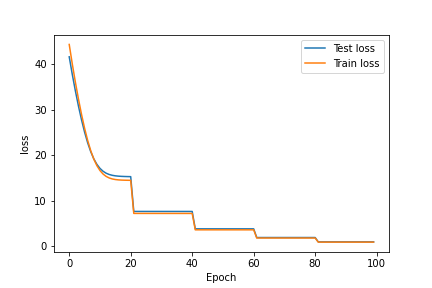
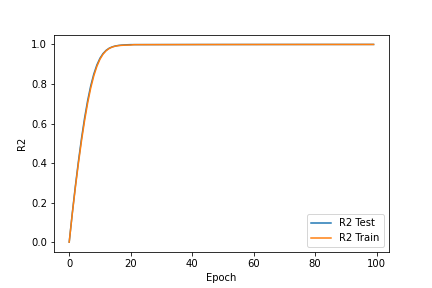
1. Minibatch starts at 64 and doubles every 20 epochs

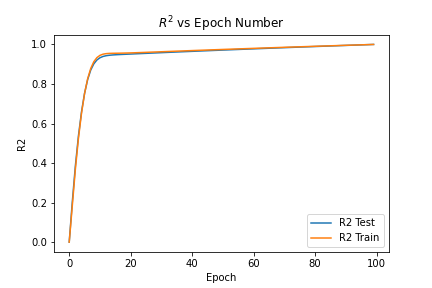
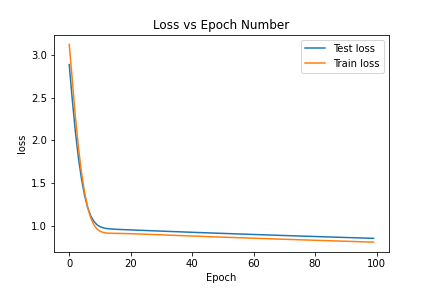


Loss function vs Epoch Number



R2 vs Epoch Number

1. Fixed minibatch = 1024



1. CNN Architecture

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Layer Description | Number of Filters | Kernel Size | Stride | Number of Parameters |
| Layer 1 | Conv2d | 16 | 5 | 1 | 1216 |
| Layer 2 | Conv2d + 2x2 max pooling | 16 | 5 | 1 | 6416 |
| Layer 3 | Conv2d + 2x2 max pooling | 32 | 5 | 1 | 12832 |
| Layer 4 | Conv2d + ReLU activation | 32 | 5 | 1 | 25632 |
| Layer 5 | Linear | 2048/1 |  |  | 2049 |
| Total Number of Parameters | 48145 |  |  |  |  |

1. Plot of residuals

Shape, arrow

Description automatically generated

#!/usr/bin/env python

# coding: utf-8

import numpy as np

import matplotlib.pyplot as plt

import torch

from torch.autograd import Variable

import torch.optim as optim

from torch.utils.data import DataLoader

import torch.nn as nn

import torch.nn.functional as F

import os

#use gpu and cuda for speedup

device = torch.device('cuda:0' if torch.cuda.is\_available() else 'cpu')

class MyDataset(torch.utils.data.Dataset):

def \_\_init\_\_(self,folder,train):

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

self.train = train

self.folder = folder

self.label\_mean = 52.90

if self.train:

self.root = os.path.join(self.folder,"cnntrain")

else:

self.root = os.path.join(self.folder,"cnntest")

self.files = os.listdir(self.root) # take all files in the root directory

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

return len(self.files)

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

#sample, label = torch.load(os.path.join(self.root, self.files[idx])) # load the features of this sample

sample = plt.imread(os.path.join(self.root, self.files[idx]))

sample = np.moveaxis(sample,-1,0)

#label = np.array(float((self.files[idx][-20:-18]))-self.label\_mean).astype(float)

label = np.array(float((self.files[idx][-20:-18]))).astype(float)

#print(label)

sample,label = torch.from\_numpy(sample).type(torch.FloatTensor),torch.from\_numpy(label).type(torch.FloatTensor)

#label = label.view(-1,1)

return sample, label

class CNN(nn.Module):

def \_\_init\_\_(self, in\_channels, out\_channels):

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

#Start the CNN architecture

#First CONV layer (16-filters)

self.conv1 = nn.Conv2d(in\_channels = in\_channels, out\_channels = 16, kernel\_size = 5, stride = 1)

#self.batch1 = nn.BatchNorm2d(16)

#First CONV -> POOL layer (16-filters)

self.conv2 = nn.Conv2d(in\_channels = 16, out\_channels = 16, kernel\_size = 5, stride = 1)

self.pool = nn.MaxPool2d(kernel\_size = 2, stride = 2)

#Second CONV -> POOL layer (32-filters)

self.conv3 = nn.Conv2d(in\_channels = 16, out\_channels = 32, kernel\_size = 5, stride = 1)

#self.batch2 = nn.BatchNorm2d(32)

#First and only CONV -> ReLU layer (32-filters)

self.conv4 = nn.Conv2d(in\_channels = 32, out\_channels = 32, kernel\_size = 5, stride = 1)

#self.relu = nn.ReLU(inplace=True)

#full connectivity layers

self.fc1 = nn.Linear(in\_features= 2048, out\_features = out\_channels)

def forward(self, x):

x = self.conv1(x)

x = self.pool(self.conv2(x))

x = self.pool(self.conv3(x))

x = self.conv4(x)

#x = self.conv5(x)

#print(x.shape)

x = torch.flatten(x, 1)

#print(x.shape)

out = self.fc1(x)

#out = self.fc2(x)

return out

def main(fixed):

source\_folder = "../datasets/cnn"

dataset\_train = MyDataset(folder=source\_folder, train=True)

dataset\_val = MyDataset(folder=source\_folder, train=False)

N\_train = dataset\_train.\_\_len\_\_()

N\_test = dataset\_val.\_\_len\_\_()

# defining the model

input\_dim = (3, 64, 64) #3 channels in 64 x 64 images

model = CNN(3, 1)

# Define loss

loss\_fn = nn.MSELoss(reduction='sum')

# Optimizer

if fixed == False:

learning\_rate = 1e-8 / N\_train

elif fixed == True:

learning\_rate = 1e-8 / N\_train

optimizer = optim.SGD(model.parameters(), lr = learning\_rate, momentum=0.9, weight\_decay = 0.0001)

# Move to GPU

model = model.to(device)

loss\_fn = loss\_fn.to(device)

test\_loss = []

train\_loss = []

R2\_test = []

R2\_train = []

for epoch in range(100):

print('EPOCH {}:'.format(epoch + 1))

batch\_size = 0

#set up for part a and b of hw

if fixed == False:

if epoch <= 20:

batch\_size = 64

elif epoch > 20 and epoch <= 40:

batch\_size = 128

elif epoch > 40 and epoch <= 60:

batch\_size = 256

elif epoch > 60 and epoch <= 80:

batch\_size = 512

else:

batch\_size = 1024

if epoch in list([0, 21, 41, 61, 81]):

loader\_train = DataLoader(dataset=dataset\_train, batch\_size = batch\_size)

loader\_test = DataLoader(dataset=dataset\_val, batch\_size = batch\_size)

optimizer = optim.SGD(model.parameters(), lr = learning\_rate / (epoch + 1), momentum=0.9, weight\_decay = 0.0001)

elif fixed == True:

batch\_size = 1024

loader\_train = DataLoader(dataset = dataset\_train, batch\_size = batch\_size)

loader\_test = DataLoader(dataset = dataset\_val, batch\_size = batch\_size)

running\_train\_loss = 0

running\_test\_loss = 0

predictions\_train, actuals\_train = [], []

#make sure gradient tracking is on

model.train()

for data in loader\_train:

inputs, labels = data

#inputs = (inputs - torch.mean(inputs)) / torch.std(inputs)

#print(inputs.shape,labels.shape)

# Zero gradients for every batch

optimizer.zero\_grad()

#send to gpu

inputs = inputs.to(device)

labels = labels.to(device)

#Make prediction for this batch

outputs = model(inputs)

labels = labels.unsqueeze(1)

#Compute loss and its gradient

loss = loss\_fn(outputs, labels)

loss.backward()

#adjust learning weight

optimizer.step()

#gather loss

running\_train\_loss += loss.item() / batch\_size

#need to get values outside of loop

model.eval()

with torch.no\_grad():

y\_predicted\_train\_batch = model(inputs)

y\_predicted\_train\_batch = y\_predicted\_train\_batch.cpu().numpy()

actual = labels.cpu().numpy() #Copy the tensor to cpu

actual = actual.reshape((len(actual), 1))

# store y values from model

for element in y\_predicted\_train\_batch:

predictions\_train.append(element)

for element in actual:

actuals\_train.append(element)

#print(np.array(predictions\_train).shape,np.array(actuals\_train).shape)

model.train()

train\_loss.append(running\_train\_loss / N\_train)

actuals\_train, predictions\_train = np.array(actuals\_train).flatten(), np.array(predictions\_train).flatten()

#actuals\_train, predictions\_train = torch.flatten(actuals\_train), torch.flatten(predictions\_train)

#print(actuals\_train.shape,predictions\_train.shape)

y\_bar\_train = np.mean(actuals\_train)

#assert actuals\_train.shape == predictions\_train.shape

R2\_train.append(1 - (np.sum(np.square(actuals\_train - predictions\_train)) / np.sum(np.square(actuals\_train - y\_bar\_train))))

predictions\_test, actuals\_test = [], []

#Do not need gradients to do validation

model.eval()

with torch.no\_grad():

for vdata in loader\_test:

vinputs, vlabels = vdata

#vinputs = (vinputs - torch.mean(vinputs)) / torch.std(vinputs)

y\_predicted\_on\_test\_batch = model(vinputs.to(device))

vlabels = vlabels.to(device)

vlabels = vlabels.unsqueeze(1)

#print(labels.size(),y\_predicted\_on\_test\_batch.size())

batch\_loss = loss\_fn(y\_predicted\_on\_test\_batch, vlabels)

running\_test\_loss += batch\_loss.item() / batch\_size

y\_predicted\_on\_test\_batch = y\_predicted\_on\_test\_batch.cpu().numpy()

actual = vlabels.cpu().numpy() #Copy the tensor to cpu

actual = actual.reshape((len(actual), 1))

# store test values

for element in y\_predicted\_on\_test\_batch:

predictions\_test.append(element)

for element in actual:

actuals\_test.append(element)

test\_loss.append(running\_test\_loss / N\_test)

actuals\_test, predictions\_test = np.array(actuals\_test).flatten(), np.array(predictions\_test).flatten()

#print(actuals\_test.shape,predictions\_test.shape)

#assert actuals\_test.shape == predictions\_test.shape

y\_bar\_test = np.mean(actuals\_test)

R2\_test.append(1 - (np.sum(np.square(actuals\_test - predictions\_test)) / np.sum(np.square(actuals\_test - y\_bar\_test))))

model.train()

print(R2\_test[-1], R2\_train[-1], test\_loss[-1], train\_loss[-1])

total\_params = 0

for name, parameter in model.named\_parameters():

if not parameter.requires\_grad: continue

params = parameter.numel()

print(name, "\tnum params:", params)

total\_params+=params

print(f"Total Trainable Params: {total\_params}")

torch.save(model.state\_dict(), os.path.join(source\_folder,"model","CNN\_A.pt"))

return R2\_test, R2\_train, test\_loss, train\_loss

R2\_test, R2\_train, test\_loss, train\_loss = main(fixed = False)

plt.plot(R2\_test,label ="R2 Test")

plt.plot(R2\_train,label="R2 Train")

plt.xlabel('Epoch')

plt.ylabel('R2')

plt.legend()

plt.savefig("R2\_varied")

plt.show()

plt.plot(test\_loss,label ="Test loss")

plt.plot(train\_loss,label="Train loss")

plt.xlabel('Epoch')

plt.ylabel('loss')

plt.legend()

plt.savefig("Loss\_varied")

plt.show()

R2\_test\_b, R2\_train\_b, test\_loss\_b, train\_loss\_b = main(fixed = True)

plt.plot(R2\_test\_b,label ="R2 Test")

plt.plot(R2\_train\_b,label="R2 Train")

plt.xlabel('Epoch')

plt.ylabel('R2')

plt.title("$R^2$ vs Epoch Number")

plt.legend()

plt.savefig("R2\_fixedbatch")

plt.show()

plt.plot(test\_loss\_b,label ="Test loss")

plt.plot(train\_loss\_b,label="Train loss")

plt.xlabel('Epoch')

plt.ylabel('loss')

plt.title("Loss vs Epoch Number")

plt.legend()

plt.savefig("Loss\_fixedbatch")

plt.show()

import pandas as pd

num\_param = 48145

results = {}

results['Layer 1'] = ['Conv2d', 16, 5, 1, 1216]

results['Layer 2'] = ['Conv2d 2x2 max pooling', 16, 5, 1, 6416]

results['Layer 3'] = ['Conv2d + 2x2 max pooling', 32, 5, 1, 12832]

results['Layer 4'] = ['Conv2d + ReLU activation', 32, 5, 1, 25632]

results['Layer 5'] = ['Linear', "2048/1",'','', 2049]

#results['Layer 6'] = ['Linear', "84/1",'','', 0]

results['Total Number of Parameters'] = [num\_param, '','','', '']

parameter\_list = pd.DataFrame(results, index = ['Layer Description' , 'Number of Filters', 'Kernel\_Size','Stride', 'Number of Parameters']).transpose()

parameter\_list

def predict():

model2 = CNN(3,1)

model2.load\_state\_dict(torch.load('../datasets/cnn/model/CNN\_A.pt'))

model2.eval()

source\_folder = "../datasets/cnn"

dataset\_train = MyDataset(folder=source\_folder,train=True)

dataset\_val = MyDataset(folder=source\_folder,train=False)

loader\_train = DataLoader(dataset=dataset\_train, batch\_size = 1024)

loader\_test = DataLoader(dataset=dataset\_val,batch\_size = 1024)

predictions\_train, actuals\_train = [], []

with torch.no\_grad():

for data in loader\_train:

inputs, labels = data

y\_predicted\_train\_batch = model2(inputs).cpu().numpy()

actual = labels.cpu().numpy()

actual = actual.reshape((len(actual), 1))

# store

for element in y\_predicted\_train\_batch:

predictions\_train.append(element)

for element in actual:

actuals\_train.append(element)

predictions\_test, actuals\_test = [], []

with torch.no\_grad():

for data in loader\_test:

inputs, labels = data

y\_predicted\_on\_current\_batch = model2(inputs).cpu().numpy()

actual = labels.cpu().numpy()

actual = actual.reshape((len(actual), 1))

# store

for element in y\_predicted\_on\_current\_batch:

predictions\_test.append(element)

for element in actual:

actuals\_test.append(element)

return predictions\_train, actuals\_train, predictions\_test, actuals\_test

y\_pred\_train, y\_train, y\_pred\_test, y\_test = predict()

residuals\_train = np.array(y\_train) - np.array(y\_pred\_train)

residuals\_test = np.array(y\_test) - np.array(y\_pred\_test)

plt.scatter(y\_train, residuals\_train, color="blue", label="train")

plt.scatter(y\_test, residuals\_test, color="red", label="test")

plt.xlabel("$Y\_i$")

plt.ylabel("Residual")

plt.title("Residuals vs $Y\_i$")

plt.legend()

plt.savefig("Residuals.png")

plt.show()