COMP 6721 Applied Artificial Intelligence (Winter 2021)

Lab Exercise #07: Introduction to Deep Learning

PyTorch is a deep learning research platform that was designed for maximum flexibility and speed.[1] To gain a basic understanding on how to implement an Artificial Neural Network using the PyTorch library, in the following questions you will implement a simple MLP and a convolutional neural network for a specific image classification task.

Question 1 Let's use PyTorch to implement a multi-layer perceptron for classifying the CIFAR10 dataset (see Figure 1).[2] The `torchvision` package[3] provides data loaders for common datasets such as Imagenet, CIFAR10, MNIST, etc.

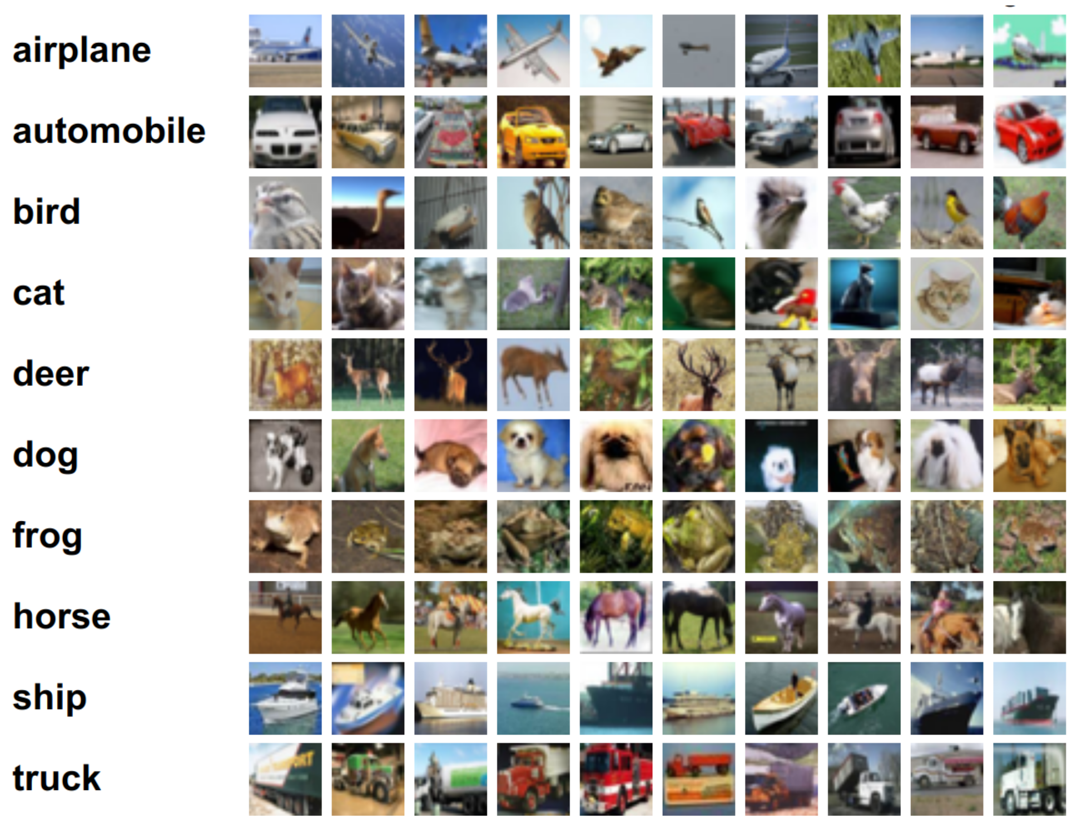


Figure 1: Some example images from the CIFAR-10 dataset

First, use the following code block, which provides Python imports and the `cifar_loader` function as a helper function to load the CIFAR-10 dataset.

[1] See https://pytorch.org/docs/stable/index.html
[2] For details on CIFAR10, see https://en.wikipedia.org/wiki/CIFAR-10
[3] https://pytorch.org/docs/stable/torchvision/index.html

This function returns the train and the test data loaders that can be used as an iterator, so to extract the data, we can use the standard Python iterators such as enumerate. After setting the hyper-parameters, where `H` is the hidden dimension and `D_out` is the output dimension, the dataset is loaded.

```python
import torch
import torchvision.transforms as transforms
import torchvision.datasets as datasets
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data as td

def cifar_loader(batch_size, shuffle_test=False):
   normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                           std=[0.225, 0.225, 0.225])
   train = datasets.CIFAR10('./data', train=True, download=True,
      transform=transforms.Compose([transforms.RandomHorizontalFlip(),
      transforms.RandomCrop(32, 4), transforms.ToTensor(),normalize]))
   test = datasets.CIFAR10('./data', train=False,
      transform=transforms.Compose([transforms.ToTensor(), normalize]))
   train_loader = torch.utils.data.DataLoader(train, batch_size=batch_size,
      shuffle=True, pin_memory=True)
   test_loader = torch.utils.data.DataLoader(test, batch_size=batch_size,
      shuffle=shuffle_test, pin_memory=True)
   return train_loader, test_loader

batch_size = 64
test_batch_size = 64
input_size = 3072

N = batch_size
D_in = input_size
H = 50
D_out = 10
num_epochs = 10

train_loader, _ = cifar_loader(batch_size)
_, test_loader = cifar_loader(test_batch_size)
```

(a) This is where the model definition takes place. The most straightforward way of creating a neural network structure in PyTorch is by creating a class inheriting from the `nn.Module`[4] superclass within PyTorch. The `nn.Module` is a very useful PyTorch class that contains all you need to construct typical deep learning networks. Define the `MultiLayerFCNet` class, which is a four-layer, fully connected network. Use the hyper-parameter `H` as the number

[4] https://pytorch.org/docs/stable/generated/torch.nn.Module.html

of hidden units for all hidden layers. For the activation function, apply ReLU to the hidden layers and use `log_softmax` in the output.

(b) Now use PyTorch's `CrossEntropyLoss`[5] to construct the loss function and define an optimizer using `torch.optim`.[6] The first argument passed to the optimizer function are the parameters we want the optimizer to train. All you have to do is pass `model.parameters()` to the function, and PyTorch keeps track of all the parameters within the model, which are required to be trained.:

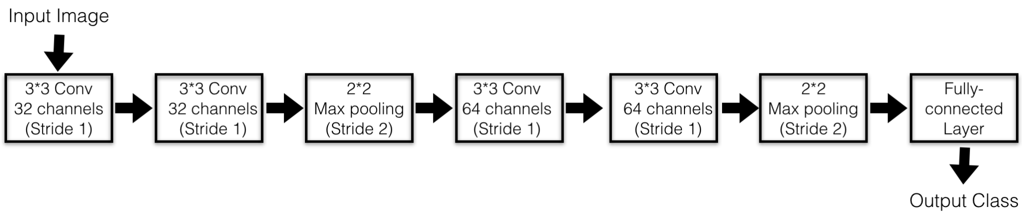
```
model = MultiLayerFCNet(D_in, H, D_out)

criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
```

Next, loop over the number of epochs and within this loop, pass the model outputs and the true labels to the `CrossEntropyLoss` function, defined as `criterion`. Then perform back-propagation and an optimized training. Call `backward()` on the loss variable to perform the back-propagation. Now that the gradients have been calculated in the back-propagation, call `optimizer.step()` to perform the optimizer training step.

(c) Finally, we need to keep track of the *accuracy* on the test set. The predictions of the model can be determined by using the `torch.max()`[7] function, which returns the index of the maximum value in a tensor. The first argument to this function is the tensor to be examined, and the second argument is the axis over which to determine the index of the maximum. Report the accuracy on the test set using the `torch.max()` function.

[5] https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html
[6] https://pytorch.org/docs/stable/optim.html
[7] https://pytorch.org/docs/stable/generated/torch.max.html

Question 2 To improve the performance for image classification, we will use PyTorch to implement more complicated, *deep learning* networks. In this question, you will implement a convolutional neural network (CNN) step-by-step to classify the CIFAR-10 dataset. The CNN architecture that we are going to build can be seen in the diagram below:



(a) First, use the following code block, which provides the Python imports, the `cifar_loader` to load the dataset, and the hyper-parameters definition:

```python
from torch.utils.data import DataLoader
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torchvision.datasets

num_epochs = 4
num_classes = 10
learning_rate = 0.001

transform = transforms.Compose(
   [transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                              download=True, transform=transform)
train_loader = torch.utils.data.DataLoader(trainset, batch_size=32,
                                shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                             download=True, transform=transform)
test_loader = torch.utils.data.DataLoader(testset, batch_size=1000,
                               shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
        'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

(b) Now create a class inheriting from the `nn.Module` to define different layers of the network based on provided network architecture above. The first step is to use the `nn.Sequential` module[8] to create sequentially ordered

[8] https://pytorch.org/docs/stable/generated/torch.nn.Sequential.html

layers in the network. It's a handy way of creating a convolution + ReLU + pooling sequence. In each convolution layer, use `LeakyRelu` for the activation function and `BatchNorm2d`[9] to accelerate the training process.

(c) Before training the model, you have to first create an instance of the `Convolution` class you defined in previous part, then define the loss function and optimizer:

```
model = CNN()

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

The following steps are similar to what you've done in previous questions: Loop over the number of epochs and within this loop, pass the model outputs and true labels to the `CrossEntropyLoss` function, defined as `criterion`. Then, perform back-propagation and an optimized training. Call `backward()` on the loss variable to perform the back-propagation. Now that the gradients have been calculated in the back-propagation, call `optimizer.step()` to perform the optimizer training step.

(d) Now keep track of the accuracy on the test set. The predictions of the model can be determined by using `torch.max()`.[10]

[9] https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm2d.html#torch.nn.BatchNorm2d

[10] https://pytorch.org/docs/stable/generated/torch.max.html