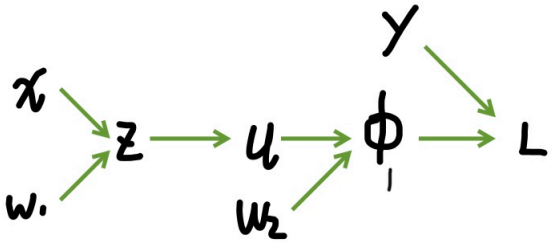
Intro to ML Homework5

Haotian Yi N18800809

April 14, 2020

1 Question 1

Computation graph:



Forward pass:

$$
\begin{aligned}
z &= w_1 * x \\
u &= \sigma(z) \\
\phi &= w_2 * u \\
L &= (\phi - y)^2
\end{aligned}
$$

Backward pass:

$$
\begin{aligned}
\partial_L L &= 1 \\
\partial_\phi L &= 2(\phi - y) \\
\partial_{w_2} \phi &= u \\
\partial_u \phi &= w_2 \\
\partial_z u &= \sigma^{'}(z) = \sigma(z)(1 - \sigma(z)) = u(1 - u) \\
\partial_{w_1} z &= x
\end{aligned}
$$

Thus:

$$
\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \phi} \frac{\partial \phi}{\partial u} \frac{\partial u}{\partial z} \frac{\partial z}{\partial w_1} = 2(\phi - y) * w_2 * u(1 - u) * x
$$

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial \phi}\frac{\partial \phi}{\partial w_2} = 2(\phi - y) * u$$

It can be seen that forward pass uses one sigmoid, two multiplication, one subtraction and a square, backward pass calculate multiple derivatives, it looks like backward pass is more complex by almost twice. However when we calculate derivative respectively on w_1 or w_2, we only need four multiplication on what what we got by derivatives, but there would be several duplicate calculations (as shown below) in forward pass which contains several mutiplication , several complex sigmoid, these will lower its efficiency.

$$\frac{\partial L}{\partial w_1} = 2(w_2\sigma(z) - y) * w_2 * \sigma(z)(1 - \sigma(z)) * x$$
$$\frac{\partial L}{\partial w_2} = 2(w_2\sigma(z) - y) * \sigma(z)$$

Roughly I think backward pass used for calculating gradient will save half calculation than forward method.

2 Question 2

(a) Z's shape: (46,62,20), U's shape: (46,62,20)

(b) number of input channel: 10, number of output channel: 20.

(c)

Add-operation: 46*62*9*10*2 = 513360
Multiplication: 46*62*9*10 = 256680

(46*62*20)*(3*3*10) = 5133600
add and mutiply-operations

(d) Number of trainable parameters is :3*3*10*20 = 1800

(3*3*10*20)+20 = 1820

3 Question 3

(a) Regularized loss:

$$L = L(w) + \lambda\|w\|$$

(b) Update Rule:

$$w_{t+1} = w_t - \alpha\frac{\partial L}{\partial w} = w_t - \alpha * (\frac{\partial L(w)}{\partial w} + 2\lambda w_t) = (1 - 2\alpha\lambda)w_t - \alpha * \frac{\partial L(w)}{\partial w}$$

(c) As we can see from (b), before each descent update, w_t is multiplied by $(1 - 2\alpha\lambda)$,which is a smaller factor than 1, so that it make weights shrunk.

(d) Increasing λ will lift up strength of regularization, it makes w shrink more intensively. Learning rate should be chosen as a small number in the range of 0 to 1, and for the update above, I think learning rate should be chosen to make $2\alpha\lambda$ much smaller than 1 to make the update stable.

4 Question 4

upload data and unzip

```
!unzip data.zip
```

import model and normalize data

```
[18] import torchvision
     import torch
     from torchvision import datasets, models, transforms
     transforms = transforms.Compose([
     transforms.Resize(256),
     transforms.CenterCrop(224),
     transforms.ToTensor(),
     transforms.Normalize([0.485, 0.456, 0.406],
     [0.229, 0.224, 0.225])
     ])
     train_set = datasets.ImageFolder("data/train",transforms)
     val_set = datasets.ImageFolder("data/val",transforms)
     model = models.resnet34(pretrained=True)

     trainloader = torch.utils.data.DataLoader(train_set, batch_size=4,
                                               shuffle=True, num_workers=2)
     valloader = torch.utils.data.DataLoader(val_set, batch_size=1,
                                             shuffle=True, num_workers=2)
     classes = ('cat','dog')
```

take a look at train set

```
import matplotlib.pyplot as plt
import numpy as np

# functions to show an image

def imshow(img):
    img = img / 2.5 + 0.25     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()

# get some random training images
dataiter = iter(trainloader)
images, labels = dataiter.next()
print ('labels:',labels)
# show images
imshow(torchvision.utils.make_grid(images))
# print labels
print(' '.join('%5s' % classes[labels[j]] for j in range(4)))
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1
labels: tensor([1, 1, 1, 0])
```

dog dog dog cat

modify last layer for 2 classes output

```
[20] import torch.nn as nn
     model.fc = nn.Linear(model.fc.in_features, 2)
```

define loss function and optimizer

```
[21] import torch.optim as optim

     criterion = nn.CrossEntropyLoss()
     optimizer = optim.SGD(model.parameters(), lr=0.001)
```

train the network with validation

```
[22] avgloss = []
     total = 0
     correct = 0
     vloss = []
     vcorrect = 0
     vtotal = 0
     for epoch in range(20):  # loop over the dataset multiple times
     # one epoch means run over whole data once
     ######################Train###################################
         running_loss = 0.0
         for i, data in enumerate(trainloader,0):
             #print(i)
             # get the inputs; data is a list of [inputs, labels]
             inputs, labels = data
             #print(labels)
             # zero the parameter gradients
             optimizer.zero_grad()

             # forward + backward + optimize
             outputs = model(inputs)

             _, predicted = torch.max(outputs, 1)
             #print(predicted)
             for j in range(4):
               total = total + 1
               if labels[j] == predicted[j]:
                 correct = correct + 1;
```

```
            #print(correct,'of',total, 'are correctly predicted')

            loss = criterion(outputs, labels)
            #print('every loss:',loss)
            loss.backward()
            optimizer.step()

            # print statistics
            running_loss += loss.item()
        # print every 15 mini-batches (15*4 = 16 = #pictures in one epoch)
        print(correct,'of',total, 'are correctly predicted')
        print('For epoch %d, every %5d mini-batches, average loss: %.3f' % (epoch + 1, 15, running_loss / 15))
        avgloss.append(running_loss / 15)
        running_loss = 0.0
    ####################Validation#################################
        with torch.no_grad():
            dataiter = iter(valloader)
            vimages, vlabels = dataiter.next()
            vimages, vlabels = data
            voutputs = model(vimages)
            VLOSS = criterion(voutputs, vlabels)
            vloss.append(VLOSS)
            _, vpredicted = torch.max(voutputs, 1)
            vtotal += vlabels.shape[0]
            vcorrect += (vpredicted == vlabels).sum().item()
            print('Validation loss: ',VLOSS.item(),'; ',vcorrect,'of',vtotal,'are correctly predicted')
```

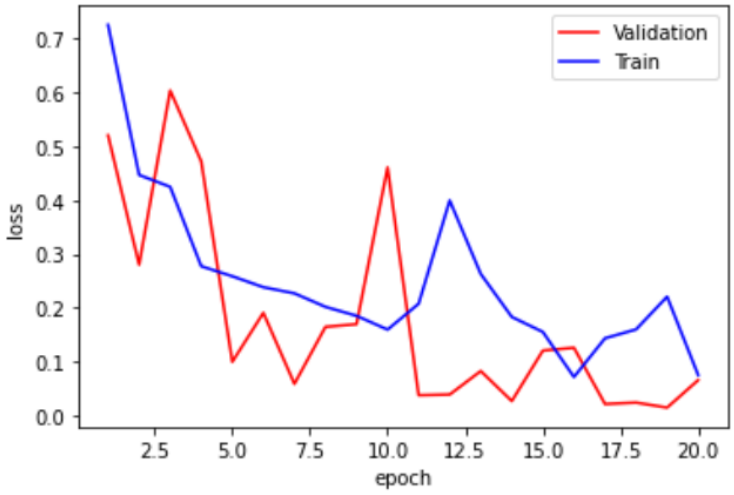
It will print the process like below. There is just a part of the process, it can be fully viewed in attached python file.

```
For epoch 14, every    15 mini-batches, average loss: 0.183
Validation loss:  0.026706388220191002 ;  53 of 56 are correctly predicted
803 of 900 are correctly predicted
For epoch 15, every    15 mini-batches, average loss: 0.155
Validation loss:  0.1204538494348526 ;  57 of 60 are correctly predicted
863 of 960 are correctly predicted
For epoch 16, every    15 mini-batches, average loss: 0.071
Validation loss:  0.12573614716529846 ;  61 of 64 are correctly predicted
921 of 1020 are correctly predicted
For epoch 17, every    15 mini-batches, average loss: 0.143
Validation loss:  0.021214816719293594 ;  65 of 68 are correctly predicted
978 of 1080 are correctly predicted
For epoch 18, every    15 mini-batches, average loss: 0.160
Validation loss:  0.02385798469185829 ;  69 of 72 are correctly predicted
1031 of 1140 are correctly predicted
For epoch 19, every    15 mini-batches, average loss: 0.221
Validation loss:  0.014515489339828491 ;  73 of 76 are correctly predicted
1091 of 1200 are correctly predicted
For epoch 20, every    15 mini-batches, average loss: 0.075
Validation loss:  0.0656055212020874 ;  77 of 80 are correctly predicted
```

Plot loss curves for train and validation

```
print('Finished Training')
print('Accuracy of the validation images: %d %%' % (100 * vcorrect / vtotal))
print('Accuracy of the the  test images: %d %%' % (100 * correct / total))
plt.figure()
vepoch = range(1,21)
plt.plot(vepoch,vloss,'r',label='Validation')
plt.legend()
epoch = range(1,21)
plt.plot(epoch,avgloss,'b',label='Train')
plt.legend()
plt.xlabel('epoch')
plt.ylabel('loss')
```

```
Finished Training
Accuracy of the validation images: 96 %
Accuracy of the the  test images: 90 %
Text(0, 0.5, 'loss')
```



Loss curves and accuracy are shown above. Accuracy of validation is 96%, Accuracy of training is 90% .

I also tried a test use all pictures in validation directory for curiosity and display accuracy for each class, it can be checked in python document attached.