CSCI 567, Spring 2020
Yan Liu

Homework #2 Programming Assignments

General Instructions

Starting point Create a folder and place all the scripts in the same folder.

- Python scripts which you need to amend:
  - `logistic.py`
  - `dnn_misc.py`
  - `pegasos.py`
- Python scripts which you are **not allowed** to modify:
  - `dnn_mlp.py`, `dnn_mlp_nononlinear.py`, `hw2_dnn_check.py`, `dnn_im2col.py` and `data_loader.py`
- Helper scripts that you will use to generage output files: `q33.sh`, `q34.sh`, `q35.sh`, `q36.sh`, `logistic_binary.sh` and `logistic_multiclass.sh`

Environment This assignment has to be done in Python 3.5.2. or higher. Make sure you have the correct version installed.

Python Packages You are allowed to use the following Python packages:

- all built-in packages in Python 3.5.2, such as `sys`.
- numpy (1.13.1)
- scipy (0.19.1)
- matplotlib (2.0.2)

You will use Numpy mostly; in contrast, Scipy is usually not needed unless you have special needs.

You will also need the following package for testing your code. Do not use any of its functions for your implementation:

- sklearn (0.19.0)

Download the data Please use the **mnist_subset.json** provided.

Problem 1 High-level descriptions

1.1 Dataset (Same as in Homework 1.) 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.

1.2 Tasks You will be asked to implement binary and multiclass classification (Sect. 2), neural networks (Sect. 3) and SVM (Sect. 4). 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.

In the next three 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 , Sect. 3 and Sect. 4, 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`

1.2.2 Neural networks

Preparation Read Sect. 3 as well as `dnn_mlp.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.
Running your code Run the scripts `q33.sh`, `q34.sh`, `q35.sh`, `q36.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`

1.2.3 SVM

You will be asked to implement the linear support vector machine (SVM) for binary classification (Sect. 4). Specifically, you will

- finish implementing the following three functions—`objective_function`, `pegasos_train`, and `pegasos_test`—in `pegasos.py`. Refer to `pegasos.py` and Sect. 4 for more information.
- run the script `pegasos.sh` after you finish your implementation. This will output `pegasos.json`.

1.3 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. That is, if you want to compute $f(\boldsymbol{x})_i$, compute $f(\tilde{\boldsymbol{x}})_i = \frac{\exp(\tilde{x}_i)}{\sum_{j=1}^{D} \exp(\tilde{x}_j)}$ instead, which is clearly mathematically equivalent but numerically more stable.
- 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 $\ln(f(\tilde{\boldsymbol{x}})_i)$. In this case you should convert the computation equivalently into $\tilde{x}_i - \ln(\sum_{j=1}^{D} \exp(\tilde{x}_j))$.

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.

Problem 2 Logistic Regression

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

Q2.1 We discussed logistic regression for binary classification. In this problem, you are given a training set $\mathcal{D} = \{(x_n, y_n)_{n=1}^N\}$, where $y_i \in \{0, 1\}\ \forall i = 1...N$. **Important**: note that here the binary labels are not -1 or $+1$, so be very careful about applying formulas.

Your task is to learn the linear model specified by $w^T x + b$ that minimizes the logistic loss. Note that we do not explicitly append the feature 1 to the data, so you need to explicitly learn the bias/intercept term b too. Specifically you need to implement function `binary_train` in `logistic.py` which uses gradient descent (*not* stochastic gradient descent) to find the optimal parameters (recall logistic regression does not admit a closed-form solution).

In addition you need to implement function `binary_predict` in `logistic.py`. We discuss two ways of making predictions in logistic regression in class: deterministic prediction or randomized prediction. Here you need to use the *deterministic prediction*.

After finishing implementation, please run `logistic_binary.sh` which generates `logistic_binary.out`.

Hint: You can match the results you are getting. It will be close to the one below.

```
Performing binary classification on synthetic data
train acc: 0.994286, test acc: 1.000000
Performing binary classification on binarized MNIST
train acc: 0.871000, test acc: 0.834000
```

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

For one-versus-rest classification in a problem with K classes, we need to train K classifiers using a black-box. Classifier k is trained on a binary problem, where the two labels corresponds to belonging or not belonging to class k. After that, the multiclass prediction is made based on the combination of all predictions from K binary classifiers.

In this problem you will implement one-versus-rest using binary logistic regression (that you have implemented in Q2.1) as the black-box. **Important**: the way to predict discussed in the lecture is to randomized over the classifiers that say "yes"; however, here since binary logistic regression naturally predicts a probability for each class (recall the sigmoid model), we will simply predict the class with the highest probability (using numpy argmax).

To sum up, you need to complete functions `OVR_train` and `OVR_predict` to perform one-versus-rest classification. After you finished implementation, please run `logistic_multiclass.sh` script, which will produce `logistic_multiclass.out`.

Hint: You can match the results you are getting. It will be close to the one below.

```
Synthetic data: 3 class classification
One-versus-rest:
train acc: 0.908571, test acc: 0.840000
Synthetic data: 5 class classification
One-versus-rest:
train acc: 0.765714, test acc: 0.800000
MNIST: 10 class classification
One-versus-rest:
train acc: 0.926800, test acc: 0.897000
```

Q2.3 Yet another multiclass classification method you learned was multinomial logistic regression. Complete the functions `multinomial_train` and `multinomial_predict` to perform multinomial logistic regression, following the same notes as in Q2.1, that is, 1) explicitly learn the biased term; 2) perform gradient descent instead of stochastic gradient descent; 3) make deterministic predictions.

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

Hint: You can match the results you are getting. It will be close to the one below.

```
Multinomial:
train acc: 0.902857, test acc: 0.840000
Multinomial:
train acc: 0.894286, test acc: 0.906667
Multinomial:
train acc: 0.945400, test acc: 0.896000
```

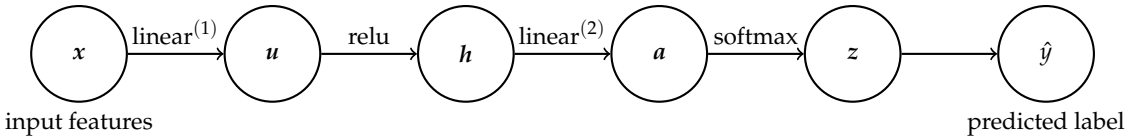

Figure 1: A diagram of a multi-layer perceptron (MLP). *The edges mean mathematical operations (modules), and the circles mean variables.* The term *relu* stands for rectified linear units.

Problem 3 Neural networks: multi-layer perceptrons (MLPs)

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. 1 we provide a diagram of a multi-layer perceptron (MLP, just another term for fully connected feedforward networks we discussed in the lecture) 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

$$\textbf{input features}: \quad \boldsymbol{x} \in \mathbb{R}^D \tag{1}$$

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

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

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

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

$$\textbf{predicted label}: \quad \hat{y} = \arg\max_k z_k. \tag{6}$$

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. Specifically we denote the cross-entropy loss with respect to the training example $(\boldsymbol{x}, y)$ by l:

$$l = -\log(z_y) = \log\left(1 + \sum_{k \neq y} e^{a_k - a_y}\right)$$

Note that one should look at l as a function of the parameters of the network, that is, $\boldsymbol{W}^{(1)}, \boldsymbol{b}^{(1)}, \boldsymbol{W}^{(2)}$ and $\boldsymbol{b}^{(2)}$. For ease of notation, let us define the one-hot (i.e., 1-of-K) encoding of a class y as

$$\boldsymbol{y} \in \mathbb{R}^K \text{ and } y_k = \begin{cases} 1, \text{ if } y = k, \\ 0, \text{ otherwise.} \end{cases} \tag{7}$$

so that

$$l = -\sum_k y_k \log z_k = -\boldsymbol{y}^T \begin{bmatrix} \log z_1 \\ \vdots \\ \log z_K \end{bmatrix} = -\boldsymbol{y}^T \log \boldsymbol{z}. \tag{8}$$

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.

$$\text{forward pass:} \quad \boldsymbol{u} = \text{linear}^{(1)}.\text{forward}(\boldsymbol{x}) = \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}, \quad \text{where } \boldsymbol{W}^{(1)} \text{ and } \boldsymbol{b}^{(1)} \text{ are its parameters.} \tag{9}$$

$$\text{backward pass:} \quad [\frac{\partial l}{\partial \boldsymbol{x}}, \frac{\partial l}{\partial \boldsymbol{W}^{(1)}}, \frac{\partial l}{\partial \boldsymbol{b}^{(1)}}] = \text{linear}^{(1)}.\text{backward}(\boldsymbol{x}, \frac{\partial l}{\partial \boldsymbol{u}}). \tag{10}$$

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). 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. 1 is

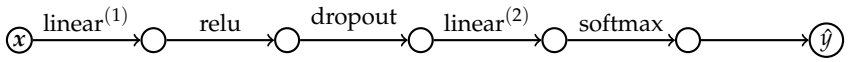

Figure 2: The diagram of the MLP implemented in `dnn_mlp.py`. The circles mean variables and edges mean modules.

$$l_{mb} = \frac{1}{N}\sum_{i=1}^{N} l(\text{softmax.forward}(\text{linear}^{(2)}.\text{forward}(\text{relu.forward}(\text{linear}^{(1)}.\text{forward}(\boldsymbol{x}_i)))), \boldsymbol{y}_i) \tag{11}$$

$$= \frac{1}{N}\sum_{i=1}^{N} l(\text{softmax.forward}(\text{linear}^{(2)}.\text{forward}(\text{relu.forward}(\boldsymbol{u}_i))), \boldsymbol{y}_i) \tag{12}$$

$$= \cdots \tag{13}$$

$$= \frac{1}{N}\sum_{i=1}^{N} l(\text{softmax.forward}(\boldsymbol{a}_i), \boldsymbol{y}_i) \tag{14}$$

$$= \frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik} \log z_{ik}. \tag{15}$$

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

$$\frac{\partial l_{mb}}{\partial \{\boldsymbol{a}_i\}_{i=1}^N} = \begin{bmatrix} (\frac{\partial l_{mb}}{\partial \boldsymbol{a}_1})^T \\ (\frac{\partial l_{mb}}{\partial \boldsymbol{a}_2})^T \\ \vdots \\ (\frac{\partial l_{mb}}{\partial \boldsymbol{a}_{N-1}})^T \\ (\frac{\partial l_{mb}}{\partial \boldsymbol{a}_N})^T \end{bmatrix}, \tag{16}$$

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

Preparation

Q3.1 Please read through `dnn_mlp.py`. The file 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 is defined in `dnn_mlp.py` Fig. 2.

Q3.2 You will modify `dnn_misc.py`. This script defines all modules that you will need to construct the MLP in `dnn_mlp.py`. 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 and derive the partial derivatives accordingly. Second, finish the implementation of `forward` and `backward` functions in `class relu`. Please follow Eqn. (3) 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.

$$\text{forward pass:} \quad \boldsymbol{s} = \text{dropout.forward}(\boldsymbol{q} \in \mathbb{R}^J) = \frac{1}{1-r} \times \begin{bmatrix} \mathbf{1}[p_1 >= r] \times q_1 \\ \vdots \\ \mathbf{1}[p_J >= r] \times q_J \end{bmatrix}, \tag{17}$$

$$\text{where } p_j \text{ is sampled uniformly from } [0,1), \forall j \in \{1, \cdots, J\}, \\ \text{and } r \in [0,1) \text{ is a pre-defined scalar named dropout rate.} \tag{18}$$

$$\text{backward pass:} \quad \frac{\partial l}{\partial \boldsymbol{q}} = \text{dropout.backward}(\boldsymbol{q}, \frac{\partial l}{\partial \boldsymbol{s}}) = \frac{1}{1-r} \times \begin{bmatrix} \mathbf{1}[p_1 >= r] \times \frac{\partial l}{\partial s_1} \\ \vdots \\ \mathbf{1}[p_J >= r] \times \frac{\partial l}{\partial s_J} \end{bmatrix}. \tag{19}$$

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: Finish the implementation of 5 functions specified above in `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:* 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.

Hint: You can match the results you are getting. It will be close to the one below.

At epoch 1
Training loss at epoch 1 is 0.28894809205071137
Training accuracy at epoch 1 is 0.919
Validation accuracy at epoch 1 is 0.908

At epoch 30
Training loss at epoch 30 is 0.032295676793784124
Training accuracy at epoch 30 is 0.9994
Validation accuracy at epoch 30 is 0.941

Q3.4 *What to do:* 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.

Hint: You can match the results you are getting. It will be close to the one below.

```
At epoch 1
Training loss at epoch 1 is 0.30055536575084546
Training accuracy at epoch 1 is 0.9118
Validation accuracy at epoch 1 is 0.919

At epoch 30
Training loss at epoch 30 is 0.04189575334960974
Training accuracy at epoch 30 is 0.9952
Validation accuracy at epoch 30 is 0.952
```

Q3.5 *What to do:* 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 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.

Hint: You can match the results you are getting. It will be close to the one below.

```
At epoch 1
Training loss at epoch 1 is 0.7636763631442427
Training accuracy at epoch 1 is 0.8394
Validation accuracy at epoch 1 is 0.852

At epoch 30
Training loss at epoch 30 is 0.28873218795235345
Training accuracy at epoch 30 is 0.9306
Validation accuracy at epoch 30 is 0.93
```

Q3.6 *What to do* 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). Essentially, without the nonlinear layer, the model is learning multinomial logistic regression similar to Q2.3.

Hint: You can match the results you are getting. It will be close to the one below.

At epoch 1
Training loss at epoch 1 is 0.3741100371813591
Training accuracy at epoch 1 is 0.888
Validation accuracy at epoch 1 is 0.886

At epoch 30
Training loss at epoch 30 is 0.11135408295300399
Training accuracy at epoch 30 is 0.9708
Validation accuracy at epoch 30 is 0.906

Problem 4 Pegasos: a stochastic gradient based solver for linear SVM

In this question, you will build a linear SVM (i.e. without kernel) classifier using the Pegasos algorithm [1]. Given a training set $\{(\mathbf{x}_n \in \mathbb{R}^D, y_n \in \{1, -1\})\}_{n=1}^N$, the primal formulation of linear SVM can be written as L2-regularized hinge loss as shown in the class:

$$\min_{\mathbf{w}} \frac{\lambda}{2}||\mathbf{w}||_2^2 + \frac{1}{N}\sum_n \max\{0, 1 - y_n \mathbf{w}^T \mathbf{x}_n\}. \tag{20}$$

Note that here we include the bias term b into parameter $\mathbf{w}$ by appending $\mathbf{x}$ with 1.

Instead of turning it into dual formulation, we are going to solve the primal formulation directly with a gradient-base algorithm. Recall for (batch) gradient descent, at each iteration of parameter update, we compute the gradients for all data points and take the average (or sum). When the training set is large (i.e., N is a large number), this is often too computationally expensive. Stochastic gradient descent with mini-batch alleviates this issue by computing the gradient on a *subset* of the data at each iteration.

Pegasos, a representative solver of eq. (20), is exactly applying stochastic gradient descent with mini-batch to this problem, with an extra step of projection described below (this is also called projected stochastic gradient descent). The pseudocode of Pegasos is given in Algorithm 1. At the t-th iteration of parameter update, we first take a mini-batch of data A_t of size K `[step (3)]`, and define $A_t^+ \subset A_t$ to be the subset of samples for which $\mathbf{w}_t$ suffers a non-zero loss `[step (4)]`. Next we set the learning rate $\eta_t = 1/(\lambda t)$ `[step (5)]`. We then perform a **two-step parameter update** as follows. We first compute $(1 - \eta_t \lambda)\mathbf{w}_t$ and for all samples $(\mathbf{x}, y) \in A_t^+$ we add the vector $\frac{y\eta_t}{K}\mathbf{x}$ to $(1 - \eta_t\lambda)\mathbf{w}_t$. We denote the resulting vector by $\mathbf{w}_{t+\frac{1}{2}}$ `[step (6)]`. This step is in fact exactly stochastic gradient descent: $\mathbf{w}_{t+\frac{1}{2}} = \mathbf{w}_t - \eta_t \nabla_t$ where

$$\nabla_t = \lambda \mathbf{w}_t - \frac{1}{K}\sum_{(\mathbf{x},y)\in A_t^+} y\mathbf{x}$$

is the (sub)gradient of the objective function on the mini-batch A_t at $\mathbf{w}_t$. Last, we set $\mathbf{w}_{t+1}$ to be the projection of $\mathbf{w}_{t+\frac{1}{2}}$ onto the set

$$B = \{\mathbf{w} : ||\mathbf{w}||_2 \leq 1/\sqrt{\lambda}\}$$

to control its L2 norm. This can be obtained simply by scaling $\mathbf{w}_{t+1}$ by $\min\left\{1, \frac{1/\sqrt{\lambda}}{||\mathbf{w}_{t+\frac{1}{2}}||}\right\}$ `[step (e)]`. For more details of Pegasos algorithm you may refer to the original paper [1].

Now you are going to implement Pegasos and train a binary linear SVM classifier on the dataset **mnist_subset.json**. You are going to implement three functions—`objective_function`, `pegasos_train`, and `pegasos_test`—in a script named `pegasos.py`. You will find detailed programming instructions in the script.

4.1 Finish the implementation of the function `objective_function`, which corresponds to the objective function in the primal formulation of SVM.

4.2 Finish the implementation of the function `pegasos_train`, which corresponds to Algorithm 1.

4.3 After you train your model, run your classifier on the test set and report the accuracy, which is defined as:

$$\frac{\text{\# of correctly classified test samples}}{\text{\# of test samples}}$$

Algorithm 1 The Pegasos algorithm

Require: a training set $S = \{(\mathbf{x}_n \in \mathbb{R}^D, y_n \in \{1, -1\})\}_{n=1}^N$, the total number of iterations T, the batch size K, and the regularization parameter λ.
Output: the last weight $\mathbf{w}_{T+1}$.

1: **Initialization:** $\mathbf{w}_1 = \mathbf{0}$
2: **for** $t = 1, 2, \cdots, T$ **do**
3: Randomly choose a subset of data $A_t \in S$ (with replacement) such that $|A_t| = K$
4: Set $A_t^+ = \{(\mathbf{x}, y) \in A_t : y\mathbf{w}_t^T\mathbf{x}_t < 1\}$
5: Set $\eta_t = \dfrac{1}{\lambda t}$
6: Set $\mathbf{w}_{t+\frac{1}{2}} = (1 - \eta_t\lambda)\mathbf{w}_t + \dfrac{\eta_t}{K}\sum_{(\mathbf{x},y)\in A_t^+} y\mathbf{x}$
7: Set $\mathbf{w}_{t+1} = \min\left\{1, \dfrac{1/\sqrt{\lambda}}{||\mathbf{w}_{t+\frac{1}{2}}||}\right\}\mathbf{w}_{t+\frac{1}{2}}$
8: **end for**

Finish the implementation of the function `pegasos_test`.

4.4 After you complete above steps, run `pegasos.sh`, which will run the Pegasos algorithm for 500 iterations with 6 settings (mini-batch size $K = 100$ with different $\lambda \in \{0.01, 0.1, 1\}$ and $\lambda = 0.1$ with different $K \in \{1, 10, 1000\}$), and output the test accuracy for the 6 settings

What to do: run script `pegasos.sh`. It will take `mnist_subset.json` as input and generate the results as below.

Hint: You can match the results you are getting. It will be close to the one below.

```
mnist test acc
k=100_lambda=0.01: test acc = 0.7980
k=100_lambda=0.1: test acc = 0.8140
k=100_lambda=1: test acc = 0.7880
k=1_lambda=0.1: test acc = 0.7790
k=10_lambda=0.1: test acc = 0.7980
k=1000_lambda=0.1: test acc = 0.8200
```

References

[1] Shai Shalev-Shwartz, Yoram Singer, Nathan Srebro, Andrew Cotter *Mathematical Programming, 2011, Volume 127, Number 1, Page 3*. Pegasos: primal estimated sub-gradient solver for SVM.