CSCI567: Machine Learning	USC, Fall 2022
Homework 3	
Instructor: Vatsal Sharan	*Due: October 26 by 2:00 pm PST*

A reminder on collaboration policy and academic integrity: Our goal is to maintain an optimal learning environment. You can discuss the homework problems at a high level with other groups, but you should not look at any other group's solutions. Trying to find solutions online or from any other sources for any homework or project is prohibited, will result in zero grade and will be reported. To prevent any future plagiarism, uploading any material from the course (your solutions, quizzes etc.) on the internet is prohibited, and any violations will also be reported. Please be considerate, and help us help everyone get the best out of this course.

Please remember the Student Conduct Code (Section 11.00 of the USC Student Guidebook). General principles of academic honesty include the concept of respect for the intellectual property of others, the expectation that individual work will be submitted unless otherwise allowed by an instructor, and the obligations both to protect one's own academic work from misuse by others as well as to avoid using another's work as one's own. All students are expected to understand and abide by these principles. Students will be referred to the Office of Student Judicial Affairs and Community Standards for further review, should there be any suspicion of academic dishonesty.

Total points: 84 points + 15 points bonus

Notes on notation:

- Unless stated otherwise, scalars are denoted by small letter in normal font, vectors are denoted by small letters in bold font and matrices are denoted by capital letters in bold font.
- $\|.\|$ means L2-norm unless specified otherwise, *i.e.*, $\|.\| = \|.\|_2$.

Instructions

We recommend that you use LaTeX to write up your homework solution. However, you can also scan handwritten notes. The homework will need to be submitted on Gradescope.

Theory-based Questions

Problem 1: Multi-class Perceptron (16pts)

Recall that a linear model for a multiclass classification problem with C classes is parameterized by C weight vectors $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_C \in \mathbb{R}^d$. In class, we derived the solution for multiclass logistic regression by minimizing the multiclass logistic loss. In this problem, you will derive the multiclass perceptron algorithm in a similar way. Specifically, the multiclass perceptron loss on a training set $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n) \in \mathbb{R}^d \times [C]$ is defined as

$$F(\boldsymbol{w}_1, \ldots, \boldsymbol{w}_C) = \frac{1}{n} \sum_{i=1}^{n} F_i(\boldsymbol{w}_1, \ldots, \boldsymbol{w}_C), \quad \text{where } F_i(\boldsymbol{w}_1, \ldots, \boldsymbol{w}_C) = \max \left\{ 0, \max_{y \neq y_i} \left(\boldsymbol{w}_y^{\mathrm{T}} \boldsymbol{x}_i - \boldsymbol{w}_{y_i}^{\mathrm{T}} \boldsymbol{x}_i \right) \right\}.$$

1.1 (8pts) To optimize this loss function, we need to first derive its gradient. Specifically, for each $i \in [n]$ and $c \in [C]$, write down the partial derivative $\frac{\partial F_i}{\partial \boldsymbol{w}_c}$ (provide your reasoning). For simplicity, you can assume that for any i, $\boldsymbol{w}_1^{\mathrm{T}} \boldsymbol{x}_i, \ldots, \boldsymbol{w}_C^{\mathrm{T}} \boldsymbol{x}_i$ are always C distinct values (so that there is no tie when taking the max over them, and consequently no non-differentiable points needed to be considered).

For each n, let $\hat{y}_i = \operatorname{argmax}_{y \in [C]} \boldsymbol{w}_y^{\mathrm{T}} \boldsymbol{x}_i$. Then by definition, F_i can be written as

$$\begin{cases} 0, & \text{if } \hat{y}_i = y_i, \\ \boldsymbol{w}_{\hat{y}_i}^{\mathrm{T}} \boldsymbol{x}_i - \boldsymbol{w}_{y_i}^{\mathrm{T}} \boldsymbol{x}_i, & \text{else.} \end{cases}$$

Its partial derivative with respect to $\boldsymbol{w}_c$ is then

$$\begin{cases} \boldsymbol{0}, & \text{if } \hat{y}_i = y_i, \\ \boldsymbol{x}_i, & \text{else if } c = \hat{y}_i, \\ -\boldsymbol{x}_i, & \text{else if } c = y_i, \\ \boldsymbol{0}, & \text{else.} \end{cases}$$

Rubrics: 2 points for each of the 4 cases. There are many other ways to write this, such as using indicator functions. It is of course also possible to combine some of these cases (such as the first and the fourth ones).

1.2 (4pts) Similarly to the binary case, multiclass perceptron is simply applying SGD with learning rate 1 to minimize the multiclass perceptron loss. Based on this information, fill in the missing details in the repeat-loop of the algorithm below (your solution cannot contain implicit quantities such as $\nabla F_i(\boldsymbol{w})$; instead, write down the exact formula based on your solution from the last question).

Algorithm 1: Multiclass Perceptron

Input: A training set $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)$
Initialization: $\boldsymbol{w}_1 = \cdots = \boldsymbol{w}_C = \boldsymbol{0}$
Repeat:
    randomly pick an example $(\boldsymbol{x}_i, y_i)$ and compute $\hat{y}_i = \operatorname{argmax}_{y \in [C]} \boldsymbol{w}_y^{\mathrm{T}} \boldsymbol{x}_i$
    if $\hat{y}_i \neq y_i$ **then**
        $\boldsymbol{w}_{\hat{y}_i} \leftarrow \boldsymbol{w}_{\hat{y}_i} - \boldsymbol{x}_i$
        $\boldsymbol{w}_{y_i} \leftarrow \boldsymbol{w}_{y_i} + \boldsymbol{x}_i$

Rubrics: 1 point for randomly picking an example, 3 points for correctly implementing the rest of SGD (again, there are many equivalent ways of doing so). Do not deduct points if the gradient is wrong solely due to mistakes from the last question.

1.3 (4pts) At this point, you should find that the parameters $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_C$ computed by Multiclass Perceptron are always linear combinations of the training points $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$, that is, $\boldsymbol{w}_c = \sum_{i=1}^{n} \alpha_{c,i} \boldsymbol{x}_i$ for some coefficients

$\alpha_{c,i}$. Just as we saw for kernelized linear regression, this means that we can kernelize Multiclass Perceptron as well. Consider some kernel function $k(\cdot,\cdot)$ with a corresponding feature map $\boldsymbol{\phi}(\cdot)$. Fill in the missing details in the repeat-loop of the algorithm below that maintains and updates the coefficient $\alpha_{c,i}$ for all c and i for the the kernalized solution $\boldsymbol{w}_c = \sum_{i=1}^{n} \alpha_{c,i}\boldsymbol{\phi}(\boldsymbol{x}_i)$. To do this, try to see how the weights $\alpha_{c,i}$ should be updated such that $\boldsymbol{w}_c = \sum_{i=1}^{n} \alpha_{c,i}\boldsymbol{\phi}(\boldsymbol{x}_i)$ is always the same as what one would get by running Algorithm 1 with $\boldsymbol{x}_i$ replaced by $\boldsymbol{\phi}(\boldsymbol{x}_i)$ for all i.

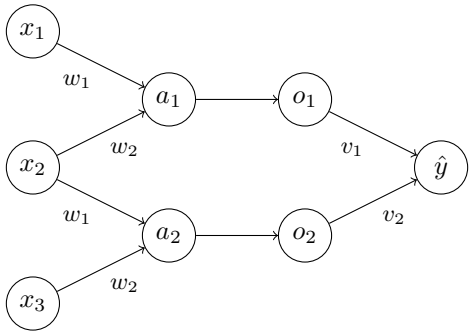
Algorithm 2: Multiclass Perceptron with kernel function $k(\cdot,\cdot)$

Input: A training set $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)$
Initialize: $\alpha_{c,n} = 0$ for all $c \in [C]$ and $i \in [n]$
Repeat:
    randomly pick an example $(\boldsymbol{x}_i, y_i)$ and compute $\hat{y}_i = \operatorname{argmax}_{y \in [C]} \left(\sum_{m=1}^{n} \alpha_{y,m} k(\boldsymbol{x}_m, \boldsymbol{x}_i)\right)$
    if $\hat{y}_i \neq y_i$ **then**
        $\alpha_{\hat{y}_i,i} \leftarrow \alpha_{\hat{y}_i,i} - 1$
        $\alpha_{y_i,i} \leftarrow \alpha_{y_i,i} + 1$

Rubrics: 1 point for randomly picking an example, 3 points for correctly implementing the rest (again, there are many equivalent ways of doing so). Storing the kernel matrix to avoid repeated calculations is of course acceptable. Do not deduct points if the mistake is solely inherited from the first question. However, deduct 2 points if there are any operations involving the feature vectors other than feeding them to the kernel function (since that is one of the most important aspects of kernel methods).

Problem 2: Backpropagation for CNN (18pts)

Consider the following mini convolutional neural net, where (x_1, x_2, x_3) is the input, followed by a convolution layer with a filter (w_1, w_2), a ReLU layer, and a fully connected layer with weight (v_1, v_2).



More concretely, the computation is specified by

$$\begin{aligned} a_1 &= x_1 w_1 + x_2 w_2 \\ a_2 &= x_2 w_1 + x_3 w_2 \\ o_1 &= \max\{0, a_1\} \\ o_2 &= \max\{0, a_2\} \\ \hat{y} &= o_1 v_1 + o_2 v_2 \end{aligned}$$

For an example $(\boldsymbol{x}, y) \in \mathbb{R}^3 \times \{-1, +1\}$, the logistic loss of the CNN is

$$\ell = \ln(1 + \exp(-y\hat{y})),$$

which is a function of the parameters of the network: w_1, w_2, v_1, v_2.

2.1 (4pts) Write down $\frac{\partial \ell}{\partial v_1}$ and $\frac{\partial \ell}{\partial v_2}$ (show the intermediate steps that use chain rule). You can use the sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ to simplify your notation.

$$\frac{\partial \ell}{\partial v_1} = \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial v_1} \quad \text{(1 point)}$$

$$= \frac{-ye^{-y\hat{y}}}{1 + e^{-y\hat{y}}} o_1 = -\sigma(-y\hat{y}) y o_1 = (\sigma(y\hat{y}) - 1) y o_1 \quad \text{(1 point)}$$

$$\frac{\partial \ell}{\partial v_2} = \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial v_2} \quad \text{(1 point)}$$

$$= \frac{-ye^{-y\hat{y}}}{1 + e^{-y\hat{y}}} o_2 = -\sigma(-y\hat{y}) y o_2 = (\sigma(y\hat{y}) - 1) y o_2 \quad \text{(1 point)}$$

Either one of the last three expressions is acceptable.

2.2 (6pts) Write down $\frac{\partial \ell}{\partial w_1}$ and $\frac{\partial \ell}{\partial w_2}$ (show the intermediate steps that use chain rule). The derivative of the ReLU function is $H(a) = \mathbb{I}[a > 0]$, which you can use directly in your answer.

$$\frac{\partial \ell}{\partial w_1} = \frac{\partial \ell}{\partial a_1} \frac{\partial a_1}{\partial w_1} + \frac{\partial \ell}{\partial a_2} \frac{\partial a_2}{\partial w_1} \quad \text{(1 point)}$$

$$= \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial o_1} \frac{\partial o_1}{\partial a_1} \frac{\partial a_1}{\partial w_1} + \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial o_2} \frac{\partial o_2}{\partial a_2} \frac{\partial a_2}{\partial w_1} \quad \text{(1 point)}$$

$$= (\sigma(y\hat{y}) - 1) y (v_1 H(a_1) x_1 + v_2 H(a_2) x_2) \quad \text{(1 point)}$$

Similarly

$$\frac{\partial \ell}{\partial w_2} = \frac{\partial \ell}{\partial a_1}\frac{\partial a_1}{\partial w_2} + \frac{\partial \ell}{\partial a_2}\frac{\partial a_2}{\partial w_2} \quad \text{(1 point)}$$

$$= \frac{\partial \ell}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial o_1}\frac{\partial o_1}{\partial a_1}\frac{\partial a_1}{\partial w_2} + \frac{\partial \ell}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial o_2}\frac{\partial o_2}{\partial a_2}\frac{\partial a_2}{\partial w_2} \quad \text{(1 point)}$$

$$= (\sigma(y\hat{y}) - 1)y(v_1 H(a_1)x_2 + v_2 H(a_2)x_3). \quad \text{(1 point)}$$

Again, other equivalent expressions are acceptable.

2.3 (8pts) Using the derivations above, fill in the missing details of the repeat-loop of the Backpropagation algorithm below that is used to train this mini CNN.

Algorithm 3: Backpropagation for the above mini CNN

Input: A training set $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)$, learning rate η

Initialize: set w_1, w_2, v_1, v_2 randomly

Repeat:

randomly pick an example $(\boldsymbol{x}_i, y_i)$

Forward propagation: compute (4 points)

$$a_1 = x_{n1}w_1 + x_{n2}w_2, a_2 = x_{n2}w_1 + x_{n3}w_2$$

$$o_1 = \max\{0, a_1\}, o_2 = \max\{0, a_2\}, \hat{y} = o_1 v_1 + o_2 v_2$$

Backward propagation: update (4 points)

$$w_1 \leftarrow w_1 - \eta(\sigma(y_n\hat{y}) - 1)y_n(v_1 H(a_1)x_{n1} + v_2 H(a_2)x_{n2})$$
$$w_2 \leftarrow w_2 - \eta(\sigma(y_n\hat{y}) - 1)y_n(v_1 H(a_1)x_{n2} + v_2 H(a_2)x_{n3})$$
$$v_1 \leftarrow v_1 - \eta(\sigma(y_n\hat{y}) - 1)y_n o_1$$
$$v_2 \leftarrow v_2 - \eta(\sigma(y_n\hat{y}) - 1)y_n o_2$$

- Deduct 1 point for writing x_1, x_2, x_3 instead of x_{n1}, x_{n2}, x_{n3} in the forward propgagation.
- Deduct 1 point for writing x_1, x_2, x_3, y instead of $x_{n1}, x_{n2}, x_{n3}, y_n$ in the backward propgagation.
- Deduct 2 points if updating w_1/w_2 with the updated value of v_1/v_2.
- Do not deduct points for using the wrong gradients solely due to mistakes from previous two questions.

Programming-based Questions

As in previous homeworks, you need to have your coding environment setup for this part. We use python3 (version $\geq$ 3.7) in our programming-based questions. There are multiple ways you can install python3, for example:

- You can use conda to configure a python3 environment for all programming assignments.
- Alternatively, you can also use virtualenv to configure a python3 environment for all programming assignments

After you have a python3 environment, you will need to install the following python packages:

- numpy
- tqdm **(a new package used in this HW, please install it to monitor training progress)**
- matplotlib (for plotting figures)

Note: You are **not allowed** to use other packages such as *tensorflow, pytorch, keras, scikit-learn, scipy*, etc. for 3.1-3.2. If you have other package requests, please ask first before using them. You are **allowed** to use any packages for 3.3-3.5.

Download the files for the programming part from `https://vatsalsharan.github.io/fall22/hw3.zip`.

Problem 3: Fashion MNIST (50pts + 15pts bonus)

We will use a subset of the Fashion MNIST dataset (`https://github.com/zalandoresearch/fashion-mnist`) dataset in this part. The entire dataset contains 70,000 grayscale images in 10 categories. In this HW, we will use 10,000 examples for training, 2,000 examples for validation and 10,000 examples for testing. The datapoints in the Fashion MNIST dataset are images of individual articles of clothing at low resolution (28 by 28 pixels), as seen in Fig. 1:

Figure 1: Fashion-MNIST samples[1]

Label	Class
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

Figure 2: Fashion MNIST class name

Fashion MNIST is intended as a drop-in replacement for the classic MNIST dataset. MNIST is often regarded as the "Hello, World" of machine learning algorithms for computer vision. The MNIST dataset contains images of handwritten digits (0, 1, 2, etc.), and is similar in format to the Fashion MNIST dataset. This problem uses Fashion MNIST instead of MNIST to add some variety (and fashion), and also because it's slightly more challenging than regular MNIST. Both datasets are relatively small and are often used to verify that an algorithm works as expected. They're good starting points to test and debug code, and to explore ideas.

[1]Figure from `https://medium.com/@ipylypenko/exploring-neural-networks-with-fashion-mnist-b0a8214b7b7b`

The images from Fashion MNIST are stored as 28x28 arrays, with pixel values ranging from 0 to 255. The labels are an array of integers, ranging from 0 to 9. The labels correspond to the item of clothing the image represents, as in Fig. 2. Each image is mapped to a single label.

Part I - Multi-Layer Perceptron (MLP) (38pts)

General instructions

- In this part you will implement neural nets. We provide the bootstrap code and you are expected to complete the functions.
- Do not import libraries other than those already imported in the original code.
- Please follow the data type annotations. You have to make the function's return values match the required type.
- Only make modifications in {`neural_networks.py`}.
- For parts where you will be running the code, we include the expected running time on Google Colab notebook in brackets (such as [5min]). If the running time of your code is much larger, it is likely that it has a bug. Your runtime maybe much faster though (indeed, the code ran faster on our laptops than on Colab).

3.1 Neural nets (24pts) In this task, you are asked to implement neural networks! You will later use this neural network to classify Fashion MNIST images into articles of clothing (0-9). The architecture of the neural network you will implement is based on the *multi-layer perceptron* (also known as MLP, just another term for fully connected feedforward networks we discussed in class). It is designed for a K-class classification problem. Let $(\boldsymbol{x}, y)$ where $x \in \mathbb{R}^d, y \in \{1, 2, ..., K\}$ be a labeled instance. A MLP predicts the label by carrying out the following computations:

$$
\begin{aligned}
&\textbf{input features} : \boldsymbol{x} \in \mathbb{R}^d \\
&\textbf{linear}^{(1)} : \boldsymbol{u} = \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)} \quad , \boldsymbol{W}^{(1)} \in \mathbb{R}^{M \times d} \text{ and } b^{(1)} \in \mathbb{R}^M \\
&\textbf{relu} : \boldsymbol{h} = \max\{0, \boldsymbol{u}\} = \begin{bmatrix} \max\{0, u_1\} \\ \vdots \\ \max\{0, u_M\} \end{bmatrix} \\
&\textbf{linear}^{(2)} : \boldsymbol{a} = \boldsymbol{W}^{(2)}\boldsymbol{h} + \boldsymbol{b}^{(2)} \\
&\textbf{softmax} : \boldsymbol{z} = \begin{bmatrix} \frac{e^{a_1}}{\sum_k e^{a_k}} \\ \vdots \\ \frac{e^{a_K}}{\sum_k e^{a_k}} \end{bmatrix} \\
&\textbf{predicted label} : \hat{y} = \underset{k}{\operatorname{argmax}}\, z_k
\end{aligned}
$$

You might find it helpful to first go through what the following parts are asking, and then read through the `main` function to see how we will be using the functions you will write. Note that to better utilize the validation set, early-stopping is provided in the starter code. Early-stopping has a hyperparameter which we call *Patience*. If the patience parameter is set to k *epochs*, then training will terminate if there is no improvement in the validation accuracy for k epochs in a row. An epoch is just one pass through the training set.

3.1.1 Linear layer (6pts) First, you need to implement the linear layer of the MLP by implementing three python functions in `class linear_layer`. This layer has two parameters `W` and `b`.

- In the function `def __init__(self, input_D, output_D)`, you need to randomly initialize the entries of `W` and `b` with mean 0 and standard deviation 0.1 using `np.random.normal`. You also need to initialize the gradients to zeroes in the same function.
- In `def forward(self, X)`, implement the forward pass of this layer. Note that the input `X` contains several examples, each of which needs to be passed through this layer. Try to use matrix operation instead of for loops to speed up your code. (As you might have realized with previous homeworks, matrix operations are much faster than for loops in Python. Please read this nice guide on *vectorized* operations to learn more: `https://www.askpython.com/python-modules/numpy/vectorization-numpy`).

- In `def backward(self, X, grad)`, implement the backward pass of this layer. Here, `grad` is the gradient with respect to the output of this layer, and you need to find and store the gradients with respect to `W` and `b`, and also find and return the gradients with respect to the input `X` of this layer.

3.1.2 ReLU (4pts) Next, you need to implement the ReLU activation by implementing 2 python functions in `class relu`. There are no parameters to be learned in this module.

- In `def forward(self, X)`, implement the forward pass of ReLU activation.
- In `def backward(self, X, grad)`, implement the backward pass of ReLU activation. Here, `X` is the gradient with respect to the output of this layer, and you need to return the gradient with respect to the input `X`.

3.1.3 Mini-batch stochastic gradient descent (2pts) Next, implement mini-batch version of stochastic gradient descent to learn the parameters of the neural network. Recall that for a general optimization problem with a parameter w, SGD iteratively computes $w_t = w_{t-1} - \eta \nabla F(w)$, where η is the step size, and $\nabla F(w)$ is the stochastic gradient (at w_{t-1} *w.r.t* loss function $F(\cdot)$). You need to complete `def miniBatchGradientDescent(model, _learning_rate)`, where we update the parameters of each layer. Note that this function is executed after the backward pass of each layer has been called and the gradients have been stored properly.

3.1.4 Backpropagation (4pts) Your network is almost ready to be trained. The only function you need to implement in this part is `backward_pass`, which calls the backward pass of each layer in the correct order and with the correct inputs.

3.1.5 Gradient checker (4pts) As you probably realized as you wrote the code for the previous parts, backpropagation is a subtle algorithm and it is easy to make mistakes. In this part, you will implement gradient checking to give you confidence that you are calculating the gradients of your model correctly. The idea behind gradient checking is as follows. As mentioned in the lecture, one naive way to approximate gradient is by

$$\frac{dF(w)}{dw} = \lim_{\epsilon \to 0} \frac{F(w+\epsilon) - F(w-\epsilon)}{2\epsilon}$$

We can use this relation to check that we are computing gradients correctly using backpropagation. For example, to check the gradient of the $(0,0)$ entry of $\boldsymbol{W}^{(1)}$, we apply a small $\epsilon = 1\text{e}{-3}$ perturbation to the $(0,0)$ entry of $\boldsymbol{W}^{(1)}$. Then, we compare the approximate gradient of the objective function with respect to the $(0,0)$ entry of $\boldsymbol{W}^{(1)}$ computed by the above equation, to the value obtained by backpropagation. These two values should be close to each other if backpropagation is implemented correctly. Please implement the `gradient_checker` function to do this. Note that to enable this function at runtime, you should turn on the `check_gradient` flag when you run the code later.

Apart from the gradient checker, we also provide a `magnitude_checker` function to help you debug the code. The gradient on any mini-batch provides an unbiased estimate of the true gradient, therefore we expect the gradient on a batch size of 50 to be similar in magnitude to the gradient on a batch size of 5,000. The `magnitude_checker` function (which is provided to you) checks that the ℓ_1-norm of $\boldsymbol{W}^{(1)}$ is of the same magnitude on these two batch sizes.

3.1.6 Screenshots and plots (4pts) [40sec] After finishing all five modules above, run the below command to get a result file `MLP_lr0.01_b5.json`. Take a screenshot of your output log, like in `screenshot_run_b5.png` (included in the zip file). (2pts) Plot the training accuracy *vs.* epochs using `plot_train_process.py`. (2pts) For the magnitude check, you should expect the two numbers are of the same magnitude. For the four gradient checks, you should expect the number from the backpropagation to be the same as that from the approximation. For training accuracy, the curve should grow as the number of epochs increases. Some fluctuation in the middle is acceptable.

```
python neural_network.py --minibatch_size 5 --check_gradient --check_magnitude
```

Screen shot: Fig. 3.

```
Training data size: 10000, Validation data size: 2000, Test data size: 10000
Check the magnitude (L1-norm of layer L1) of gradient with batch size 50: 148.290031 and with batch size 5k: 117.085383
Check the gradient of W in the L1 layer from backpropagation: 0.000000 and from approximation: 0.000000
Check the gradient of b in the L1 layer from backpropagation: 0.129579 and from approximation: 0.129579
Check the gradient of W in the L2 layer from backpropagation: 0.004011 and from approximation: 0.004011
Check the gradient of b in the L2 layer from backpropagation: 0.121975 and from approximation: 0.121975
At epoch 1
100%|██████████| 2000/2000 [00:01<00:00, 1846.66it/s]
Training loss at epoch 1 is 6.05685388211968
Training accuracy at epoch 1 is 0.7893
Validation accuracy at epoch 1 is 0.7765
At epoch 2
100%|██████████| 2000/2000 [00:00<00:00, 2034.73it/s]
Training loss at epoch 2 is 5.173094948088359
Training accuracy at epoch 2 is 0.8187
Validation accuracy at epoch 2 is 0.8015
```

Figure 3: Screenshot of running with the command.

Training accuracy: Fig. 4.

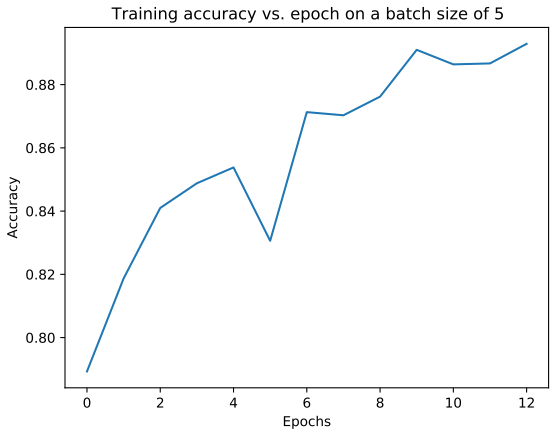


Figure 4: Training accuracy with the command.

3.2 Batch size (14pts) [6min] We now explore the effect of different batch size on training. Run the command from the previous part with 5 different batch sizes: 1, 5, 50, 500, 5000.

3.2.1 Accuracy and time (4pts) Plot testing accuracy and training time *w.r.t.* batch size using `plot_batch.py`.

3.2.2 Epochs and gradient updates (4pts) What is the number of training epochs required to get the best model for each batch size? How many gradient updates are required get the best model for each batch size? (We define a gradient update to be a single stochastic gradient descent step, where the gradient may have been computed over some mini-batch).

13, 10, 43, 39, 158 training epochs; 130000, 20000, 8600, 780, 316 gradient updates, to get the best model for batch size 1, 5, 50, 500, 5000, respectively (160000, 26000, 9200, 840, 322 gradient updates are also acceptable for the second part).

3.2.3 Analysis (6pts) Answer the following question based on the previous plots and results:

(i). Does smaller batch size guarantee faster training? Why do you think this is the case?

(ii). Does larger batch size imply higher test accuracy? Why do you think this is the case?

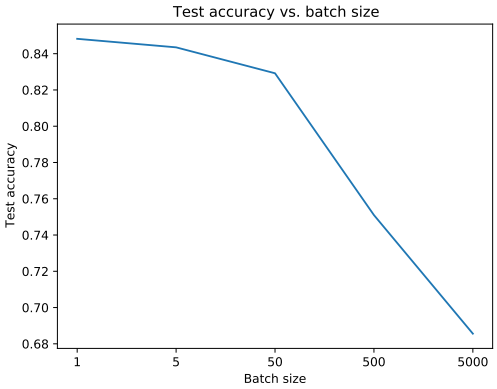


Figure 5: Test accuracy.

Figure 6: Train time.

(iii). Does larger batch size imply less gradient updates to converge? Why do you think this is the case?

(i). No, more gradient updates are needed to converge, which takes time. Also, because of the way computer memory is organized, it is often as efficient to take the gradient with respect to a few samples as it is to take the gradient with respect to a single sample.

(ii). No, the stochasticity in smaller batch sizes helps with generalization. See https://wandb.ai/ayush-thakur/dl-question-bank/reports/What-s-the-Optimal-Batch-Size-to-Train-a-Neural-Network---VmlldzoyMDkyNDU and https://stats.stackexchange.com/questions/164876/what-is-the-trade-off-between-batch-size-and-number-of-iterations-to-train-a-neu.

(iii). Yes, all are unbiased estimates of the gradient but larger batch size implies smaller variance.

Part II - Explore on Colab (12pts)

The two-layer MLP from the previous part is our base model, and in the rest of this HW we will explore some variations on it. We will do this on a Colab notebook `fashion_mnist.ipynb`. To run the notebook, upload the `fashion_mnist.ipynb` to your USC Google Drive. Then, add Google Colab to your Google App by New → More → Connect more apps → Type in Google Colab and install it, as in Fig. 7 and Fig. 8. After that, you can run the notebook with your browser. The notebook is based on *TensorFlow*, a popular library for neural networks. TensorFlow enables us to build and train neural networks with a few simple lines of code. We recommend that you go through the TensorFlow code we provide and understand what's happening, but for the purpose of this homework you will mainly have to run the code and understand the results.

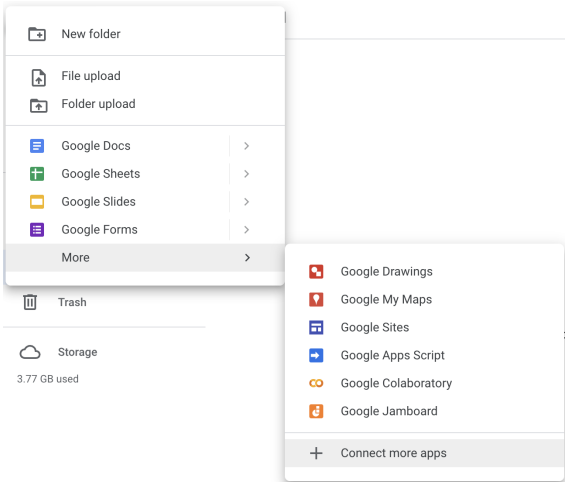


Figure 7: Colab install 1

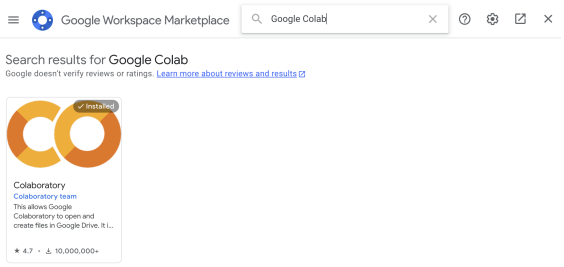


Figure 8: Colab install 2

3.3 The effect of early-stopping (6pts) [3.5min] We provided built-in early-stopping in the starter code, but did not explore the effect that it has on training. In this question, we will explore the effect of early-stopping. To do this, we evaluate the training, validation and test accuracy for each training epoch. We then plot the training, validation and test accuracy throughout the training process to see how early-stopping works. We train 30 epochs on a batch size of 5 without early-stopping.

(i). Plot the training accuracy vs. the number of epochs. On the same graph, plot the validation and test accuracy vs. the number of epochs. Also, mark the point on the validation accuracy curve where we previously early-stopped. (We provide the plotting code, you only need to run it). (2pts)

(ii). What is the trend of training and test accuracy after the early-stopped point? (2pts)

(iii). Based on the plot, what do you think could go wrong if the patience parameter for early-stopping is too small? (Recall that if the patience parameter is set to k epochs, then training will terminate if there is no improvement in the validation accuracy for k epochs in a row.) (2pts)

(i). Train-val-test plot: Fig. 9
(ii). Training curve goes up, but validation and test curve fluctuate.
(iii). Stop at suboptimal points due to small fluctuations.

3.4 SGD with momentum (6pts) [14min] We mentioned in class that adding a "momentum" term, which encourages the model to continue along the previous gradient direction helps the network to converge. Concretely, with an initial velocity $\boldsymbol{v} = \mathbf{0}$, we update the gradient by

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} + \nabla F(\boldsymbol{w})$$
$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \boldsymbol{v}$$

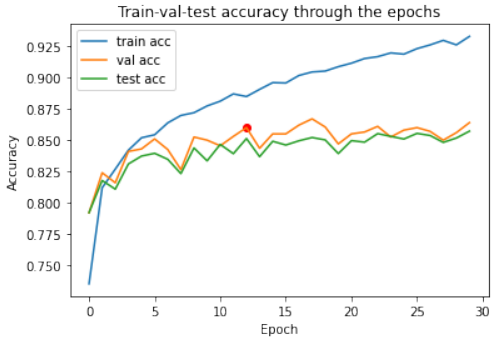


Figure 9: Train-val-test accuracy for 30 epochs

where η is the learning rate and α is the factor describing how much weight we put on the previous gradients. $\alpha = 0$ is equivalent to gradient update without momentum.

We provide the `sgd_with_momentum` function with argument α. In this question, we will explore how the momentum factor α effects training loss and test accuracy.

(i). Run the code to call the function `sgd_with_momentum` with α = 0.1, 0.3, 0.5, 0.7, 0.9. We store the training loss and test accuracy returned by the function. We visualize the training loss by plotting 5 curves for the 5 different values of α on the same graph. Each curve has the epoch on the x-axis and the training loss on the y-axis. (4pts)

(ii). Based on this, what is a suitable value of α? Therefore, how should training ideally rely on previous gradients for better convergence? (2pts)

(i). SGD momentum curve: Fig. 10

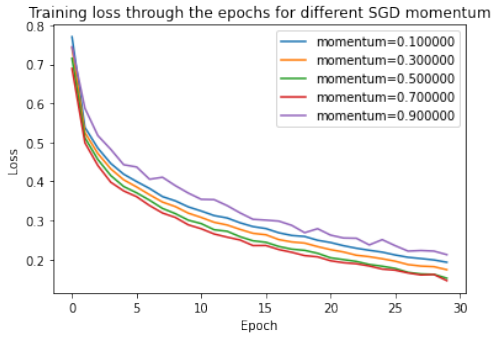


Figure 10: SGD momentum with different α

(ii). 0.5 or 0.7. Relying on previous gradients helps convergence, but too much is harmful.

Part III (Bonus question) - Exploring Out-of-Distribution (OOD) generalization on Colab (15pts) [15min]

This question is *optional*. You will get bonus points for finishing this.

A major research direction at present is to ensure that our ML models not only do well on test data drawn from the same distribution as training data, but that they also do well when they get data from distributions different from the original distribution. This is known as Out-of-Distribution (OOD) generalization. In the Fashion MNIST dataset, the training and test set are drawn from the same distribution. Let's explore how our models do on test data coming from a slightly different distribution.

3.5 Translation and rotation (15pts) We modify the original test datapoints, by moving up the images of the test set by 4 pixels. By doing this, we create a new translated test set. Run the code to see the original and modified images. You will notice that to a human eye, the new test set is not any more difficult than the original test set.

(i). But can our MLP model still do well on the new test set? What's the test accuracy on the two-layer MLP? (1pt)

(ii). What if we try a different model architecture? For example, in class we saw that convolutional neural networks are good at dealing with image translation. We replace the first linear layer with a convolutional layer of 64 kernels with size 7×7, followed by max-pooling. This can be done with just one or two lines of code in TensorFlow. Calculate the number of parameters of the CNN model and the original 2-layer MLP model (show your calculation), and verify that the CNN has fewer parameters than the MLP model. (3pts)

(iii). Train the 2-layer CNN on the original training data and test it on both the original and the translated test sets. What is the in-domain test accuracy and the translated test accuracy of the CNN model? Can you provide some intuition behind these numbers? (2pts) [6min]

(iv). Going one step further, we can make the CNN deeper by adding one more convolutional layer of 64×128 (64 input channels and 128 output channels) kernels with size 2×2 between the two existing layers, followed by max-pooling. Verify that the deeper 3-layer CNN model has fewer parameters than the 2-layer CNN model (show your calculation). (3pts)

(v). What is the in-domain and the translated test accuracy of the deeper 3-layer CNN model? Provide some intuition on why it is better than the 2-layer CNN on the translated set. (3pts) [9min] You will notice that the deeper model takes some time to train. If we trained on GPUs instead of CPUs, training would be much faster. This is because GPUs are much more efficient at matrix computations, which is exactly what neural network training demands.

Next, we create 3 more OOD test sets by rotation. We rotate the images in the original test set by 90, 180 and 270 degrees.

(vi). Provide the test accuracy of the 2-layer MLP model, the 2-layer CNN model and the 3-layer CNN model on the three rotation test sets. Are the 2-layer CNN and the 3-layer CNN still doing well? (3pts)

(i). No, 47.6
(ii). 2-layer MLP: $128\times784+128+128\times10+10 = 101770$. 2-layer CNN: $64\times7\times7+64+7744\times10+10 = 80650$
(iii). 87.9, 54.7. CNN captures spatial feature better than MLP. Notice the input to MLP is one vector of size 784 for each example, and the input to CNN is one image of size 28×28 for each example.
(iv). 3-layer CNN: $64 \times 49 + 64 + 64 \times 128 \times 2 \times 2 + 128 + 10 \times 3200 + 10 = 68106$.
(v). 88.0, 57.0. Becomes more translation invariant with another convolutional and max-pooling layer (or is able to capture farther feature through translation by another convolution and max-pooling)
(vi). 2-layer MLP: 1.9, 19.7, 5.6; 2-layer CNN: 5.6, 20.6, 5.3; 3-layer CNN: 5.4, 5.6, 5.6.

Note: Tensorflow version 2.9.2 has slightly different numbers for model accuracy. Results within a reasonable range (e.g., ±1) are acceptable.

Deliverables for Problem 3: Code for 3.1 as a separate Python file `neural_networks.py`. Screenshot for 3.1. Plots for part 3.1, 3.2, 3.3 and 3.4. Number of epochs and gradient updates for part 3.2. Analysis and explanation for parts 3.2, 3.3, 3.4. For the optional bonus question 3.5, submit the test accuracy on in-domain and OOD test sets of the three models, the number of parameters of the three models, and the explanations we ask for.