COMP 6721 Applied Artificial Intelligence (Summer 2023)

Assignment #08: Introduction to Deep Learning and 09: CNNs and Optimization

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.

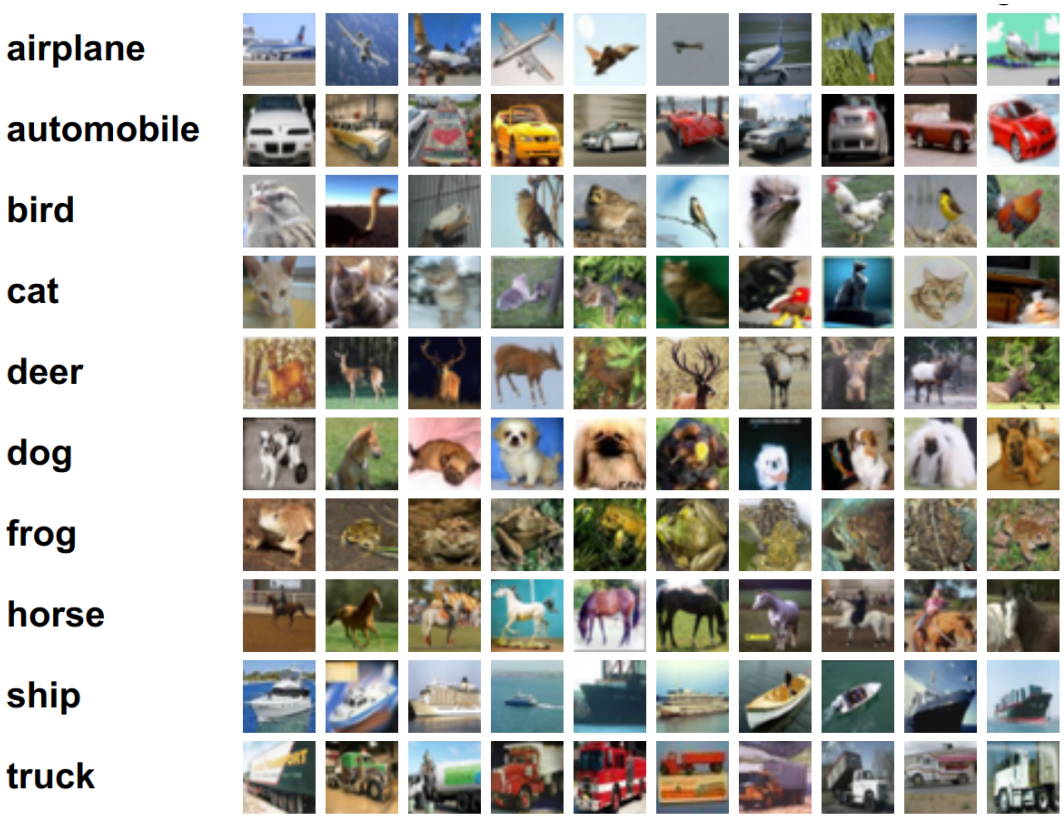


Figure 1: Some example images from the CIFAR-10 dataset

[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

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. 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.

```
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

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

deep learning networks. Define the `MultiLayerFCNet` class, which is a four-layer, fully connected network. Use the hyper-parameter `H` as the number 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 Assume you have a 4x3 one-channel (BW) image with the following pixel intensities. A 2x2 filter is applied to the image followed by a ReLu (the activation function of the convolutional layer is ReLu). Find the resulting activation map (output volume) before and after pooling when

- No padding is applied followed by 2x2 max-pooling
- 1 pixel padding is applied, followed by 2x2 average pooling

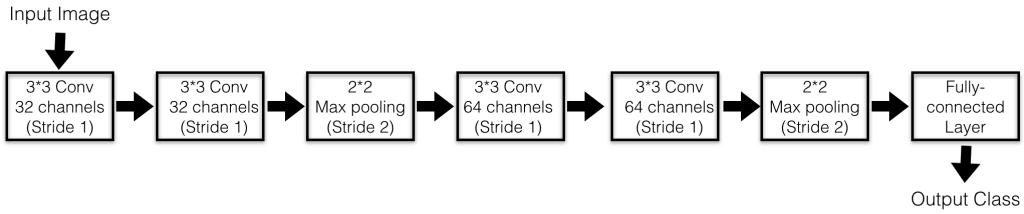
-1	0	1
2	3	4
5	6	7
8	9	10

Table 1: 4x3 image

2	1
0	-1

Table 2: 2x2 filter (kernel)

Question 3 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:

```
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]

(e) In PyTorch, the learnable parameters (i.e., weights and biases) of a model are contained in the model's parameters. A `state_dict` is simply a Python dictionary object that maps each layer to its parameter tensor. Because `state_dict` objects are Python dictionaries, they can be easily saved, updated, altered, and restored.[11]

Saving the model's `state_dict` with the `torch.save()` function will give you the most flexibility for restoring the model later. To load the models, first initialize the models and optimizers, then load the dictionary locally using `torch.load()`.

Inspect the PyTorch documentation to understand how you can use these functionalities.

[9] https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm2d.html#torch.nn.BatchNorm2d
[10] https://pytorch.org/docs/stable/generated/torch.max.html
[11] https://pytorch.org/tutorials/beginner/saving_loading_models.html

Question 4 The goal of `skorch`[12] is to make it possible to use PyTorch with scikit-learn. This is achieved by providing a wrapper around PyTorch that has a scikit-learn interface. Additionally, `skorch` abstracts away the training loop, a simple `net.fit(X, y)` is enough. It also takes advantage of scikit-learn functions, such as `predict`.

(a) In this section, we will train the same CNN model you developed in the previous question using `skorch` with fewer lines of code.
Let's install `skorch` first:

```
pip install skorch
```

Once the library is installed, we can import the libraries. The next step is to prepare the dataset before training.

```
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.datasets
import torchvision.transforms as transforms
from sklearn.metrics import accuracy_score
from sklearn.metrics import plot_confusion_matrix
from skorch import NeuralNetClassifier
from torch.utils.data import random_split
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)
testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                               download=True, transform=transform)
m = len(trainset)
train_data, val_data = random_split(trainset, [int(m - m * 0.2), int(m * 0.2)])
DEVICE = torch.device("cpu")
y_train = np.array([y for x, y in iter(train_data)])
classes = ('plane', 'car', 'bird', 'cat',
        'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

`skorch.NeuralNetClassifier`[13] is a Neural Network class used for classification tasks. Initialize the NeuralNetClassifier class then train the CNN

[12] https://skorch.readthedocs.io/en/stable/index.html
[13] https://skorch.readthedocs.io/en/stable/classifier.html

model using the method fit. Finally print the accuracy score and confusion matrix. Note that CNN is the model you already developed from the previous question using torch.

(b) Now let's evaluate the score using K-fold cross-validation. Before the evaluation, we need to make the training set *sliceable* using the class `SliceDataset`, which wraps a torch dataset to make it work with scikit-learn. Use the function `cross_val_score(estimator, X, y=None, cv=None)` of the scikit-learn library to evaluate the validation accuracy obtained using $K = 5$ folds for K-fold cross-validation.

To achieve this, replace the training process from the previous section with a K-fold cross-validation.

Question 5 As discussed in the course, one of the techniques that helped us to mitigate the drawback of need for lots of labeled data in Deep Learning (DL) was *Transfer Learning*. In Transfer learning, we use a pretrained model so that the "optimal" encoder weights have already been determined, and we may just do further fine tuning using our own data.

(a) Fine Tuning: Pretrained weights are not frozen, but they change with a very low learning rate when we train the model with our data. We expect to converge faster compared to training from scratch.

(b) Freezing the rates: We fix the pretrained rates, but we may add a few new layers or replace the few last layers with ours and train the whole model using our data. This approach could be considered "less adaptive" but faster than updating all weights.

The PyTorch tutorial can be followed here https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html. We use *resnet18* pretrained model on the CIFAR10 dataset.

Steps to load the data are the same as before (We did in the first question).

```
data_transforms = {
   'train': transforms.Compose([
      transforms.RandomResizedCrop(224),
      transforms.RandomHorizontalFlip(),
      transforms.ToTensor(),
      transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
   ]),
   'val': transforms.Compose([
      transforms.Resize(256),
      transforms.CenterCrop(224),
      transforms.ToTensor(),
      transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
   ]),
}

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

dataloaders={'train':train_loader, 'val':test_loader}

dataset_sizes = {'train': len(trainset),'val': len(testset) }
```

```
print(dataset_sizes)

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

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

However, the model that will be passed to the training will change as follows:

```
from torchvision import models
model_ft = models.resnet18(weights='IMAGENET1K_V1')
num_ftrs = model_ft.fc.in_features

# Alternatively, it can be generalized to ``nn.Linear(num_ftrs, len(class_names))``.
model_ft.fc = nn.Linear(num_ftrs, len(class_names))

model_ft = model_ft.to(device)
```

Question 6 As discussed in the lecture, the main hyperparameters in a deep neural network are architectural hyperparameters such as the number of hidden layers and the number of neurons per layer.

There are other parameters that affect the performance and speed of training such as the number of epochs, batch size and the initial learning rate (or the way that it is updated). The choice of optimizer can also be considered a parameter significantly affecting the speed of training, as different optimizers may have different performance rates, depending on the type of data and structure of the network. There is no optimizer that is necessarily consistently superior to all the other optimizers for every problem.

Every optimizer may have its own parameters that need further adjustments, so it is difficult to compare several optimizers even for the same problem. However, we want to simply compare three known optimizers for the problem we saw in Question 1.

For Question 1, how can you change the optimizer and try others such as 'Adam' and 'RMSprop'?