Deep learning Assignment 2

Tianzheng Hu (276027)

November 2023

1 Answers

question 1 For a function: $f(X, Y) = \frac{X}{Y}$ where X and Y are two matrices and the division is element-wise. Let $Z = \frac{X}{Y}$, we aim to find $\triangledown X$ and $\triangledown Y$, which derive to the gradient for X and for Y. $\triangledown Z$ denotes the backward of loo function $\frac{\partial L}{\partial Z}$. In order to find the gradient for the matrices, try to find the gradient of a single element in the matrix.

Derivation for $\triangledown X$**:**
Let $z_{ij} = \frac{X_{ij}}{Y_{ij}}$, where i and j denote the position of the single element in the matrices. Using chain rule:

$$\triangledown x_{ij} = \sum_{kl} \triangledown z_{kl} \frac{\partial z_{kl}}{\partial x_{ij}} = \sum_{kl} \triangledown z_{kl} \frac{x_{kl}}{y_{kl}} \frac{\partial}{\partial x_{ij}}$$

This is non-zero only when i = k, j = l. So

$$\frac{\partial x_{kl}}{\partial x_{ij}} \frac{1}{y_{kl}} = \begin{cases} \frac{1}{y_{kl}} & \text{if } i = k, j = l, \\ 0 & \text{if } otherwise. \end{cases}$$

Extending this to whole X matrix,

$$\triangledown X = \triangledown Z \odot \frac{1}{Y}$$

Derivation for $\triangledown Y$:

$$\triangledown y_{ij} = \sum_{kl} \triangledown z_{kl} \frac{\partial z_{kl}}{\partial y_{ij}} = \sum_{kl} \triangledown z_{kl} \frac{x_{kl}}{y_{kl}} \frac{\partial}{\partial y_{ij}}$$

So

$$\frac{x_{kl}}{y_{kl}} \frac{\partial}{\partial y_{ij}} = \begin{cases} -\frac{x_{ij}}{(y_{ij})^2} & \text{if } i = k, j = l, \\ 0 & \text{if } otherwise. \end{cases}$$

Extending this to whole Y matrix,

$$\triangledown Y = -\triangledown Z \odot \frac{X}{Y^2}$$

question 2 The input of tensor-to-tensor function F(X) is tensor X, and the output is denoted as Y after applying scalar-to-scalar function f element-wisely. Input the gradient $\triangledown Y$, and the backward of function F(X) should give a output $\triangledown X$, which can represent as $\frac{\partial l}{\partial X}$, also $\frac{\partial l}{\partial Y}\frac{\partial Y}{\partial X}$.So:

$$\triangledown Y = \frac{\partial l}{\partial Y}, \triangledown X = \frac{\partial l}{\partial X} = \frac{\partial l}{\partial Y}\frac{\partial Y}{\partial X}$$

Since the function F(x) applies f(x) on each single element of X, every single element of Y can be denoted as $y_{ij} = f(x_{ij})$. So the gradient of each element in X:

$$\triangledown x_{ij} = \frac{\partial l}{\partial x_{ij}} = \frac{\partial l}{\partial Y}\frac{\partial Y}{\partial x_{ij}} = \sum_k \sum_l \frac{\partial l}{\partial y_{kl}}\frac{\partial y_{kl}}{\partial x_{ij}} = \sum_k \sum_l \frac{\partial l}{\partial y_{kl}}\frac{\partial f(x_{kl})}{\partial x_{ij}}$$

It is non-zero only when i=k and k=l, so:

$$\triangledown x_{ij} = \begin{cases} \frac{\partial l}{\partial y_{ij}} f'(x_{ij}) & \text{if } i = k, j = l, \\ 0 & \text{if } otherwise. \end{cases}$$

Extending the element to the whole tensor X, $\triangledown X = \frac{\partial l}{\partial Y} F'(X) = \triangledown Y \odot F'(X)$. The notation $\odot$ represents element-wise multiplication. $F'(X)$ denotes the tensor-to-tensor function that applied scalar-to-scalar function $f'(x)$ on every element of it.

question 3 The layer output Y(dimension n-by-m) should be operated by a matrix multiplied by input X and weight W with the shape of dimensions n-by-f and dimensions f-by-m individually. So F(X) = XW = Y.

A single element in Y denoted as y_{ij} is the result of multiplying column i of W and row j of X:

$$y_{ij} = \sum_{r=1}^{n} x_{ir} w_{rj} = x_{i1} w_{1j} + x_{i2} w_{2j} + \cdots + x_{in} w_{nj}$$

Derivation of X: Same with before, calculate the derivation of a single element:

$$\triangledown x_{ir} = \frac{\partial l}{\partial x_{ir}} = \frac{\partial l}{\partial y_{kl}}\frac{\partial y_{kl}}{\partial x_{ir}} = \sum_k \sum_l \triangledown y_{kl} \frac{\partial \sum_r x_{kl} w_{rl}}{\partial x_{ir}}$$

whereby

$$\frac{\partial \sum_r x_{kl} w_{rl}}{\partial x_{ir}} = \begin{cases} w_{rl} & \text{if } i = k, \\ 0 & \text{if } otherwise. \end{cases}$$

So:

$$\triangledown x_{ir} = \sum_l \triangledown y_{kl} w_{rl} = \triangledown y_k w_r^T$$

Extending to the whole matrix:

$$\triangledown X = \triangledown Y W^T$$

Derivation of W: First, calculate a single element of W:

$$\nabla w_{rj} = \frac{\partial l}{\partial y_{kl}} \frac{\partial y_{kl}}{\partial w_{rj}} = \sum_k \sum_l \nabla y_{kl} \frac{\partial \sum_r x_{kr} w_{rl}}{\partial w_{rj}}$$

whereby

$$\frac{\partial \sum_r x_{kr} w_{rl}}{\partial w_{rj}} = \begin{cases} x_{kr} & \text{if } j = l, \\ 0 & \text{if } otherwise. \end{cases}$$

So:

$$\nabla w_{ir} = \sum_k \nabla y_{kl} x_{kr} = x_r^T \nabla y_l$$

Extending to the whole matrix:

$$\nabla W = X^T \nabla Y$$

question 4 In the original function f(x) = Y, where x is a vector and Y is a matrix consisting of 16 columns that are all equal to x.

$$Y = [x_1, x_2 ..., x_n]$$

Each column of Y is a repetition of the vector x. let's consider Y as a matrix with elements

$$Y_{ij} = x_i$$

where i represents the index of the elements in x, and j represents the column index in Y. The goal is to find the partial derivatives of each element of Y with respect to each element of x.

$$\nabla x_i = \sum_j \nabla y_{ij} \frac{\partial y_{ij}}{\partial x_i}$$

It can be deduced $y_{ij} = x_i$, because y is the duplication of x columns in 16 times. So:

$$\nabla x_i = \sum_j \nabla y_{ij} \frac{\partial x_i}{\partial x_i} = \sum_j \nabla y_{ij}$$

So, in matrix form:

$$\nabla X = \sum_j \nabla Y_j = \nabla Y$$

question 5 First, create two TensorNodes a and b, then use the plus operator to generate the third TensorNode c.

```python
import vugrad as vg
import numpy as np
a = vg.TensorNode(np.random.randn(2, 2))
b = vg.TensorNode(np.random.randn(2, 2))
c = a + b
```

1) c.value is a 2-D array with shape (2,2), containing the additional value of TensorNode a and TensorNode b.

2) c.source refers to an OpNode, which produced this TensorNode c. At the same time, OpNode is a Computation graph.

3) c.source.inputs[0].value refers to TensorNode a's value, which is the first input of the addition operation indicated by index 0. For the same reason, index 1 would represent TensorNode b.

4) a.grad refers to a tensor containing the gradient of TensorNode a. It will be updated by every backward propagation by applying the Multivariate Chain Rule. The current value is array([[0., 0.],[0., 0.]]).

question 6 Understanding how class TensorNode and OpNode are implemented.

1) An OpNode is defined by its inputs, its outputs, and the specific operation it represents (i.e. summation, multiplication). What kind of object defines this operation?

Answer: The operation is defined by Object Op, where the real operations are implemented.

2) In the computation graph of question 5, we ultimately added one numpy array to another (albeit wrapped in a lot of other code). In which line of code is the actual addition performed?

Answer: The actual operation is in line 324 in the original file code.py, class Add(Op), which is inherited from the Op class. The forward function, the code: ”return a+b”.

```
class Add(Op):
    """
    Op for element-wise matrix addition.
    """
    @staticmethod
    def forward(context, a, b):
        assert a.shape == b.shape, f'Arrays not the same sizes' \
                                   f' ({a.shape} {b.shape}).'
        return a + b
```

3) When an OpNode is created, its inputs are immediately set, together with a reference to the op that is being computed. The pointer to the output node(s) is left None at first. Why is this? In which line is the OpNode connected to the output nodes?

Answer: When an OpNode is created, the pointer to its output node is set to None as the initial value, because the shape of the output depends on the operation and inputs, which cannot be predicted till the actual operation is implemented. The OpNode and the output nodes are connected in the do_forward function of class Op. The specific lines are 246 to 249. First create a new OpNode object with args including cls, context, and inputs, then the outputs of this OpNode are calculated by a for loop.

```
class Op:
    @classmethod
      def do_forward(cls, *inputs, **kwargs):
          ...
          opnode = OpNode(cls, context, inputs)
          outputs = [TensorNode(value=output, source=opnode) for
                      output in outputs_raw]
          opnode.outputs = outputs
          ...
```

question 7 When we have a complete computation graph, resulting in a TensorNode called loss, containing a single scalar value, we start backpropagation by calling "loss.backward()". Ultimately, this leads to the backward() functions of the relevant Ops being called, which do the actual computation. In which line of the code does this happen?

Answer: When the backward function of a TensorNode object is called, it is the last node of the computation graph. Otherwise, the conditional judgment "if" wouldn't go to lines 94 to 97, the code "self.source.backward()". The backward function of the TensorNode object source will be called along with it, which is a backward function in class OpNode.

```
class TensorNode:
    ...
    def backward(self, start=True):
        ...
        # If we've been visited by all parents, move down the tree
        if self.visits == self.numparents or start:
            if self.source is not None:
                    self.source.backward()
        ...
```

The backward function in OpNode then triggers the backward function in class Op(code lines 159). Before that, the gradients of each input from TensorNode to OpNode are calculated to give the output(code lines 156).

```
class OpNode:
    ...
     def backward(self):
        ...
        # extract the gradients over the outputs (these have been
        #computed already)
        goutputs_raw = [output.grad for output in self.outputs]
        # compute the gradients over the inputs
        ginputs_raw = self.op.backward(self.context, *goutputs_raw)
        ...
```

question 8 I chose the Normalize function from Vugrad. In the original code, the forward function of the Normalize process implements row-wise:

$$Y = \frac{x_{ij}}{\sum_j x_{ij}}$$

where i, j denote row and column index separately. First, we can calculate the derivation of a single element:

$$\nabla x_{ij} = \frac{\partial l}{\partial y_{kl}} \frac{\partial y_{kl}}{\partial x_{ij}} = \sum_{kl} \nabla y_{kl} \frac{\partial y_{kl}}{\partial x_{ij}} = \sum_{kl} \nabla y_{kl} \frac{\partial}{\partial x_{ij}} \frac{x_{kl}}{\sum_l x_{kl}}$$

whereby

$$\frac{\partial}{\partial x_{ij}} \frac{x_{kl}}{\sum_l x_{kl}} = \begin{cases} 0 & \text{if } i \neq k, j \neq l \\ -\frac{x_{kl}}{(\sum_l x_{kl})^2} & \text{if } i = k, j \neq l \\ \frac{\sum_l x_{kl} - x_{kl}}{(\sum_l x_{kl})^2} & \text{if } i = k, j = l. \end{cases}$$

Thus, sum the second and third situations up:

$$\nabla x_{ij} = -\sum_{j \neq l} \nabla y_{il} \frac{x_{il}}{(\sum_l x_{il})^2} + \nabla y_{ij} \frac{\sum_j x_{ij} - x_{ij}}{(\sum_j x_{ij})^2}$$

$$\nabla x_{ij} = (-\sum_{j \neq l} \nabla y_{il} \frac{x_{il}}{(\sum_l x_{il})^2} - \nabla y_{ij} \frac{x_{ij}}{(\sum_j x_{ij})^2}) + \nabla y_{ij} \frac{1}{(\sum_j x_{ij})^2}$$

$$\nabla x_{ij} = -\sum_j \nabla y_{ij} \frac{x_{ij}}{(\sum_j x_{ij})^2} + \nabla y_{ij} \frac{1}{\sum_j x_{ij}}$$

Extending it to the whole matrix:

$$\nabla X = \frac{\nabla Y}{\sum_j X} - \sum_j \nabla Y \odot \frac{X}{(\sum_j X)^2}$$

Since $\sum_j X$ has the same size as X, also the identical value in X. So the backward of the Normalize function matches what is implemented in code. In code, ∇Y denotes as **go**, $\sum_j X$ denotes **sumd** and X is **X**.

```
class Normalize(Op):
    ...
    @staticmethod
    def backward(context, go):
        x, sumd = context['x'], context['sumd']
        return (go / sumd) - ((go * x)/(sumd * sumd)).sum(axis=1, keepdims=True)
```

question 9 In order to explore the effect of different activation functions on neural network learning, the Sigmoid function is replaced with Relu. The formula for the Relu function is $Relu(x) = max(x, 0)$. Add a rule class to the file ops.py.

```
class ReLU(Op):
    @staticmethod
    def forward(context, input):
        context['relu_input'] = input
        # if the input is greater than 0, keep input
        # otherwise, replace it as 0
        return np.where(input > 0, input, 0)
    @staticmethod
    def backward(context, goutput):
        input = context['relu_input']
        return goutput * np.where(input > 0, 1, 0)
```

Add a Relu function in the functions.py. In this function, $ReLU.do_forward(x)$ can call the forward function of class ReLU in ops.py.

```
def relu(x):
    return ReLU.do_forward(x)
```

After running two activation functions, the Relu activation function leads to a higher loss and faster improvement in accuracy in the beginning(see Fig. 1).

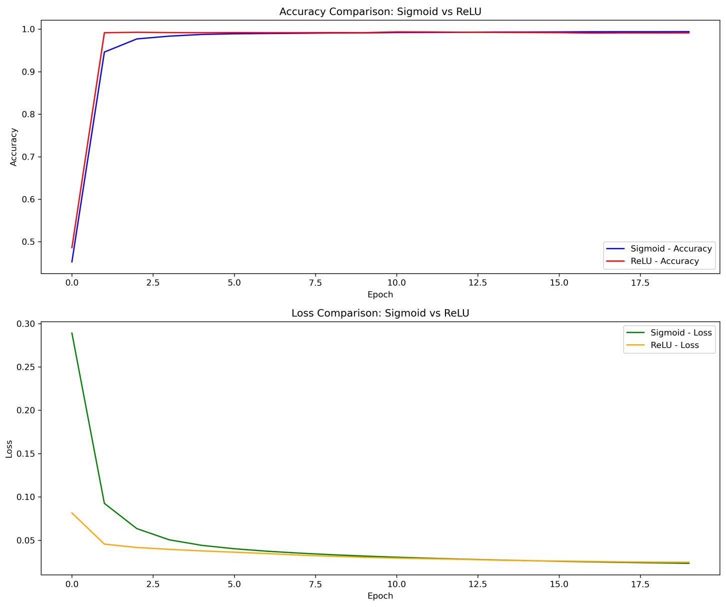

Figure 1: loss and accuracy of sigmoid and relu

question 10 In order to change the architecture of the network, tried to add a new hidden *layer3* and a boolean parameter *residual* in class MLP.

```
def __init__(self, input_size, output_size, residual, hidden_mult=4):
    super().__init__()
    hidden_size = hidden_mult * input_size
    self.layer1 = vg.Linear(input_size, hidden_size)
    self.layer2 = vg.Linear(hidden_size, hidden_size)
    self.layer3 = vg.Linear(hidden_size, output_size)
    self.residual = residual
```

At the same time, add an if branch based on residual value. If the residual is **True**, add the first hidden layer1 to the result of the sigmoid of the second hidden layer.

```
def forward(self, input):
    assert len(input.size()) == 2
    # first layer
    hidden_1 = self.layer1(input)
    hidden_1 = vg.Sigmoid.do_forward(hidden_1)
    # second layer
    hidden_2 = self.layer2(hidden_1)
    # softmax activation
    if self.residual:
        hidden_2 = vg.Sigmoid.do_forward(hidden_2) + hidden_1
    else:
        hidden_2 = vg.Sigmoid.do_forward(hidden_2)
    output = vg.logsoftmax(self.layer3(hidden_2))
    return output
```

Also, the parameters function has to be modified.

```
def parameters(self):
    return self.layer1.parameters() + self.layer2.parameters() +
    self.layer3.parameters()
```

After running the network with and without the residual layer, the result shows that the residual layer made lower accuracy and loss at the beginning. Meanwhile, the residual layer is due to a sharper drop of loss. Although the accuracy and loss value after a few epochs are very similar, it still improves the performance of the network gently(see Fig. 2).

question 11 After building the classifier for dataset CIFAR_10 based on the 60-minute tutorial, I trained the following hyperparameters to find the best accuracy.

$$learning\_rate = [0.0005, 0.001, 0.01]$$

$$momentum = [0.9, 0.95]$$

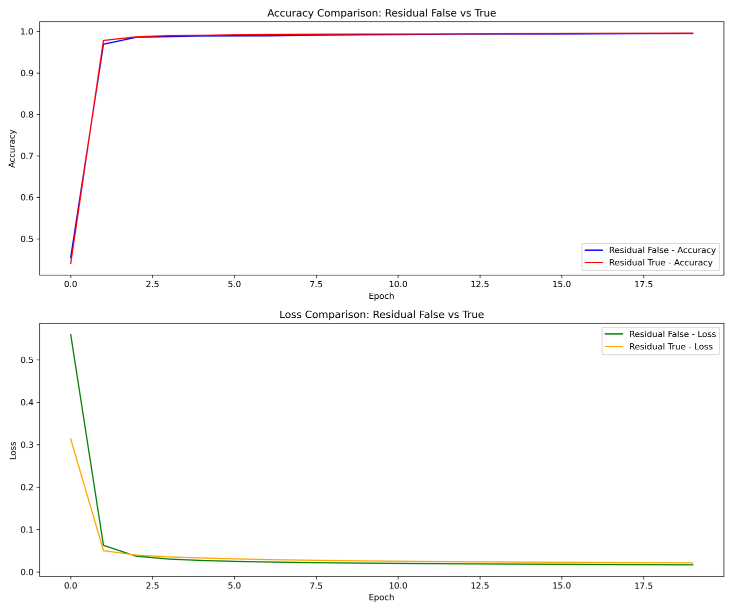

Figure 2: Comparison of Residual False and True

Under the influence of different parameters, it can be seen that when the learning rate is 0.01, the performance of the model is the worst on both the training dataset and the test dataset. This is because the step size is too large, which prevents the model from descending to the optimal point.

The model performs best when the learning rate is 0.001 and momentum is 0.95, and the accuracy on the test data set can reach 0.6391 on the 8th epoch. Different momentums have very little impact on the training effect when the learning rate is 0.001. When the momentum is reduced by 0.05, the accuracy is reduced by about 0.001. But lower momentum makes the accuracy more volatile(see Fig. 3, 5 and 4).

At a smaller learning rate of 0.0005, the impact of momentum is more obvious. Higher momentum results in higher accuracy at the beginning and lower loss on both the training and test data sets.

question 12 In previous question, I used the SGD method. However, SGD uses a fixed learning rate, which usually requires manual adjustment. If the learning rate is set too small, the training process may converge slowly; if the learning rate is set too large, the training may be unstable. And SGD often uses momentum to help speed up convergence. Momentum can be thought of as the accumulated velocity of the simulated object in the direction of the gradient, helping to jump out of local minima. In each iteration, SGD uses a single gradient to update parameters.

In this problem, I plan to use Adam. Because Adam introduces an adaptive

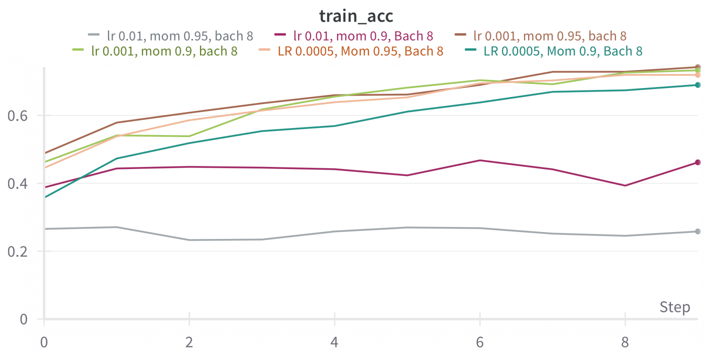

Figure 3: Training Accuracy

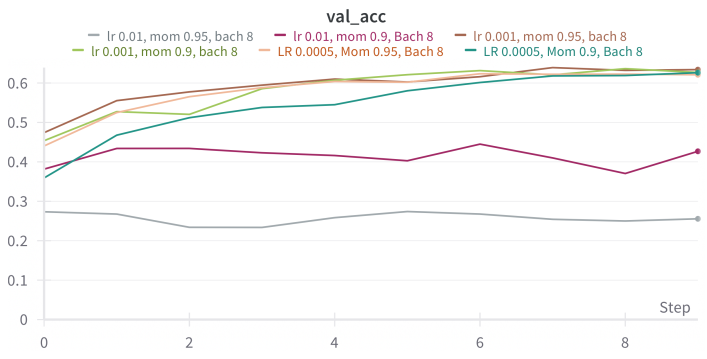

Figure 4: Testing Accuracy

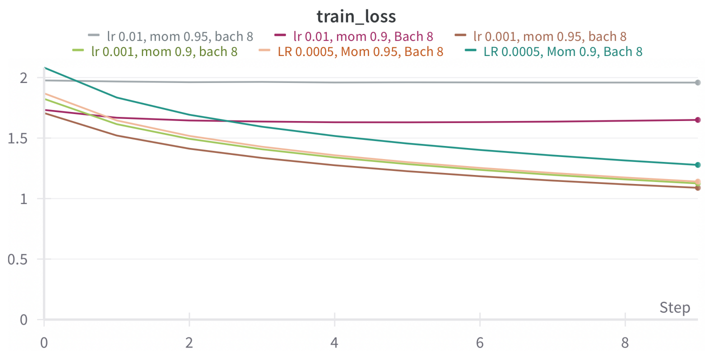

Figure 5: Training Loss

learning rate mechanism, it dynamically adjusts the learning rate based on the gradient of each parameter. This helps to automatically adjust the learning rate at different stages of training, allowing for more efficient optimization. And it comes with momentum-like effects because it uses an exponential moving average of the gradient[1].

SGD is generally more scalable when training on large-scale datasets because it uses only one sample or a small batch of samples in each iteration. Adam generally performs better on smaller data sets and default parameter settings. But in some cases, it may over-adapt to noise and therefore may not work as well as SGD on some problems [2]. As mentioned before, Adam showed instability on the test data set in the experiment(see Fig. 6).

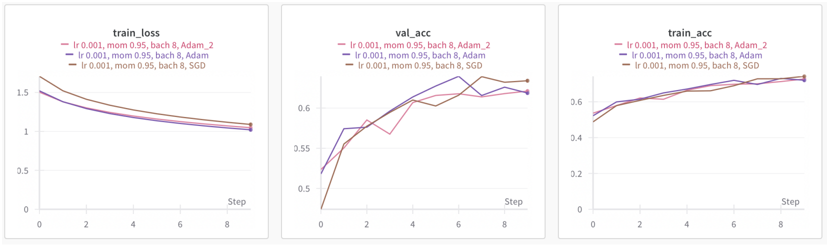

Figure 6: Comparsion of Adam and SGD

References

[1] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[2] Pan Zhou et al. “Towards theoretically understanding why sgd generalizes better than adam in deep learning”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 21285–21296.

2 Appendix

Part 3

```
import torch
import numpy as np
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from sklearn.model_selection import ParameterGrid
from tqdm import tqdm
import wandb
import random
import pickle

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

```
def set_up(batch_size):
    transform = transforms.Compose([transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
    trainset = torchvision.datasets.CIFAR10(root='./data',
    train=True, download=False, transform=transform)
    trainloader = torch.utils.data.DataLoader(trainset,
    batch_size=batch_size, shuffle=True, num_workers=2)
    testset = torchvision.datasets.CIFAR10(root='./data',
    train=False, download=False, transform=transform)
    testloader = torch.utils.data.DataLoader(testset,
    batch_size=batch_size, shuffle=False, num_workers=2)
    return trainloader, testloader, trainset, testset


def training(hpts, device):
    results_dict_tune = {}

    for param_run, params in enumerate(hpts):
        lr = params['learning_rate']
        mom = params['momentum']
        batch_size = params["batch_size"]
        trainloader, testloader, trainset, testset = set_up(batch_size)
        net = Net()
        # net.to(device)
        criterion = nn.CrossEntropyLoss()
        optimizer = optim.SGD(net.parameters(), lr=lr, momentum=mom)

        loss_list = list()

        # start a new wandb run to track this script
        wandb.init(
            # set the wandb project where this run will be logged
            project="DeepL_Assignment2_Question11",
            # track hyperparameters and run metadata
            config={
                "learning_rate": lr,
                "architecture": "CNN",
                "dataset": "CIFAR10",
                "epochs": 10,
            }
        )

        for epoch in range(10):
            # train
            for i, data in tqdm(enumerate(trainloader, 0)):
```

```
    # get the inputs; data is a list of [inputs, labels]

    inputs, labels = data
    inputs, labels = inputs.to(device), labels.to(device)

    # zero the parameter gradients
    optimizer.zero_grad()

    # forward + backward + optimize
    outputs = net(inputs)
    loss = criterion(outputs, labels)
    loss_list.append(loss.item())
    loss.backward()
    optimizer.step()
train_loss = np.average(loss_list)

# evaluate training
correct_train = 0
total_train = 0
# since we're not training, we don't need to calculate
the gradients for our outputs
with torch.no_grad():
    for data in trainloader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)

        # calculate outputs by running images through the network
        outputs = net(images)
        # the class with the highest energy is what we choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        total_train += labels.size(0)
        correct_train += (predicted == labels).sum().item()
train_acc = correct_train / total_train

# evaluate test
correct_val = 0
total_val = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)

        # calculate outputs by running images through the network
        outputs = net(images)
        # the class with the highest energy is what we
        choose as prediction
```

```
                    _, predicted = torch.max(outputs.data, 1)
                    total_val += labels.size(0)
                    correct_val += (predicted == labels).sum().item()

                val_acc = correct_val / total_val
                print(f'Training Accuracy with lr {lr} and mom {mom}: {train_acc}')
                print(f'Validation Accuracy with lr {lr} and mom {mom}: {val_acc}')

                # log metrics to wandb
                wandb.log({"train_acc": train_acc, "val_acc":
                val_acc,"train_loss": train_loss})
                results_dict_tune[param_run] = [train_acc, val_acc, loss_list]
            # [optional] finish the wandb run, necessary in notebooks
            wandb.finish()

            print('Finished Training')
        return results_dict_tune

    if __name__ == '__main__':
        # batch_size = 4
        # trainloader, testloader, trainset, testset = set_up(batch_size)
        device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
        # Hyperparameter Tuning: Question 11

        learning_rate = [0.0005, 0.001, 0.01]
        momentum = [0.9, 0.95]
        batch_size = [8, 32, 64]
        ###Define the grid of parameters to search
        hyper_grid = {'learning_rate': learning_rate, 'momentum':
        momentum, "batch_size": batch_size}
        hpts = ParameterGrid(hyper_grid)
        results_dict_tune = training(hpts, device)

        # data_to_pickle = {'example': 'data'}
        with open('./tuning_results.pkl', 'wb') as f:
            pickle.dump(results_dict_tune, f)
```