# Appendix 1: Project Code

## analytical\_solve.py

'''

Alex Knowlton

12/7/2024

This file presents the analytical solution of the problem defined in the

project proposal

'''

import numpy as np

def analytical\_solve(vector, constellations):

'''

Given a signal vector and a list of constellations, returns the index of

the constellation with the lowest mean Euclidean distance from the given

vector.

'''

results = []

for constellation in constellations:

dE = get\_mean\_euclidean\_distance(vector, constellation)

results.append(dE)

results = np.array(results)

return np.argmin(results)

def get\_mean\_euclidean\_distance(vector, constellation):

'''

Given a complex signal vector and a constellation vector, computes the most

likely index within the constellation and computes the euclidean distance

from that point in the signal vector and that point in the constellation,

and returns the mean of all the euclidean distances.

'''

vector = vector.T.reshape((2, 128)).T

vector = vector[:, 0] + 1j \* vector[:, 1]

vector = vector.reshape((vector.shape[0], 1))

constellation = constellation.T

diff = vector - constellation

diff = np.abs(diff \* np.conj(diff)) # squared euclidean distance

min = np.argmin(diff, axis=1)

selected = constellation[:, min].T

diff = vector - selected

diff = np.abs(diff \* np.conj(diff))

return np.mean(diff)

## generate\_dataset.py

'''

Alex Knowlton

12/6/24

Generates constellation dataset for EECS453 and saves as .npy files

'''

import numpy as np

import matplotlib.pyplot as plt

def normalize\_constellation(constellation):

'''

Normalizes such that each element in the constellation have energy 1

'''

E = np.sqrt(np.mean(constellation \* np.conj(constellation)))

Eb = E / np.log2(len(constellation))

return constellation / Eb

def draw\_samples(constellation, n\_samples=512):

'''

draws n\_samples from the given constellation

'''

M, n = constellation.shape

indices = np.random.randint(0, M, n\_samples)

data = constellation[indices, :]

real = np.real(data)

imag = np.imag(data)

data = np.vstack((real, imag))

return data

def corrupt\_data(samples, n0=0.05, n0\_range=True, channel=1):

'''

Adds noise with standard deviation n0 to the given samples

'''

samples = samples.T.reshape((2, -1)).T

samples = samples[:, 0] + 1j \* samples[:, 1]

samples = samples \* channel

samples = np.hstack((np.real(samples), np.imag(samples))).T

if n0\_range:

n0 = np.random.uniform(0, n0)

noise = np.random.normal(0, n0, samples.shape)

data = samples + noise

data = data.reshape((-1, 1))

return data

def get\_qam\_constellation(n):

'''

Return QAM constellation with n points. Note: n must be an even power of 2

'''

b = int(np.sqrt(n)) - 1

data = np.arange(-b, b+1, 2)

N = len(data)

real = data.reshape((1, N))

imag = 1j \* data.reshape((N, 1))

data = real + imag

L = data.shape[0] \* data.shape[1]

data = data.reshape((L, 1))

return normalize\_constellation(data)

def get\_pam\_constellation(n):

'''

Returns normalized PAM constellation with N points

'''

theta = np.arange(0, 2 \* np.pi, 2 \* np.pi / n)

points = np.exp(-1j \* theta)

points = points.reshape((points.shape[0], 1))

return normalize\_constellation(points)

def get\_bpsk\_constellation():

return np.array([[-1], [1]])

def get\_apsk32\_constellation():

inner = get\_pam\_constellation(4)

middle = 3 \* get\_pam\_constellation(12) \* np.exp(-1j \* 2 \* np.pi / 24)

outer = 5 \* get\_pam\_constellation(16)

constellation = np.vstack((inner, middle, outer))

constellation = normalize\_constellation(constellation)

return constellation

def display\_data(data, label):

'''

Plot data vector and display image

'''

M = data.shape[0] // 2

real = data[:M]

imag = data[M:]

max\_val = np.max(data) \* 1.1

plt.figure()

plt.scatter(real, imag)

plt.ylabel('Quadrature [Q]')

plt.xlabel('In-phase [I]')

plt.title(f'True Label: {label}')

plt.xlim(-max\_val, max\_val)

plt.ylim(-max\_val, max\_val)

plt.grid()

def get\_dataset(constellations, n\_samples, n0=0.05, n0\_range=True,

channels=None):

channels = [1] if channels is None else channels

sample\_length = 512

results = np.zeros((sample\_length \* 2,

len(channels) \* len(constellations) \* n\_samples))

for j in range(len(constellations)):

for i in range(len(channels)):

print(f'Drawing sample {j},{i}')

for k in range(n\_samples):

constellation = constellations[j]

channel = channels[i]

new\_data = draw\_samples(constellation)

noisy\_data = corrupt\_data(new\_data, n0, n0\_range, channel)

idx = k + j \* n\_samples + i \* len(constellations)

results[:, idx] = noisy\_data[:, 0]

outputs = np.array(range(len(constellations)))

ones\_list = np.ones((n\_samples \* len(channels), 1))

outputs = outputs \* ones\_list

outputs = outputs.T.reshape((outputs.shape[0] \* outputs.shape[1]))

outputs = outputs.astype(int)

return results, outputs

def generate\_dataset(n\_samples, n0=0.05, n0\_range=True,

channels=None, is\_train=False):

name = 'train' if is\_train else 'test'

constellations = []

constellations.append(get\_pam\_constellation(4))

constellations.append(get\_pam\_constellation(8))

constellations.append(get\_apsk32\_constellation())

constellations.append(get\_bpsk\_constellation())

constellations.append(get\_qam\_constellation(4))

constellations.append(get\_qam\_constellation(16))

X, y = get\_dataset(constellations,

n\_samples=n\_samples,

n0=n0, channels=channels, n0\_range=n0\_range)

# reshape and form result arrays and save

np.save(f'./data/{name}\_results', y)

np.save(f'./data/{name}\_data', X.T)

print('Finished Generating Dataset')

if \_\_name\_\_ == '\_\_main\_\_':

generate\_dataset(10, 5)

## train\_model.py

import torch

import torch.nn as nn

import torch.nn.functional as F

import torch.optim as optim

import numpy as np

import time

from torch.utils.data import Dataset, DataLoader

'''

Alex Knowlton

12/6/24

This module defines a PyTorch neural network, imports the generated

communications dataset, and evaluates the network performance on the data.

Network Structure:

Convolutional layer:

Input: 512 points, with 2 channels

Input size: 512x2

Kernel: size (1, 2), so operate on one point at a time

# channels: 48, representing a bunch of different points in constellation

No padding,

Standard stride of 1

'''

class CommsDataset(Dataset):

'''

Defines a dataset that takes in the communications dataset as numpy arrays,

not tensors.

'''

def \_\_init\_\_(self, features, labels):

'''

Class initializer

Inputs:

features: NxD numpy features array

labels: (N,) numpy labels array with class numbers

'''

self.features = torch.tensor(features, dtype=torch.float)

self.labels = torch.tensor(labels, dtype=torch.long)

self.height = 512

self.width = 2

def \_\_len\_\_(self):

return len(self.features)

def \_\_getitem\_\_(self, index):

constellation = self.features[index].reshape(1, 2, 512)

label = self.labels[index]

return constellation, label

class CommsNet(nn.Module):

'''

Neural network for constellation prediction

'''

def \_\_init\_\_(self):

super(CommsNet, self).\_\_init\_\_()

self.n\_channels = 64

self.conv = nn.Sequential(

nn.Conv2d(in\_channels=1, out\_channels=self.n\_channels,

kernel\_size=(2, 1), stride=1, padding=0)

)

self.fc = nn.Sequential(

nn.Linear(self.n\_channels, 48),

nn.ReLU(),

nn.Linear(48, 48),

nn.ReLU(),

nn.Linear(48, 6)

)

def forward(self, x):

'''

Forward pass of X vector

'''

conv\_output = self.conv(x)

softmax\_output = F.softmax(conv\_output, dim=1)

x = torch.sum(softmax\_output \* conv\_output, dim=3, keepdim=True)

x = x.view(x.size(0), -1)

x = self.fc(x)

return x

def train(trainloader, net, criterion, optimizer, device):

start = time.time()

for epoch in range(10): # loop over the dataset multiple times

start = time.time()

running\_loss = 0.0

for i, (images, labels) in enumerate(trainloader):

# send images and labels to device

images = images.to(device)

labels = labels.to(device)

# zero the parameter gradients

optimizer.zero\_grad()

# forward pass

yhat = net.forward(images)

# loss function - cross entropy

loss = criterion(yhat, labels)

# backward pass

loss.backward()

# optimize the network

optimizer.step()

# print statistics

running\_loss += loss.item()

if i % 100 == 99: # print every 2000 mini-batches

end = time.time()

print('[epoch %d, iter %5d] loss: %.5f, time: %.2f seconds' %

(epoch + 1, i + 1, running\_loss / 100, end-start))

start = time.time()

running\_loss = 0.0

stop = time.time()

print(f'Finished Training: {(stop - start) / 60} minutes')

def test(testloader, net, device, is\_test=True):

data\_name = 'test' if is\_test else 'training'

correct = 0

total = 0

with torch.no\_grad():

for data in testloader:

images, labels = data

images = images.to(device)

labels = labels.to(device)

outputs = net(images)

\_, predicted = torch.max(outputs.data, 1)

total += labels.size(0)

correct += (predicted == labels).sum().item()

print(f'Accuracy: {data\_name} constellations: %.2f %%' % (

100 \* correct / total))

return correct / total

def train\_model():

X\_train = np.load('./data/train\_data.npy')

y\_train = np.load('./data/train\_results.npy')

trainset = CommsDataset(X\_train, y\_train)

trainloader = DataLoader(trainset, batch\_size=32, shuffle=True)

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

net = CommsNet().to(device)

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(net.parameters(), lr=0.001)

train(trainloader, net, criterion, optimizer, device)

return net

def test\_model(net):

# X\_train = np.load('./data/train\_data.npy')

X\_test = np.load('./data/test\_data.npy')

# y\_train = np.load('./data/train\_results.npy')

y\_test = np.load('./data/test\_results.npy')

# trainset = CommsDataset(X\_train, y\_train)

testset = CommsDataset(X\_test, y\_test)

# trainloader = DataLoader(trainset, batch\_size=32, shuffle=True)

testloader = DataLoader(testset, batch\_size=32, shuffle=True)

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

# net = CommsNet().to(device)

# criterion = nn.CrossEntropyLoss()

# optimizer = optim.Adam(net.parameters(), lr=0.001)

# optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

# # train(trainloader, net, criterion, optimizer, device)

# train\_error = test(trainloader, net, device, is\_test=False)

test\_error = test(testloader, net, device, is\_test=True)

# return train\_error, test\_error

return test\_error

if \_\_name\_\_ == "\_\_main\_\_":

test\_model()

## train\_fading\_model.py

import torch

import torch.nn as nn

import torch.nn.functional as F

import torch.optim as optim

import numpy as np

import time

from torch.utils.data import Dataset, DataLoader

'''

Alex Knowlton

12/6/24

This module defines a PyTorch neural network, imports the generated

communications dataset, and evaluates the network performance on the data.

Network Structure:

Convolutional layer:

Input: 512 points, with 2 channels

Input size: 512x2

Kernel: size (1, 2), so operate on one point at a time

# channels: 48, representing a bunch of different points in constellation

No padding,

Standard stride of 1

'''

class CommsDataset(Dataset):

'''

Defines a dataset that takes in the communications dataset as numpy arrays,

not tensors.

'''

def \_\_init\_\_(self, features, labels):

'''

Class initializer

Inputs:

features: NxD numpy features array

labels: (N,) numpy labels array with class numbers

'''

self.features = torch.tensor(features, dtype=torch.float)

self.labels = torch.tensor(labels, dtype=torch.long)

self.height = 512

self.width = 2

def \_\_len\_\_(self):

return len(self.features)

def \_\_getitem\_\_(self, index):

constellation = self.features[index].reshape(1, 2, 512)

label = self.labels[index]

return constellation, label

class CommsNet(nn.Module):

'''

Neural network for constellation prediction with fading channels

'''

def \_\_init\_\_(self):

super(CommsNet, self).\_\_init\_\_()

self.n\_channels = 8

self.n\_constellation\_points = 64

self.conv1 = nn.Sequential(

nn.Conv2d(in\_channels=1,

out\_channels=self.n\_channels,

kernel\_size=(4, 1), stride=(2, 1), padding=(2, 0)),

nn.MaxPool1d

)

self.conv2 = nn.Sequential(

nn.Conv2d(in\_channels=self.n\_channels,

out\_channels=self.n\_channels\*self.n\_constellation\_points,

kernel\_size=(2, 1), groups=self.n\_channels)

)

self.fc = nn.Sequential(

nn.Linear(self.n\_channels \* self.n\_constellation\_points, 48),

nn.ReLU(),

nn.Linear(48, 48),

nn.ReLU(),

nn.Linear(48, 6)

)

def forward(self, x):

'''

Forward pass of X vector

'''

x = self.conv1(x)

conv\_output = self.conv2(x)

softmax\_output = F.softmax(conv\_output, dim=1)

x = torch.sum(softmax\_output \* conv\_output, dim=3, keepdim=True)

x = x.view(x.size(0), -1)

x = self.fc(x)

return x

def train(trainloader, net, criterion, optimizer, device):

for epoch in range(10): # loop over the dataset multiple times

start = time.time()

running\_loss = 0.0

for i, (images, labels) in enumerate(trainloader):

# send images and labels to device

images = images.to(device)

labels = labels.to(device)

# zero the parameter gradients

optimizer.zero\_grad()

# forward pass

yhat = net.forward(images)

# loss function - cross entropy

loss = criterion(yhat, labels)

# backward pass

loss.backward()

# optimize the network

optimizer.step()

# print statistics

running\_loss += loss.item()

if i % 100 == 99: # print every 2000 mini-batches

end = time.time()

print('[epoch %d, iter %5d] loss: %.5f, time %.2f seconds' %

(epoch + 1, i + 1, running\_loss / 100, end-start))

start = time.time()

running\_loss = 0.0

print('Finished Training')

def test(testloader, net, device, is\_test=True):

data\_name = 'test' if is\_test else 'training'

correct = 0

total = 0

with torch.no\_grad():

for data in testloader:

images, labels = data

images = images.to(device)

labels = labels.to(device)

outputs = net(images)

\_, predicted = torch.max(outputs.data, 1)

total += labels.size(0)

correct += (predicted == labels).sum().item()

print(f'Accuracy: {data\_name} constellations: %.2f %%' % (

100 \* correct / total))

return correct / total

def train\_fading\_model():

X\_train = np.load('./data/train\_data.npy')

y\_train = np.load('./data/train\_results.npy')

trainset = CommsDataset(X\_train, y\_train)

trainloader = DataLoader(trainset, batch\_size=32, shuffle=True)

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

net = CommsNet().to(device)

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(net.parameters(), lr=0.001)

train(trainloader, net, criterion, optimizer, device)

return net

def test\_fading\_model(net):

X\_test = np.load('./data/test\_data.npy')

y\_test = np.load('./data/test\_results.npy')

testset = CommsDataset(X\_test, y\_test)

testloader = DataLoader(testset, batch\_size=32, shuffle=True)

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

test\_error = test(testloader, net, device, is\_test=True)

return test\_error

if \_\_name\_\_ == "\_\_main\_\_":

test\_fading\_model()

## report\_vals.py (toplevel script)

'''

Alex Knowlton

12/7/24

Generates plots for report

'''

import numpy as np

import matplotlib.pyplot as plt

import torch

from generate\_dataset import get\_pam\_constellation, get\_qam\_constellation, \

draw\_samples, corrupt\_data, generate\_dataset

from analytical\_solve import get\_mean\_euclidean\_distance

from train\_model import train\_model, test\_model

from train\_fading\_model import train\_fading\_model, test\_fading\_model

def show\_analytical\_solution():

'''

Plots a visual aid for the analytical solution of the problem

'''

qam = get\_qam\_constellation(16)

pam = get\_pam\_constellation(8)

sample = draw\_samples(pam, 128)

sample = corrupt\_data(sample, n0=0.25, n0\_range=False)

dE\_pam = np.round(get\_mean\_euclidean\_distance(sample, pam), 3)

dE\_qam = np.round(get\_mean\_euclidean\_distance(sample, qam), 3)

# reshape sample vector for plotting

sample = sample.T.reshape((2, 128)).T

sample = sample[:, 0] + 1j \* sample[:, 1]

# plot all and save image

plt.figure(figsize=(5, 10))

# plot PAM constellation with data

plt.subplot(211)

plt.title(f'PAM: $ d\_E = {dE\_pam} $')

plt.scatter(np.real(sample), np.imag(sample), label='Sample Data')

plt.scatter(np.real(pam), np.imag(pam), label='PAM Constellation')

plt.grid()

plt.legend(loc='upper right')

plt.xlabel('In-phase [I]')

plt.ylabel('Quadrature [Q]')

# plot QAM constellation with data

plt.subplot(212)

plt.title(f'QAM: $ d\_E = {dE\_qam} $')

plt.scatter(np.real(sample), np.imag(sample), label='Sample Data')

plt.scatter(np.real(qam), np.imag(qam), label='QAM Constellation')

plt.grid()

plt.legend(loc='upper right')

plt.xlabel('In-phase [I]')

# save figure

plt.tight\_layout()

plt.savefig('./img/analytical\_comparison.png')

def plot\_constellation\_points():

'''

Plots 3 constellations on top of each other and save image

'''

qam4 = get\_qam\_constellation(4)

qam16 = get\_qam\_constellation(16)

pam = get\_pam\_constellation(8)

# plot figure

plt.figure()

plt.grid()

plt.scatter(np.real(qam16), np.imag(qam16), label='$QAM16$')

plt.scatter(np.real(pam), np.imag(pam), label='$PAM$')

plt.scatter(np.real(qam4), np.imag(qam4), label='$QAM4$', marker='+')

plt.legend(loc='upper right')

plt.xlabel('In-phase [I]')

plt.ylabel('Quadrature [Q]')

plt.title('Superimposed Constellations')

plt.savefig('./img/constellation\_compare.png')

def plot\_constellation\_with\_channel():

'''

Plots a data sample with a constellation, but with a noisy flat fading

channel, which looks rotated and shrunk

'''

qam = get\_qam\_constellation(16)

data = draw\_samples(qam)

data = corrupt\_data(data, n0=0.25, n0\_range=False,

channel=0.4\*np.exp(-1j \* np.pi / 6))

data = data.T.reshape((2, -1)).T

data = data[:, 0] + data[:, 1] \* 1j

# plot figure

plt.figure()

plt.scatter(np.real(data), np.imag(data), label='Data')

plt.scatter(np.real(qam), np.imag(qam), label='Constellation')

plt.xlabel('In-phase [I]')

plt.ylabel('Quadrature [Q]')

plt.title('Data with Noisy Flat Fading Channel')

plt.grid()

plt.legend(loc='upper right')

plt.savefig('./img/channel\_fading.png')

def plot\_constellation\_with\_noise():

'''

Plots and saves a QAM16 constellation with two different SNR values, and

saves the image

'''

qam = get\_qam\_constellation(16)

data = draw\_samples(qam)

data\_low\_snr = corrupt\_data(data, n0=1.5, n0\_range=False)

data\_low\_snr = data\_low\_snr.reshape((2, -1)).T

data\_low\_snr = data\_low\_snr[:, 0] + 1j \* data\_low\_snr[:, 1]

data\_high\_snr = corrupt\_data(data, n0=0.5, n0\_range=False)

data\_high\_snr = data\_high\_snr.reshape((2, -1)).T

data\_high\_snr = data\_high\_snr[:, 0] + 1j \* data\_high\_snr[:, 1]

# plot data

plt.figure(figsize=(5, 10))

plt.subplot(211)

plt.scatter(np.real(data\_low\_snr), np.imag(data\_low\_snr))

plt.scatter(np.real(qam), np.imag(qam))

plt.title('Low SNR')

plt.ylabel('Quadrature [Q]')

plt.grid()

plt.subplot(212)

plt.scatter(np.real(data\_high\_snr), np.imag(data\_high\_snr), label='Data')

plt.scatter(np.real(qam), np.imag(qam), label='Constellation')

plt.title('High SNR')

plt.ylabel('Quadrature [Q]')

plt.xlabel('In-phase [I]')

plt.legend(loc='upper right')

plt.grid()

plt.savefig('./img/constellation\_with\_noise.png')

def plot\_correct\_vs\_noise(n\_samples\_train=5000, n\_samples\_test=500):

'''

Generates data with a signal to noise ratio and trains and tests the model

to see how it behaves with more and more noisy data

'''

snr\_db = np.linspace(-4, 1, num=10)

n0\_list = 1 / np.power(10, snr\_db / 10)

train\_acc = []

test\_acc = []

# generate dataset with base noise level

generate\_dataset(n\_samples\_train, 1.0,

n0\_range=True, is\_train=True)

model = train\_model()

# test model with various noise levels

for n0 in n0\_list:

print(f'Generating dataset for n0 = {n0}')

generate\_dataset(n\_samples\_test, n0, n0\_range=False, is\_train=False)

e\_test = test\_model(net=model)

test\_acc.append(e\_test)

# plot output vs. sndr

train\_acc = np.array(train\_acc)

test\_acc = np.array(test\_acc)

plt.figure()

plt.plot(snr\_db, test\_acc, label='Test Accuracy')

plt.xlabel('$ E\_b/N\_0 $ [dB]')

plt.ylabel('Accuracy')

plt.title('Test Accuracy vs. SNR')

plt.yscale('log')

plt.grid()

plt.tight\_layout()

plt.savefig('./img/error\_vs\_snr.png')

# save model

dummy\_input = torch.rand((1, 1, 2, 512), dtype=torch.float32)

torch.onnx.export(

model,

(dummy\_input,),

'./model/CommsNet.onnx',

input\_names=['input']

)

def plot\_channel\_correct\_vs\_noise(n\_samples\_train=1000, n\_samples\_test=100,

channels=None):

'''

Generates data with a signal to noise ratio and trains and tests the model

to see how it behaves with more and more noisy data

'''

snr\_db = np.linspace(-4, 3, num=10)

n0\_list = 1 / np.power(10, snr\_db / 10)

test\_acc = []

# generate dataset

generate\_dataset(n\_samples\_train, 0.5,

n0\_range=True, channels=channels, is\_train=True)

model = train\_fading\_model()

for n0 in n0\_list:

print(f'Generating dataset for n0 = {n0}')

generate\_dataset(n\_samples\_test, n0, n0\_range=False,

is\_train=False)

e\_test = test\_fading\_model(net=model)

test\_acc.append(e\_test)

# plot output vs. sndr

test\_acc = np.array(test\_acc)

plt.figure()

plt.plot(snr\_db, test\_acc)

plt.xlabel('$ E\_b/N\_0 $ [dB]')

plt.ylabel('Accuracy')

plt.title('Test Accuracy vs. SNR with Fading Channel')

plt.yscale('log')

plt.grid()

plt.savefig('./img/channel\_error\_vs\_snr.png')

def main():

show\_analytical\_solution()

plot\_constellation\_points()

plot\_constellation\_with\_channel()

plot\_constellation\_with\_noise()

plot\_correct\_vs\_noise(5000)

# amps = np.array([0.9, 0.7, 0.95, 1])

# angles = 1j \* np.pi \* np.random.uniform(-1/4, 1/4, size=amps.shape)

# angles[-1] = 1j \* np.pi \* 0

# channels = amps \* np.exp(angles)

# plot\_channel\_correct\_vs\_noise(1000, channels=channels)

if \_\_name\_\_ == '\_\_main\_\_':

main()