COMP 6721 Applied Artificial Intelligence (Fall 2023)

Lab Exercise #07: Introduction to Deep Learning

PyTorch is a deep learning research platform designed for maximum flexibility and speed.[1] While scikit-learn offers user-friendly tools for a wide range of machine learning algorithms, focusing mainly on traditional methods, PyTorch caters specifically to deep learning. It provides a dynamic environment that allows for intricate model designs and optimizations. To gain a basic understanding of how to implement an Artificial Neural Network using the PyTorch library, in the subsequent questions, you will implement both a simple MLP and a convolutional neural network for a specific image classification task.

Installation. To set up the necessary environment using `conda`, follow the steps below.

1. First, create a new conda environment. This helps in maintaining a clean workspace and avoids conflicts with other packages. Open your terminal or Anaconda prompt and run:

```
conda create --name pytorch_env python=3.8
```

2. Activate the environment:

```
conda activate pytorch_env
```

3. Install PyTorch and torchvision using the official channel:[2]

```
conda install pytorch torchvision -c pytorch
```

4. Lastly, ensure you have other required libraries such as `matplotlib` for visualization, if not included in your current environment:

```
conda install matplotlib
```

With these steps completed, you should have a working conda environment named `pytorch_env` with all the necessary libraries installed.

[1] See https://pytorch.org/docs/stable/index.html
[2] See https://pytorch.org/get-started/locally/ for all options

Question 1 Let's use PyTorch to implement a multi-layer perceptron for classifying the CIFAR10 dataset (see Figure 1).[3] The `torchvision` package[4] provides data loaders for common datasets such as Imagenet, CIFAR10, MNIST, etc.

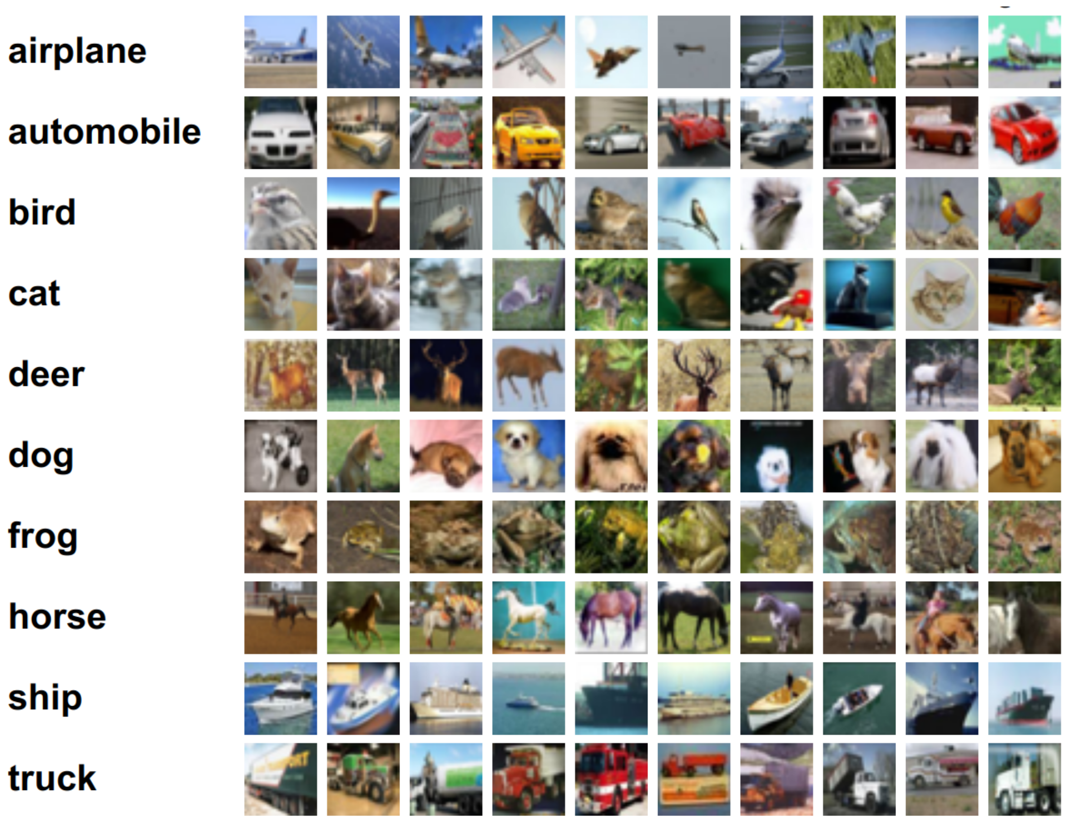


Figure 1: Some example images from the CIFAR-10 dataset

First, utilize the provided code block below, which contains essential Python imports and the `cifar_loader` function, a utility to load the CIFAR-10 dataset. Each CIFAR-10 image is a 32×32 RGB image, giving an input size of $3 \times 32 \times 32 = 3072$. The `cifar_loader` provides train and test data loaders that can be used as iterators. To retrieve the data, standard Python iterators like `enumerate` can be employed. The training dataset uses data augmentation techniques, specifically `RandomHorizontalFlip` and `RandomCrop`, to artificially expand the dataset.[5] Such augmentations introduce variability, making the model more robust and less prone to overfitting on the training data. After setting the hyper-parameters, where `hidden_size` specifies the hidden dimension and `output_size` represents the output dimension, the dataset is loaded.

[3] For details on CIFAR10, see https://en.wikipedia.org/wiki/CIFAR-10

[4] https://pytorch.org/docs/stable/torchvision/index.html

[5] The `RandomHorizontalFlip` augmentation mirrors images, simulating the appearance of objects from different orientations. The `RandomCrop` augmentation helps in focusing on various parts of an image, teaching the model to recognize features irrespective of their position.

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

import torchvision.transforms as transforms
import torchvision.datasets as datasets


# Function to load CIFAR10 dataset
def cifar_loader(batch_size, shuffle_test=False):
   # Normalization values for CIFAR10 dataset
   normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                    std=[0.225, 0.225, 0.225])

   # Loading training dataset with data augmentation techniques
   train_dataset = datasets.CIFAR10('./data', train=True, download=True,
                                    transform=transforms.Compose([
                                       transforms.RandomHorizontalFlip(),
                                       transforms.RandomCrop(32, 4),
                                       transforms.ToTensor(),
                                       normalize
                                    ]))

   # Loading test dataset
   test_dataset = datasets.CIFAR10('./data', train=False,
                                   transform=transforms.Compose([
                                      transforms.ToTensor(),
                                      normalize
                                   ]))

   # Creating data loaders for training and testing
   train_loader = td.DataLoader(train_dataset, batch_size=batch_size,
                                shuffle=True, pin_memory=True)
   test_loader = td.DataLoader(test_dataset, batch_size=batch_size,
                               shuffle=shuffle_test, pin_memory=True)

   return train_loader, test_loader

# Hyperparameters and settings
batch_size = 64
test_batch_size = 64
input_size = 3 * 32 * 32 # 3 channels, 32x32 image size
hidden_size = 50 # Number of hidden units
output_size = 10 # Number of output classes (CIFAR-10 has 10 classes)
num_epochs = 10
```

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

(a) This is the stage where you'll define the model. You've previously worked with scikit-learn's MLP, so think of this step as specifying the architecture of the neural network. In PyTorch, the preferred method to create a neural network is to define a class that inherits from the `nn.Module`[6] superclass. The `nn.Module` class in PyTorch acts as a blueprint that offers functionalities essential for building a variety of deep learning models.

For this exercise, you'll be defining the `MultiLayerFCNet` class, representing a four-layer, fully connected network:

```
model = MultiLayerFCNet(input_size, hidden_size, output_size)
```

This will include an input layer, two hidden layers, and an output layer. Use the hyper-parameter `hidden_size` to specify the number of neurons in each hidden layer. As for the activation functions, employ the ReLU (Rectified Linear Unit) activation for the hidden layers. For the output layer, leverage the `log_softmax` function, which is especially suitable for multi-class classification tasks. The `log_softmax` function provides the logarithm of the softmax values, which, when combined with the negative log likelihood loss, can be used to train models efficiently for classification tasks in PyTorch.

(b) Now use PyTorch's `CrossEntropyLoss`[7] to construct the loss function. This loss is particularly suited for classification problems where the task is to predict one out of multiple classes.

Define an optimizer using `torch.optim`[8]. The optimizer is responsible for updating the weights of our neural network based on the gradients computed during back-propagation. Specifically, we will be using the *Stochastic Gradient Descent (SGD)* optimizer. The learning rate (`lr`) controls the step size during optimization, and `momentum` helps accelerate gradients vectors in the right directions, leading to faster converging.

The first argument passed to the optimizer function contains the parameters we want the optimizer to train. By passing `model.parameters()` to the function, PyTorch effortlessly tracks all the parameters within the model that require training:

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

Next, loop over the number of epochs. Within this loop, pass the model outputs and the true labels to the `CrossEntropyLoss` function, defined as

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

`criterion`. Back-propagation is then performed to compute the gradient of the loss with respect to the model parameters. Call `backward()` on the loss variable to execute this. After calculating the gradients using back-propagation, invoke `optimizer.step()` to perform the optimizer's weight update step.

(c) Finally, we need to monitor the *accuracy* on the test set. To determine the model's predictions, we can use the `torch.max()`[9] function. This function returns the index of the maximum value in a tensor (an array-like structure). In the context of classification, this index corresponds to the predicted class label. When using `torch.max()`, the first argument should be the output from the model you're examining, and the second argument specifies the dimension over which to find the maximum value. Your task is to report the accuracy on the test set by comparing the predicted class labels (obtained using `torch.max()`) to the actual labels.

After running the accuracy calculation code, you might get a result like this:

```
Accuracy of the network on the 10000 test images: 45 %
```

Is this what you'd expect, given our approach?

(d) *Bonus visualization:* If you want to print some random images from the test set with their classification, you can add the code below to your program:

```python
# Plot some example images with their predicted labels
import numpy as np
import matplotlib.pyplot as plt

def denormalize(tensor, mean, std):
   for t, m, s in zip(tensor, mean, std):
      t.mul_(s).add_(m)
   return tensor

def plot_results(x=2, y=5):
   # Width per image (inches)
   width_per_image = 2.4

   # Get a batch of images and labels
   data_iter = iter(test_loader)
   images, labels = next(data_iter)

   # Select x*y random images from the batch
   indices = np.random.choice(images.size(0), x*y, replace=False)
   random_images = images[indices]
   random_labels = labels[indices]
```

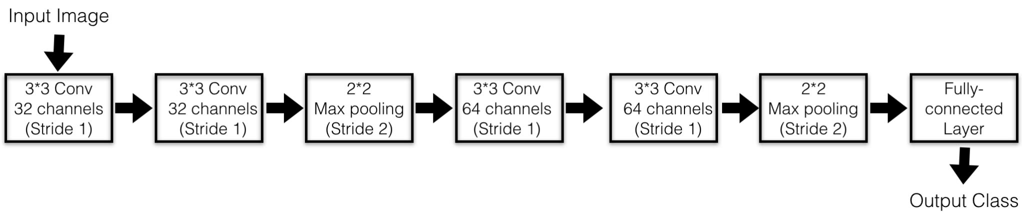
[9] https://pytorch.org/docs/stable/generated/torch.max.html

```
    # Get predictions for these images
    random_images_reshaped = random_images.reshape(-1, 3 * 32 * 32)
    outputs = model(random_images_reshaped)
    _, predicted = torch.max(outputs.data, 1)

    # Fetch class names from CIFAR10
    classes = test_loader.dataset.classes

    fig, axes = plt.subplots(x, y, figsize=(y * width_per_image, x *
        ↪ width_per_image))

    # Iterate over the random images and
    # display them along with their predicted labels
    for i, ax in enumerate(axes.ravel()):
        # Denormalize image
        img = denormalize(random_images[i], [0.485, 0.456, 0.406],
            ↪ [0.225, 0.225, 0.225])
        img = img.permute(1, 2, 0).numpy() # Convert image from CxHxW to
            ↪ HxWxC format for plotting

        true_label = classes[random_labels[i]]
        pred_label = classes[predicted[i]]

        ax.imshow(img)
        ax.set_title(f"true='{true_label}', pred='{pred_label}'",
            ↪ fontsize=8)
        ax.axis('off')

    plt.tight_layout()
    plt.show()

# Call with optional x, y values
plot_results()
```

This will give you an output similar to the one shown in Figure 2.

Figure 2: Some classification results on the CIFAR10 data

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:

```
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[10] to create sequentially ordered

[10] 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`[11] 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()`.[12]

(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.[13]

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.

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

Question 3 The goal of `skorch`[14] 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`[15] is a Neural Network class used for classification tasks. Initialize the NeuralNetClassifier class then train the CNN

[14] https://skorch.readthedocs.io/en/stable/index.html
[15] 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.