Homework #3

CSE 446: Machine Learning

Eric Boris: 1976637

Conceptual Questions

Problem 1

a. **False**: SVM only maximizes the margin, it doesn't minimize generalization error among linear classifiers.

b. **Decrease**: Lower values of σ allow the model to be more expressive.

c. **True**: Neural networks often have non-convex loss functions with only local minima.

d. **False**: This can lead to problems. It's better to initialize weights to random values.

e. **True**: Without nonlinearities in the network, the network can only learn linear functions.

f. **True**: We mitigate this by using SGD to improve the runtime of the backward pass.

Kernels

Problem 2

Show that $K(x, x') = e^{-\frac{(x-x')^2}{2}}$ is a kernel function for this feature map, i.e., $\phi(x) \cdot \phi(x') = e^{-\frac{(x-x')^2}{2}}$.

$$
\begin{aligned}
\phi(x) \cdot \phi(x') &= \left(\sum_{i=0}^{\infty} \frac{1}{\sqrt{i!}} e^{-\frac{x^2}{2}} x^i\right) \left(\sum_{i=0}^{\infty} \frac{1}{\sqrt{i!}} e^{-\frac{x'^2}{2}} x'^i\right) \\
&= e^{-\frac{x^2+x'^2}{2}} \sum_{i=0}^{\infty} \left(\frac{1}{i!} (xx')^i\right) \\
&= e^{-\frac{x^2+x'^2}{2}} e^{xx'} && \text{Talor Series of } e^x = \textstyle\sum_{i=0}^{\infty} \frac{x^i}{i!} \\
&= e^{-\frac{(x-x')^2}{2}}
\end{aligned}
$$

Problem 3: Answers

a. Polynomial Kernel: Lambda=1E-05 d=11.0
RBF Kernel: Lambda=1E-03 gamma=2.0802

b. See Figures 1 and 2

Problem 3: Code

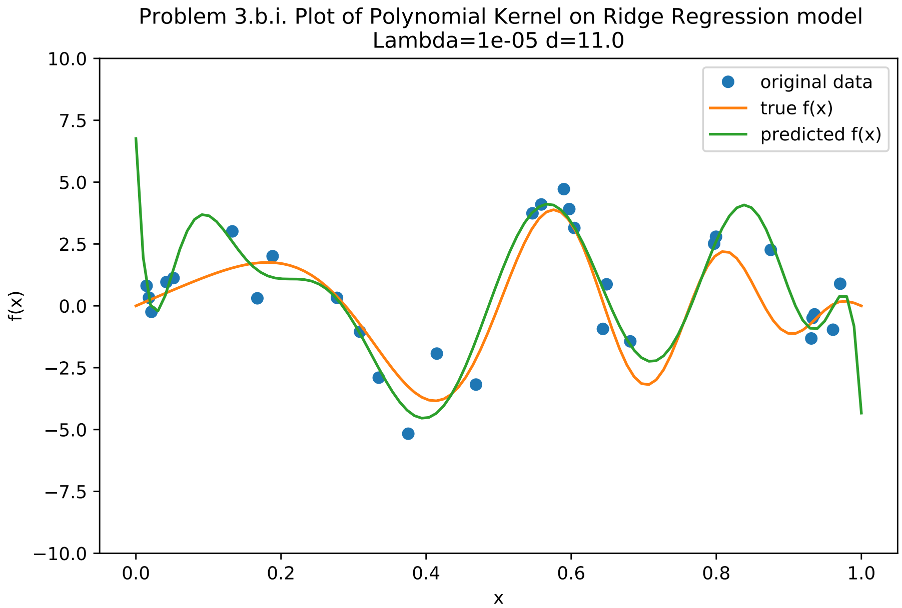

Figure 1

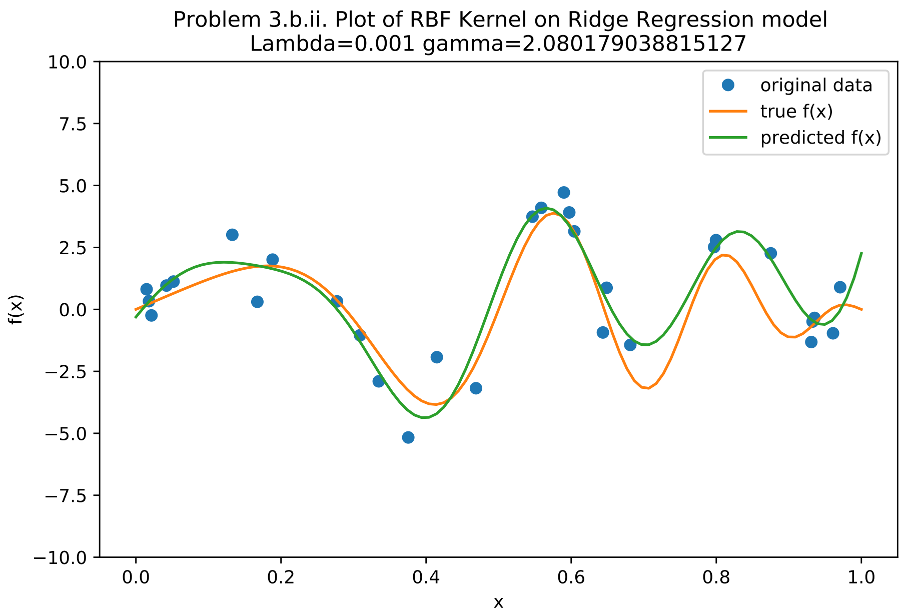

Figure 2

```
# Problem 3: Kernel Ridge Regression

import numpy as np
import matplotlib.pyplot as plt

def generate_data(f, n):
    ''' Generate n random samples of data and actual output using function f. '''
    # Let x_i be uniformly random on [0, 1].
    x = np.random.rand(n)
    # Let epsilon_i ~ N(0, 1).
```

```
    epsilon = np.random.randn(n)
    # Let y_i = f(x_i) + epsilon_i.
    y = f(x) + epsilon
    # Return x as a column vector.
    return x.reshape(-1, 1), y

def f_star(x):
    ''' Compute the f star function given in the spec. '''
    return 4 * np.sin(np.pi * x) * np.cos(6 * np.pi * x ** 2)

def polynomial_kernel(x, z, d):
    ''' Define the polynomial kernel given in the spec where d is a hyperparameter. '''
    return (1 + x.dot(z.T)) ** d

def rbf_kernel(x, z, gamma):
    ''' Define the RBF kernel given in the spec where gamma is a hyperparameter. '''
    return np.exp(-gamma * squared_difference(x, z))

def squared_difference(x, z):
    ''' Return the squared difference between x and z. '''
    return np.sum((x[:, :, None] - z[:, :, None].T) ** 2, axis=1)

def leave_one_out_cv(model, X, y, regularized_lambdas, hyperparameters):
    ''' Compute the error of lambda and hypermeter combinations on the model using LOOCV. '''

    def error(model, X, y, regularized_lambda, hyperparameter):
        ''' Perform LOOCV on one lambda / hyperparameter pair and return the model mean error. '''
        # Set the model to use the parameters
        model.regularized_lambda = regularized_lambda
        model.hyperparameter = hyperparameter

        # Perform LOOCV.
        error = []
        for i in range(len(X)):
            # Use this to determine which indices to include in the "in" subsets.
            indices = np.full(len(X), True)
            indices[i] = False

            # Define the cross validation subsets.
            X_in, y_in = X[indices], y[indices]
            X_out, y_out = X[i].reshape(1, -1), y[i].reshape(1, )

            # Interpolate with the model.
            model.train(X_in, y_in)
            y_hat = model.predict(X_out)

            # Compute and store the mean squared error.
            error.append(np.mean((y_hat - y_out) ** 2))

        return np.mean(error)

    # Perform LOOCV.
    results = [(error(model, X, y, rl, hp), rl, hp) for rl in regularized_lambdas for hp in hyperparameters]
    return np.array(results)

def plot(title, subtitle, x_label, y_label, file_path, original_x, original_y, original_label, true_x, true_y, true_label, pred_y, pred_label, argsort, dim=(8, 5)):
    ''' Plot the graphs for Problem 3 Part b. '''
    plt.figure(figsize=dim)
    plt.title(f'{title}\n{subtitle}')
    plt.xlabel(x_label)
    plt.ylabel(y_label)
    plt.plot(original_x[argsort, 0], original_y[argsort], 'o', label=original_label)
    plt.plot(true_x[:, 0], true_y, label=true_label)
    plt.plot(true_x[:, 0], pred_y, label=pred_label)
    plt.legend()
    plt.ylim([-10, 10])
```

```python
    plt.savefig(file_path)
    plt.show()

class KernelRidgeRegression:
    def __init__(self, kernel, regularized_lambda=1E-8, hyperparameter=None):
        self.kernel = kernel
        self.regularized_lambda = regularized_lambda
        self.hyperparameter = hyperparameter

    def train(self, X_train, y_train):
        ''' Train the model. '''
        self.mean = np.mean(X_train, axis=0)
        self.std = np.std(X_train, axis=0)

        X_train = (X_train - self.mean) / self.std
        self.X_train = X_train

        K = self.kernel(X_train, X_train, self.hyperparameter)
        self.alpha = np.linalg.solve(K + self.regularized_lambda * np.eye(K.shape[0]), y_train)

        return self.alpha

    def predict(self, X):
        ''' Predict using the model. '''
        X = (X - self.mean) / self.std
        K = self.kernel(self.X_train, X, self.hyperparameter)
        return K.T.dot(self.alpha)

def main():
    # Generate training data.
    X_train, y_train = generate_data(f=f_star, n=30)

    # Set the hyperparameter bounds.
    regularized_lambdas = 10.0 ** (-np.arange(2, 10))
    ds = np.arange(4, 20)
    gammas = (1 / np.median(squared_difference(X_train, X_train))) * np.linspace(0, 2, 10)

    # Build and score both models.
    poly_model = KernelRidgeRegression(polynomial_kernel)
    poly_results = leave_one_out_cv(poly_model, X_train, y_train, regularized_lambdas, ds)

    rbf_model = KernelRidgeRegression(rbf_kernel)
    rbf_results = leave_one_out_cv(rbf_model, X_train, y_train, regularized_lambdas, gammas)

    # Part a. Find and assign the best lambdas and hyperparameters for and to the models.
    poly_model.regularized_lambda = poly_results[np.argmin(poly_results[:, 0])][1]
    poly_model.hyperparameter = poly_results[np.argmin(poly_results[:, 0])][2]

    rbf_model.regularized_lambda = rbf_results[np.argmin(rbf_results[:, 0])][1]
    rbf_model.hyperparameter = rbf_results[np.argmin(rbf_results[:, 0])][2]

    print(f'Problem 3.a.i. Best Lambda and d values with polynomial kernel: Lambda={poly_model.regularized_lambda}\td={poly_model.hyperparameter}')
    print(f'Problem 3.a.ii. Best Lambda and gamma values with RBF kernel: Lambda={rbf_model.regularized_lambda}\tgamma={rbf_model.hyperparameter}')

    # Part b. Plot the learned functions using the best lambdas and hyperparameters from part a.
    X_evenly_spaced = np.linspace(0, 1, 100).reshape(-1, 1)
    y_evenly_spaced = f_star(X_evenly_spaced[:, 0])
    argsort = np.argsort(X_train[:, 0])

    poly_model.train(X_train, y_train)
    y_hat_evenly_spaced_poly = poly_model.predict(X_evenly_spaced)

    plot(title='Problem 3.b.i. Plot of Polynomial Kernel on Ridge Regression model',
        subtitle=f'Lambda={poly_model.regularized_lambda} d={poly_model.hyperparameter}',
        x_label='x',
```

```
            y_label='f(x)',
            file_path='../plots/3bi.pdf',
            original_x=X_train,
            original_y=y_train,
            original_label='original data',
            true_x=X_evenly_spaced,
            true_y=y_evenly_spaced,
            true_label='true f(x)',
            pred_y=y_hat_evenly_spaced_poly,
            pred_label='predicted f(x)',
            argsort=argsort)

    rbf_model.train(X_train, y_train)
    y_hat_evenly_spaced_rbf = rbf_model.predict(X_evenly_spaced)

    plot(title='Problem 3.b.ii. Plot of RBF Kernel on Ridge Regression model',
        subtitle=f'Lambda={rbf_model.regularized_lambda} gamma={rbf_model.hyperparameter}',
        x_label='x',
        y_label='f(x)',
        file_path='../plots/3bii.pdf',
        original_x=X_train,
        original_y=y_train,
        original_label='original data',
        true_x=X_evenly_spaced,
        true_y=y_evenly_spaced,
        true_label='true f(x)',
        pred_y=y_hat_evenly_spaced_rbf,
        pred_label='predicted f(x)',
        argsort=argsort)

if __name__ == '__main__':
    main()
```

Neural Networks for MNIST

Problem 4: Answers

a., b., c.

We see that a deep network performs significantly better than a wide network with the same number of parameters. The deep network reaches comparable levels of performance to the wide in one epoch compared to eighteen epochs. This performance is shown in the following tables and plotted in figures 3 and 4. The reason for the deep network's better performance is likely due to there being more nonlinearities in the deeper model so the data are more immediately fit.

Model Performance

	Wide Net		Deep Net	
	Training	Testing	Training	Testing
Accuracy	0.9904	0.9749	0.9911	0.9718
Loss	0.0336	0.0844	0.0320	0.0928

Model Details

	Wide Net	Deep Net
Number of Parameters	50890	50890
Epochs trained	18	1

Problem 4: Code

```
# Problem 4: Neural Networks for MNIST

import numpy as np
```

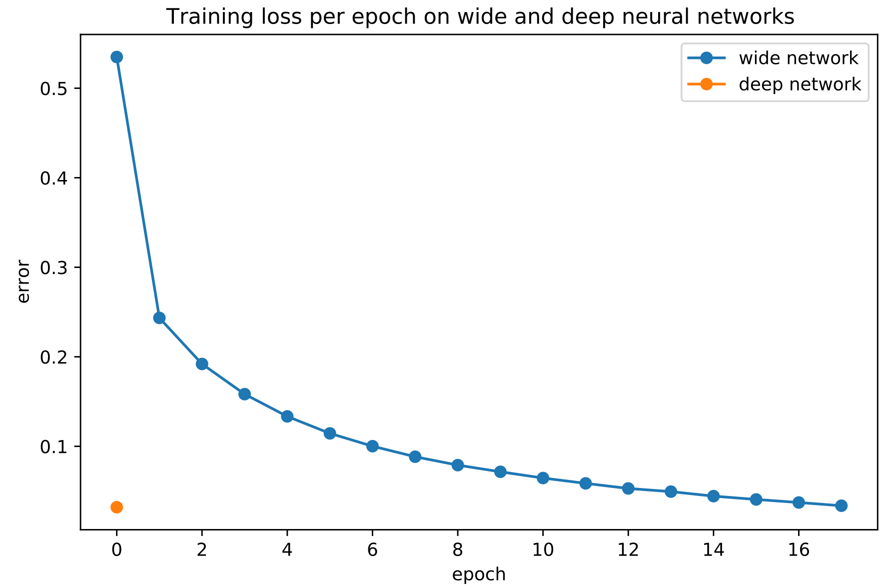

Figure 3

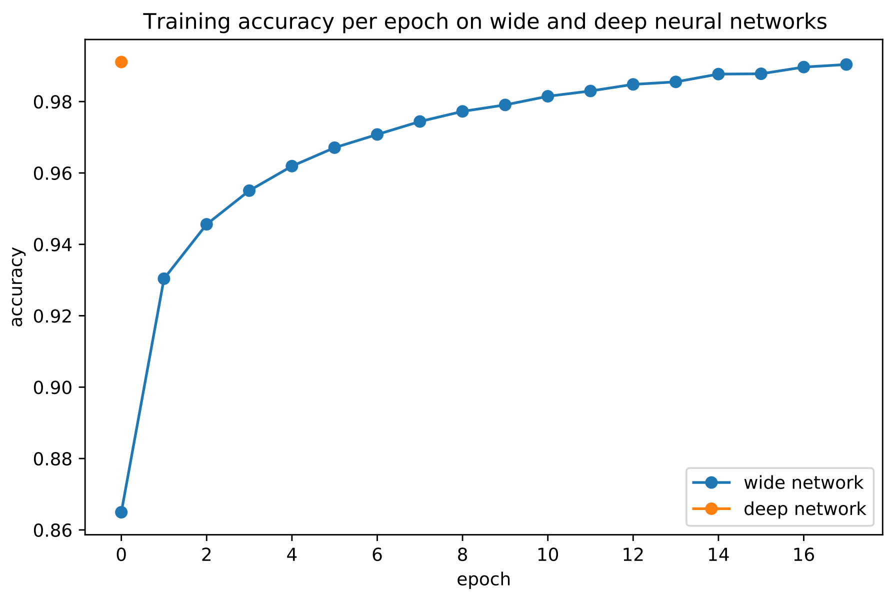

Figure 4

```
import torch
import torch.nn as nn
import torchvision.datasets as datasets
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
from tqdm import tqdm

def get_loaders(root_path):
    ''' Return MNIST train and test DataLoaders. '''
    train = torch.utils.data.DataLoader(
```

```
        datasets.MNIST(
            root=root_path,
            train=True,
            download=True,
            transform=transforms.ToTensor()),
        batch_size=128,
        shuffle=True)
    test = torch.utils.data.DataLoader(
        datasets.MNIST(
            root=root_path,
            train=False,
            download=True,
            transform=transforms.ToTensor()),
        batch_size=128,
        shuffle=True)
    return train, test

def count_parameters(weights, biases):
    ''' Return the total number of parameters used in the given weights and biases. '''
    parameters = sum([np.prod(w.shape) for w in weights])
    parameters += sum([np.prod(b.shape) for b in biases])
    return parameters

def plot(title, x_label, y_label, file_name, x_1, y_1, label_1, x_2, y_2, label_2, dim=(8, 5)):
    ''' Plot two lines on single chart. '''
    plt.figure(figsize=dim)
    plt.title(title)
    plt.xlabel(x_label)
    plt.ylabel(y_label)
    plt.plot(x_1, y_1, '-o', label=label_1)
    plt.plot(x_2, y_2, '-o', label=label_2)
    plt.xticks(np.arange(0, max(x_1)+1, 2.0))
    plt.legend()
    plt.savefig(file_name)
    plt.show()

class NeuralNetwork:
    def __init__(self, data_loader, learning_rate, n_neurons, n_layers, input_dim, output_dim):
        self.data_loader = data_loader
        self.learning_rate = learning_rate

        alpha = 1 / (np.sqrt(input_dim))
        self._weights(alpha, n_neurons, n_layers, input_dim, output_dim)
        self._biases(alpha, n_neurons, n_layers, output_dim)

    def _weights(self, alpha, n_neurons, n_layers, input_dim, output_dim):
        ''' Create the model's weights. '''
        weights = []

        # Define the input layer weights.
        weights.append(-2 * alpha * torch.rand(n_neurons, input_dim) + alpha)
        weights[-1].requires_grad = True

        # Define the hidden layer weights.
        # Don't add the input and output layer weights.
        for _ in range(n_layers - 2):
            weights.append(-2 * alpha * torch.rand(n_neurons, n_neurons) + alpha)
            weights[-1].requires_grad = True

        # Define the output layer weights.
        weights.append(-2 * alpha * torch.rand(output_dim, n_neurons) + alpha)
        weights[-1].requires_grad = True

        self.weights = weights

    def _biases(self, alpha, n_neurons, n_layers, output_dim):
```

```
        ''' Create the model's biases. '''
        biases = []

        # Define the input and hidden layer biases.
        # Don't add the output layer biases.
        for _ in range(n_layers - 1):
            biases.append(-2 * alpha * torch.rand(n_neurons) + alpha)
            biases[-1].requires_grad = True

        # Define the output layer biases.
        biases.append(-2 * alpha * torch.rand(output_dim) + alpha)
        biases[-1].requires_grad = True

        self.biases = biases

    def train(self, n_epochs, accuracy_threshold=0.99, verbose=False):
        ''' Train the model. '''
        # Use the weights and biases as the parameter.
        optimizer = torch.optim.Adam(self.weights + self.biases, lr=self.learning_rate)

        accuracies = []
        losses = []
        for epoch in range(n_epochs):
            # Perform forward and backwards passes through the data and return the model performance.
            accuracy, loss = self.measure_performance(self.data_loader, optimizer=optimizer, train=True)

            accuracies.append(accuracy)
            losses.append(loss)

            if verbose:
                print(f'Epoch={epoch}\tTraining Loss={losses[-1]}\tAccuracy={accuracy}')

            # End training if minimium accuracy threshold is met.
            if accuracy > accuracy_threshold:
                break

        return accuracies, losses, self.weights, self.biases

    def measure_performance(self, data_loader, optimizer=None, train=False):
        ''' Perform forwards and backwards passes on the model and return the performance. '''
        accuracy = 0
        loss = 0

        for X, y in tqdm(iter(data_loader)):
            # Change the dimensions of X.
            X = torch.flatten(X, start_dim=1, end_dim=3)

            # Perform the forward pass and get the predictions.
            logits = self._forward(X)
            y_hat = torch.argmax(logits, 1)

            # Compute the accuracy and loss.
            accuracy += torch.sum(y == y_hat)
            loss_tmp = torch.nn.functional.cross_entropy(logits, y, size_average=False)

            # Gradient descent backward pass.
            if train:
                optimizer.zero_grad()
                loss_tmp.backward()
                optimizer.step()

            loss += loss_tmp

```

```python
        # Normalize the performance measures.
        loss /= len(data_loader.dataset)
        accuracy = accuracy.to(dtype=torch.float) / len(data_loader.dataset)

        return accuracy, loss

    def _forward(self, x):
        ''' Perform a pass through the network using the given input x and ReLU nonlinearities. '''
        y = torch.matmul(x, self.weights[0].T) + self.biases[0]
        for i in range(1, len(self.weights)):
            y = torch.matmul(nn.functional.relu(y), self.weights[i].T) + self.biases[i]
        return y

def main():
    # Load the training and test data.
    train_loader, test_loader = get_loaders('../data/python_mnist/')

    # Part a: Build, train, and test a wide neural network.
    wide_net = NeuralNetwork(data_loader=train_loader, learning_rate=1E-3, n_neurons=64, n_layers=2, input_dim=784, output_dim=10)

    wide_train_accuracies, wide_train_losses, wide_weights, wide_biases = wide_net.train(n_epochs=500, verbose=True)
    wide_parameters = count_parameters(wide_weights, wide_biases)
    print(f'Wide net training results:\nAccuracy={wide_train_accuracies[-1]}\tLoss={wide_train_losses[-1]}\tN Parameters={wide_parameters}\n')

    wide_test_accuracy, wide_test_loss = wide_net.measure_performance(test_loader)
    print(f'Wide net test results:\nAccuracy={wide_test_accuracy}\tLoss={wide_test_loss}\n')

    # Part b: Build, train, and test a deep neural network.
    deep_net = NeuralNetwork(data_loader=train_loader, learning_rate=1E-3, n_neurons=32, n_layers=3, input_dim=784, output_dim=10)

    deep_train_accuracies, deep_train_losses, deep_weights, deep_biases = wide_net.train(n_epochs=500, verbose=True)
    deep_parameters = count_parameters(deep_weights, deep_biases)
    print(f'Deep net training results:\nAccuracy={deep_train_accuracies[-1]}\tLoss={deep_train_losses[-1]}\tN Parameters={deep_parameters}\n')

    deep_test_accuracy, deep_test_loss = wide_net.measure_performance(test_loader)
    print(f'Deep Net Test Results:\nAccuracy={deep_test_accuracy}\tLoss={deep_test_loss}\n')

    # Plot the performance of both models.
    x_wide_evenly_spaced = range(len(wide_train_losses))
    x_deep_evenly_spaced = range(len(deep_train_losses))

    plot(title='Training loss per epoch on wide and deep neural networks',
        x_label='epoch',
        y_label='error',
        file_name='../plots/4_losses.pdf',
        x_1=x_wide_evenly_spaced,
        y_1=[float(loss) for loss in wide_train_losses],
        label_1='wide network',
        x_2=x_deep_evenly_spaced,
        y_2=[float(loss) for loss in deep_train_losses],
        label_2='deep network')

    plot(title='Training accuracy per epoch on wide and deep neural networks',
        x_label='epoch',
        y_label='accuracy',
        file_name='../plots/4_accuracies.pdf',
        x_1=x_wide_evenly_spaced,
        y_1=wide_train_accuracies,
        label_1='wide network',
```

```
            x_2=x_deep_evenly_spaced,
            y_2=deep_train_accuracies,
            label_2='deep network')


if __name__ == '__main__':
    main()
```

Using Pretrained Networks and Transfer Learning

Problem 5: Answers

a. Fixed Feature Extractor:

- Plot for training loss: See figure 5

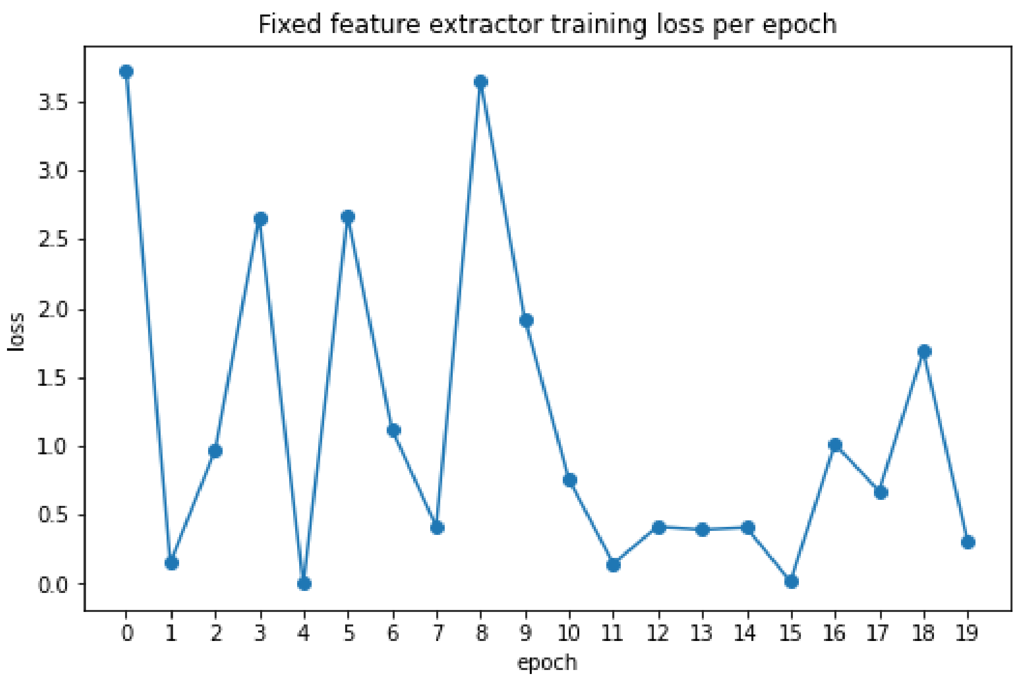

Figure 5

- Plot for validation loss: See figure 6
- Highest validation accuracy: 0.8046
- Test accuracy: 79.99
- Test loss: 0.6293

b. Fine-Tuning:

- Plot for training loss: See figure 7
- Plot for validation loss: See figure 8
- Highest validation accuracy: 0.9088
- Test accuracy: 90.73

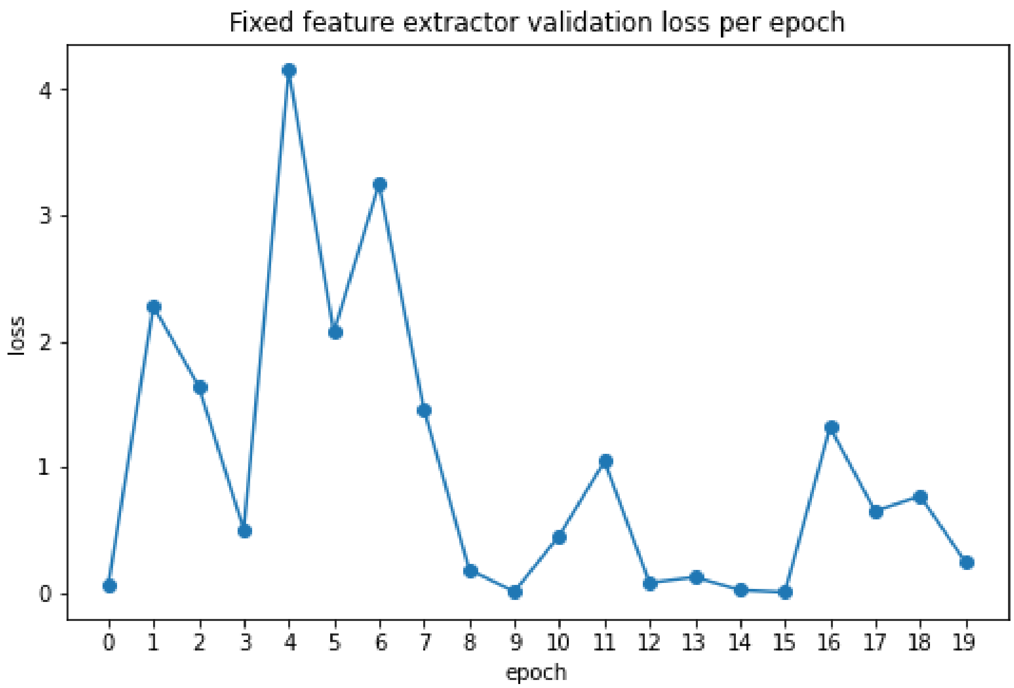

Figure 6

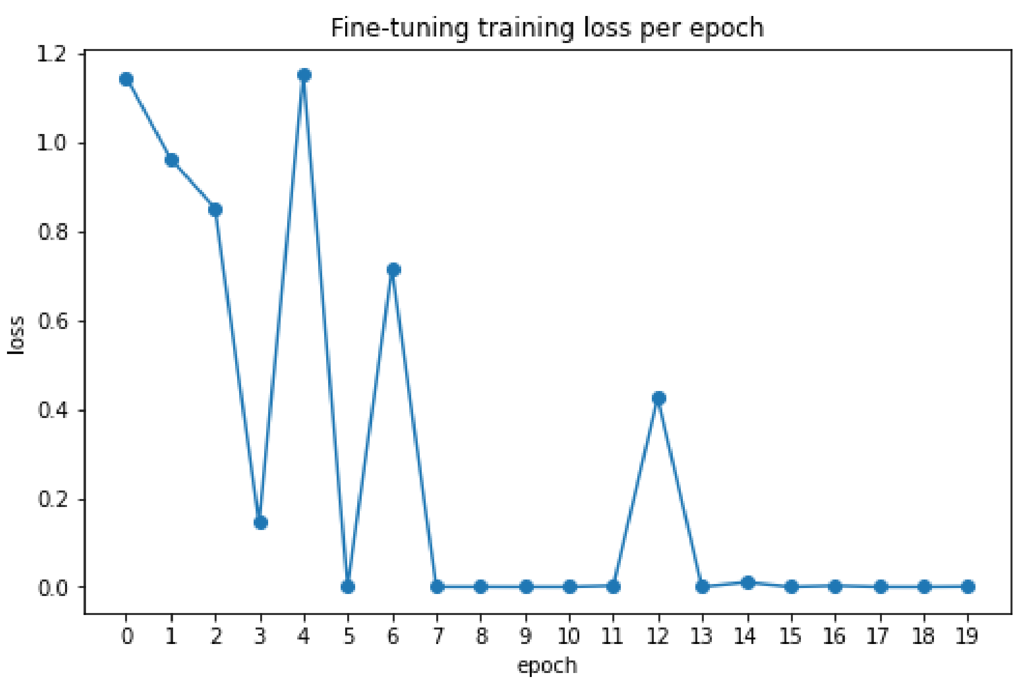

Figure 7

- Test loss: 0.4235

Problem 5: Code

```
# -*- coding: utf-8 -*-
"""
CSE446 hw3 p5.ipynb
```

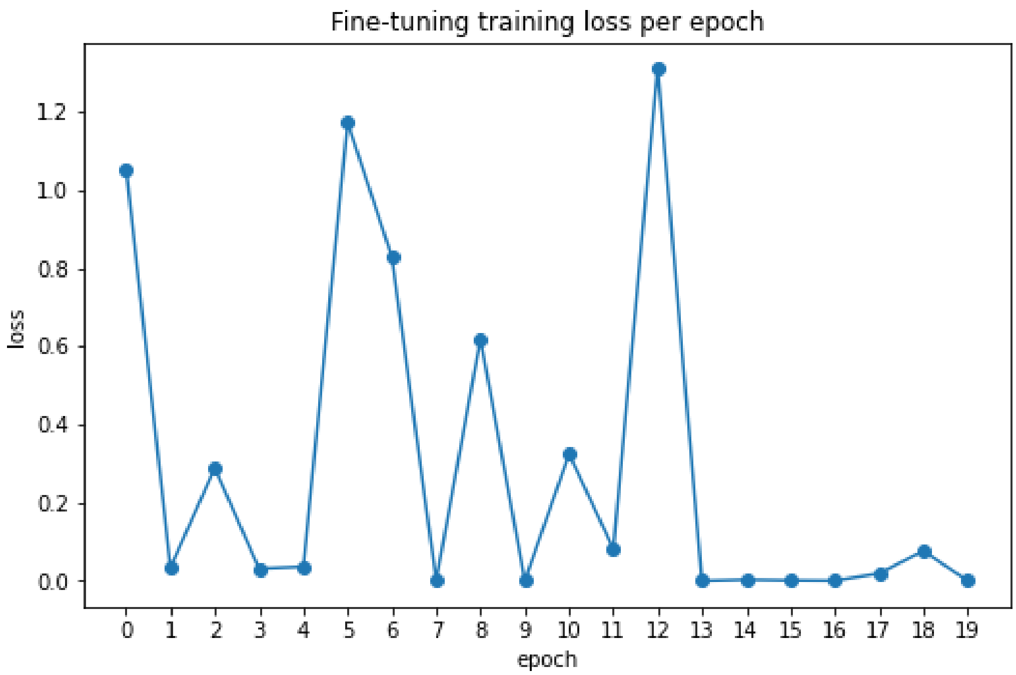

Figure 8

```

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1EtJYFeZDQPBH3zOfZk9kJQ2KNAagt9FT

Written using code from the following tutorials:
    https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html
    https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html#load-data
    https://pytorch.org/tutorials/beginner/finetuning_torchvision_models_tutorial.html
"""

# Access google drive for saving and loading trained models.
from google.colab import drive
drive.mount('/content/drive')

import torch
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
import torch.nn as nn
import torch.optim as optim
from tqdm import tqdm
from torch.utils.data import random_split
import time
import copy

# Use GPU if it's available, and CPU otherwise.
train_on_gpu = torch.cuda.is_available()
train_on_multi_gpus = (torch.cuda.device_count() >= 2)
gpus = torch.cuda.device_count()

# Define transforms for the data to work with the AlexNet model.
transform = transforms.Compose([transforms.Resize(256),
                                transforms.ToTensor(),
                                transforms.Normalize((0.5, 0.5, 0.5),
                                                     (0.5, 0.5, 0.5))])
```

```python

# Load the dataset.
dataset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)

# Split the dataset into training / validation such that validation is 10% of
# the training.

# Use a seed so that the training / validation split is the same each time.
torch.manual_seed(43)

# We set the validation set to be 10% of the training data.
val_size = 5000
train_size = len(dataset) - val_size

# Split the set.
trainset, valset = random_split(dataset, [train_size, val_size])

# Define the training and validation dataloaders.
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)

valloader = torch.utils.data.DataLoader(valset, batch_size=4,
                                        shuffle=True, num_workers=2)

# Join train and validation into a single dataloader
dataloaders = {'train': trainloader, 'val': valloader}
dataset_sizes = {'train': train_size, 'val': val_size}

# Load and define the test dataloader.
testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=2)
test_size = len(testset)

# Define the dataset class names.
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',
           'ship', 'truck')

def train_model(model, criterion, optimizer, scheduler, num_epochs=25):
    ''' Train the model and return the trained model as well as training
        performance data. '''

    # Handle training on gpu or cpu.
    if train_on_multi_gpus:
        print(f"\nTraining on {gpus} GPUs!\n")
        model = torch.nn.DataParallel(model).cuda()
    elif train_on_gpu:
        print('\nTraining on GPU!\n')
        model = model.cuda()
    else:
        print('\nTraining on CPU; consider making n_epochs very small.\n')

    since = time.time()

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    train_losses = []
    valid_losses = []

    for epoch in tqdm(range(num_epochs)):
        # Each epoch has a training and validation phase
        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()  # Set model to training mode
            else:
                model.eval()   # Set model to evaluate mode
```

```python

            running_loss = 0.0
            running_corrects = 0

            # Iterate over data.
            for inputs, labels in dataloaders[phase]:
                if train_on_multi_gpus or train_on_gpu:
                    inputs, labels = inputs.cuda(), labels.cuda()

                # zero the parameter gradients
                optimizer.zero_grad()

                # forward
                # track history if only in train
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    # backward + optimize only if in training phase
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                # statistics
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)
            if phase == 'train':
                scheduler.step()

            epoch_loss = running_loss / dataset_sizes[phase]
            epoch_acc = running_corrects.double() / dataset_sizes[phase]

            print('{} Loss: {:.4f} Acc: {:.4f}'.format(
                phase, epoch_loss, epoch_acc))

            # deep copy the model
            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())

            # Store this epoch's training and validation losses.
            if phase == 'train':
                train_losses.append(loss)
            else:
                valid_losses.append(loss)

    time_elapsed = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.format(
        time_elapsed // 60, time_elapsed % 60))
    print('Best val Acc: {:4f}'.format(best_acc))

    # load best model weights
    model.load_state_dict(best_model_wts)
    return model, train_losses, valid_losses

def performance(model):
    ''' Return the loss and performace of the model on the test data. '''
    correct = 0
    total = 0
    running_loss = 0.0
    with torch.no_grad():
        for data in testloader:
            images, labels = data

            # Handle for gpu.
            if train_on_multi_gpus or train_on_gpu:
                images, labels = images.cuda(), labels.cuda()

```

```
            # Compute the accuracy.
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            # Compute the loss.
            loss = criterion(outputs, labels)
            running_loss += loss.item() * images.size(0)

    loss = running_loss / test_size
    accuracy = 100 * correct / total

    return loss, accuracy

def plot(title, x_label, y_label, x, y, file_dir, dim=(8, 5)):
    ''' Plot the given data. '''
    plt.figure(figsize=dim)
    plt.title(title)
    plt.xlabel(x_label)
    plt.xticks(np.arange(0, max(x)+2, 1))
    plt.ylabel(y_label)
    plt.plot(x, y, '-o')
    plt.show()
    plt.savefig(file_dir)

# Define the fixed feature extractor model and it's related components.
ffe_model = torchvision.models.alexnet(pretrained=True)

# Prevent all but the last layer from training.
for param in ffe_model.parameters():
    param.requires_grad = False

ffe_model.classifier[6] = nn.Linear(4096, 10)

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(ffe_model.parameters(), lr=0.001, momentum=0.9)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=7, gamma=0.1)

# Train and save the ffe model.
ffe_model, ffe_train_losses, ffe_valid_losses = train_model(ffe_model, criterion, optimizer,
    scheduler, 20)

PATH = '/content/drive/MyDrive/CSE446_hw3/cifar_ffe_model.pth'
torch.save(ffe_model.state_dict(), PATH)

plot(title='Fixed feature extractor training loss per epoch',
     x_label='epoch',
     y_label='loss',
     x=[x for x in range(len(ffe_train_losses))],
     y=ffe_train_losses,
     file_dir='/content/drive/MyDrive/CSE446_hw3/ffe_t.png')

plot(title='Fixed feature extractor validation loss per epoch',
     x_label='epoch',
     y_label='loss',
     x=[x for x in range(len(ffe_valid_losses))],
     y=ffe_valid_losses,
     file_dir='/content/drive/MyDrive/CSE446_hw3/ffe_v.png')

# Display the test performance of the ffe model
loss, accuracy = performance(ffe_model)
print(f'test Loss: {loss} Accuracy: {accuracy}')

# Define the fine tuning model and it's related components.
ft_model = torchvision.models.alexnet(pretrained=True)

# Train every layer.
for param in ft_model.parameters():
```

```
        param.requires_grad = True

ft_model.classifier[6] = nn.Linear(4096, 10)

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(ft_model.parameters(), lr=0.001, momentum=0.9)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=7, gamma=0.1)

# Train and save the ft model.
ft_model, ft_train_losses, ft_valid_losses = train_model(ft_model, criterion, optimizer, scheduler, 20)

PATH = '/content/drive/MyDrive/CSE446_hw3/cifar_ft_model.pth'
torch.save(ffe_model.state_dict(), PATH)

plot(title='Fine-tuning training loss per epoch',
     x_label='epoch',
     y_label='loss',
     x=[x for x in range(len(ft_train_losses))],
     y=ft_train_losses,
     file_dir='/content/drive/MyDrive/CSE446_hw3/ft_t.png')

plot(title='Fine-tuning training loss per epoch',
     x_label='epoch',
     y_label='loss',
     x=[x for x in range(len(ft_valid_losses))],
     y=ft_valid_losses,
     file_dir='/content/drive/MyDrive/CSE446_hw3/ft_v.png')

# Display the test performance of the ft model
loss, accuracy = performance(ft_model)
print(f'test Loss: {loss} Accuracy: {accuracy}')
```