Submission format

Submit a single PDF-version report for all problems, including explanations, results, codes (if needed), and discussions. Other formats are not acceptable.

Problem 1. Neural Network Optimizer

Review lecture 9 and answer the following questions.

In this problem, you are asked to train and test a multilayer perceptron (MLP), i.e., a fully connected neural network, for entire MNIST handwritten digit dataset. The network structure is 784-200-50-10, where 784 means the input layer has 784 input neurons. This is because each image in MNIST dataset is 28x28 and you need to stretch them to a length-784 vector. 200 and 50 are the number of neurons in hidden layers. 10 is the number of neurons in output layer since there are 10 types of digits. Use the softmax as the output layer. For other hyperparameters, like activation functions, dropout, batchnorm, etc., you can adjust or decide if adding them by yourself.

- a) Summarize briefly the differences among different network optimizers, including Conventional SGD, SGD with Momentum, AdaGrad, and RMSprop, as shown in Lecture 9. Also, explain briefly why the optimizer “Adam” is so popular.

- Conventional SGD:
  - Update Rule: $w := w - \eta \cdot \nabla J(w)$
  - We apply gradients directly
  - We struggles with ravines: slow progress in shallow directions and jittery in steep ones
  - It is sensitive to learning rate
  - We easily get stuck in local minimas or saddle points
- SGD with Momentum:
  - Update Rule: $v := \mu v - \eta \nabla J(w)$

    $w := w + v$
  - This adds a “velocity” term that accumulates past gradients
  - We dampens oscillations and accelerates convergence.
  - The common momentum value is: μ = 0.9
- AdaGrad
  - It adapts learning rate per parameter, based on gradient history
  - Frequent updates → smaller effective LR; infrequent → larger LR
  - However, the issue: LR decays too much over time → stops learning early
  - We use when: data is sparse or features vary in frequency
- RMSProp

- Update Rule: $E[g^2]_t := \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$

$$w := w - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t$$

- This introduces exponential moving average
- We solve AdaGrad's vanishing learning rate issue

- Adam
  - Solves AdaGrad's vanishing learning rate issue
  - Maintains both 1st moment (mean) and 2nd moment (variance) of gradients
  - Update Rule: $m_t := \beta_1 m_{t-1} + (1-\beta_1) g_t$

$$v_t := \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

$$\hat{m}_t := \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t := \frac{v_t}{1-\beta_2^t}$$

$$w := w - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

  - **Why is Adam so popular?**
    - It has a fast convergence
    - It works well right out of the box
    - Barely and tuning required

- b) Implement the above MLP and train with different optimizers (Conventional SGD, SGD with Momentum, AdaGrad, RMSprop, and Adam) and step size schedulers, and report your test accuracies and compare them.

  Note: you can implement them manually or via built-in libraries.

main.py:

```
# main.py
import torch
import platform
from model import MLP
from train import train
from evaluate import evaluate
from utils import get_dataloaders
from torch.optim import SGD, Adam, RMSprop, Adagrad
from torch.optim.lr_scheduler import StepLR
```

```
# Device Selection
if torch.cuda.is_available():
    device = torch.device('cuda')
elif platform.system() == 'Darwin' and platform.machine() ==
'arm64' and torch.backends.mps.is_available():
    device = torch.device('mps')
else:
    device = torch.device('cpu')

print(f"Using device: {device}")

# Data Loaders
train_loader, test_loader = get_dataloaders()

# Optimizers Configuration
optimizers_config = {
    "SGD": lambda params: SGD(params, lr=0.01),
    "SGD+Momentum": lambda params: SGD(params, lr=0.01,
momentum=0.9),
    "AdaGrad": lambda params: Adagrad(params, lr=0.01),
    "RMSprop": lambda params: RMSprop(params, lr=0.001),
    "Adam": lambda params: Adam(params, lr=0.001),
}

# Training and Evaluation
results = {}

for name, opt_func in optimizers_config.items():
    print(f"Training with {name}...")
    model = MLP().to(device)
    optimizer = opt_func(model.parameters())
    scheduler = StepLR(optimizer, step_size=5, gamma=0.5)
    train(model, optimizer, scheduler, train_loader, device,
epochs=10)
    acc = evaluate(model, test_loader, device)
    results[name] = acc
```

```
    print(f"{name} Test Accuracy: {acc:.2f}%\n")

# Final Results
print("=== Final Results ===")
for name, acc in results.items():
    print(f"{name}: {acc:.2f}%")
```

model.py:

```
# model.py
import torch.nn as nn

class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(784, 200),
            nn.ReLU(),
            nn.Linear(200, 50),
            nn.ReLU(),
            nn.Linear(50, 10),
        )

    def forward(self, x):
        return self.layers(x)
```

train.py:

```
# train.py
import torch
from torch.nn import CrossEntropyLoss

def train(model, optimizer, scheduler, train_loader, device,
epochs=10):
    model.train()
    criterion = CrossEntropyLoss()
```

```
    for epoch in range(epochs):
        for batch in train_loader:
            x, y = batch
            x = x.view(x.size(0), -1).to(device)
            y = y.to(device)

            optimizer.zero_grad()
            outputs = model(x)
            loss = criterion(outputs, y)
            loss.backward()
            optimizer.step()
        if scheduler:
            scheduler.step()
```

evaluate.py:

```
# evaluate.py
import torch

def evaluate(model, test_loader, device):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for x, y in test_loader:
            x = x.view(x.size(0), -1).to(device)
            y = y.to(device)
            outputs = model(x)
            _, preds = outputs.max(1)
            correct += (preds == y).sum().item()
            total += y.size(0)
    return 100.0 * correct / total
```

utils.py:

```
# utils.py
```

```
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

def get_dataloaders(batch_size=64):
    transform = transforms.ToTensor()
    train = datasets.MNIST(root='.', train=True,
transform=transform, download=True)
    test = datasets.MNIST(root='.', train=False,
transform=transform, download=True)

    return (
        DataLoader(train, batch_size=batch_size, shuffle=True),
        DataLoader(test, batch_size=batch_size)
    )
```

Problem 2. Neural Network Approximation

In modern deep learning, It is very important to learn how to read and understand codes. In this problem, you are given a piece of Python code written in a Jupyter notebook (see “Intro2DL HW4.ipynb”), which includes two neural network models, i.e., MLP and Fourier model. Both networks have the same number of parameters and learning rate.

Please understand and run the provided code to answer the following questions.

This problem is to examine your code understanding skills; you do not need to modify the code. Also, you can still answer the questions even if you're not familiar with the Fourier model.

- What is the input and output of the model? What does this code do?

Input:
A scalar value x between 0 and 1, representing a 1D continuous input. It is expanded into a 4000×1 tensor for training.

Output:
A single scalar f(x) that approximates the value of a target function:
$f(x) = \sin(2\pi x) + 0.5 \cdot \sin(6\pi x) + 0.25 \cdot \sin(10\pi x)$

What this code actually does?:
This code trains two neural networks to approximate the above sinusoidal target function:
- A standard 3-layer MLP using ReLU activations
- A Fourier model that includes a Learnable Fourier Feature (LFF) layer to encode high-frequency components

It compares how well each model learns the function over 4000 training epochs and plots:
- Training loss curves
- The final learned functions vs. the original target

- What a role does the LFF layer shown in the code play? Explain briefly.

The LFF layer applies a sinusoidal transformation, which is (sin(Wx + b)), to the input. This helps the model to capture high-frequency variations in the signal by projecting the input into a Fourier feature space. In contrast to an MLP that must learn those features indirectly, the LFF layer can directly inject periodic structure into the model. This helps make it easier for the Fourier model to fit functions composed of sinusoids, like the one in the Jupyter Notebook.

- Run the code and describe what you observe. How do the results relate to the Universal Function Approximation Theorem of neural networks?

After running the code, we observe that both the MLP and the Fourier model are able to learn and approximate the target function over time. However, the Fourier model clearly converges faster and achieves a lower training loss throughout the 4000 epochs. This is reflected in the loss curve, where the Fourier model consistently outperforms the MLP. Additionally, when we compare the learned functions against the true target function, the Fourier model captures the high-frequency oscillations more accurately,

particularly in regions where the MLP appears to oversmooth or flatten out the waveform. This behavior aligns with the Universal Approximation Theorem, which states that neural networks with sufficient parameters can approximate any continuous function. Both models demonstrate this capacity in principle, but the Fourier model does so more effectively in practice. Thanks to the LFF layer, which encodes sinusoidal features directly, the Fourier model is better equipped to represent periodic functions like the one in this task. This highlights the importance of model architecture and feature representation: even with the same number of parameters and learning rate, the Fourier model leverages its structural advantage to approximate the function with greater precision and efficiency.