Solutions 4

Jumping Rivers

```
knitr::opts_chunk$set(eval = FALSE)
```

A

Predict a person based on accelerometer data from walking

We have accelerometer data on 15 individuals who all walked for a period of time. Each observation is a set of values from the x,y and z axis of an accelerometer in 5 seconds sampled at a frequency of 52Hz. One sample is 260 observations. We have between 300-600 observations on each individual

```
import jrpytorch

walking = jrpytorch.datasets.load_walking()
```

The following code will produce a visualisation of one sample

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

sub = walking[walking['sample'] == 400]
sub.loc[:,'time'] = np.arange(260)
fig, (ax1,ax2,ax3) = plt.subplots(1,3, figsize = (18,10))
sub.plot(x = 'time', y = 'acc_x', ax = ax1)
sub.plot(x = 'time', y = 'acc_y', ax = ax2)
sub.plot(x = 'time', y = 'acc_z', ax = ax3)
plt.show()
```

At present this data is not in a particularly convenient structure for training my model. The following code will create the (n,260,3) (n observations of 260 inputs for 3 channels) input shape and class labels as seperate array objects

```
dims = ['acc_x','acc_y','acc_z']
x = np.dstack([walking[[d]].values.reshape(-1,260) for d in dims])
y = walking['person'].values[::260] - 1
```

Each observation is a sequence of 260 values with 3 channels. This is the sort of data structure where we would use a convolutional neural network with 1d convolutions.

Create a model structure that takes in the 260 input features for each of 3 data channels and returns 15 output features (one for each

person). We will want a set of convolution and pooling layers to begin with before having some linear layers to get to the final output. The convolutions and pooling extract features from the 3 channels of sequences, the linear layers then map those features to the final output.

```
import torch
import torch.nn as nn

class net(nn.Module):

    def __init__(self):
        super(net,self).__init__()
        self.conv1 = nn.Conv1d(3,40,30,stride=2)
        self.conv2 = nn.Conv1d(40,40,10)
        self.pool = nn.MaxPool1d(2)
        self.lin1 = nn.Linear(960,100)
        self.lin2 = nn.Linear(100,15)

    def forward(self,x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = torch.sigmoid(self.lin1(x.view(-1,24*40)))
        x = self.lin2(x)
        return x
```

partition your data into training and test sets

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(x,y, shuffle = True, test_size = 0.2)
```

pytorch expects a tensor shape of (samples, channels, inputdim) for convolutional nets. we can transpose the dimensions of our data to match this

```
X_train = X_train.transpose((0,2,1))
X_test = X_test.transpose((0,2,1))
```

Create a data loader to create batches from this data for training (Hint: `torch.utils.data.TensorDataset` and `torch.utils.data.DataLoader`)

```
import torch

X_train = torch.tensor(X_train).float()
X_test = torch.tensor(X_test).float()
y_train = torch.tensor(y_train).long()
```

```
y_test = torch.tensor(y_test).long()

trainset = torch.utils.data.TensorDataset(X_train,y_train)
trainloader = torch.utils.data.DataLoader(trainset,batch_size = 200)
testset = torch.utils.data.TensorDataset(X_test, y_test)
testloader = torch.utils.data.DataLoader(testset, batch_size = 20)
```

Train your model using the training set, if you want to you could keep track of the loss for the test set too.

```
loaders = {
    'train': trainloader,
    'test' : testloader
}

sizes = {
    'train': len(trainset),
    'test': len(testset)
}

def train(model,criterion,optimiser,epochs,device = 'cpu'):
    model = model.to(device)
    for epoch in range(epochs):
        print('Epoch %d / %d' %(epoch + 1, epochs))

        for stage in ['train','test']:
            if stage == 'train':
                model.train()
            else:
                model.eval()

            running_loss = .0
            running_correct = 0

            for input, label in loaders[stage]:
                input,label = input.to(device), label.to(device)
                optimiser.zero_grad()

                with torch.set_grad_enabled(stage == 'train'):
                    output = model(input)
                    _, pred = torch.max(output,1)
                    loss = criterion(output,label)

                    if stage == 'train':
                        loss.backward()
```

```
                optimiser.step()

            running_loss += loss.item() * input.size(0)
            running_correct += torch.sum(pred == label.data)

        stage_loss = running_loss/sizes[stage]
        stage_accuracy = running_correct.double() / sizes[stage]

        print('%s Loss: %.3f Acc: %.3f' % (stage,stage_loss, stage_accuracy))


model = net()
loss_fn = nn.CrossEntropyLoss()
optimiser = torch.optim.Adam(model.parameters())
train(model,loss_fn,optimiser,25)
```

For me this gives 95% plus accuracy on classifying a person purely on the accelerometer data while walking. Pretty neat!

B

Transfer learning

Some images of dogs and cats are available on github `https://github.com/jamieRowen/jrpytorch_data`

Download the images and use a pre trained model as a fixed feature extractor on this data. Try using data augmentation to obtain a model that does well for predicting objects from the images.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
import numpy as np
import torchvision
from torchvision import datasets, models, transforms
model = models.resnet18(pretrained=True)
# get the number of input features for the final layer
num_ftrs = model.fc.in_features
transform = torchvision.transforms.Compose(
  [
    torchvision.transforms.RandomResizedCrop(224),
    torchvision.transforms.RandomHorizontalFlip(),
    torchvision.transforms.ColorJitter(
      hue=.05, saturation=.05
```

```
    ),
    torchvision.transforms.RandomRotation(90),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize(
      (0.5, 0.5, 0.5),
      (0.5, 0.5, 0.5)
    )
  ]
)
test_transform = torchvision.transforms.Compose(
  [
    torchvision.transforms.Resize((224,224)),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize(
      (0.5, 0.5, 0.5),
      (0.5, 0.5, 0.5)
    )
  ]
)
trainset = torchvision.datasets.ImageFolder(
  root="./dogs_cats/training/", transform=transform
)
trainloader = torch.utils.data.DataLoader(
  trainset,batch_size=10,shuffle=True,num_workers=4
)
testset = torchvision.datasets.ImageFolder(
  root="./dogs_cats/testing/", transform=test_transform
)
testloader = torch.utils.data.DataLoader(
  testset,batch_size=10, shuffle=True
)
loaders = {
  'train': trainloader,
  'test': testloader
}
sizes = {
  'train': len(trainset),
  'test' : len(testset)
}

def train(
    model,criterion,optimiser,
    scheduler,epochs= 10,device= "cpu"
):
```

```
    for epoch in range(epochs):
        print('Epoch %d / %d' % (epoch+1,epochs))

        # we have both training and testing stages
        for stage in ['train','test']:
            if stage == 'train':
                scheduler.step()
                model.train()
            else:
                model.eval()

            running_loss = .0
            running_correct = 0

            # run through data
            for input,label in loaders[stage]:
                input,label = input.to(device), label.to(device)
                optimiser.zero_grad()

                # forward pass, only calculate gradients when training
                with torch.set_grad_enabled(stage == 'train'):
                    output = model(input)
                    _, pred = torch.max(output,1)
                    loss = criterion(output,label)

                    # only backward pass when training
                    if stage == 'train':
                        loss.backward()
                        optimiser.step()

                # update stats
                running_loss += loss.item() * input.size(0)
                running_correct += torch.sum(pred == label.data)

            # give stats for phase
            stage_loss = running_loss/sizes[stage]
            stage_accuracy = running_correct.double() / sizes[stage]

            print('%s Loss: %.3f Acc: %.3f' %
                (stage,stage_loss,stage_accuracy)
            )

for param in model.parameters():
  param.requires_grad = False
```

```
model.fc = nn.Linear(num_ftrs, 2)
device = torch.device(
  "cuda:0" if torch.cuda.is_available() else "cpu"
)
model.to(device)
loss = nn.CrossEntropyLoss()
# we are only optimising the final layer parameters
optimiser = torch.optim.Adam(
  model.fc.parameters(), lr = 0.1,weight_decay=0.1
)
scheduler = torch.optim.lr_scheduler.StepLR(
  optimiser, step_size = 2, gamma = .1
)
epochs = 25
train(model,loss,optimiser,scheduler, epochs,device)
```