notebook

October 27, 2023

[ ]:
```
#from google.colab import drive
#drive.mount('/content/drive')

%load_ext autoreload
%autoreload 2
%autosave 180

import sys
#sys.path.append("/content/drive/MyDrive/137-codes/assignment3-solution")
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

```
Autosaving every 180 seconds
```

1 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 are expected to construct a 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.

In the classifcation task, you need to construct a CNN in `conv_net.py`. Then you should 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.

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 27, 23:59pm, 2023.

[ ]:
```
# 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)
```

```
cpu
```

1.1 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 the
# 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␣
 ↪implementations is ",
          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␣
 ↪implementations is ",
          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␣
 ↪implementations is ",
          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
is  9.53356e-06
Calculation from the second batch: the difference between the two
implementations is  9.58232e-06
Calculation from the third batch: the difference between the two implementations
is  9.637566e-06
Calculation from testing: the difference between the two implementations is
1.9003392e-07
```

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 CNN.
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_stride, w_stride], padding=padding)
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=padding)

with torch.no_grad():
    print('Difference between your implementation and torch calculation:',␣
 ↪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_stride, w_stride], padding=padding)
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=padding)

with torch.no_grad():
    print('Difference between your implementation and torch calculation:',␣
 ↪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_stride, w_stride], padding=padding)
```

```
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=padding)

with torch.no_grad():
    print('Difference between your implementation and torch calculation:',␣
 ↪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_stride, w_stride], padding=padding)
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=padding)

with torch.no_grad():
    print('Difference between your implementation and torch calculation:',␣
 ↪mean_diff(out, t_out))
```

```
Difference between your implementation and torch calculation:
5.698660970665515e-08
Difference between your implementation and torch calculation:
9.773080819286406e-08
Difference between your implementation and torch calculation:
8.67787295848959e-08
Difference between your implementation and torch calculation:
6.240106150319964e-08
```

2 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], padding='VALID')
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('The difference between your implementation and torch operation is:␣
 ↪', 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], padding='VALID')
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('The difference between your implementation and torch operation is:␣
 ↪', 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], padding='VALID')
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('The difference between your implementation and torch operation is:␣
↪', mean_diff(my_out, t_out))
```

```
The difference between your implementation and torch operation is:  0.0
The difference between your implementation and torch operation is:  0.0
The difference between your implementation and torch operation is:  0.0
```

3 Implement a CNN with Torch 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 if the test accuracy is over 0.7 * 4 points if the test accuracy is over 0.8 * 6 points if the test accuracy is over 0.85

3.1 Train the CNN

3.1.1 Load Data

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

[ ]:
```
from bean_dataset import BeanImageDataset

trainset = BeanImageDataset("data/train")
validset = BeanImageDataset("data/validation")

train_loader = DataLoader(trainset, batch_size=4, shuffle=True)
valid_loader = DataLoader(validset, batch_size=len(validset), shuffle=False)
```

3.1.2 Train the model

Then you need to train your network below. You are supposed to implementing the training loop in the notebook cell below. Please pay attention to TODOs. Feel free to change hyper-parameters.

[ ]:
```
from conv_net import ConvNet, train
import numpy as np

device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
```

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

max_epoch = 50
learning_rate = 1e-3

# Load your CNN model
model = ConvNet()

# Train the CNN model
# the train function is supposed to return the average training loss,
# the average validation loss, and the validation accuracy in each epoch
# then we can plot the training behavior below.
train_loss, val_loss, val_acc = train(model, train_loader, valid_loader,␣
 ↪learning_rate, max_epoch, device)

# The print-out lines are from my own training function, and you don't have to␣
 ↪reproduce these lines.
```

```
Epoch [1]: Training Loss: 0.8950826561358904 Validation Loss: 0.8439488410949707
Accuracy: 75.0
Epoch [2]: Training Loss: 0.9569483051889192 Validation Loss: 0.847313404083252
Accuracy: 86.0
Epoch [3]: Training Loss: 0.7980117857111007 Validation Loss: 0.7634483575820923
Accuracy: 91.0
Epoch [4]: Training Loss: 0.8280635093514984 Validation Loss: 0.7478405833244324
Accuracy: 92.0
Epoch [5]: Training Loss: 0.7905444463707766 Validation Loss: 0.681023895740509
Accuracy: 97.0
Epoch [6]: Training Loss: 0.7976064091138397 Validation Loss: 0.6407067775726318
Accuracy: 99.0
Epoch [7]: Training Loss: 0.740417046724139 Validation Loss: 0.6163605451583862
Accuracy: 102.0
Epoch [8]: Training Loss: 0.7698891959135137 Validation Loss: 0.6449576616287231
Accuracy: 93.0
Epoch [9]: Training Loss: 0.7591903095194732 Validation Loss: 0.6592562198638916
Accuracy: 97.0
Epoch [10]: Training Loss: 0.7369287398330954 Validation Loss:
0.6074304580688477 Accuracy: 100.0
Epoch [11]: Training Loss: 0.7452383129408 Validation Loss: 0.6005153656005859
Accuracy: 97.0
Epoch [12]: Training Loss: 0.7191354463804642 Validation Loss:
0.5989725589752197 Accuracy: 102.0
Epoch [13]: Training Loss: 0.703332005895703 Validation Loss: 0.5897401571273804
```

```
Accuracy: 99.0
Epoch [14]: Training Loss: 0.7183904876810243 Validation Loss: 0.538832426071167
Accuracy: 102.0
Epoch [15]: Training Loss: 0.7149914997885126 Validation Loss:
0.6109691858291626 Accuracy: 101.0
Epoch [16]: Training Loss: 0.7152190534362002 Validation Loss:
0.5848734974861145 Accuracy: 104.0
Epoch [17]: Training Loss: 0.7281066680967118 Validation Loss:
0.6645515561103821 Accuracy: 98.0
Epoch [18]: Training Loss: 0.6887466067512984 Validation Loss:
0.5797222852706909 Accuracy: 104.0
Epoch [19]: Training Loss: 0.6993056514433452 Validation Loss:
0.5419667959213257 Accuracy: 106.0
Epoch [20]: Training Loss: 0.6941562708050127 Validation Loss:
0.5591424703598022 Accuracy: 104.0
Epoch [21]: Training Loss: 0.6838402902585199 Validation Loss:
0.6039848327636719 Accuracy: 104.0
Epoch [22]: Training Loss: 0.7795123654960665 Validation Loss:
0.6196224093437195 Accuracy: 93.0
Epoch [23]: Training Loss: 0.6503564564647822 Validation Loss:
0.5458844304084778 Accuracy: 102.0
Epoch [24]: Training Loss: 0.689612859896021 Validation Loss: 0.576741635799408
Accuracy: 101.0
Epoch [25]: Training Loss: 0.648139359983238 Validation Loss: 0.5164759755134583
Accuracy: 105.0
Epoch [26]: Training Loss: 0.6914037958535448 Validation Loss:
0.4951029121875763 Accuracy: 111.0
Epoch [27]: Training Loss: 0.727850048353313 Validation Loss:
0.48834872245788574 Accuracy: 101.0
Epoch [28]: Training Loss: 0.6294517078972691 Validation Loss:
0.5287407040596008 Accuracy: 103.0
Epoch [29]: Training Loss: 0.6688409303261046 Validation Loss:
0.5758522152900696 Accuracy: 102.0
Epoch [30]: Training Loss: 0.6188938757800228 Validation Loss:
0.5742987394332886 Accuracy: 104.0
Epoch [31]: Training Loss: 0.6569674903361493 Validation Loss:
0.5147915482521057 Accuracy: 107.0
Epoch [32]: Training Loss: 0.5888592460153186 Validation Loss:
0.5263767242431641 Accuracy: 101.0
Epoch [33]: Training Loss: 0.6406123147248992 Validation Loss: 0.538011372089386
Accuracy: 103.0
Epoch [34]: Training Loss: 0.5989242068214996 Validation Loss:
0.3912202715873718 Accuracy: 114.0
Epoch [35]: Training Loss: 0.5776820838393852 Validation Loss:
0.45738649368286133 Accuracy: 109.0
Epoch [36]: Training Loss: 0.5845773338581144 Validation Loss:
0.4224592447280884 Accuracy: 106.0
Epoch [37]: Training Loss: 0.6100964229985438 Validation Loss:
```

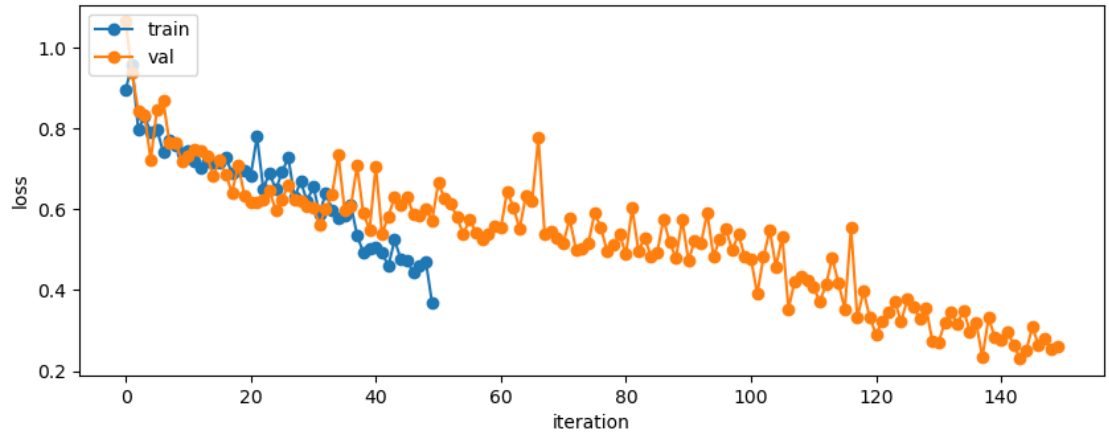
```
0.40897366404533386 Accuracy: 108.0
Epoch [38]: Training Loss: 0.5363613173552577 Validation Loss:
0.4793238937854767 Accuracy: 103.0
Epoch [39]: Training Loss: 0.49217094140105727 Validation Loss:
0.5535002946853638 Accuracy: 105.0
Epoch [40]: Training Loss: 0.5033427084073366 Validation Loss:
0.3320170044898987 Accuracy: 114.0
Epoch [41]: Training Loss: 0.5070878662255279 Validation Loss:
0.3450523316860199 Accuracy: 112.0
Epoch [42]: Training Loss: 0.4936710948333087 Validation Loss:
0.37696003913879395 Accuracy: 117.0
Epoch [43]: Training Loss: 0.45894602630071657 Validation Loss:
0.35683250427246094 Accuracy: 111.0
Epoch [44]: Training Loss: 0.5242413366033649 Validation Loss:
0.32119548320770264 Accuracy: 119.0
Epoch [45]: Training Loss: 0.4760338212561193 Validation Loss:
0.3507697880268097 Accuracy: 110.0
Epoch [46]: Training Loss: 0.47200753601469303 Validation Loss:
0.23444581031799316 Accuracy: 121.0
Epoch [47]: Training Loss: 0.4442567810071021 Validation Loss:
0.2765005826950073 Accuracy: 116.0
Epoch [48]: Training Loss: 0.45904468796773307 Validation Loss:
0.23155392706394196 Accuracy: 121.0
Epoch [49]: Training Loss: 0.4711463250306122 Validation Loss:
0.2633047103881836 Accuracy: 120.0
Epoch [50]: Training Loss: 0.36752923930416237 Validation Loss:
0.26018887758255005 Accuracy: 116.0
```

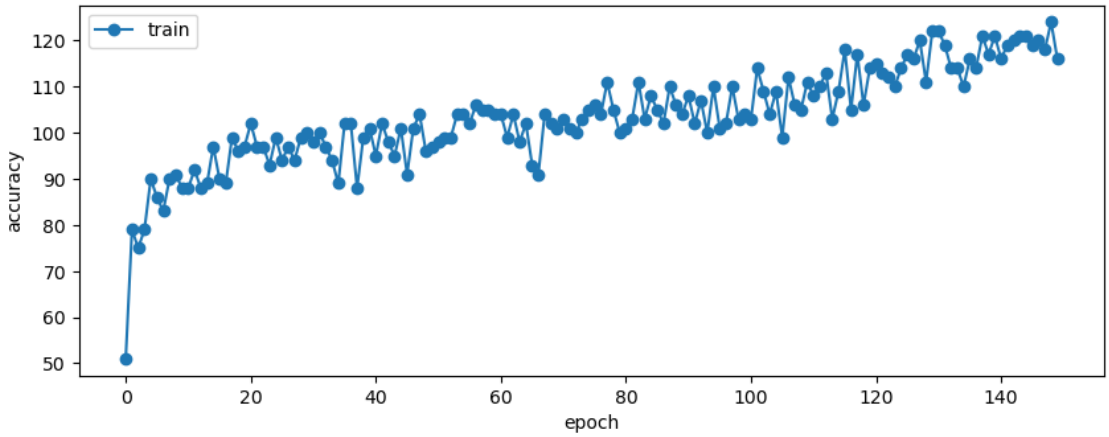
3.1.3 Plot the training history

One important way of diagnosing model training is to plot training and validation losses during the training procedure. In general, we should see a decreasing of training and validation losses. With my naive CNN model and only 4 epochs, the validation loss is even smaller than training loss, which is somewhat unusual, but a typical model should get higher validation losses.

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

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





3.1.4 Save the model

[ ]:
```
torch.save(model, "bean_classifier.sav")
```

3.1.5 Test the saved model on a test set

[ ]:
```
# Load the previously saved model.
testset = BeanImageDataset("data/test")
test_loader = DataLoader(testset, batch_size=1, shuffle=True)

model = torch.load("bean_classifier.sav")
model.eval()


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
```