } \boldsymbol{b}^{(1)} \text{ are its parameters.}$$

$$\text{backward pass:} \qquad [\frac{\partial l}{\partial \boldsymbol{x}}, \frac{\partial l}{\partial \boldsymbol{W}^{(1)}}, \frac{\partial l}{\partial \boldsymbol{b}^{(1)}}] = \text{linear}^{(1)}.\text{backward}(\boldsymbol{x}, \frac{\partial l}{\partial \boldsymbol{u}}). \tag{20}$$

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 intermediated variables (i.e., $\boldsymbol{u}$, $\boldsymbol{h}$, etc.) will all be computed along the forward pass. Similarly, getting the partial derivatives of the loss function w.r.t. the parameters is equivalent to running the backward pass of each module in a reverse order, given $\frac{\partial l}{\partial \boldsymbol{z}}$.

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 same 10-class classification problem introduced in Sect. 4. 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 deeper understanding, check out, e.g., the seminal work of Yann LeCun et al. "Gradient-based learning applied to document recognition," written in 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 Fig. 2 is

$$
\begin{aligned}
l_{mb} &= \frac{1}{N}\sum_{i=1}^{N} l(\text{softmax.forward}(\text{linear}^{(2)}.\text{forward}(\text{relu.forward}(\text{linear}^{(1)}.\text{forward}(\boldsymbol{x}_i)))), \boldsymbol{y}_i) && (21)\\
&= \frac{1}{N}\sum_{i=1}^{N} l(\text{softmax.forward}(\text{linear}^{(2)}.\text{forward}(\text{relu.forward}(\boldsymbol{u}_i))), \boldsymbol{y}_i) && (22)\\
&= \cdots && (23)\\
&= \frac{1}{N}\sum_{i=1}^{N} l(\text{softmax.forward}(\boldsymbol{a}_i), \boldsymbol{y}_i) && (24)\\
&= \frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik}\log z_{ik}. && (25)
\end{aligned}
$$

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 Fig. 2, if we are now to pass the partial derivatives of the loss w.r.t. $\{\boldsymbol{a}_i\}_{i=1}^{N}$ to linear$^{(2)}$.backward, then

$$
\frac{\partial l_{mb}}{\partial \{\boldsymbol{a}_i\}_{i=1}^{N}} = \begin{bmatrix} (\frac{\partial l_{mb}}{\partial \boldsymbol{a}_1})^T \\ (\frac{\partial l_{mb}}{\partial \boldsymbol{a}_2})^T \\ \vdots \\ (\frac{\partial l_{mb}}{\partial \boldsymbol{a}_{N-1}})^T \\ (\frac{\partial l_{mb}}{\partial \boldsymbol{a}_N})^T \end{bmatrix}. \qquad (26)
$$

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

Preparation

Q5.1 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. 3 and Fig. 4, respectively.

What to submit: Nothing.

Coding: Modules

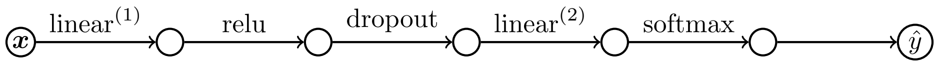

Figure 3: The diagram of the MLP implemented in `dnn_mlp.py`. The circles mean variables and edges mean modules.

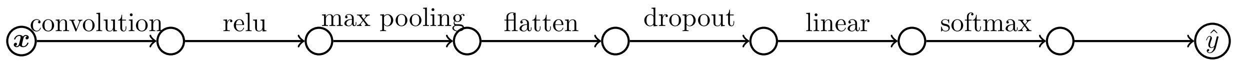

Figure 4: The diagram of the CNN implemented in `dnn_cnn.py`. The circles correspond to variables and edges correspond to modules. Note that the input to CNN may not be a vector (e.g., in `dnn_cnn.py` it is an image, which can be represented as a 3-dimensional tensor). The flatten layer is to reshape its input into vector.

Q5.2 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. First, finish the implementation of `forward` and `backward` functions in `class linear_layer`. Please follow Eqn. (2) for the forward pass. Second, finish the implementation of `forward` and `backward` functions in `class relu`. Please follow Eqn. (3) for the forward pass and Eqn. (11) for deriving the partial derivatives (note that relu itself has no parameters). Third, finish the the implementation of `backward` function in `class dropout`. We define the forward pass and the backward pass as follows.

$$\text{forward pass:} \qquad \boldsymbol{s} = \text{dropout.forward}(\boldsymbol{q} \in \mathbb{R}^J) = \frac{1}{1-r} \times \begin{bmatrix} \mathbf{1}[p_1 >= r] \times q_1 \\ \vdots \\ \mathbf{1}[p_J >= r] \times q_J \end{bmatrix}, \tag{27}$$

where p_j is sampled uniformly from $[0, 1), \forall j \in \{1, \cdots, J\}$,
and $r \in [0, 1)$ is a pre-defined scalar named dropout rate.

$$\text{backward pass:} \qquad \frac{\partial l}{\partial \boldsymbol{q}} = \text{dropout.backward}(\boldsymbol{q}, \frac{\partial l}{\partial \boldsymbol{s}}) = \frac{1}{1-r} \times \begin{bmatrix} \mathbf{1}[p_1 >= r] \times \frac{\partial l}{\partial s_1} \\ \vdots \\ \mathbf{1}[p_J >= r] \times \frac{\partial l}{\partial s_J} \end{bmatrix}. \tag{28}$$

Note that $p_j, j \in \{1, \cdots, J\}$ and r are not be learned so we do not need to compute the derivatives w.r.t. to them. Moreover, $p_j, j \in \{1, \cdots, J\}$ are re-sampled every forward pass, 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 do read carefully.

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

Testing dnn_misc.py

Q5.3 *What to do and submit:* run script `q53.sh`. It will output `MLP_lr0.01_m0.0_w0.0_d0.0.json`. Add, commit, and push this file before the due date.
What it does: `q53.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.

Q5.4 *What to do and submit:* run script `q54.sh`. It will output `MLP_lr0.01_m0.0_w0.0_d0.5.json`. Add, commit, and push this file before the due date.
What it does: `q54.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.

Q5.5 *What to do and submit:* run script `q55.sh`. It will output `MLP_lr0.01_m0.0_w0.0_d0.95.json`. Add, commit, and push this file before the due date.
What it does: `q55.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 Q5.4 will give better validation accuracy (at epoch 30) compared to Q5.3. Specifically, dropout is widely-used to prevent over-fitting. However, if we use a too large dropout rate (like the one in Q5.5), the validation accuracy (together with the training accuracy) will be relatively lower, essentially under-fitting the training data.

Q5.6 *What to do and submit:* run script `q56.sh`. It will output `LR_lr0.01_m0.0_w0.0_d0.0.json`. Add, commit, and push this file before the due date.
What it does: `q56.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 Q5.3, except that we remove the relu (nonlinear) layer. You will see that the validation accuracies drop significantly (the gap is around 0.03). Essentially, without the nonlinear layer, the model is learning multinomial logistic regression similar to Q4.2.

Q5.7 *What to do and submit:* run script `q57.sh`. It will output `CNN_lr0.01_m0.0_w0.0_d0.5.json`. Add, commit, and push this file before the due date.
What it does: `q57.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.

$$\begin{tikzcd}
x \arrow[r, "\text{conv}"] & \bigcirc \arrow[r, "\text{relu}"] & \bigcirc \arrow[r, "\text{max-p}"] & \bigcirc \arrow[r, "\text{conv}"] & \bigcirc \arrow[r, "\text{relu}"] & \bigcirc \arrow[r, "\text{max-p}"] & \bigcirc \arrow[r, "\text{flatten}"] & \bigcirc \arrow[r, "\text{dropout}"] & \bigcirc \arrow[r, "\text{linear}"] & \bigcirc \arrow[r, "\text{softmax}"] & \bigcirc \arrow[r] & \hat{y}
\end{tikzcd}$$

Figure 5: 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 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 flatten layer is to reshape its input into vector.

Q5.8 *What to do and submit:* run script `q58.sh`. It will output `CNN_lr0.01_m0.9_w0.0_d0.5.json`. Add, commit, and push this file before the due date.
What it does: `q58.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 Q5.8 will lead to faster convergence than Q5.7 (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.

Coding: Building a deeper architecture

Q5.9 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 Fig. 5 using the modules you implemented in Q5.2. 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 a few parts marked as `TODO`. You need to fill in your code so as to construct the CNN in Fig. 5.

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`.

Testing dnn_cnn_2.py

Q5.10 *What to do and submit:* run script `q510.sh`. It will output `CNN2_lr0.001_m0.9_w0.0_d0.5.json`. Add, commit, and push this file before the due date.
What it does: `q510.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 Q5.8.