CS37300 Homework — Homework 4— Last update: Aug 10, 2023
Theory Due: Mar. 29 (Friday), 2024 (11:59 pm)
Programming Due: Apr. 5 (Friday), 2024 (11:59 pm)

Name: Derrick Khoo PUID: 0039028290

Instructions and Policy: You are allowed to study with others and use online resources for reference, however, the work you turn in must be your own. This means do not copy/paste from Stack Exchange (or from another student.) If you have worked closely with other students, provide their name(s) and a brief (at most one paragraph) description of the interaction; if we feel this oversteps the bounds, we will discuss it with you. Each student should write up their own solutions independently.

The requirements below are supposed to be followed in this and further homework assignments.

- For your theoretical submission:
  - YOU MUST INCLUDE YOUR NAME IN THE HOMEWORK.
  - The answers MUST be submitted via Gradescope.
  - Please write clearly and concisely - clarity and brevity will be rewarded. Refer to known facts as necessary.
  - Theoretical questions MUST include the intermediate steps to the final answer.
- For your programming answers:
  - The Python scripts will be submitted separately via Gradescope
  - Zero points will be given in any question where the Python code answer doesn't match the answer on Gradescope.
  - If the answer is/includes a plot, it should be added to your theoretical submission.

Your code is REQUIRED to run on Python 3.11 in an environment detailed in Homework 0 under the Python Installation section. While your code may run in other environments, any points lost due to environmental issues will not be regraded.

Please make sure you didn't use any library/source explicitly forbidden to use. If such library/source code is used, you will get 0 pt for the coding part of the assignment.

If your code doesn't run on Gradescope, then even if it compiles in another computer, it will still be considered not-running and the respective part of the assignment will receive 0 pt.

General Instructions

For this assignment, please download the provided `hw4_handout.zip` file on **Brightspace**.

For those questions that ask you to plot figures, do analysis, etc., please include your answers along with solutions to theoretical questions in the written PDF report. For other coding questions, you need to package all of your python files into a `.tar.gz` file and submit to the data server via `turnin` command. Detailed instructions of how to submit your code are in the last page.

Theoretical Questions (8+16+28=52 pts)

Please submit your answers on Gradescope.

Q1 (8 pts): True or False questions

Answer the following as True or False with a justification or example. Points are uniformly distributed among the questions.

1. **(2 pts)** The activation values of the hidden units are always bounded between 0 and 1 when using the sigmoid activation function.

True. This is because the sigmoid function is defined as: $\frac{1}{1+e^{-x}}$, $\forall x$. Thus, the lower bound of the sigmoid function is 0 while the upper bound of the sigmoid function is 1.

2. **(4 pts)** The derivative of the sigmoid function is greater than the derivative of the tanh function near 0.

False. Derivative of sigmoid $\frac{d}{dx}(\sigma(x)) = \frac{1}{1+e^{-x}}\left(1-\frac{1}{1+e^{-x}}\right) = \sigma(x)(1-\sigma(x))$
while derivative of tanh $\frac{d}{dx}(\tanh(x)) = 1-\left(\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}\right)^2 = 1-\tanh^2(x)$
When $x=0$, $\sigma(0)=0.5$, $\tanh(0)=0$.
$\therefore \lim_{x\to 0}\frac{d}{dx}(\sigma(x)) = 0.5(1-0.5) = 0.25$, $\lim_{x\to 0}\frac{d}{dx}(\tanh(x)) = 1-0^2 = 1 > \frac{d}{dx}(\sigma(x))$

3. **(2 pts)** In a fully connected layer of an Artificial Neural Network (ANN), all neurons share the same set of weights, which are learned to model complex relationships between inputs and outputs.

False. Each neuron has its own set of weights for its own inputs, which are learned and adjusted during backpropagation. Hence, all neurons do not share the same set of weights.

Q2 (18 pts): Deep Learning/Activations

In this question, we will dive deeper into training ReLU-activated neurons (ReLU neurons).

Inner product layer (fully connected layer) As the name suggests, every output neuron of inner product layer is fully connected to the input neurons. The output is a vector resulting from the multiplication of the input with a weight matrix W plus a bias offset, i.e.:

$$\mathbf{z}(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b}, \tag{1}$$

where the input $\mathbf{x}$ is d dimensional column vector, and $\mathbf{W}$ is a $n \times d$ matrix and $\mathbf{b}$ is n dimensional column vector. This is just a linear transformation of the input. Often, we will concatenate the bias $\mathbf{b}$ into W by concatenating the constant 1 to $\mathbf{x}$. The weight parameter $\mathbf{W}$ and bias parameter $\mathbf{b}$ are learnable in this layer.

Activation. We add nonlinear activation functions after the inner product layers to model non-linearity (non-polynomial, to be more precise). Here are variants of the ReLU activation function, popular choices for non-polynomial activations:

- **ReLU**:
$$\text{relu}(z) = \max(0, z) \tag{2}$$
- **Leaky ReLU**:
$$\text{leaky\_relu}(z) = \begin{cases} z & \text{if } z \geq 0, \\ \alpha z & \text{if } z < 0, \end{cases} \tag{3}$$
- **ELU**:
$$\text{elu}(z) = \begin{cases} z & \text{if } z \geq 0, \\ \alpha(\exp(z) - 1) & \text{if } z < 0, \end{cases} \tag{4}$$

  where α is a small constant.

If the input to the above functions is a vector $\mathbf{z}$*, then each function is applied to each element of the vector.*

Consider the negative log-likelihood as the score and gradient descent as the search algorithm used to find the optimal parameters of the neural network.

1. (**6 pts**) Compute the derivatives of the ReLU, leaky ReLU and ELU units with respect to variable z in equations (2), (3) and (4), respectively. (For points where the gradient is undefined, the derivative at that point may be designated as either the left-hand derivative or the right-hand derivative.)

Derivative of ReLU : $\frac{d}{dz}(\text{ReLU}(z)) = \begin{cases} 1 & \text{for } z>0 \\ 0 & \text{for } z \le 0 \end{cases}$ ∵ gradient is undefined when output is 0, thus take right-hand derivative

Derivative of leaky ReLU $= \begin{cases} 1 & \text{for } z \ge 0 \\ \alpha & \text{for } z<0 \end{cases}$

Derivative of ELU $= \begin{cases} 1 & \text{for } z \ge 0 \\ \alpha \exp(z) & \text{for } z<0 \end{cases}$

2. (**6 pts**) Compute $\partial \mathbf{h}(\mathbf{x})/\partial \mathbf{w}_j$, where $\mathbf{w}_j$ is the element j of the weight vector for

(a) $\mathbf{h}(\mathbf{x}) = \text{relu}(\mathbf{w}^T\mathbf{x})$

(b) $\mathbf{h}(\mathbf{x}) = \text{leaky\_relu}(\mathbf{w}^T\mathbf{x})$

(c) $\mathbf{h}(\mathbf{x}) = \text{elu}(\mathbf{w}^T\mathbf{x})$

(you will need to use the chain rule; see class notes).

(For points where the gradient is undefined, the derivative at that point may be designated as either the left-hand derivative or the right-hand derivative.)

For simplicity, let $z = w^T x$.

For $h(x) = \text{relu}(z)$, $\frac{\partial h(x)}{\partial w_j} = \frac{\partial h(x)}{\partial z}\frac{\partial z}{\partial w_j}$ (using chain rule)

$= \begin{cases} 1 \cdot x_j & \text{if } z>0 \\ 0 \cdot x_j & \text{if } z \le 0 \end{cases}$ $(\because w^T x = w_1x_1 + w_2x_2 + \dots + w_jx_j \; \therefore \frac{\partial z}{\partial w_j} = x_j)$

$= \begin{cases} x_j & \text{if } z>0 \\ 0 & \text{if } z \le 0 \end{cases}$

For $h(x) = \text{leaky\_relu}(z)$, $\frac{\partial h(x)}{\partial w_j} = \frac{\partial h(x)}{\partial z}\frac{\partial z}{\partial w_j}$ (using chain rule)

$= \begin{cases} 1 \cdot x_j & \text{if } z \ge 0 \\ \alpha \cdot x_j & \text{if } z<0 \end{cases}$

$= \begin{cases} x_j & \text{if } z \ge 0 \\ \alpha x_j & \text{if } z<0 \end{cases}$

Q2.

for $h(x) = \text{elu}(z)$, $\frac{\partial h(x)}{\partial w_j} = \frac{\partial h(x)}{\partial z}\frac{\partial z}{\partial w_j}$

$= \begin{cases} 1 \cdot x_j & \text{if } z \ge 0 \\ \alpha \exp(z) \cdot x_j & \text{if } z<0 \end{cases}$

$= \begin{cases} x_j & \text{if } z \ge 0 \\ \alpha \exp(z) \cdot x_j & \text{if } z<0 \end{cases}$

3. (**4 pts**) We will now use ReLU gradient computed above to showcase a significant drawback of using ReLUs on neural networks. Recall that a ReLU **neuron** of a neural network is simply a logistic regression where we replace the sigmoid activation $\sigma()$ of the logistic by the ReLU activation. We have seen how the gradient of a weight vector $\mathbf{w}$ of a single logistic unit is computed by averaging the gradient computed over each training example $\{\mathbf{x}_i, y_i\}_{i=1}^n$. Clearly, if the resulting gradient of a parameter is zero, this parameter will not change in the next gradient step. Define the general mathematical condition relating $\{\mathbf{x}_i\}_{i=1}^n$ and $\mathbf{w}$ and b under which the gradient of a ReLU neuron $\mathbf{h}(\mathbf{x}) = \text{relu}(\mathbf{w}^T\mathbf{x} + \mathbf{b})$ is zero for all the inputs $\{\mathbf{x}_i\}_{i=1}^n$. When this happens, we say that the neuron has died.
Hint: Pay attention to the fact that the gradient of a parameter is the sum of the gradients over all training examples of the data.

The general mathematical condition is as follows:

$w^T x_i + b \leq 0, \forall i \in \{1, 2, \ldots, n\}$

This is because for all $\frac{\partial h(x)}{\partial w_j}$, $\frac{\partial h(x)}{\partial w_j} = \sum_{i=1}^{n} \frac{\partial h(x)}{\partial y_i} \cdot \frac{\partial y_i}{\partial w_j}$.

When $w^T x_i + b \leq 0$, $\frac{\partial h(x)}{\partial y_i} = 0$, thus, $\frac{\partial h(x)}{\partial w_j} = 0$, and this would mean that the neuron has died as gradient is zero for all inputs $\{x_i\}_{i=1}^n$.

4. (**2 pts**) Can Leaky Relu and ELU neurons die like ReLU neurons? Explain using answers in 2.1 and 2.2.

Leaky Relu and ELU neurons will not die like ReLU neurons. This is because when $w^T x + b \leq 0$, from part 2.1, the derivative of leaky ReLU $= \begin{cases} 1 & \text{if } z \geq 0 \\ \alpha & \text{if } z < 0 \end{cases}$ and derivative of ELU $= \begin{cases} 1 & \text{if } z \geq 0 \\ \alpha \exp(z) & \text{if } z < 0 \end{cases}$. Thus, the derivative is always non-zero for all values of $w^T x + b$. Furthermore, from part 2.2, we can observe that the gradient for leaky Relu and ELU are all non-zero for all inputs $\{x_i\}_{i=1}^n$. Hence, this means that both leaky ReLU and ELU neurons will not die like ReLU neurons.

Q3 (28 pts): Convolutional Neural Networks

In this problem, consider only the convolutional layer of a standard implementation of a CNN.

1. (**8 pts**) We are given image X and filter F below.

X =

0	0	0	0	0	0	0	0
0	8	1	7	-2	5	0	0
0	-2	1	4	6	3	3	0
0	5	1	1	3	-2	6	0
0	-1	5	0	3	-2	9	0
0	8	-2	-1	-1	4	9	0
0	-2	0	-2	9	-1	7	0
0	0	0	0	0	0	0	0

F =

2	-1	2	-1
1	3	1	3
2	-1	2	-1
1	3	1	3

Y =

a	b	c	d	e
f	g	h	i	j
k	l	m	n	o
p	q	r	s	t
u	v	w	x	y

(a) (**4 pts**) Let X be convolved with F using zero-padding of 1 and a stride of 1 to produce an output Y. What is value of e and v in the output Y?

$e = -2(1) + 5(3) + 0(1) + 0(3) + 6(2) + 3(-1) + 3(2) + 0(-1)$
$+ 3(1) + (-2)(3) + 6(1) + 0(3)$
$= \underline{\underline{31}}$

$v = -1(2) + 5(-1) + 0(2) + 3(-1) + 8(1) + (-2)(3) + (-1)(1)$
$+ (-1)(3) + (-2)(2) + 0(-1) + (-2)(2) + 9(-1)$
$= \underline{\underline{-29}}$

(b) (**4 pts**) Suppose you had an input feature map of size 10x12 and filter size 6x6, using a zero-padding of 2 and a stride of 2; what would be the resulting output size? Write your answer in terms of height × width.
Hint: Describe your reasoning for earning partial credit.

Input : $W_1 \times H_1 = 10 \times 12$ Using formula from lecture,

Filter: $F \times F = 6 \times 6$

Padding (P) : 2

Stride (S) : 2

Output : $W_2 \times H_2$

$$W_2 = \frac{W_1 - F + 2P}{S} + 1 = \frac{10 - 6 + 2(2)}{2} + 1 = 5$$

$$H_2 = \frac{H_1 - F + 2P}{S} + 1 = \frac{12 - 6 + 2(2)}{2} + 1 = 6$$

$\therefore$ Final answer $= H_2 \times W_2 = 6 \times 5$

2. (**20 pts**) These questions ask about the parameters of a convolutional layer and the layer's output (or intermediate outputs before the activation function).

(a) (**4 pts**) Consider a convolutional neural network with the following properties: the input image has a spatial dimension of 32 × 32 pixels, the convolutional layer uses a filter of size $n \times n$, the stride is 1 and padding is 2. The output has dimensions same as the input.
Select all that apply. Show your calculations.

☐ A. The value of n is 4
[✓ B. The value of n is 5]
☐ C. If the padding is doubled, the value of n will be 10.
[✓ D. If the padding is doubled, the value of n will be 9.]

$\because$ Output has dimensions same as input, applying the formula from lecture,

$$32 = \frac{32 - n + 2(2)}{1} + 1$$

$31 = 32 - n + 4$

$n = 36 - 31$
$= 5$

If padding is doubled :

$$32 = \frac{32 - n + 2(2 \times 2)}{1} + 1$$

$31 = 32 - n + 8$

$n = 40 - 31 = 9$

(b) (**6 pts**) Which of the following are hyperparameters of a convolutional layer? **Select all that apply. No need to clarify your selections.**

- ☑ stride size
- ☑ padding size
- ☐ image size
- ☑ filter size
- ☑ weights in the filter
- ☐ None of the above.

(c) (**2 pts**) Suppose for the convolutional layer, we are given colored images of size 12×12. Using one single 3×3 filter with a stride of 1 and zero-padding of 2, what is the number of parameters you are learning in this layer with and without bias?

with bias, Number of parameters $= (3 \times 3) + 1$
$= 10$
without bias, number of parameters $= 3 \times 3$
$= 9$

(d) (**2 pts**) Now suppose you are given a 40×40 black and white image (so your input is a $40 \times 40 \times 1$ tensor). Using 4×4 filters with a stride of 2 and no padding, what is the number of parameters you are learning in this layer? Consider that every filter adds a bias term.
Hint: Please detail your calculations to get partial credit.

Number of parameters $= (F^2 \times K) + K$ (for bias)
were $F =$ size of filter and $K =$ number of filters.
$\therefore$ Number of parameters $= (4 \times 4 \times 1) + 1$ (since $\exists$ a bias term for every filter)
$= 17$

(e) (**6 pts**) Suppose for the convolutional layer, we are given black and white images of size 28×28. We use four 6×6 filters with a stride of 2 and zero-padding of 2 in the first convolutional layer. The second convolutional layer has three 3×3 filters with a stride of 1 and no padding. What is the total number of parameters in these two layers? Consider that every filter adds a bias term.
Hint: Please detail your calculations to get partial credit.

Total number of parameters $=(F^2 \times K) + K$ (for bias terms)
, where F = size of filter and K = number of filters.

Number of parameters in first layer $= (6 \times 6 \times 4) + 4$ ($\because$ 4 filters, each filter has a bias term)
$= 148$

Number of parameters in second layer $= (3 \times 3 \times 3) + 3$ ($\because$ 3 filters, each filter has a bias term)
$= 30$

$\therefore$ Total number of parameters $= 148 + 30$
$= 178$ //

Programming Part (18+14+10=42)

General Instructions

For this assignment, please download the provided `hw3.zip` file on **Brightspace**.

For those questions that ask you to plot figures, do analysis, etc., please include your answers along with solutions to theoretical questions in the written PDF report. For other coding questions, you need to package all of your python files into a `.tar.gz` file and submit to the data server via `turnin` command. Detailed instructions of how to submit your code are in the last page.

Overview

In this assignment, you will be playing around with neural networks and using them to classify images. You will first implement an artificial neural network (ANN) from scratch using only `numpy`, and then you will use `PyTorch` to implement an ANN and a convolutional neural network (CNN).

Files

You will fill in functions in `numpy_ann.py`, `pytorch_ann.py`, and `pytorch_cnn.py`. After you finish, you will submit these, as well as any other files mentioned in the assignment, to Gradescope.

Aside from these three files, you will also find `utils.py` in the same folder. You do not need to modify it. `utils.py` contains some helper functions.

Packages

You will need the following packages for this assignment:

- `numpy`
- `scikit-learn`
- torch, installation detailed below
- torchvision, installation detailed below
- matplotlib

These packages should be installed if you have installed Anaconda. If you are not using Anaconda, you can install them using `pip` or `conda`. For example, you can run `pip install numpy` in the terminal to install numpy.

Note that for this assignment, you should not use any other packages. You may see some other packages in the skeleton code, but they are only used for testing and grading. If you use any other packages, you may lose points. If you are not sure whether you can use a package, please ask the course staff.

Evaluation

Unless otherwise specified, your code will be evaluated by an auto grader on Gradescope using the same environment as detailed below. You should make sure your code runs without errors in this environment. Otherwise, you may lose points. You can submit your code to Gradescope as many times as you want. We will only grade the latest submission.

There will be two types of test cases in the autograder: public (local) and hidden. Public test cases are visible to you, and you can see the results after you submit your code. You can also run the public test cases locally by running `python <filename>.py` in the same folder as your code. Passing all public test cases **does not** guarantee you will get full credit for the assignment.

Hidden test cases, however, are not visible to you, and you will not be able to see the results until your grade is published. The final score you get for this assignment is the sum of the scores of all public and hidden test cases.

Getting Help

If you have any questions about this assignment, please contact the course staff for help, preferably during office hours. You can also post your questions on the course forum. However, please do not post any code publicly. When asking questions, you should describe your problem in words and post the relevant code snippet. Please avoid showing a screenshot of your code to TAs and ask "why my code is not working".

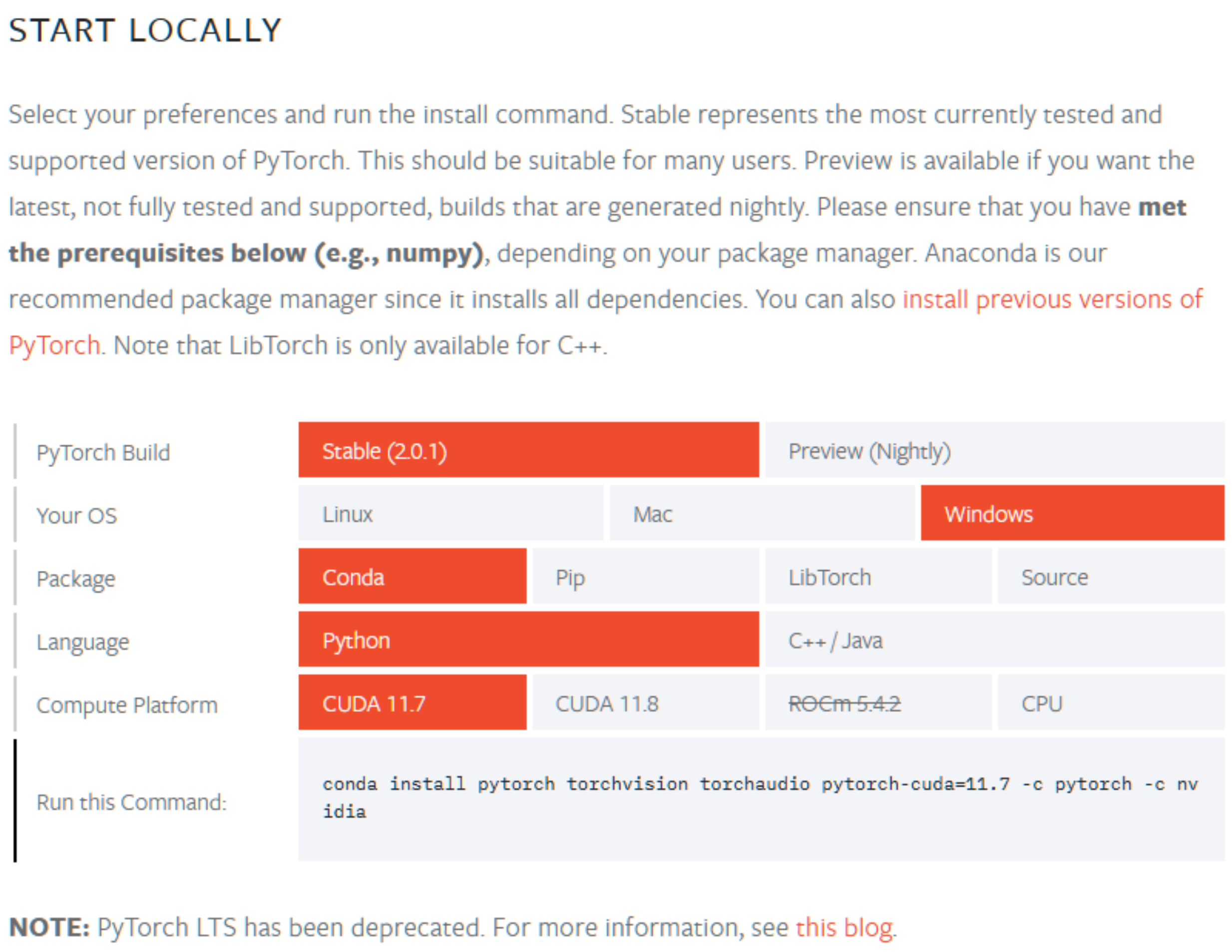
Setting Up PyTorch

Installing PyTorch

PyTorch is a popular deep-learning framework that we will use in this assignment. You can install PyTorch by following the instructions on the PyTorch website.

Once you open the website, you will see a few options. You should select the following options:

- Pytorch Build: `Stable`
- Your OS: (the OS you are using)
- Package: We recommend `Conda`, but you can also use `Pip` if you prefer
- Language: `Python`
- Compute Platform
  - If you have selected `Mac` as your OS, the only option is `Default`.



START LOCALLY

Select your preferences and run the install command. Stable represents the most currently tested and supported version of PyTorch. This should be suitable for many users. Preview is available if you want the latest, not fully tested and supported, builds that are generated nightly. Please ensure that you have **met the prerequisites below (e.g., numpy)**, depending on your package manager. Anaconda is our recommended package manager since it installs all dependencies. You can also install previous versions of PyTorch. Note that LibTorch is only available for C++.

PyTorch Build	[✓ Stable (2.0.1)]	Preview (Nightly)		
Your OS	Linux	Mac	[✓ Windows]	
Package	[✓ Conda]	Pip	LibTorch	Source
Language	[✓ Python]	C++ / Java		
Compute Platform	[✓ CUDA 11.7]	CUDA 11.8	~~ROCm 5.4.2~~	CPU
Run this Command:	`conda install pytorch torchvision torchaudio pytorch-cuda=11.7 -c pytorch -c nvidia`			

NOTE: PyTorch LTS has been deprecated. For more information, see this blog.

Figure 1: Pytorch Installation

- If you have an NVIDIA GPU, you should select the one that starts with `CUDA`.
- Otherwise, you should select `CPU`.

After you have selected the options, you will see a command that you can run in the terminal to install PyTorch. This command will look something like `conda install pytorch torchvision torchaudio pytorch-cuda=11.7 -c pytorch -c nvidia` if you have selected `Conda` as the package and `CUDA 11.7` as the computing platform. As you can see, this command will install PyTorch, torchvision, and torchaudio. We will only use PyTorch and torchvision in this assignment.

You should run this command in the terminal and wait for the installation to finish, which may take a few minutes. After the installation is finished, you can run `python` in the terminal and run `import torch` to check if PyTorch is installed correctly.

Checking GPU Support

Note: A GPU is not required for this assignment. You can still complete this assignment without a GPU in a reasonable amount of time. You can skip this section if you do not have a GPU.

Deep learning models are usually trained on GPUs because they are much faster than CPUs due to their parallel architecture. If you have an NVIDIA GPU or an Apple Silicon Mac (M1, M2, and their variants) with a Neural Engine, you can use it to train your models. If you do not have a GPU, you can still train your models on your CPU, but it will be much slower.

After you have installed PyTorch, you can check if your GPU is supported by running the following code in the terminal:

```python
import torch

# If you have an NVIDIA GPU, this should print True
print(torch.cuda.is_available())

# If you have an Apple Silicon Mac, this should print True
print(torch.backends.mps.is_available())
```

1 (18 pts) Artificial Neural Network with Numpy

In this part, you will implement an artificial neural network (ANN) from scratch using only `numpy`. You will then use the ANN to classify images from the Fashion MNIST dataset.

You will write your code in `numpy_ann.py`.

1.1 (3 pts) Numpy ANN - Constructor

Under `class NumpyANN`, fill in the `__init__` method so that it initializes four instance variables:

- `self.input_size`: the number of features in the input
- `self.hidden_size`: the number of neurons in the hidden layer
- `self.output_size`: the number of classes in the output
- `self.lr`: the learning rate

You should also set the random see using `np.random.seed()`. Then use `np.random.randn` to initialize the weights `W1` and `W2`. `W1` should be of size `(self.input_size, self.hidden_size)`, and `W2` should be of size `(self.hidden_size, self.output_size)`. You should also initialize the biases `b1` and `b2` using `np.zeros`. `b1` should be of size `(1, self.hidden_size)`, and `b2` should be of size `(1, self.output_size)`.

You will see that a coefficient of 0.01 is used to scale the weights. It is used to make sure the weights are initialized to small values, but not zero, so that the gradients are not too large or too small.

1.2 (1 pts) Numpy ANN - ReLU

Under `class NumpyANN`, fill in the `_relu` method.

Recall that the ReLU of the input is 0 if the input is less than 0, and the input itself otherwise. Mathematically, this can be written as:

$$\text{ReLU}(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$$

1.3 (1 pts) Numpy ANN - ReLU Derivative

Under `class NumpyANN`, fill in the `_relu_grad` method so that it applies the derivative of the ReLU function to the input. Recall that the derivative of the ReLU function is 0 if the input is less than 0, and 1 otherwise. Mathematically, this can be written as:

$$\frac{\partial \text{ReLU}(x)}{\partial x} = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{otherwise} \end{cases}$$

1.4 (1 pts) Numpy ANN - Cross Entropy Loss

Under `class NumpyANN`, fill in the `_cross_entropy` method so that it takes in the true label `y_true` and the predicted label `y_pred` and returns the cross entropy loss between them. The cross-entropy loss is defined as:

$$\text{CrossEntropyLoss}(y_{true}, y_{pred}) = -\sum_{i=1}^{n} y_{true,i} \log(y_{pred,i})$$

To keep the magnitude of the loss reasonable, we will divide the loss by the number of classes. Mathematically, this can be written as:

$$\text{CrossEntropyLoss}(y_{true}, y_{pred}) = -\frac{1}{n}\sum_{i=1}^{n} y_{true,i} \log(y_{pred,i})$$

where n is the number of classes.

1.5 (2 pts) Numpy ANN - Forward

Under `class NumpyANN`, fill in the `_forward` method so that it takes in an input `x` and a boolean `training` and returns the output of the neural network. Since we are using a neural network with one hidden layer, the output of the neural network is the output of the hidden layer passed through a sigmoid activation function. Mathematically, the forward pass can be written as:

$$
\begin{aligned}
z_1 &= xW_1 + b_1 &(5)\\
a_1 &= \text{Activation}(z_1) &(6)\\
z_2 &= a_1 W_2 + b_2 &(7)\\
a_2 &= \text{Activation}(z_2) &(8)
\end{aligned}
$$

where Activation is ReLU for the hidden layer and softmax for the output layer.

Note that the forward pass is different during training and testing. During training, we need to keep track of the outputs of the hidden layer, so we can use them in the backward pass. Therefore, when `training=True`, the output of those layers should be stored in `self` (we already did this for you).

1.6 (3 pts) Numpy ANN - Backward

Under `class NumpyANN`, fill in the `_backward` method so that it takes in an input `X` and the true label `y_true` and performs the backward propagation.

The mathematical formulas for the backward pass of the second layer are:

$$
\begin{aligned}
\frac{\partial \text{Loss}}{\partial z_2} &= (a_2 - y_{true}) &(9)\\
\frac{\partial \text{Loss}}{\partial W_2} &= \frac{\partial \text{Loss}}{\partial z_2}\frac{\partial z_2}{\partial W_2} = a_1^\top \frac{\partial \text{Loss}}{\partial z_2} &(10)\\
\frac{\partial \text{Loss}}{\partial b_2} &= \frac{\partial \text{Loss}}{\partial z_2}\frac{\partial z_2}{\partial b_2} = \frac{\partial \text{Loss}}{\partial z_2} &(11)
\end{aligned}
$$

We then divide the weight gradients and bias gradients by the number of samples to get the average gradients.

Similarly, the mathematical formulas for the backward pass of the first layer are:

$$
\begin{aligned}
\frac{\partial \text{Loss}}{\partial z_1} &= \frac{\partial \text{Loss}}{\partial z_2}\frac{\partial z_2}{\partial a_1}\frac{\partial a_1}{\partial z_1} = \frac{\partial \text{Loss}}{\partial z_2} W_2^\top \frac{\partial a_1}{\partial z_1} &(12)\\
\frac{\partial \text{Loss}}{\partial W_1} &= \frac{\partial \text{Loss}}{\partial z_1}\frac{\partial z_1}{\partial W_1} = X^\top \frac{\partial \text{Loss}}{\partial z_1} &(13)\\
\frac{\partial \text{Loss}}{\partial b_1} &= \frac{\partial \text{Loss}}{\partial z_1}\frac{\partial z_1}{\partial b_1} = \frac{\partial \text{Loss}}{\partial z_1} &(14)
\end{aligned}
$$

Again, we then divide the weight gradients and bias gradients by the number of samples to get the average gradients.

After you have computed the gradients, you should update the weights and biases using gradient descent. Recall that the update rule for gradient descent is:

$$\theta = \theta - \alpha \frac{\partial \text{Loss}}{\partial \theta}$$

where θ is a parameter, α is the learning rate, and $\frac{\partial \text{Loss}}{\partial \theta}$ is the gradient of the loss with respect to θ.

1.7 (2 pts) Numpy ANN - Train

Under `class NumpyANN`, fill in the `train` method so that it takes in an input `X_train`, a true label `y_train`, a validation set `X_val` and `y_val`, the number of `epochs` epochs, and the batch size `batch_size`, and trains the neural network using the training set and validates it using the validation set.

We will use mini-batch gradient descent to train the neural network for faster convergence. In each epoch, we will split the training set into mini-batches of size `batch_size`. Then we will train the neural network using each mini-batch. After each epoch, we will compute the loss and accuracy of the neural network on the validation set.

You should use the `_forward` and `_backward` methods you implemented above to train the neural network. You should also use the `_cross_entropy` method to compute the loss.

1.8 (1 pts) Numpy ANN - Predict

Under `class NumpyANN`, fill in the `predict` method so that it takes in an input `X` and returns the predicted probabilities of the neural network. Recall that the output of the neural network is the output of a forward pass without saving the outputs of the hidden layer (i.e., `training=False`). You should use the `_forward` method you implemented above to compute the output of the neural network.

1.9 (2 pts) Numpy ANN - Accuracy

Under `class NumpyANN`, fill in the `accuracy` method so that it takes in the data `X` and the true label `y_true` and returns the accuracy of the neural network on the data. You should use the `predict` method you implemented above to compute the predicted probabilities of the neural network. Then you should use `np.argmax` to get the predicted labels. Finally, calculate the accuracy.

1.10 (2 pts) Numpy ANN - Overall Test

Once you have finished implementing the above methods, you can run `python numpy_ann.py` in the terminal to test your implementation. Your model will run on the Fashion MNIST dataset, which is a classification dataset containing 10 classes of images. You should get an accuracy of at least 0.77 on the test set.

2 (14 pts) PyTorch ANN

In this part, you will implement an artificial neural network (ANN) using `PyTorch`. You will then use the ANN to classify images from the Fashion MNIST dataset.

Building a neural network using `PyTorch` is very similar to building a neural network using `numpy`. The main difference is that you do not need to implement the forward and backward passes yourself. Instead, you can use the `torch.nn` module to build the neural network and the `torch.optim` module to perform gradient descent. In most cases, you only need to specify the modules (layers, loss functions, optimizers, etc) and hyperparameters of the neural network. `PyTorch` will take care of the rest.

Before you start, you should make sure you have installed PyTorch correctly. You can run `python -c "import torch; print(torch.__version__)"` in the terminal to check the version of PyTorch you have installed. Version >= 2.0.0 is highly recommended and will be used in the autograder.

You will write your code in `pytorch_ann.py`. You will see `torch.Tensor` objects in the skeleton code. `torch.Tensor` is the basic data structure in `PyTorch`. It is similar to `numpy.ndarray`, but it can also be used to represent a tensor on a GPU. For now, you can think of it as a `numpy.ndarray`.

2.1 (3 pts) PyTorch ANN - Constructor

Under `class PyTorchANN`, fill in the `__init__` method so that it initializes three instance variables:

- `self.input_size`: the number of features in the input
- `self.hidden_size`: the number of neurons in the hidden layer
- `self.output_size`: the number of classes in the output
- `self.seed`: the random seed

Then, use `torch.manual_seed` to set the random seed. Finally, inside `nn.Sequential`, use `nn.Linear` to initialize the layers of the neural network. You should use `self.input_size` and `self.hidden_size` as the input size and output size of the first layer and `self.hidden_size` and `self.output_size` as the input size and output size of the second layer. You should also use `nn.ReLU` as the activation function of the first layer. You do not need to specify the activation function of the second layer because we will use `nn.CrossEntropyLoss` as the loss function, which already includes a softmax activation function in it.

2.2 (1 pts) PyTorch ANN - Forward

Under `class PyTorchANN`, fill in the `forward` method so that it takes in an input `X` and returns the output of the neural network. To run the input through the neural network, simply call `self.model(X)`, where `self.model` is the neural network you initialized in the constructor.

While `self.model` is a neural network, it can also be treated as a function. When you call `self.model(X)`, it will run the input `X` through the neural network and return the output. This is the same as calling `self.model.forward(X)`, because in Python, `self.model(X)` is equivalent to `self.model.__call__(X)`, which is equivalent to `self.model.forward(X)` for `nn.Module` objects.

We do not need a backward pass here because we will use `torch.optim` optimizer to perform gradient descent.

2.3 (5 pts) PyTorch ANN - Train

Under `class PyTorchANN`, fill in the `train` method so that it takes in an input `X_train`, a true label `y_train`, a validation set `X_val` and `y_val`, the number of `epochs` epochs, the batch size `batch_size`, and the learning rate. It trains the neural network using the training set and validates it using the validation set.

First, you need to define the loss function and optimizer. You should use `nn.CrossEntropyLoss` as the loss function and `torch.optim.SGD` (stochastic gradient descent) as the optimizer. You should use the learning rate you passed in as the learning rate of the optimizer and use `self.model.parameters()` as the parameters of the optimizer. This will tell the optimizer to update the parameters of the neural network.

Then, you can train the neural network using the mini-batches. For each mini-batch, you should first set the gradients to zero using `optimizer.zero_grad()`. Then you should run the mini-batch through the neural network using `self.forward` and compute the loss using the loss function you defined above. After that, you should call `loss.backward()` to perform the backward propagation. Finally, you should call `optimizer.step()` to update the parameters of the neural network using gradient descent.

2.4 (1 pts) PyTorch ANN - Predict

Under `class PyTorchANN`, fill in the `predict` method so that it takes in an input `X` and returns the predicted labels of the neural network. To run the input through the neural network, simply call `self.forward(X)`. Then you can use `torch.argmax` to get the predicted labels.

2.5 (1 pts) PyTorch ANN - Visualize Learning Curve

Under `class PyTorchANN`, fill in the `plot_train_val_metrics` method so that it creates a side-by-side plot of the training and validation loss and accuracy curves. You should use the predefined `ax` to plot the curves. Include the plot generated by this method in your theoretical write-up.

2.6 (2 pts) PyTorch ANN - Overall Test

Once you have finished implementing the above methods, you can run `python pytorch_ann.py` in the terminal to test your implementation. Your model will run on the Fashion MNIST dataset. You should get an accuracy of at least 0.8 on the test set.

2.7 (1 pt) PyTorch ANN - Analysis of Learning Curve

Based on the learning curve you generated above and the program output, under `class PyTorchANN`, fill in the `report_my_judgement` method so that it returns a string that describes the performance of the neural network by the end of training. Return `underfitting` if the neural network is underfitting, `overfitting` if the neural network is overfitting, and `generalizing` if the neural network is generalizing.

3 (10 pts) PyTorch CNN

In this part, you will implement a convolutional neural network (CNN) using `PyTorch`. You will then use the CNN to classify images from the Fashion MNIST dataset.

You will write your code in `pytorch_cnn.py`.

3.1 (6 pts) PyTorch CNN - Constructor

You should use `nn.Sequential` and use `nn.Conv2d`, `nn.ReLU`, `nn.MaxPool2d`, `nn.BatchNorm2d`, `nn.Dropout`, `nn.Flatten`, and `nn.Linear` to define the layers. You do not need to specify the activation function of the last layer because we will use `nn.CrossEntropyLoss` as the loss function, which already includes a softmax activation function in it.

You might be wondering what the parameters of `nn.Conv2d` mean. `nn.Conv2d` is a convolutional layer that takes in a 4D tensor as input and outputs a 4D tensor. The first dimension of the input tensor is the batch size, the second dimension is the number of channels, and the last two dimensions are the height and width of the input. The first dimension of the output tensor is the batch size, the second dimension is the number of channels, and the last two dimensions are the height and width of the output. The parameters of `nn.Conv2d` are:

- `in_channels`: the number of channels in the input
- `out_channels`: the number of channels in the output
- `kernel_size`: the size of the convolutional kernel
- `stride`: the stride of the convolution
- `padding`: the padding of the convolution

`stride` is the number of pixels the kernel moves each time. `padding` is the number of pixels added to each side of the input. For example, if the input is a 4x4 image and the kernel size is 3x3, then the output will be a 2x2 image if `stride=1` and `padding=0`, a 3x3 image if stride=1 and padding=1, and a 4x4 image if `stride=2` and `padding=1`. The basic formula for calculating the output size is:

$$\text{Output Size} = \frac{\text{Input Size} - \text{Kernel Size} + 2 \times \text{Padding}}{\text{Stride}} + 1$$

`nn.MaxPool2d` is a max pooling layer that is used to reduce the size of the input. You can think of it as a layer that takes the maximum value of each `kernel_size x kernel_size` block of the input.

`nn.BatchNorm2d` is a batch normalization layer that is used to normalize the input. You can think of it as a layer that normalizes the input to have mean 0 and standard deviation 1.

`nn.Dropout` is a dropout layer that is used to prevent overfitting. You can think of it as a layer that randomly sets some of the input to 0 or "kills" some of the neurons.

Now, under `class PyTorchCNN`, fill in the model definition so that it defines a convolutional neural network with the following architecture:

- (Block 1)
  - A convolutional layer with `in_channels=1`, `out_channels=32`, `kernel_size=3`, `stride=1`, and `padding=1`
  - A ReLU activation function
  - A max pooling layer with `kernel_size=2`
  - A BatchNorm2D layer with `num_features=32`
  - A Dropout layer with `p=0.3`
- (Block 2)
  - A convolutional layer with `in_channels=32`, `out_channels=64`, `kernel_size=3`, `stride=1`, and `padding=1`
  - A ReLU activation function
  - A max pooling layer with `kernel_size=2`
  - A BatchNorm2D layer with `num_features=64`
  - A Dropout layer with `p=0.3`
- (Block 3)
  - A convolutional layer with `in_channels=64`, `out_channels=128`, `kernel_size=3`, `stride=1`, and `padding=1`
  - A ReLU activation function
  - A max pooling layer with `kernel_size=2`
  - A BatchNorm2D layer with `num_features=128`
  - A Dropout layer with `p=0.3`
- (Block 4)
  - A convolutional layer with `in_channels=128`, `out_channels=256`, `kernel_size=3`, `stride=1`, and `padding=1`
  - A ReLU activation function
  - A max pooling layer with `kernel_size=2`
  - A BatchNorm2D layer with `num_features=256`
  - A Dropout layer with `p=0.3`

- (Block 5)
  - A Flatten layer
  - A fully connected layer with `in_features=256`, `out_features=128`
  - A ReLU activation function
  - A fully connected layer with `in_features=128`, `out_features=self.num_classes`

3.2 (2 pts) PyTorch CNN - Overall Test

Now, you can run python `pytorch_cnn.py` in the terminal to test your implementation. Your model will run on the Fashion MNIST dataset. You should get an accuracy of at least 0.85 on the test set.

3.3 (2 pts) PyTorch CNN - Analysis of Learning Curve

Once you have run the program, you will see a plot of the training and validation loss and accuracy curves. Based on the graph and the output of the program, fill in the `report_my_judgement` method so that it returns a string that describes the performance of the neural network by the end of training. Return `underfitting` if the neural network is underfitting, `overfitting` if the neural network is overfitting, and `generalizing` if the neural network is generalizing.

Don't forget to include the plot generated by this method in your theoretical write-up.

Submission

You will submit the following files to Gradescope:

- `numpy_ann.py`
- `pytorch_ann.py`
- `pytorch_cnn.py`
- `pytorch_ann_train_val_metrics.png`
- `pytorch_cnn_train_val_metrics.png`