Cheyenne Hwang

Data Mining

Professor Huajie (Jay) Shao

Assignment 3: Neural Networks

```
!pip3 install torch
```

```
!pip install wandb
```

- Pip install torch is used to install PyTorch library in our google colab notebook, an open source machine learning library used for neural networks
- Pip install wandb is used to install the weights and biases library in python, a tool that helps with experiment tracking, and visualization

```
'''Import packages'''

import numpy as np
import time
import argparse
import os.path
from pathlib import Path
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.optim as optim
import torchvision
import torch.nn as nn
import wandb ##weight and bias
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
```

- Import numpy as np : numerical computations in python
- Import time: time related functions
- Import argparse: parse command-line arguments for python script
- Import os.path: functions for working with file paths and directories

- From pathlib import path: object oriented approach to working with file paths and directories
- Import torch: import pytorch library
- From torch.autograd import Variable: imports variable class from torch.autograd which allows pytorch automation differentiation
- Import torch.nn as nn: imports nn module from Pytorch which contains neural network layers, loss functions
- Import torch.optim as optim: imports the optim module from python, which provides optimization algorithms like SGD for training neural networks
- Transforms: image transformation
- DataLoader: create data loaders for batching and iterating over datasets

```python
#  Part 1 - Loads in data from DATA_PATH variable using se the data loader about MNIST from Pytorch libs
def _load_data(DATA_PATH, batch_size):
    ## for training
    rotation = 15
    train_trans = transforms.Compose([transforms.RandomRotation(rotation),\
                                      transforms.RandomHorizontalFlip(),\
                                      transforms.ToTensor(),\
                                      transforms.Normalize((0.5), (0.5))])
    train_dataset = torchvision.datasets.MNIST(root=DATA_PATH, download=True,\
                                               train=True, transform=train_trans)
    train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size,\
                              shuffle=True, num_workers=0)
    ## for testing
    test_trans = transforms.Compose([transforms.ToTensor(),\
                                     transforms.Normalize((0.5), (0.5))])
    test_dataset = torchvision.datasets.MNIST(root=DATA_PATH,\
                                              download=True, train=False, transform=test_trans)
    test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size,\
                             shuffle=False, num_workers=0)

    return train_loader, test_loader
```

This function "_load_data" takes in data from a specified 'DATA_PATH' using Pytorch's Dataloader for the MNIST dataset. For the training dataset, transforms.Compose allows for chaining multiple transformations together, rotation allows for a rotation up to 15 degrees, randomHorizontalFlip() randomly flips the images horizontally, ToTensor() converts the images to PyTorch tensors. transforms.Normalize(0.5, 0.5) normalizes the tensor data to have a mean of 0.5 and a standard deviation of 0.5. For the testing data, transforms.Compose is used again, and then ToTensor and Normalize is applied to it. torchvision.datasets.MNIST is used to load the MNIST dataset by specifying a directory path, if it's downloaded, which one is the training set, which one is the testing set, and what transformations will be performed. Train_loader and test_loader was created to manage the training and testing datasets, and then returns those objects containing the batches of training and testing data.

```python
# test the 3 layers and 2 layers
class MLPModel(nn.Module):
    def __init__(self):
        super(MLPModel, self).__init__()
        # Convolutional layers
        self.conv1 = nn.Conv2d(1, 4, kernel_size=3, stride=1, padding=1)  # 4 output channels
        self.relu1 = nn.ReLU()
        # Reduce spatial dimensions
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        # Dropout after the first convolutional layer
        self.dropout1 = nn.Dropout(0.2)

        # 8 output channels
        self.conv2 = nn.Conv2d(4, 8, kernel_size=3, stride=1, padding=1)
        self.relu2 = nn.ReLU()
         # Reduce spatial dimensions
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
         # Dropout after the second convolutional layer
        self.dropout2 = nn.Dropout(0.2)

        # Calculate the number of features after convolutional and pooling operations
        self.num_features = 8 * 7 * 7  # Adjusted based on the last pooling layer output size

        # Fully connected layers

        # Reduced number of units in the fully connected layer
        self.fc1 = nn.Linear(self.num_features, 32)
        self.relu3 = nn.ReLU()
        # Dropout after the first fully connected layer
        self.dropout3 = nn.Dropout(0.5)

        self.fc2 = nn.Linear(32, 10)  # Output layer with 10 classes
```

```python
    def forward(self, x):
        # Input shape: (batch_size, 1, 28, 28)
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.pool1(x)

        # Apply dropout after the first convolutional layer
        x = self.dropout1(x)

        x = self.conv2(x)
        x = self.relu2(x)
        x = self.pool2(x)

        # Apply dropout after the second convolutional layer
        x = self.dropout2(x)

        # Flatten the output of conv layers and reshape to (batch_size, num_features)
        x = torch.flatten(x, 1)

        # Fully connected layers
        x = self.fc1(x)
        x = self.relu3(x)

        # Apply dropout after the first fully connected layer
        x = self.dropout3(x)

        x = self.fc2(x)

        return x
```

The code above is the neural network model named MLPModel for the image classification. The self.relu1 = nn.ReLu() applies the ReLu activation function after the first convolution layer with 1 input channel and 4 output channels. This model contains two convolution layers and then two connected layers. Each channel represents a different learned feature that is extracted from the input. The kernel size defines the region, which then the convolution operation is performed. Padding of 1 makes sure that the output size matches the input size. The ReLu function is applied to create non linearity, max pooling, and reduces spatial dimensions by taking in the maximum value of each pooling region, and dropout. Dropout helps with overfitting by randomly setting a fraction of inputs to zero during training.

```
[ ]  ## compute accuracy of training and testing
     def _compute_counts(y_pred, y_batch, mode='train'):
         return (y_pred==y_batch).sum().item()
```

```
     def adjust_learning_rate(learning_rate, optimizer, epoch, decay):
         """initial LR decayed by 1/10 every args.lr epochs"""
         lr = learning_rate
         if (epoch > 5):
             lr = 0.001
         if (epoch >= 10):
             lr = 0.0001
         if (epoch > 20):
             lr = 0.00001
         for param_group in optimizer.param_groups:
             param_group['lr'] = lr
```

```
[ ]  def _save_checkpoint(ckp_path, model, epoch, optimizer, global_step):
         ## save checkpoint to ckp_path: 'checkpoint/step_100.pt'
         ckp_path = ckp_path + 'ckp_{}.pt'.format(epoch+1)
         checkpoint = {'epoch': epoch,
                       'global_step': global_step,
                       'model_state_dict': model.state_dict(),
                       'optimizer_state_dict': optimizer.state_dict()}
         torch.save(checkpoint, ckp_path)
```

```
[ ]  from google.colab import drive
     drive.mount('/content/drive')
```

The _compute_counts function computes the number of correct predictions in a batch during training or testing. The three arguments it takes are y_pred, y_batch, and an optional argument mode. After finishing the code for both the training and testing batches, we proceeded with the training data. We completed the loss function and backpropagation, set up a writer to track accuracy, and then moved on to the testing code. This involved working with the testing batches, making predictions, computing accuracy, and finally plotting the prediction accuracy to visualize our final results.

```python
# set up and parameters for model
def main():
    ## choose cpu or gpu
    seed = 1
    torch.manual_seed(seed)
    ## numpy.rand(1), 1.1
    ## choose GPU id
    gpu_id = 0  ## 1, 2, 3,4
    use_cuda = torch.cuda.is_available()
    if use_cuda:
        device = torch.device('cuda', gpu_id)
    else:
        device = torch.device('cpu')
    print("device: ", device)
    ## random seed for cuda
    if use_cuda:
        torch.cuda.manual_seed(72)

    ## initialize hyper-parameters
    num_epochs = 10
    decay = 0.01
    learning_rate = 0.0001
    batch_size = 50 #100
    ckp_path = './checkpoint/'
    os.makedirs(ckp_path, exist_ok=True)

    ## Part 1: loading in data using MNIST (see _load_data function definition above)
    DATA_PATH = '.drive/shareddrives/hw3_csci436/data/'
    train_loader, test_loader=_load_data(DATA_PATH, batch_size)

    ## Part 2: load the MLP model in Model class definition above, this has the architecture for the CNN model
    model =  MLPModel()
    ## load model to gpu or cpu
    model.to(device)

    ## Part 3: define the Opimization method and LOSS FUNCTION: cross-entropy
    ## optimizer
    optimizer = optim.Adam(model.parameters(),lr=learning_rate)
    ## cross entropy loss
    loss_fun = nn.CrossEntropyLoss()

    # Visualization - Lists to store logged data
    iterations = []
```

```python
##  model training
iteration = 0
if True:
    model = model.train() ## model training
    for epoch in range(num_epochs): #10-50
        ## learning rate
        adjust_learning_rate(learning_rate, optimizer, epoch, decay)
        for batch_id, (x_batch,y_labels) in enumerate(train_loader):

            iteration += 1
            x_batch,y_labels = Variable(x_batch).to(device), Variable(y_labels).to(device)

            output_y = model(x_batch)
            ##---------------------------------------------------------------
            ## Step 4: compute loss between ground truth and predicted result
            ##---------------------------------------------------------------
            loss = loss_fun(output_y, y_labels)

            ##-----------------------------------------------
            ## Step 5: write back propagation steps below
            ##-----------------------------------------------
            optimizer.zero_grad()
            loss.backward()
            optimizer.step() # update params

            ##-------------------------------------------------------------
            ## Step 6: get the predict result and then compute accuracy
            ##-------------------------------------------------------------
            y_pred = torch.argmax(output_y.data, 1)
            accy = _compute_counts(y_pred, y_labels)/batch_size
            ##-------------------------------------------------------------
            ## Step 7: print loss values [I have done it]
            ##-------------------------------------------------------------
            if iteration%10==0:
                print('iter: {} loss: {}, accy: {}'.format(iteration, loss.item(), accy))
                wandb.log({'iter': iteration, 'loss': loss.item()})
                wandb.log({'iter': iteration, 'accy': accy})

                # Append data for visualization
                iterations.append(iteration)
                losses.append(loss.item())

        ##-------------------------------------------------------
```

```python
    ##--------------------------------------
    ##    model testing code below
    ##--------------------------------------
    total = 0
    accy_count = 0
    model.eval() ##test
    with torch.no_grad(): ## no gradient update
        for batch_id, (x_batch,y_labels) in enumerate(test_loader):
            x_batch, y_labels = Variable(x_batch).to(device), Variable(y_labels).to(device)
            ##---------------------------------------
            ## Step 8: write the predict result below
            ##---------------------------------------
            output_y = model(x_batch)
            y_pred = torch.argmax(output_y.data, 1)

            ##----------------------------------------------------
            ## Step 9: computing the test accuracy
            ##----------------------------------------------------
            total += len(y_labels)
            accy_count += _compute_counts(y_pred, y_labels)
    accy = accy_count/total
    print("testing accy: ", accy)
    wandb.log({'Testing Accuracy': accy})

    # Log training loss data as a table
    table = wandb.Table(data=list(zip(iterations, losses)), columns=["Iteration", "Loss"])
    wandb.log({"Training Loss Table": table})

    # Visualize training loss with wandb
    wandb.log({"Training Loss Plot": wandb.plot.line(table, x='Iteration', y='Loss', title='Training Loss vs Iteration')})
```

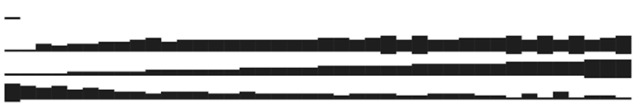
RESULTS:

We had a high accuracy of 90.87 for our testing data, which can also be seen in the picture below which is a screenshot of our accuracy output:

```
iter: 12000 loss: 0.5106592178344727, accy: 0.86
testing accy:  0.9087
```

0.133 MB of 0.133 MB uploaded

Run history:

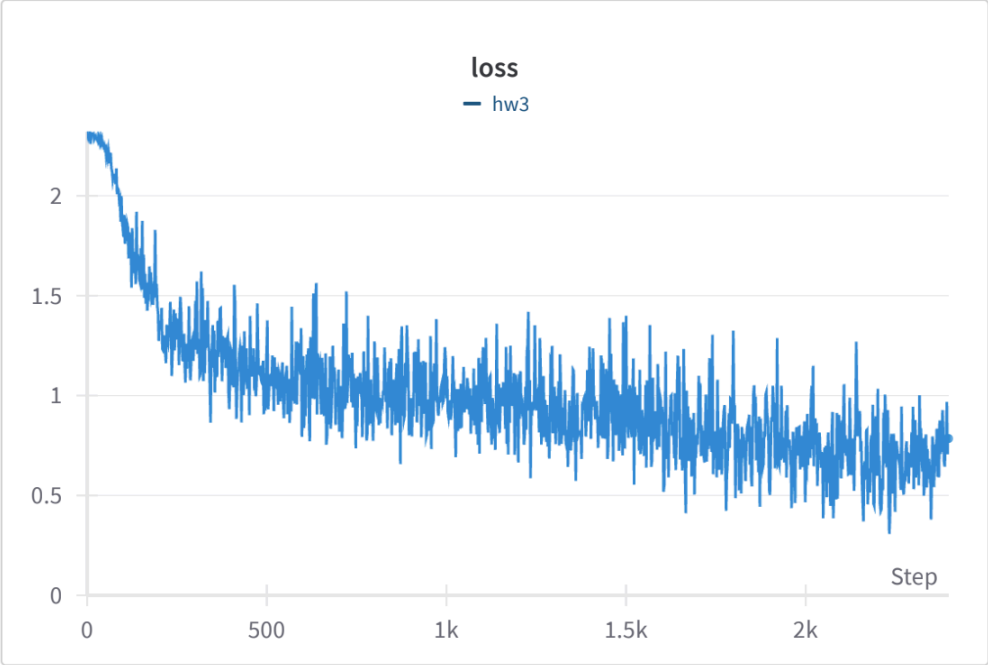


Run summary:

Testing Accuracy	0.9087
accy	0.86
iter	12000
loss	0.51066

View run **hw3** at: https://wandb.ai/hw6_csci436/CSCI_436/runs/db15qdjh
View project at: https://wandb.ai/hw6_csci436/CSCI_436
Synced 5 W&B file(s), 1 media file(s), 1 artifact file(s) and 0 other file(s)

CNN Model Training Loss over iterations



Our analysis of the results indicates that our convolutional neural network performed well on the MNIST dataset. Over time, the epochs showed consistently higher testing accuracy. The plot demonstrates a decreasing trend in training loss across iterations, resulting in improved image classification accuracy. Overall, our model achieved a low cross-entropy loss, showcasing its effectiveness in accurately classifying the dataset's images.

BONUS

```
[ ]  # test the 3 layers and 2 layers
     class MLPModel(nn.Module):
         def __init__(self):
             super(MLPModel, self).__init__()
             # Convolutional layers
             self.conv1 = nn.Conv2d(1, 4, kernel_size=3, stride=1, padding=1)  # 4 output channels
             # BONUS - add batch normalization
             self.bn1 = nn.BatchNorm2d(4)
             self.relu1 = nn.ReLU()
             # BONUS - max pooling to reduce dimensions
             self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
             # Dropout after the first convolutional layer
             self.dropout1 = nn.Dropout(0.2)

             # 8 output channels
             self.conv2 = nn.Conv2d(4, 8, kernel_size=3, stride=1, padding=1)
             # BONUS - add batch normalization
             self.bn2 = nn.BatchNorm2d(8)
             self.relu2 = nn.ReLU()
              # BONUS - max pooling to reduce dimensions
             self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
              # Dropout after the second convolutional layer
             self.dropout2 = nn.Dropout(0.2)

             # Calculate the number of features after convolutional and pooling operations
             self.num_features = 8 * 7 * 7  # Adjusted based on the last pooling layer output size

             # Fully connected layers

             # Reduced number of units in the fully connected layer
             self.fc1 = nn.Linear(self.num_features, 32)
             # BONUS - add batch normalization
             self.bn_fc1 = nn.BatchNorm1d(32)
             self.relu3 = nn.ReLU()
             # Dropout after the first fully connected layer
             self.dropout3 = nn.Dropout(0.5)

             self.fc2 = nn.Linear(32, 10)  # Output layer with 10 classes
```

This code defines a convolutional neural network (CNN) model called MLPModel using PyTorch's nn.Module class for image classification tasks, particularly suited for the MNIST dataset with hand-written digit images. It starts with two convolutional layers (conv1 and conv2) followed by batch normalization (bn1 and bn2), ReLU activation functions (relu1, relu2, and relu3), max-pooling layers (pool1 and pool2) for dimensionality reduction, and dropout layers (dropout1, dropout2, and dropout3) to prevent overfitting. The first convolutional layer (conv1) processes single-channel input images using a 3x3 kernel to produce 4 output channels, normalized by bn1 and activated by relu1, followed by max-pooling and dropout. The second convolutional layer (conv2) takes the output from the first layer, increases the channels to 8, and applies similar operations as the first layer. The fully connected layers (fc1 and fc2) at the end handle feature reduction and class prediction, with appropriate batch normalization, ReLU activation, and dropout for regularization.