**Machine Learning Assignment 2 Report**

**2016112083 김연웅**

**<Q1 - training a math function with 4 parameters to approximate the sine function>**

**Code**

#Q1

"""

We use our custom function to approximate the sine function.

"""

import torch

import math

class CustomFunction(torch.autograd.Function):

    @staticmethod

    def forward(ctx, input):

        ctx.save\_for\_backward(input)

        return 0.5 \* (5 \* input \*\* 3 - 3 \* input)

    @staticmethod

    def backward(ctx, grad\_output):

        input, = ctx.saved\_tensors

        # grad\_output: Tensor containing gradient of the loss with respect to the output.

        # computed the gradient of the loss with respect to the input which means bacward propagation.

        return grad\_output \* 0.5 \* (15 \* input \*\* 2 - 3)

dtype = torch.float

device = torch.device("cuda:0")

x = torch.linspace(-math.pi, math.pi, 2000, device=device, dtype=dtype)

y = torch.sin(x) # We approximate this sine function.

# In our model, we have 4 weights to train: y = a + b \* P3(c + d \* x).

# These weights need to be initialized.

# Setting requires\_grad=True indicates that we want to compute gradients with

# respect to these Tensors during the backward pass.

a = torch.full((), 0.0, device=device, dtype=dtype, requires\_grad=True)

b = torch.full((), -1.0, device=device, dtype=dtype, requires\_grad=True)

c = torch.full((), 0.0, device=device, dtype=dtype, requires\_grad=True)

d = torch.full((), 0.3, device=device, dtype=dtype, requires\_grad=True)

learning\_rate = 5e-6

for t in range(2000):

    P3 = CustomFunction.apply

    # Forward pass: predict y.

    # P3 using our custom backward function.

    y\_pred = a + b \* P3(c + d \* x)

    # Compute and print loss

    loss = (y\_pred - y).pow(2).sum()

    if t % 100 == 99:

        print(t, loss.item())

    # Use autograd to compute the backward pass.

    loss.backward()

    # Update weights using gradient descent

    with torch.no\_grad():

        a -= learning\_rate \* a.grad

        b -= learning\_rate \* b.grad

        c -= learning\_rate \* c.grad

        d -= learning\_rate \* d.grad

        # Manually zero the gradients after updating weights

        a.grad = None

        b.grad = None

        c.grad = None

        d.grad = None

print(f'Result: y = {a.item()} + {b.item()} \* P3({c.item()} + {d.item()} x)')

**Explanation**

First [FILL ME] part was about returning value of backward(ctx, grad\_output) function in class CustomKernel. This class inherits torch.autograd.Function and it implements custom pytorch autograd function for approximate the sine function. For [FILL ME] part I wrote “grad\_output \* 0.5 \* (15 \* input \*\* 2 - 3)”. This function “backward” is an implementation of back propagation progress using chain rule. I computed the gradient of the loss with respect to the input and returned it.

Second [FILL ME] part was about using pytorch autograd to compute backward pass. Above of this part, there are a line which ‘loss’ is defined. Since we are minimizing loss function of y\_pred with gradient descent method, we must compute gradient of loss function. The ‘loss’ variable is a tensor which contains computation of loss function of our prediction model. So by using tensor method ‘loss.backward()’ I performed backpropagation and saved the gradients in a.grad, b.grad, c.grad, d.grad respectively

Last 4 of [FILL ME] part is about updating gradients values respectively to zero after updating weight using gradient descent method. The code “with torch.no\_grad(): “ part means that corresponding blocks` computation won`t be tracked. According to googling, there are two ways to updating weights to zero. One is to put None just like I did on my assignment, and the other is using .zero\_grad(). I used None since [FILL ME] part for this assignment is intending.

**Execution Result**

For result, I found out interesting fact. When I changed first [FILL ME] part from 1)  to 2)  , although it is same equation with different expression , the execution result yielded different values for each parameters. Just in case, I attached both of result screen captures.

Result of 1) 



Result of 2)  ,



**<Q2 – Defining my own custom kernels(using kernel tricks)>**

**Code – Load Datasets**

#Q2

"""

We will implement many custom kernels. Try to improve the classification accuracy and F-1 scores.

"""

from sklearn.svm import SVC

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score

from sklearn.metrics import f1\_score

from sklearn.metrics import roc\_auc\_score

import sklearn

import pandas as pd

import numpy as np

from google.colab import files

import io

# Upload data

uploaded = files.upload()

# load data using pandas and cast them to numpy

X = pd.read\_csv(io.BytesIO(uploaded['trainX.csv']),header = None)

Y = pd.read\_csv(io.BytesIO(uploaded['trainY.csv']),header = None)

XTe = pd.read\_csv(io.BytesIO(uploaded['testX.csv']),header = None)

YTe = pd.read\_csv(io.BytesIO(uploaded['testY.csv']),header = None)

X = (X.to\_numpy())

Y = (Y.to\_numpy())

XTe = (XTe.to\_numpy())

YTe = (YTe.to\_numpy())

I used pandas library for loading data. Because I worked this project on Google Colaboratory, I used ‘file’ library from google.colab. This enables user to upload files. After uploads 4 dataset that we will use, I load each dataset using pandas.read\_csv function. After loading data I type casted pd.Dataframe to numpy for convenience of computation.

**Code – custom kernels**

# Custom kernels that I implemented

def euclid\_distance(X1, X2):

    # It have two parameter for each dataset.

    # It returns Euclid Distance between two dataset matrixes

    term1 = (X1 \*\* 2).sum(axis=1).reshape(-1,1)

    term2 = (X2 \*\* 2).sum(axis=1)

    term3 = -2 \* np.dot(X1,X2.T)

    return np.abs(term1 + term2 - term3)

def linear\_kernel(X1, X2):

    #  Linear kernel : k (xi , xj) = xi . xj

    return X1.dot(X2.T)

def poly\_kernel\_homo(X1,X2,degree=2):

    #Polynomial (homogeneous) : k(xi,xj) = (xi . xj)^d

    return (X1.dot(X2.T)) \*\* degree

def poly\_kernel\_inhomo(X1,X2,degree=2, c=10):

    #Polynomial (homogeneous) : k(xi,xj) = (xi . xj + c)^d

    return (X1.dot(X2.T) + c) \*\* degree

def rbf\_kernel(X1,X2,gamma = 0.1):

    #  Gaussian radial basis function (rbf) : k(xi, yj) = e^(-gamma||xi - yj||^2)

    return np.exp(-gamma \* euclid\_distance(X1,X2))

def cosine\_kernel(X1,X2):

    # Cosine :  K(xi, xj) = xi.xj / (||x|| . ||y||)

    term1 = np.sqrt((X1 \*\* 2).sum(axis=1)).reshape(-1, 1)

    term2 = np.sqrt((X2 \*\* 2).sum(axis=1)).reshape(-1, 1)

    return X1.dot(X2.T) / (term1.dot(term2.T))

def multiquadric\_kernel(X1,X2,c=2):

    # multiquadric :    k(xi, yj) = 1 / sqrt(|| xi- yj ||^2 + c^2)

    return np.sqrt(euclid\_distance(X1,X2) + c \*\* 2)

def cauchy\_kernel(X1,X2, sigma= 2):

    # cauchy : k(xi, xj) = 1 / (1 + || xi - xj ||^2 / sigma ^ 2)

    distance = euclid\_distance(X1, X2)

    return 1 / (1 + distance / sigma)

def tstudent\_kernel(X1,X2, d=2):

    # tstudent :  k(xi , xj) = 1 / (1 + ||xi - yj||^d)

    return 1 / (1 + (euclid\_distance(X1,X2) \*\* d / 2))

def log\_kernel(X1,X2, d=2):

    #  k(xi, xj) = -log(|| xi-xj ||^d) + 1

    return -np.log(euclid\_distance(X1, X2) \*\* d / 2.) + 1

def thin\_plate\_kernel(X1,X2,n=3):

    # thin-plate: k(xi, xj) = || xi-xj ||^2n+1

    return euclid\_distance(X1, X2) \*\* ((2\*n+1) / 2)

**Explanation**

* SUB function -> Euclid\_distance(X1,X2)

For convenience of implementation, I defined sub-helper function name Euclid\_distnace. This returns Euclidean distance of given Data Input.

* About *“Kernel Trick”*

In the assignment explanation, there are a statement about kernel trick (*You must use a kernel trick if available (i.e., reducing complexity when a workaround is available). If not using a trick, when available, you may not have full credits.*) I was little unclear about this statement.

What I understood about “Kernel Trick” in the lecture is that 1)We can apply kernel which expands datasets to higher dimension in order to enable classification. 2) When we are applying kernel, we can do calculation in the kernel function first, in order to avoid cumbersome calculation in much higher dimension.

And I think 2) part is kernel trick, and in this assignment defining customize kernel(my codes) is equal as *“using kernel trick”.* Since it is inevitable to use kernel tricks when I implement custom kernel functions, I had a confuse when I read the explanation saying “*If not using a trick, when available, you may not have full credits”.*

In conclusion, I defined 10 custom kernel functions using kernel tricks.

* (I referred to information of 10 different kernels from this source : https://www.longdom.org/open-access/a-complete-list-of-kernels-used-in-support-vector-machines-2167-0501-1000195.pdf)
* Some of kernel functions require several additional parameters such as gamma, degree, n. I set those parameters in function definition, in a way that yields the highest accuracy and f1-score through several tests.

**Code for Test and Result Analysis**

Y= Y.reshape(160,)

# apply custom kernels to SVM

clf = SVC()

clf = SVC(kernel = linear\_kernel, random\_state=2011)

clf.fit(X, Y)

yp = clf.predict(XTe)

print("Linear Kernel DIY")

print("AcuuracyScore: ", accuracy\_score(YTe, yp))

print("F1 score: ", f1\_score(YTe, yp, average='macro'))

print()

clf = SVC()

clf = SVC(kernel = 'linear', random\_state=2011)

clf.fit(X, Y)

yp = clf.predict(XTe)

print("Linear Kernel sklearn version")

print("AcuuracyScore: ", accuracy\_score(YTe, yp))

print("F1 score: ", f1\_score(YTe, yp, average='macro'))

print()

clf = SVC(kernel = poly\_kernel\_homo, random\_state=2011)

clf.fit(X, Y)

yp = clf.predict(XTe)

print("Homo Polynomial Kernel DIY")

print("AcuuracyScore: ", accuracy\_score(YTe, yp))

print("F1 score: ", f1\_score(YTe, yp, average='macro'))

print()

clf = SVC(kernel = 'poly', gamma=1,degree=2, coef0=0, random\_state=2011)

clf.fit(X, Y)

yp = clf.predict(XTe)

print("Homo Polynomial Kernel sklearn version")

print("AcuuracyScore: ", accuracy\_score(YTe, yp))

print("F1 score: ", f1\_score(YTe, yp, average='macro'))

print()

clf = SVC(kernel = poly\_kernel\_inhomo, random\_state=2011)

clf.fit(X, Y)

yp = clf.predict(XTe)

print("Inhomo Polynomial Kernel DIY")

print("AcuuracyScore: ", accuracy\_score(YTe, yp))

print("F1 score: ", f1\_score(YTe, yp, average='macro'))

print()

clf = SVC(kernel = "poly",gamma= 1, degree = 2, coef0= 10, random\_state=2011)

clf.fit(X, Y)

yp = clf.predict(XTe)

print("Inhomo Polynomial Kernel sklearn version")

print("AcuuracyScore: ", accuracy\_score(YTe, yp))

print("F1 score: ", f1\_score(YTe, yp, average='macro'))

print()

clf = SVC(kernel = rbf\_kernel, random\_state=2011)

clf.fit(X, Y)

yp = clf.predict(XTe)

print("Gaussian Radial Basis Function Kernel DIY version")

print("AcuuracyScore: ", accuracy\_score(YTe, yp))

print("F1 score: ", f1\_score(YTe, yp, average='macro'))

print()

clf = SVC(kernel = 'rbf', gamma = 0.1, random\_state=2011)

clf.fit(X, Y)

yp = clf.predict(XTe)

print("Gaussian Radial Basis Function Kernel sklearn version")

print("AcuuracyScore: ", accuracy\_score(YTe, yp))

print("F1 score: ", f1\_score(YTe, yp, average='macro'))

print()

clf = SVC(kernel = cosine\_kernel, random\_state=2011)

clf.fit(X, Y)

yp = clf.predict(XTe)

print("Cosine Kernel")

print("AcuuracyScore: ", accuracy\_score(YTe, yp))

print("F1 score: ", f1\_score(YTe, yp, average='macro'))

print()

clf = SVC(kernel = multiquadric\_kernel, random\_state=2011)

clf.fit(X, Y)

yp = clf.predict(XTe)

print("Multiquadric Kernel")

print("AcuuracyScore: ", accuracy\_score(YTe, yp))

print("F1 score: ", f1\_score(YTe, yp, average='macro'))

print()

clf = SVC(kernel = cauchy\_kernel, random\_state=2011)

clf.fit(X, Y)

yp = clf.predict(XTe)

print("Cauchy Kernel")

print("AcuuracyScore: ", accuracy\_score(YTe, yp))

print("F1 score: ", f1\_score(YTe, yp, average='macro'))

print()

clf = SVC(kernel = tstudent\_kernel, random\_state=2011)

clf.fit(X, Y)

yp = clf.predict(XTe)

print("Tstudent Kernel")

print("AcuuracyScore: ", accuracy\_score(YTe, yp))

print("F1 score: ", f1\_score(YTe, yp, average='macro'))

print()

clf = SVC(kernel = log\_kernel, random\_state=2011)

clf.fit(X, Y)

yp = clf.predict(XTe)

print("Log Kernel")

print("AcuuracyScore: ", accuracy\_score(YTe, yp))

print("F1 score: ", f1\_score(YTe, yp, average='macro'))

print()

clf = SVC(kernel = thin\_plate\_kernel, random\_state=2011)

clf.fit(X, Y)

yp = clf.predict(XTe)

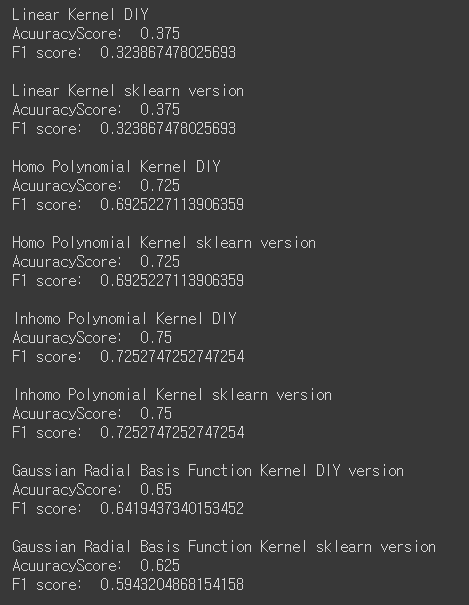
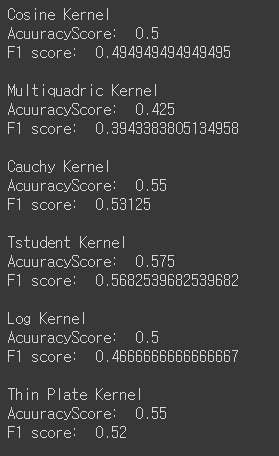
print("Thin Plate Kernel")

print("AcuuracyScore: ", accuracy\_score(YTe, yp))

print("F1 score: ", f1\_score(YTe, yp, average='macro'))

print()

**Result**

* For kernels which exist in sklearn as default API (such as linear, poly-homo poly-inhomo, rbf), I compared output values between sklearn version and DIY version.
* Although, I carefully set parameters and implementation of kernel, there are slight difference in accuracy score and F1 score between my custom version rbf kernel and sklearn version rbf kernel function.
* In case of linear, poly-homo, poly-inhomo kernels, they all yielded exact same values in accuracy score and f1 score
* Two of my custom kernels(homo polynomial kernel, and inhomo polynomial kernel) showed accuracy scores 0.725, 0.75 and f1-score 0.6925, 0.7252 respectively, which is larger than limitation of assignment2,( 0.7 for accuracy and 0.67 for f1-score).