Top Pytorch commands

```
# back-propagation step
optimizer.zero_grad() # clear last gradients
logits = model(images) # forward pass
loss = criterion(logits, labels) # calculate loss
loss.backward() # calculate gradients
optimizer.step() # update weight
# turn off gradients calculation and evaluation
model.eval()
with torch.no_grad():
    ...
# image transformation
transforms.ToTensor()
transforms.Normalize((0.5,),(0.5,))
transforms.Resize(255)
transforms.CenterCrop(224)
transforms.RandomRotation(30)
transforms.RandomResizedCrop(224)
transforms.RandomHorizontalFlip()
# load dataset from directory
data_dir = 'path/to/folder'
dataset = datasets.ImageFolder(data_dir, transform=transform)
dataloader = torch.utils.data.DataLoader(dataset, batch_size=32, shuffle=True)
for images, labels in dataloader:
    ...
# use GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device) # put everything to corresponding device for calculation
inputs, labels = inputs.to(device), labels.to(device)
```

Common Pytorch commands

```
# 5 random normal variables
features = torch.randn((1, 5))
# random weights with same size
torch.randn_like(features)
# set random seed
torch.manual_seed(7)
# torch operations
torch.sum()
torch.mm()
# check shape of tensor
tensor.shape
# change shape of tensor
weight.reshape(a,b) # copy data
weight.resize_(a,b) # same tensor different shape, remove elements if mismatch
weight.view() # new tensor with same data
# convert btw numpy and torch
b = torch.from_numpy(a)
b.numpy()
# torchvision related commands
transform = transforms.Compose([transforms.ToTensor(),transforms.Normalize((0.5,),
↪  (0.5,)),])
# download and load the training data
trainset = datasets.MNIST('~/.pytorch/MNIST_data/', download=True, train=True,
↪  transform=transform)
# load training set to dataloader
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)
# show image quickly
plt.imshow(images[1].numpy().squeeze(), cmap='Greys_r');
# flatten the 2D images
inputs = images.view(images.shape[0], -1)
# create softmax function
def softmax(x):
    return torch.exp(x)/torch.sum(torch.exp(x), dim=1).view(-1, 1)
# reshape the torch to (64,1)
view(-1, 1)
# sum at specific dimension
probabilities.sum(dim=1)
```

Structure of a network

```
class Network(nn.Module):
    def __init__(self): # define here
        super().__init__()
        # hidden layer
        self.hidden = nn.Linear(784, 256)
        # output layer
        self.output = nn.Linear(256, 10)
        # activation and softmax
        self.sigmoid = nn.Sigmoid()
        self.softmax = nn.Softmax(dim=1)
    def forward(self, x):
        # stack the layers together
        x = self.hidden(x)
        x = self.sigmoid(x)
        x = self.output(x)
        x = self.softmax(x)
        return x
# or use built-in activation function
import torch.nn.functional as F
class Network(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden = nn.Linear(784, 256)
        self.output = nn.Linear(256, 10)
    def forward(self, x):
        x = F.sigmoid(self.hidden(x))
        x = F.softmax(self.output(x), dim=1)
        return x
# or use sequential to build feed forward nn
model = nn.Sequential(nn.Linear(input_size, hidden_sizes[0]),
            nn.ReLU(),
            nn.Linear(hidden_sizes[0],hidden_sizes[1]),
            nn.ReLU(),
            nn.Linear(hidden_sizes[1], output_size),
            nn.Softmax(dim=1))
```

Common commands continue

```
# set biases to all zeros
model.fc1.bias.data.fill_(0)
# sample from random normal with standard dev = 0.01
model.fc1.weight.data.normal_(std=0.01)
# get next iteration of data
dataiter = iter(trainloader)
images, labels = dataiter.next()
# forward pass
ps = model.forward()
# view the probability
helper.view_classify(img.view(1, 28, 28), ps)
```

Backpropagation

```
# define loss
criterion = nn.CrossEntropyLoss()
from torch import optim
# define optimizer
optimizer = optim.SGD(model.parameters(), lr=0.01)
# dont forget
optimizer.zero_grad()
# forward pass
logits = model(images)
# calculate loss and backpropagate
loss = criterion(logits, labels)
loss.backward()
# update weight
optimizer.step()
```

Training Example

```
1st step: define model
2nd step: define loss and optimizer (if use logsoftmax, then use NLL loss)
epochs = 5
for e in range(epochs):
    running_loss = 0
    for images, labels in trainloader:
        # flatten
        images = images.view(images.shape[0], -1)
        # zero grad
        optimizer.zero_grad()
        # forward
        output = model(images)
        loss = criterion(output, labels)
        # backward
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    else:
        # print running loss
        print(f"Training loss: {running_loss/len(trainloader)}")
```

Testing Example

```
model.eval()
# flatten
# img = images[0].view(1, 784)
dataiter = iter(testloader)
images, labels = dataiter.next()
img = images[0]
# turn of auto grad to test
with torch.no_grad():
    logps = model.forward(img) # model(img)
# if use logsoftmax
ps = torch.exp(logps)
# then view classified
```

Common commands continue

```
# check grad function
y.grad_fn
# check most likely classes
top_p, top_class = ps.topk(1, dim=1)
# check if predicted label correct
equals = top_class == labels.view(*top_class.shape)
```

Validation Example

```
test_loss = 0
accuracy = 0
# turn off gradient
with torch.no_grad():
        for images, labels in testloader:
            log_ps = model(images)
            test_loss += criterion(log_ps, labels)
            ps = torch.exp(log_ps)
            top_p, top_class = ps.topk(1, dim=1)
            equals = top_class == labels.view(*top_class.shape)
            accuracy += torch.mean(equals.type(torch.FloatTensor))
        train_losses.append(running_loss/len(trainloader))
        test_losses.append(test_loss/len(testloader))
        print("Epoch: {}/{}.. ".format(e+1, epochs),
              "Training Loss: {:.3f}.. ".format(running_loss/len(trainloader)),
              "Test Loss: {:.3f}.. ".format(test_loss/len(testloader)),
              "Test Accuracy: {:.3f}".format(accuracy/len(testloader)))
# plot train loss and test loss
plt.plot(train_losses, label='Training loss')
plt.plot(test_losses, label='Validation loss')
plt.legend(frameon=False)
```

Sample network with dropout

```
class Classifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 64)
        self.fc4 = nn.Linear(64, 10)
        # dropout p=0.2
        self.dropout = nn.Dropout(p=0.2)
    def forward(self, x):
        x = x.view(x.shape[0], -1)
        x = self.dropout(F.relu(self.fc1(x)))
        x = self.dropout(F.relu(self.fc2(x)))
        x = self.dropout(F.relu(self.fc3(x)))
        x = F.log_softmax(self.fc4(x), dim=1)
        return x
```

Save and load networks

```
import fc_model
print("The state dict keys: \n\n", model.state_dict().keys()) # check model
# save the architecture and trained model
checkpoint = {'input_size': 784,
              'output_size': 10,
              'hidden_layers': [each.out_features for each in model.hidden_layers],
              'state_dict': model.state_dict()}
torch.save(checkpoint, 'checkpoint.pth')
def load_checkpoint(filepath):
    checkpoint = torch.load(filepath)
    model = fc_model.Network(checkpoint['input_size'],
                             checkpoint['output_size'],
                             checkpoint['hidden_layers'])
    model.load_state_dict(checkpoint['state_dict'])
    return model
model = load_checkpoint('checkpoint.pth')
```

Common image transform

```
transforms.ToTensor()
transforms.Normalize((0.5,),(0.5,))
transforms.Resize(255)
transforms.CenterCrop(224)
transforms.RandomRotation(30)
transforms.RandomResizedCrop(224)
transforms.RandomHorizontalFlip()
```

Loading image from folder

```
data_dir = 'Cat_Dog_data/train'
# define transform
# load dataset from directory
dataset = datasets.ImageFolder(data_dir, transform=transform)
dataloader = torch.utils.data.DataLoader(dataset, batch_size=32, shuffle=True)
```

Common commands continue

```
# load pretrained model
model = models.densenet121(pretrained=True)
# use gpu maybe
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# freeze parameters of pretrained model
for param in model.parameters():
    param.requires_grad = False
# define a classifier
model.classifier = ...
model.to(device)
inputs, labels = inputs.to(device), labels.to(device)
```