CSCI567 (Spring 2018) Programming Assignment 2

General instructions

Due date and time Feb. 25th, 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.py`, which you will be amending by adding code for questions in Sect. 2.
- python script `dnn_misc.py`, which you will be amending by adding code for questions in Sect. 3.
- python script `dnn_cnn_2.py`, which you will be amending by adding code for questions in Sect. 3.
- Various other python scripts: `dnn_mlp.py`, `dnn_mlp_nononlinear.py`, `dnn_cnn.py`, `hw2_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`, `q310.sh`, `logistic_binary.sh` and `logistic_multiclass.sh`; you will use these to generate output files.

Download the data Please download **mnist_subset.json** from Piazza/Resource/Homework. Please **DO NOT** push it into your repository when you submit your results. Otherwise, you will get 20% deduction of your score of this assignment.

Submission Instructions The following will constitute your submission:

- The three python scripts above, amended with the code you added for Sect. 2 and Sect. 3. Be sure to commit your changes!
- Seven .json files and two output 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_binary.out
logistic_multiclass.out
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

1 High-level descriptions

1.1 Dataset

We will use **mnist_subset** (images of handwritten digits from 0 to 9). 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.

Besides, for logistic regression in Sect. 2, you will be using synthetic datasets with two, three and five classes. Synthetic data that you will use is illustrated in Figure 1.

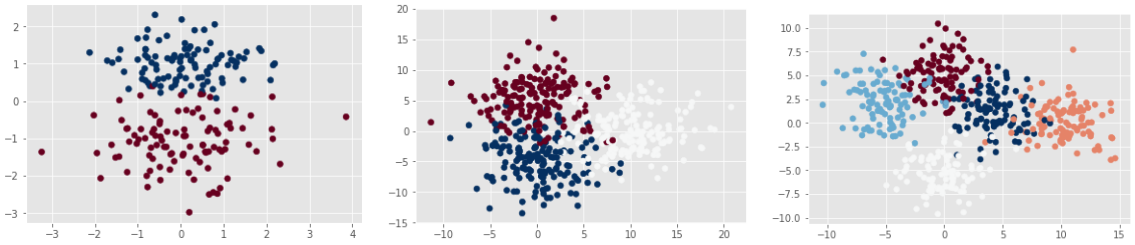

Figure 1: Synthetic datasets

1.2 Tasks

You will be asked to implement binary and multiclass classification (Sect. 2) and neural networks (Sect. 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.
- add, commit, and push (1) all `*.py` files, and (2) all `*.json` and `*.out` files that **you have amended or created**.

In the next two subsections, 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 text in Sect. 2 and Sect. 3, as well as corresponding python scripts.

1.2.1 Logistic regression

Coding In `logistic.py`, finish implementing the following functions: `binary_train`, `binary_predict`, `multinomial_train`, `multinomial_predict`, `ovr_train` and `ovr_predict`. Refer to `logistic.py` and Sect. 2 for more information.

Running your code Run the scripts `logistic_binary.sh` and `logistic_multiclass.sh` after you finish your implementation. This will output:

- `logistic_binary.out`
- `logistic_multiclass.out`

What to submit Submit `logistic.py`, `logistic_binary.out`, `logistic_multiclass.out`.

1.2.2 Neural networks

Preparation Read Sect. 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 Sect. 3 for more information.

Second, in `dnn_cnn_2.py`, finish implementing the `main` function. There are five TODO items. Refer to `dnn_cnn_2.py` and Sect. 3 for more information.

Running your code Run the scripts `q33.sh`, `q34.sh`, `q35.sh`, `q36.sh`, `q37.sh`, `q38.sh`, `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.

1.3 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.

1.4 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 problems, so if you follow the instructions and settings provided in the python files, you should not encounter overflow or underflow.

2 Logistic Regression

For this assignment you are asked to implement Logistic Regression for binary and multiclass classification.

Q2.1 Recall from lecture 6, that binary classification with Logistic regression is performed by the following rule:

$$p(y == 1|x) = \sigma(w^T x + b) \tag{1}$$

$$\sigma(a) = \frac{1}{1 + e^{-a}} \tag{2}$$

Given a training set $\mathcal{D} = \left\{(x_n, y_n)_{n=1}^N\right\}$, where $y_i \in \{0, 1\}\, \forall i = 1...N$, you will need to write a function that will find the optimal parameters w.

You will need to implement functions `binary_train` and `binary_predict` in `logistic.py`. Recall also, that logistic regression does not have a closed form solution, so for the training you will need to implement gradient descent.

After you finished implementation, please run `logistic_binary.sh` script, which will produce `logistic_binary.out`.
What to submit: `logistic.py` and `logistic_binary.out`.

Q2.2 In the lectures you learned several methods to perform multiclass classification. One of them was one-versus-rest approach.

For one-versus-rest classification in a problem with K classes, we need to train K classifiers. Each classifier is trained on a binary problem, where belonging to the class corresponding to the classifier is a positive outcome, and belonging to any other class is a negative outcome. After that, the multiclass prediction is made based on the combination of all predictions from K binary classifiers. [*Hint*: use numpy argmax function for combination.]

You will need to complete functions `OVR_train` and `OVR_predict` to perform one-versus-rest classification. If needed, you can make calls to the binary logistic regression train and predict functions that you implemented before.

After you finished implementation, please run `logistic_multiclass.sh` script, which will produce `logistic_multiclass.out`.
What to submit: `logistic.py` and `logistic_multiclass.out`.

Q2.3 Yet another multiclass classification method you learned was multinomial logistic regression. Here, for the conditional probability we replaced the *sigmoid* function from binary classification with a *softmax* function:

$$p(y == k|x) = \frac{e^{\mathbf{w}_k^T \mathbf{x}}}{\sum_{k'} e^{\mathbf{w}_k'^T \mathbf{x}}} \tag{3}$$

In this setup we again need to train K binary classifiers, and each point is assigned to the class, that maximizes the conditional probability from Eq 3.

Complete the functions `multinomial_train` and `multinomial_predict` to perform multinomial classification.

After you finished implementation, please run `logistic_multiclass.sh` script, which will produce `logistic_multiclass.out`.

What to submit: `logistic.py` and `logistic_multiclass.out`.

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

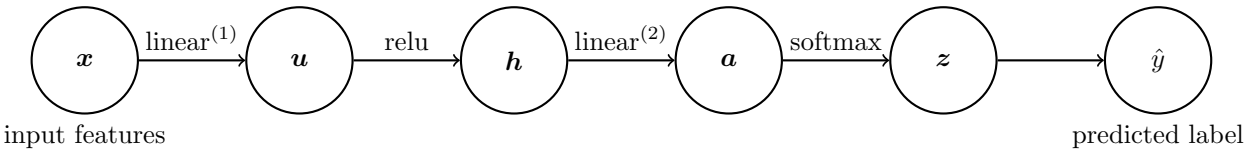

Figure 2: A diagram of a multi-layer perceptron (MLP). *The edges mean mathematical operations (modules), and the circles mean variables.* The term *relu* stands for rectified linear units.

Background

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 (or mathematical operation) modules*.

For example, in Fig. 2 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 $(\boldsymbol{x} \in \mathbb{R}^D, y \in \{1, 2, \cdots, K\})$ be a labeled instance, such an MLP performs the following computations

$$\textbf{input features}: \quad \boldsymbol{x} \in \mathbb{R}^D \tag{4}$$

$$\textbf{linear}^{(1)}: \quad \boldsymbol{u} = \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)} \quad , \boldsymbol{W}^{(1)} \in \mathbb{R}^{M \times D} \text{ and } \boldsymbol{b}^{(1)} \in \mathbb{R}^M \tag{5}$$

$$\textbf{relu}: \quad \boldsymbol{h} = \max\{0, \boldsymbol{u}\} = \begin{bmatrix} \max\{0, u_1\} \\ \vdots \\ \max\{0, u_M\} \end{bmatrix} \tag{6}$$

$$\textbf{linear}^{(2)}: \quad \boldsymbol{a} = \boldsymbol{W}^{(2)}\boldsymbol{h} + \boldsymbol{b}^{(2)} \quad , \boldsymbol{W}^{(2)} \in \mathbb{R}^{K \times M} \text{ and } \boldsymbol{b}^{(2)} \in \mathbb{R}^K \tag{7}$$

$$\textbf{softmax}: \quad \boldsymbol{z} = \begin{bmatrix} \frac{e^{a_1}}{\sum_k e^{a_k}} \\ \vdots \\ \frac{e^{a_K}}{\sum_k e^{a_k}} \end{bmatrix} \tag{8}$$

$$\textbf{predicted label}: \quad \hat{y} = \arg\max_k z_k. \tag{9}$$

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

$$l = -\sum_k \mathbf{1}[y == k] \log z_k, \tag{10}$$

$$\text{where } \mathbf{1}[\text{True}] = 1; \text{ otherwise, } 0. \tag{11}$$

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

$$\boldsymbol{y} \in \mathbb{R}^K \text{ and } y_k = \begin{cases} 1, & \text{if } y = k, \\ 0, & \text{otherwise.} \end{cases} \tag{12}$$

so that

$$l = -\sum_k y_k \log z_k = -\boldsymbol{y}^T \begin{bmatrix} \log z_1 \\ \vdots \\ \log z_K \end{bmatrix} = -\boldsymbol{y}^T \log \boldsymbol{z}. \tag{13}$$

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.

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 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.

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

$$\text{forward pass:} \qquad \boldsymbol{u} = \text{linear}^{(1)}.\text{forward}(\boldsymbol{x}) = \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}, \tag{14}$$
$$\text{where } \boldsymbol{W}^{(1)} \text{ and } \boldsymbol{b}^{(1)} \text{ are its parameters.}$$

$$\text{backward pass:} \qquad [\frac{\partial l}{\partial \boldsymbol{x}}, \frac{\partial l}{\partial \boldsymbol{W}^{(1)}}, \frac{\partial l}{\partial \boldsymbol{b}^{(1)}}] = \text{linear}^{(1)}.\text{backward}(\boldsymbol{x}, \frac{\partial l}{\partial \boldsymbol{u}}). \tag{15}$$

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 $\dfrac{\partial l}{\partial \boldsymbol{z}}$.

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. 2. 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

$$
\begin{aligned}
l_{mb} &= \frac{1}{N}\sum_{i=1}^{N} l(\text{softmax.forward}(\text{linear}^{(2)}.\text{forward}(\text{relu.forward}(\text{linear}^{(1)}.\text{forward}(\boldsymbol{x}_i)))), \boldsymbol{y}_i) && (16)\\
&= \frac{1}{N}\sum_{i=1}^{N} l(\text{softmax.forward}(\text{linear}^{(2)}.\text{forward}(\text{relu.forward}(\boldsymbol{u}_i))), \boldsymbol{y}_i) && (17)\\
&= \cdots && (18)\\
&= \frac{1}{N}\sum_{i=1}^{N} l(\text{softmax.forward}(\boldsymbol{a}_i), \boldsymbol{y}_i) && (19)\\
&= \frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik} \log z_{ik}. && (20)
\end{aligned}
$$

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, given the partial derivatives of the loss w.r.t. $\{\boldsymbol{a}_i\}_{i=1}^N$

$$
\frac{\partial l_{mb}}{\partial \{\boldsymbol{a}_i\}_{i=1}^N} = \begin{bmatrix} (\frac{\partial l_{mb}}{\partial \boldsymbol{a}_1})^T \\ (\frac{\partial l_{mb}}{\partial \boldsymbol{a}_2})^T \\ \vdots \\ (\frac{\partial l_{mb}}{\partial \boldsymbol{a}_{N-1}})^T \\ (\frac{\partial l_{mb}}{\partial \boldsymbol{a}_N})^T \end{bmatrix}, \tag{21}
$$

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

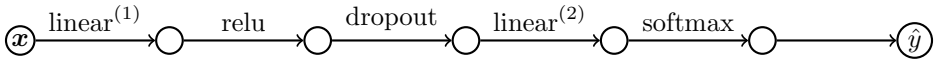

Figure 3: The diagram of the MLP implemented in `dnn_mlp.py`. The circles mean variables and edges mean modules.

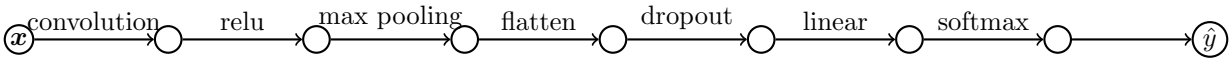

Figure 4: The diagram of the CNN implemented in `dnn_cnn.py`. The circles correspond to variables and edges correspond to modules. Note that the input to CNN may not be a vector (e.g., in `dnn_cnn.py` it is an image, which can be represented as a 3-dimensional tensor). The flatten layer is to reshape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

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. 3 and Fig. 4, respectively.

What to submit: Nothing.

Coding: Modules

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. First, finish the implementation of `forward` and `backward` functions in `class linear_layer`. Please follow Eqn. (5) for the forward pass and derive the partial derivatives accordingly. Second, finish the implementation of `forward` and `backward` functions in `class relu`. Please follow Eqn. (6) for the forward pass and derive the partial derivatives accordingly. Third, finish the the implementation of `backward` function in `class dropout`. We define the forward and the backward passes as follows.

$$\text{forward pass:} \qquad \boldsymbol{s} = \text{dropout.forward}(\boldsymbol{q} \in \mathbb{R}^J) = \frac{1}{1-r} \times \begin{bmatrix} \mathbf{1}[p_1 >= r] \times q_1 \\ \vdots \\ \mathbf{1}[p_J >= r] \times q_J \end{bmatrix}, \tag{22}$$

$$\text{where } p_j \text{ is sampled uniformly from } [0,1), \forall j \in \{1, \cdots, J\},$$
$$\text{and } r \in [0,1) \text{ is a pre-defined scalar named dropout rate.} \tag{23}$$

$$\text{backward pass:} \qquad \frac{\partial l}{\partial \boldsymbol{q}} = \text{dropout.backward}(\boldsymbol{q}, \frac{\partial l}{\partial \boldsymbol{s}}) = \frac{1}{1-r} \times \begin{bmatrix} \mathbf{1}[p_1 >= r] \times \dfrac{\partial l}{\partial s_1} \\ \vdots \\ \mathbf{1}[p_J >= r] \times \dfrac{\partial l}{\partial s_J} \end{bmatrix}. \tag{24}$$

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 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 `hw2_dnn_check.py` to check your implementation.**

Testing dnn_misc.py with multi-layer perceptron (MLP)

Q3.3 *What to do and submit:* run script `q33.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: `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`. Add, commit, and push this file before the due date.
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`. Add, commit, and push this file before the due date.
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 over-fitting. 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 under-fitting 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`. Add, commit, and push this file before the due date.
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). Essentially, without the nonlinear layer, the model is learning multinomial logistic regression similar to Q4.2.

Testing dnn_misc.py with convolutional neural networks (CNN)

Q3.7 *What to do and submit:* run script `q37.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: `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`. Add, commit, and push this file before the due date.
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.

Coding: Building a deeper architecture

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 Fig. 5 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 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

Q3.10 *What to do and submit:* run script `q310.sh`. It will output `CNN2_lr0.001_m0.9_w0.0_d0.5.json`. Add, commit, and push this file before the due date.

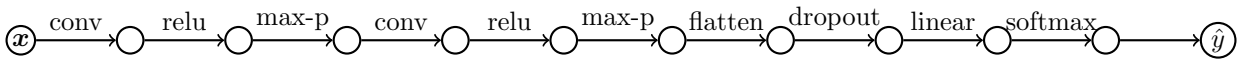

Figure 5: The diagram of the CNN you are going to implement in `dnn_cnn_2.py`. The term *conv* stands for convolution; *max-p* stands for max pooling. The circles correspond to variables and edges correspond to modules. Note that the input to CNN may not be a vector (e.g., in `dnn_cnn_2.py` it is an image, which can be represented as a 3-dimensional tensor). The flatten layer is to reshape its input into vector.

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.