Homework 4

Pan Du

December 12, 2022

Problem 1:

A. The filter is implemented inside the kernel class:

```
class kernel:
    def __init__(self, K, R, d, J, lamb_max):
        # -- filter properties
        self.R = float(R)
        self.J = J
        self.K = K
        self.d = d
        self.lamb_max = torch.tensor(lamb_max)

        # -- Half-Cosine kernel
        self.a = self.R*np.log10(self.lamb_max.detach().numpy())/(self.J-self.R+1)
        self.g_hat = lambda lamb:sum([ele*np.cos(2*np.pi*k*(lamb/self.a+0.5))*1*(-
                                                lamb>=0 and -lamb<self.a) for k,
                                                ele in enumerate(self.d)])

    def wavelet(self, lamb, j):
        """
            constructs wavelets ($j\in [2, J]$).
        :param lamb: eigenvalue (analogue of frequency).
        :param j: filter index in the filter bank.
        :return: filter response to input eigenvalues.
        """
        # compute filter at this sacle:
        assert(j>=2 and j<=self.J), 'j must be between [2,J]'

        # calculate critical value
        lamb_star = 10**(self.a*((j-1)/self.R-1))
        if  lamb < lamb_star:
            # print(lamb)
            lamb = lamb_star
        return torch.tensor(self.g_hat(np.log10(lamb) - self.a*(j-1)/self.R))
    def scaling(self, lamb):
        """
            constructs scaling function (j=1).
        :param lamb: eigenvalue (analogue of frequency).
        :return: filter response to input eigenvalues.
        """
        return torch.tensor(np.sqrt(self.R*self.d[0]**2+ self.R/2*sum([ele**2 for ele
```

```
                                                    in self.d])-sum([abs(self.wavelet
                                                    (lamb,i))**2 for i in range(2,self
                                                    .J+1)])))
```

One thing worth noting is that when $\lambda \to 0$. The value "$log(\lambda)$" in the translated and dilated input value "$\log(\lambda) - \frac{a(j-1)}{R}$" will go to "$-\infty$" and return "Nan" values due to numerical error. According to equation (1),

$$\hat{g}(\lambda) := \sum_{k=0}^{\mathcal{K}} d_k \left[\cos\left(2\pi k \left(\frac{\lambda}{a} + \frac{1}{2}\right)\right) \cdot \mathbf{1}\{0 \le -\lambda < a\} \right] \tag{1}$$

if the input value of $\hat{g}(x = \log(\lambda) - \frac{a(j-1)}{R})$ satisfies "$-x \ge a$", the term "$\mathbf{1}\{0 \le x < a\}$" will always return 0. Hence, to avoid $\lambda \to 0$, we can set "$-x = a$" when "$-x > a$", in other words, setting"$\lambda = \lambda^*$" when $\lambda > \lambda^*$ where the critical value $\lambda^* = 10^{a(\frac{i-1}{R}-1)}$. This is reflected in the code snippet:

```
        # calculate critical value
        lamb_star = 10**(self.a*((j-1)/self.R-1))
        if  lamb < lamb_star:
            # print(lamb)
            lamb = lamb_star
        return torch.tensor(self.g_hat(np.log10(lamb) - self.a*(j-1)/self.R))
```

B. Plot out the filters as follow:

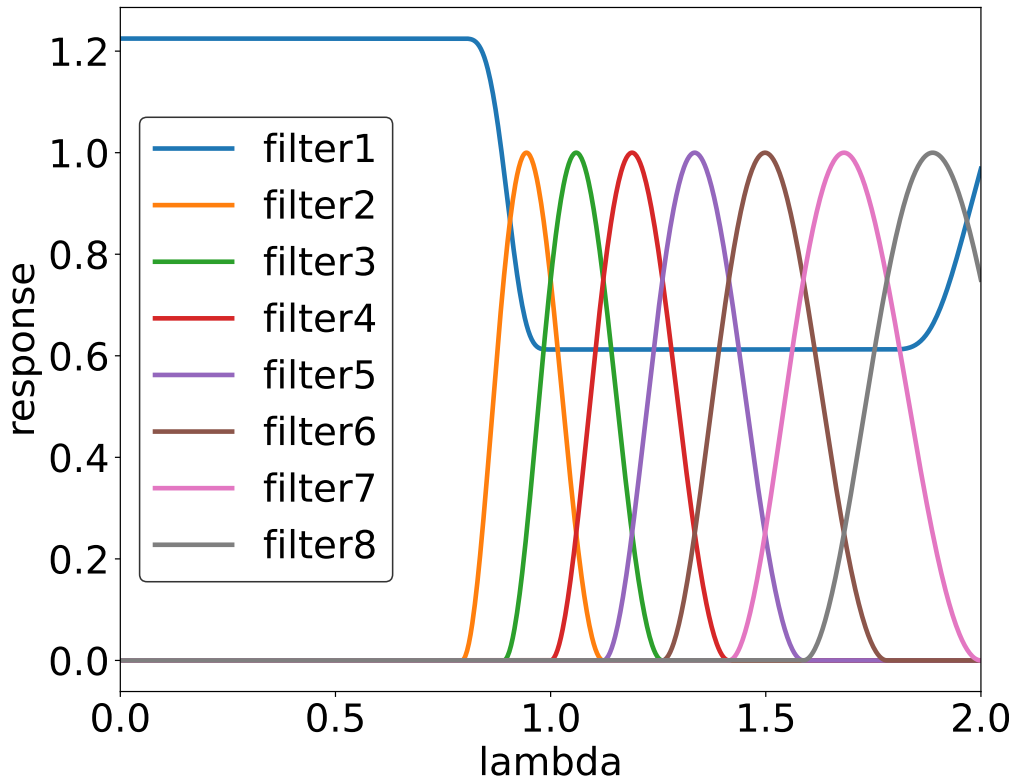

Figure 1: filter bank

Problem 2:

A. The scatter feature map is constructed in scatering class

```
class scattering(nn.Module):
    def __init__(self, J, L, V, d_f, K, d, R, lamb_max):
        super(scattering, self).__init__()

        # -- graph parameters
        self.n_node = V
        self.n_atom_features = d_f

        # -- filter parameters
        self.K = K
        self.d = d
        self.J = J
        self.R = R
        self.lamb_max = lamb_max
        self.filters = kernel(K=1, R=3, d=[0.5, 0.5], J=8, lamb_max=2)

        # -- scattering parameters
        self.L = L  # number of layers

    def compute_spectrum(self, W):
        """
            Computes eigenvalues of normalized graph Laplacian.
        :param W: tensor of graph adjacency matrices.
        :return: eigenvalues of normalized graph Laplacian
        """

        # -- computing Laplacian
        # L = ...
        # W = W+torch.eye(W.size()[1])
        L = torch.diag(W.sum(1)) - W

        # -- normalize Laplacian
        diag = W.sum(1)
        dhalf = torch.diag_embed(1. / torch.sqrt(torch.max(torch.ones(diag.size()),
                                                           diag)))
        L = torch.chain_matmul(dhalf,L,dhalf)

        # -- eig decomposition
        E, V = torch.symeig(L, eigenvectors=True)
        return abs(E), V

    def filtering_matrices(self, W):
        """
            Compute filtering matrices (frames) for spectral filters
        :return: a collection of filtering matrices of each wavelet kernel and the
                                                           scaling function in the filter-
                                                           bank.
        """

        filter_matrices = []
        E, V = self.compute_spectrum(W)
        filter_matrices.append(V @ torch.diag(torch.tensor([self.filters.scaling(ele)
```

```
                                                    for ele in E])) @ V.T)

        # -- wavelet frame
        for j in range(2, self.J+1):

            filter_matrices.append(V @ torch.diag(torch.tensor([self.filters.wavelet(
                                                        ele,j) for ele in E])) @ V.T)
        return torch.stack(filter_matrices)

    def forward(self, W, f):
        """
            Perform wavelet scattering transform
        :param W: tensor of graph adjacency matrices.
        :param f: tensor of graph signal vectors. f has shape of  num of vertices *
                                                    num of channels
        :return: wavelet scattering coefficients
        """
        # -- filtering matrices
        g = self.filtering_matrices(W)

        # --
        U_ = [f]

        # -- zero-th layer
        # S = ...  # S_(0,1)
        S = torch.mean(f,dim = 0)

        for l in range(self.L):
            U = U_.copy()
            U_ = []
            for f_ in U:
                for g_j in g:
                    U_.append(abs(g_j@f_))
                    # -- append scattering feature S_(l,i)
                    S_li = torch.mean(abs(g_j@f_),dim=0)
                    S = torch.cat((S, S_li))
        return S
```

Then we construct $z_{\mathbf{G}}$ using scattering feature map:

```
# -- initialize scattering function
scat = scattering(L=2, V=9, d_f=5, K=1, R=3, d=[0.5, 0.5], J=8, lamb_max=2)

# -- load data
data_size = 6000
data = MolecularDataset(N=data_size)
# print(data[0][1].size())
# -- Compute scattering feature maps
feature_maps = []
from tqdm import tqdm
for i in tqdm(range(data_size)):
    feature_maps.append(scat(data[i][0],data[i][1]))
feature_maps = torch.stack(feature_maps)
```

B. Implement PCA:

```
# -- PCA projection
from sklearn.decomposition import PCA
X = feature_maps.detach().numpy()
pca = PCA(n_components=2)
latent = pca.fit_transform(X)
```

Plot out the 2D map:

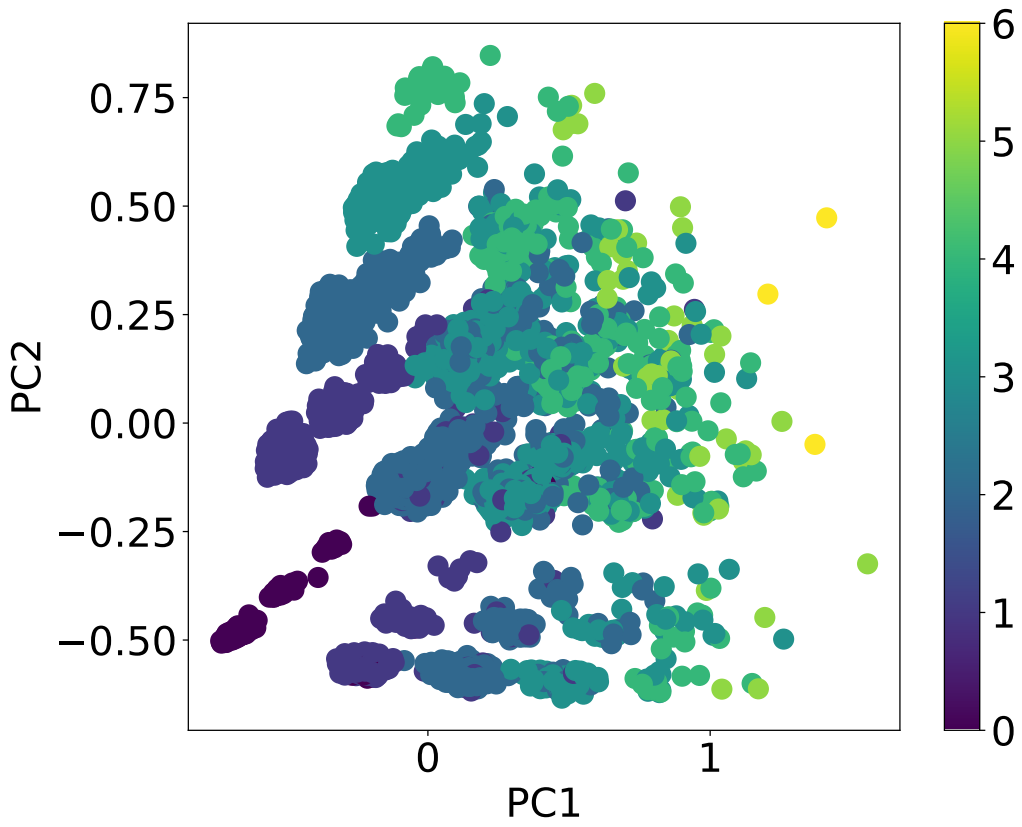

Figure 2: 2D map

C. Input and output number of features are 365 and 1 respectively. For the training, I used a MLP of the following structure:

```
My_model(
  (layers): Sequential(
    (conv_0): Linear(in_features=365, out_features=365, bias=True)
    (relu_0): ReLU()
    (conv_1): Linear(in_features=365, out_features=1, bias=True)
  )
)
```

Adam optimizer is used with initial learning rate “lr = 0.0001”. I also applied normalization to the

input along the dimension of number of samples.

$$\{\mathbf{f}_{normalized}\}_i = \frac{\{\mathbf{f}\}_i - f_{mean}}{f_{std}} \quad for \quad i = \{1, ..., N\} \tag{2}$$

where N=5000 is the total number of training samples. I trained it for 2000 epoches and here is the training error plot:

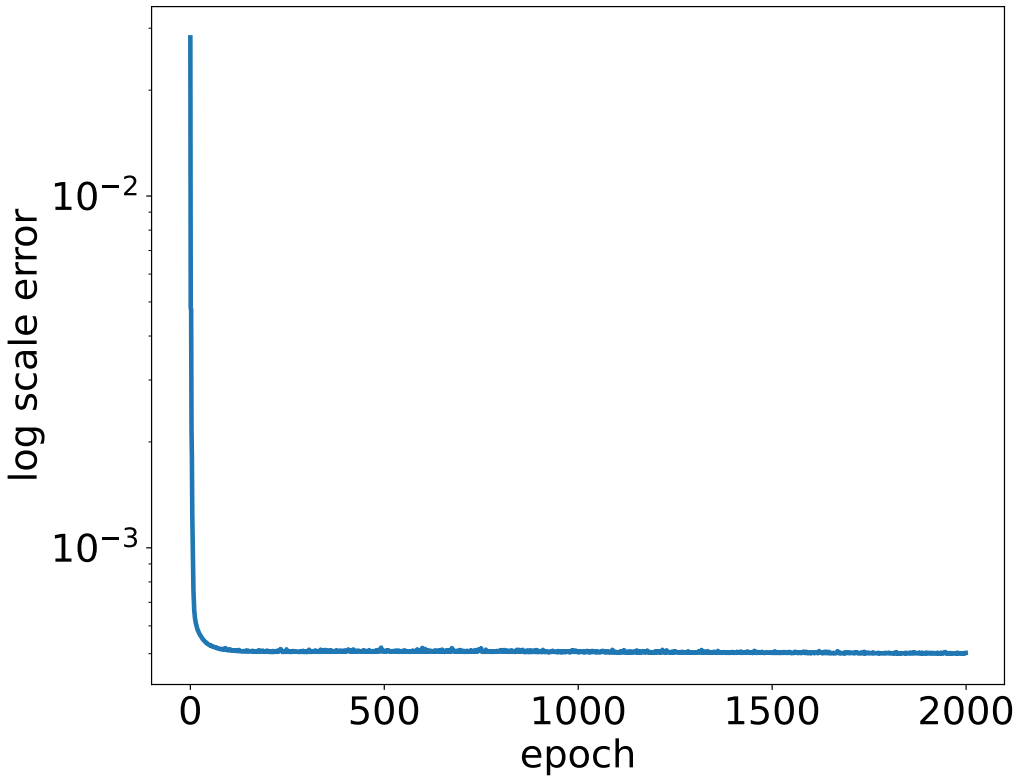

Figure 3: training error

D. Test it on 1000 samples, here is the result:

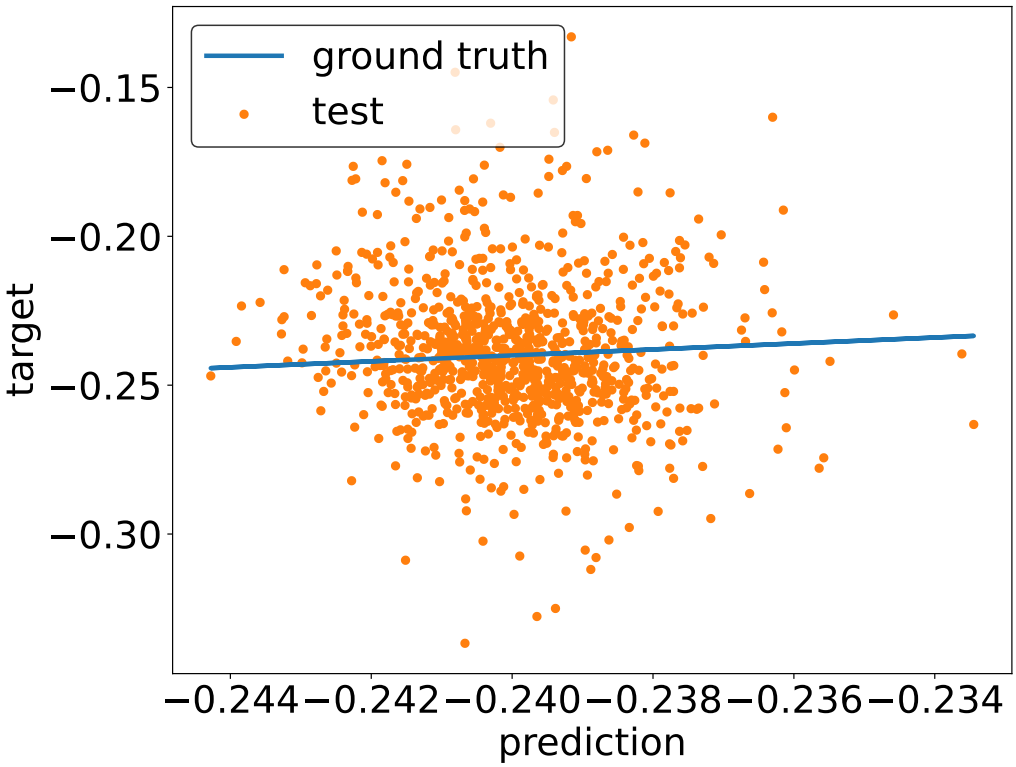

Figure 4: training error