```
from google.colab import drive
drive.mount('/content/drive')

%load_ext autoreload
%autoreload 2
%autosave 180

import sys
sys.path.append("/content/drive/MyDrive/cs137assignments/assignment3")
```

```
    Drive already mounted at /content/drive; to attempt to forcibly remount, call dr
    Autosaving every 180 seconds
```

Convolutional Neural Networks

Objective: This assignment aims to help you understand CNNs through implementing layers for a CNN network. It also provide you a chance to tune a CNN.

Tasks: The first part of the assignment asks you to implement three types of layers (batch normalization, convolution, and pooling) that are commonly used in CNNs. To reduce the level of difficulty, you can implement convolution and pooling layers with `numpy`, and you are allowed to use for-loops. Your calculation will be compared against `torch` functions.

The second part of the assignment asks you to construct and train a convolutional neural network on the **bean-leaf dataset (link)**. In this part work, you need to code up your own convolutional neural network using `torch` layers. Then you apply the CNN to the dataset. You need to tune the model and try to get the highest test accuracy.

Detailed Instructions: Please implement all functions in `my_layers.py` and run through tests of your layers in this ipython notebook. You also need to construct a CNN in `conv_net.py`.

Then you need to train the CNN you have implemented. You may want to tune your CNN through multiple runs of training and validation to find a good model. You can consider `pytorch_lightning` to tune hyperparameters.

At the end, you need to save your model to this folder for submission. We will test your saved model to decide your points for the last part of work.

Grading: Please finish your work in the folder `assignment3`. If you use google colab, please upload folder `assignment3` to your google drive and run your code there. When you have finished your work, please print your notebook to a pdf file and put it to `assignment3`. You need to submit the entire `assignment3` folder, which should contain:

- all your implementations;

- the notebook that can run with your implementations;
- your saved model; and
- a pdf print of your notebook.

(If Gradescope does not accept your submission due to size limit, please let us know).

When we grade your code, we check your result notebook as well as your code. If your code cannot generate the result of a problem in your notebook file, you will get zero point for that problem.

GPU resources: In this assignment you will need to use GPU to tune your model for the bean-leaf task. You can use GPU from google colab and Kaggle. You have up to 30 GPU hours/week from Kaggle (link). Google colab does not have an anounced limit, but you should be able to get more hours than Kaggle. Here are some tips for GPU usage.

1. Only use GPU when necessary. You may want to garantee that your model runs before you put it to GPU.
2. Start your work early, so you have more GPU time
3. Remember to release GPU when you don't run your model.

Deadline: Oct 25, 23:59pm, 2022.

```
# As usual, a bit of setup
import numpy as np
import torch
import matplotlib.pyplot as plt

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython


def mean_diff(x, y):
    with torch.no_grad():
        if isinstance(x, torch.Tensor):
            x = x.numpy()
        if isinstance(y, torch.Tensor):
            y = y.numpy()
        err = np.mean(np.abs(x - y))
    return err

# If you have cuda, do the following
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
print(device)
```

```
cuda:0
```

Batch normalization

Question 1 (4 points). Please implement the batch normalization layer in `bn_layer.py`. The following code block will test your implementation against the `torch` implementation. Please refer to the documentation of `torch.nn.BatchNorm2d` (link) for the detailed calculation of batch normalization.

```
from bn_layer import BNLayer
from torch.nn import BatchNorm2d
# initialize a tensor X and pretend that the tensor is a feature map in a CNN.
# In this case, we do normalization over different channels. As we have mentioned in
# class, all pixels in a channel share the same normalization parameters.

np.random.seed(137)
N, C, H, W = 200, 5, 50, 60
X = torch.tensor(np.random.randn(N, C, H, W) * 0.2 + 5, dtype = torch.float32)

# initialize parameters for batch normalization

momentum = 0.1
epsilon = 0.01


# batch_normalization with torch.

torch_bn = BatchNorm2d(5, eps = epsilon, momentum = momentum, dtype = torch.float32)

# we use the same parameters as torch batch normalization.
my_bn = BNLayer(5, eps=epsilon, momentum=momentum)


# compare computation results over different training batches.

res_tf = torch_bn(X[0:10])
res_my = my_bn(X[0:10])


print("Calculation from the first batch: the difference between the two implementatio
          mean_diff(res_tf, res_my))

# run batch normalization on the second batch

res_tf = torch_bn(X[10:20])
res_my = my_bn(X[10:20])

print("Calculation from the second batch: the difference between the two implementati
          mean_diff(res_tf, res_my))
```

```
# run batch normalization on the third batch

res_tf = torch_bn(X[20:30])
res_my = my_bn(X[20:30])

print("Calculation from the third batch: the difference between the two implementatio
          mean_diff(res_tf, res_my))

# compare computation results over a testing batch
# my_bn.eval()
# torch_bn.eval()

res_tf = torch_bn(X[40:50])
res_my = my_bn(X[40:50])
print("Calculation from testing: the difference between the two implementations is ",
       mean_diff(res_tf, res_my))
```

```
    Calculation from the first batch: the difference between the two implementations
    Calculation from the second batch: the difference between the two implementation
    Calculation from the third batch: the difference between the two implementations
    Calculation from testing: the difference between the two implementations is  1.2
```

The convolution operation

Question 2 (4 points). Please implement a convolutional operation in `my_layers.py`. The following code block will test your implementation against the corresponding `torch` function. Please refer to the documentation of `torch.nn.Conv2d` for the detailed calculation of the convolution operation.

Note: You only need to consider `stride = 1`, `padding` is `same` or `valid`.

```
from my_layers import conv_forward
import torch.nn as nn

# initialize a tensor in the NCHW format and pretend the tensor is a feature map in a
x = np.random.randn(*[1, 5, 10, 20]).astype(np.float32)


# The first test case:
h_stride = 1
w_stride = 1
padding = "same"
t_mod = nn.Conv2d(in_channels=5, out_channels = 4, kernel_size = (3,3), stride=[h_str
t_out = t_mod(torch.tensor(x, dtype = torch.float32))
with torch.no_grad():
    w = t_mod.weight.numpy()
```

```
        b = t_mod.bias.numpy()
    out = conv_forward(input=x, filters=w, bias = b, stride=[h_stride, w_stride], padding

    with torch.no_grad():
        print('Difference:', mean_diff(out, t_out))

    # The second test case:
    h_stride = 1
    w_stride = 1
    padding = "same"
    t_mod = nn.Conv2d(in_channels=5, out_channels = 4, kernel_size = (7,7), stride=[h_str
    t_out = t_mod(torch.tensor(x, dtype = torch.float32))
    with torch.no_grad():
        w = t_mod.weight.numpy()
        b = t_mod.bias.numpy()
    out = conv_forward(input=x, filters=w, bias = b, stride=[h_stride, w_stride], padding

    with torch.no_grad():
        print('Difference:', mean_diff(out, t_out))



    # The third test case:
    h_stride = 1
    w_stride = 1
    padding = "valid"
    t_mod = nn.Conv2d(in_channels=5, out_channels = 4, kernel_size = (7,3), stride=[h_str
    t_out = t_mod(torch.tensor(x, dtype = torch.float32))
    with torch.no_grad():
        w = t_mod.weight.numpy()
        b = t_mod.bias.numpy()
    out = conv_forward(input=x, filters=w, bias = b, stride=[h_stride, w_stride], padding

    with torch.no_grad():
        print('Difference:', mean_diff(out, t_out))



    # The third test case:
    w = np.random.randn(3, 3, 5, 4).astype(np.float32)
    h_stride = 1
    w_stride = 1
    padding = "valid"

    t_mod = nn.Conv2d(in_channels=5, out_channels = 4, kernel_size = (3, 3), stride=[h_st
    t_out = t_mod(torch.tensor(x, dtype = torch.float32))
    with torch.no_grad():
        w = t_mod.weight.numpy()
        b = t_mod.bias.numpy()
    out = conv_forward(input=x, filters=w, bias = b, stride=[h_stride, w_stride], padding

    with torch.no_grad():
        print('Difference:', mean_diff(out, t_out))
```

```
Difference: 4.9762893468141553e-08
Difference: 9.633135050535202e-08
Difference: 8.84093525302079e-08
Difference: 5.717609181172318e-08
```

The max-pooling operation

Question 4 (4 points). Please implement the forward pass for the max-pooling operation in the function `pooling_forward_naive` in the file `my_layers.py`. Your implementation will be compared against torch implementation.

```
from my_layers import max_pool_forward

np.random.seed(137)
# shape is NCHW
x = np.random.randn(2, 4, 10, 20).astype(np.float32)

pool_height = 3
pool_width = 3
stride_h = pool_height
stride_w = pool_width

my_out = max_pool_forward(x, [pool_height, pool_width], stride=[stride_h, stride_w],
t_mod = nn.MaxPool2d([pool_height, pool_width], stride=[stride_h, stride_w])
t_out = t_mod(torch.tensor(x, dtype = torch.float32))

with torch.no_grad():
    t_out = t_mod(torch.tensor(x, dtype = torch.float32))
    print('Difference: ', mean_diff(my_out, t_out))



pool_height = 4
pool_width = 4
stride_h = 2
stride_w = 2
my_out = max_pool_forward(x, [pool_height, pool_width], stride=[stride_h, stride_w],
t_mod = nn.MaxPool2d([pool_height, pool_width], stride=[stride_h, stride_w])
t_out = t_mod(torch.tensor(x, dtype = torch.float32))

with torch.no_grad():
    t_out = t_mod(torch.tensor(x, dtype = torch.float32))
    print('Difference: ', mean_diff(my_out, t_out))


pool_height = 5
pool_width = 4
stride_h = 2
```

```
stride_w = 1

my_out = max_pool_forward(x, [pool_height, pool_width], stride=[stride_h, stride_w],
t_mod = nn.MaxPool2d([pool_height, pool_width], stride=[stride_h, stride_w])
with torch.no_grad():
    t_out = t_mod(torch.tensor(x, dtype = torch.float32))
    print('Difference: ', mean_diff(my_out, t_out))
```

```
Difference:  0.0
Difference:  0.0
Difference:  0.0
```

Implement a CNN with Keras and train it

In this task, you need to build a CNN with `torch` layers and train it on the `beans` dataset. The `beans` dataset is an image classification task. The dataset contains images of bean leafs, and these images falls into three categories. The task is to predict the label of a leaf.

Question 5 (5 points). Implementation: in your implementation of the neural network, you need to construct your CNN using existing `torch` layers. You cannot directly load a CNN such as ResNet from `torch`. You can check the example code provided by chapter 8 of D2L book.

NOTE: you cannot copy code from any resources.

Question 6 (6 points). Model tuning: depending the final performance of your trained model, you will get

- 2 points for the test accuracy being over 0.7
- 2 points for the test accuracy being over 0.8
- 2 points for the test accuracy being over 0.85

Train the CNN

Load Data

```
from torchvision.transforms import ToTensor, Compose
from torch.utils.data import random_split, DataLoader
import torch
```

```
from torchvision import datasets, transforms
train_dir = "/content/drive/MyDrive/cs137assignments/assignment3/archive/train/train"
```

```
validation_dir = "/content/drive/MyDrive/cs137assignments/assignment3/archive/validat
test_dir = "/content/drive/MyDrive/cs137assignments/assignment3/archive/test/test"

mean = [0.485, 0.456, 0.406]
std = [0.229, 0.224, 0.225]
train_transform = transforms.Compose([
        transforms.RandomResizedCrop((224,224)),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])

valid_transform=transforms.Compose([
            transforms.CenterCrop((224,224)),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ])

test_transform=transforms.Compose([
            transforms.CenterCrop((224,224)),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ])

train_set = datasets.ImageFolder(train_dir, transform=train_transform)
validation_set = datasets.ImageFolder(validation_dir, transform=valid_transform)
test_set = datasets.ImageFolder(test_dir, transform=test_transform)

batch_size = 32
train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True)
validation_loader = DataLoader(validation_set, batch_size=batch_size)
test_loader = DataLoader(test_set, batch_size=batch_size)



#from bean_dataset import BeanImageDataset
# import torchvision

# from bean_dataset import BeanImageDataset

# trainset = BeanImageDataset("../archive/train/train/")
# validset = BeanImageDataset("../archive/validation/validation")

# train_loader = DataLoader(trainset, batch_size=4, shuffle=True)
# valid_loader = DataLoader(validset, batch_size=4, shuffle=True)


from conv_net import ConvNet
import numpy as np

if device.type == "cuda":
    total_mem = torch.cuda.get_device_properties(0).total_memory
```

```
else:
    total_mem = 0

epochs = 60
learning_rate = 0.0001

model = ConvNet()
model.to(device)

optimizer =  torch.optim.Adam(model.parameters(), lr = learning_rate)
loss_function = torch.nn.CrossEntropyLoss()

# Recording the loss
total_train_loss = []
total_val_loss   = []
total_val_acc    = []

# TODO: implement the training procedure. Please remember to zero out previous gradie

for i in range(epochs):
    training_loss = 0
    for j, data in enumerate(train_loader):
        x, y = data
        ## If GPU is available, move to cuda
        if device.type == "cuda":
            x = x.to(device)
            y = y.to(device)

        # TODO: Implement parameter updates here
            optimizer.zero_grad()
            y_hat = model(x)
            loss = loss_function(y_hat, y)
            loss.backward()
            optimizer.step()



        # TODO: Record loss values to some variable
        if device.type == "cuda":
            loss = loss.cpu()
            x = x.cpu()
            y = y.cpu()
            y_hat = y_hat.cpu()
            loss = loss.cpu()
        training_loss += loss.item()
    training_loss /= (j+1)
    total_train_loss.append(training_loss)

    # validate
    with torch.no_grad():
        # TODO: compute validation loss and validation accuracy
        training_val_loss = 0
        training_val_acc  = 0
        for k, data in enumerate(validation_loader):
            x, y = data

            x = x.to(device)
            y = y.to(device)

            y_hat = model(x)

            if device.type == "cuda":
              y = y.to("cpu")
              x = x.to("cpu")
              y_hat = y_hat.to("cpu")

            v_loss  = loss_function(y_hat, y).item()
            training_val_loss += v_loss
            acc = np.average(y.numpy() == np.argmax(y_hat.numpy(), axis = 1))
            training_val_acc += acc

        training_val_loss, training_val_acc = (training_val_loss/(k+1), training_val_
        total_val_loss.append(training_val_loss)
        total_val_acc.append(training_val_acc)

    # check GPU memory if necessary
    if device.type == "cuda":
        alloc_mem = torch.cuda.memory_allocated(0)
    else:
        alloc_mem = 0

    # print out
    print(f"Epoch [{i+1}]: Training Loss: {training_loss} Validation Loss: {training_
        f" Allocated/Total GPU memory: {alloc_mem}/{total_mem}" if device.type == "cu
    ))
```

```
 Epoch [3]: Training Loss: 0.8206150278900609 Validation Loss: 0.7698398351669311
 Epoch [4]: Training Loss: 0.786556115656188 Validation Loss: 0.6793647229671478
 Epoch [5]: Training Loss: 0.7657428817315535 Validation Loss: 0.6781855404376984
 Epoch [6]: Training Loss: 0.7776985078146963 Validation Loss: 0.8841259241104126
 Epoch [7]: Training Loss: 0.7025936562003512 Validation Loss: 0.6188271284103394
 Epoch [8]: Training Loss: 0.7165577095566373 Validation Loss: 0.7600942939519882
 Epoch [9]: Training Loss: 0.6786588999358091 Validation Loss: 0.696013069152832
 Epoch [10]: Training Loss: 0.6412433421973026 Validation Loss: 0.876171836256980
 Epoch [11]: Training Loss: 0.6805603486118894 Validation Loss: 0.591373443603515
 Epoch [12]: Training Loss: 0.6026856330308047 Validation Loss: 0.760087746381759
 Epoch [13]: Training Loss: 0.5621951412070881 Validation Loss: 0.591192510724067
 Epoch [14]: Training Loss: 0.5351477427916094 Validation Loss: 0.513831844925880
 Epoch [15]: Training Loss: 0.4640839637228937 Validation Loss: 0.534323483705520
 Epoch [16]: Training Loss: 0.4727653888138858 Validation Loss: 0.671479524299502
 Epoch [17]: Training Loss: 0.5820854712616313 Validation Loss: 0.71819489300251
 Epoch [18]: Training Loss: 0.4835134247938792 Validation Loss: 0.444689518213272
 Epoch [19]: Training Loss: 0.4469553223161986 Validation Loss: 0.549432602524757
 Epoch [20]: Training Loss: 0.42783578598138056 Validation Loss: 0.49006284475326
 Epoch [21]: Training Loss: 0.4286458243926366 Validation Loss: 0.582991778850555
 Epoch [22]: Training Loss: 0.42987464865048725 Validation Loss: 0.48875359296798
```

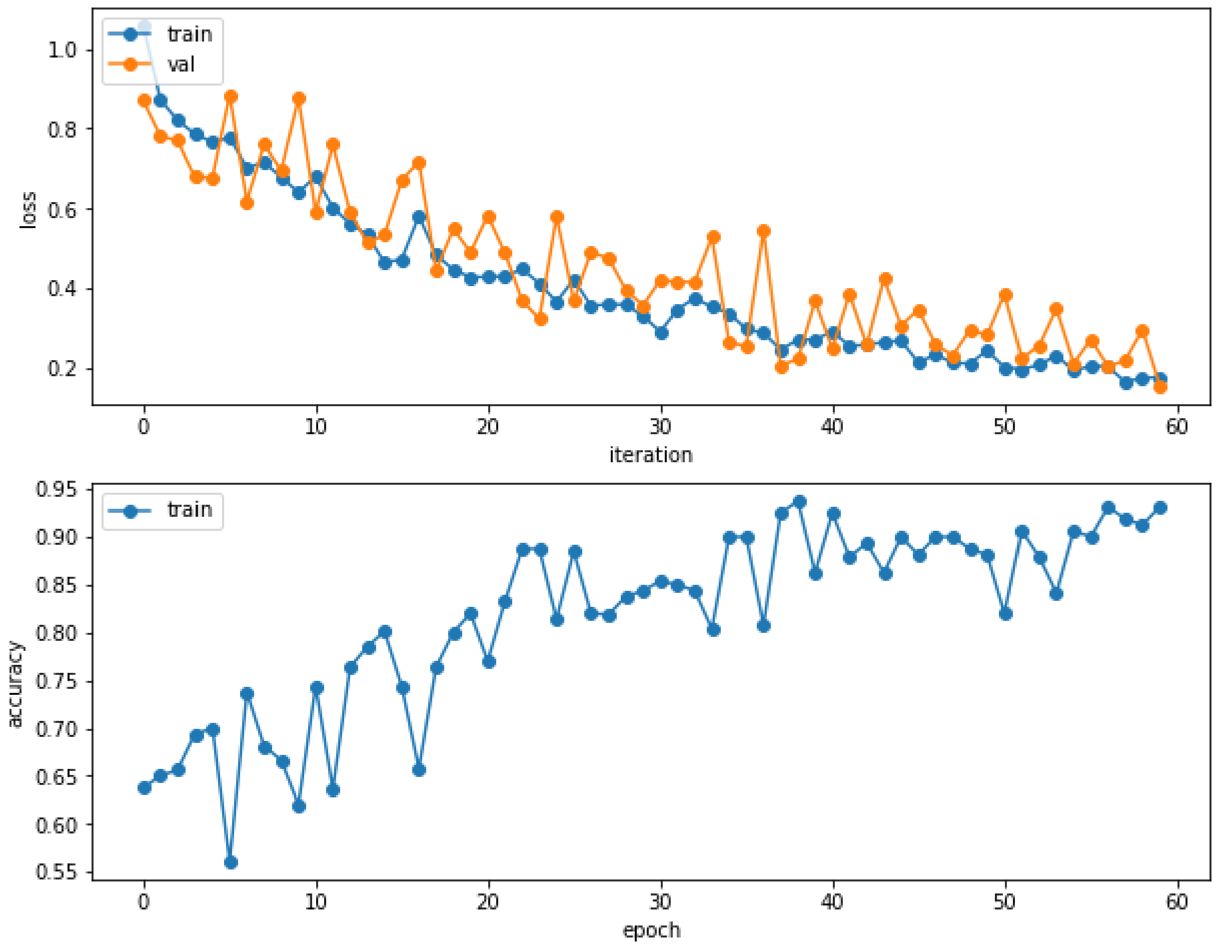
```
    Epoch [23]: Training Loss: 0.4482383326147542 Validation Loss: 0.369703894853591
    Epoch [24]: Training Loss: 0.4122926425753218 Validation Loss: 0.322497072815895
    Epoch [25]: Training Loss: 0.3661608528910261 Validation Loss: 0.581027925014495
    Epoch [26]: Training Loss: 0.4222362714283394 Validation Loss: 0.368297955393791
    Epoch [27]: Training Loss: 0.3559672584136327 Validation Loss: 0.490562999248504
    Epoch [28]: Training Loss: 0.3597569601102309 Validation Loss: 0.476062782853841
    Epoch [29]: Training Loss: 0.35878342254595325 Validation Loss: 0.39517568647861
    Epoch [30]: Training Loss: 0.3303100904732039 Validation Loss: 0.357242203503847
    Epoch [31]: Training Loss: 0.2909055623141202 Validation Loss: 0.420786845684051
    Epoch [32]: Training Loss: 0.34696978556387353 Validation Loss: 0.41630772650241
    Epoch [33]: Training Loss: 0.3752685288588206 Validation Loss: 0.415809451043605
    Epoch [34]: Training Loss: 0.3529290900085912 Validation Loss: 0.528716686367988
    Epoch [35]: Training Loss: 0.3371438113125888 Validation Loss: 0.264657016098499
    Epoch [36]: Training Loss: 0.2974024952361078 Validation Loss: 0.254050475358963
    Epoch [37]: Training Loss: 0.2875986261801286 Validation Loss: 0.544741147756576
    Epoch [38]: Training Loss: 0.2461167164711338 Validation Loss: 0.206755152344703
    Epoch [39]: Training Loss: 0.27062771424199594 Validation Loss: 0.22197623550891
    Epoch [40]: Training Loss: 0.2714466905842225 Validation Loss: 0.367904815636575
    Epoch [41]: Training Loss: 0.2912393167163386 Validation Loss: 0.248806607723236
    Epoch [42]: Training Loss: 0.255178565441659 Validation Loss: 0.3858248502016067
    Epoch [43]: Training Loss: 0.2582171536756284 Validation Loss: 0.257076944410800
    Epoch [44]: Training Loss: 0.26466590572487225 Validation Loss: 0.42389469072222
    Epoch [45]: Training Loss: 0.26846058440930914 Validation Loss: 0.30622303485870
    Epoch [46]: Training Loss: 0.2134114694640492 Validation Loss: 0.346758684329688
    Epoch [47]: Training Loss: 0.2338607297702269 Validation Loss: 0.258991048485040
    Epoch [48]: Training Loss: 0.2126855845704223 Validation Loss: 0.231091912835836
    Epoch [49]: Training Loss: 0.21100753071633252 Validation Loss: 0.29589540436863
    Epoch [50]: Training Loss: 0.24359095232053238 Validation Loss: 0.28239976763725
    Epoch [51]: Training Loss: 0.20168624016823192 Validation Loss: 0.38546718060970
    Epoch [52]: Training Loss: 0.19657616012475707 Validation Loss: 0.22436645105481
    Epoch [53]: Training Loss: 0.2075515134316502 Validation Loss: 0.256105181574821
    Epoch [54]: Training Loss: 0.2316239406213616 Validation Loss: 0.349119497090578
    Epoch [55]: Training Loss: 0.1922045504730759 Validation Loss: 0.211107121407985
    Epoch [56]: Training Loss: 0.2052416360626618 Validation Loss: 0.271776863932609
    Epoch [57]: Training Loss: 0.20298377988916455 Validation Loss: 0.20344508513808
    Epoch [58]: Training Loss: 0.16470755941488527 Validation Loss: 0.21909681260585
    Epoch [59]: Training Loss: 0.17499835868224953 Validation Loss: 0.29675106839276
    Epoch [60]: Training Loss: 0.17622903130496992 Validation Loss: 0.15594081915915
```

Plot training losses, validation losses, training accuracies, and validation accuracies. If these numbers are from the training on a small dataset, you should see clear overfitting.

```
plt.subplot(2, 1, 1)
plt.plot(total_train_loss, '-o')
plt.plot(total_val_loss, '-o')
plt.legend(['train', 'val'], loc='upper left')
plt.xlabel('iteration')
plt.ylabel('loss')

plt.subplot(2, 1, 2)
plt.plot(total_val_acc, '-o')
plt.legend(['train', 'val'], loc='upper left')
plt.xlabel('epoch')
```

```
    plt.ylabel('accuracy')
    plt.show()
```



```
torch.save(model, "bean_classifier.sav")
```

Predict labels of the test set using the saved model

```
# Load the previously saved model.
test_loader = DataLoader(test_set, batch_size=batch_size)


t_acc = []
with torch.no_grad():
    # only one item in the iterator
    # Add more batches if your device couldn't handle the computation
    for _, data in enumerate(test_loader):
        x, y = data
        x = x.to(device)
        y = y.to(device)
        y_hat = model(x)

        if device.type == "cuda":
          x = x.to("cpu")
          y = y.to("cpu")
          y_hat = y_hat.to("cpu")
        acc = np.average(y.numpy() == np.argmax(y_hat.numpy(), axis = 1))
        t_acc.append(acc.item())
print(f"Test accuracy: {np.average(t_acc)}")

    Test accuracy: 0.8671875
```

Colab paid products - Cancel contracts here

✓ 0s completed at 4:33 PM