Name: Honda Li Class: CSE 546 SPRING 2021 HW3A

A1: Conceptual Questions

A.1.a

Decraese σ. This makes the function $\exp\left(-\frac{\|u-v\|_2^2}{2\sigma^2}\right)$ thinner, make the inner product between different points more distinct.

A.1.b

True. It's an non-convex objective function, assuming thta deep means more than one hidden layer of course and assuming that activation function is not linear.

A.1.c

Yes it is. This ties to the fact that the Loss function of the Neural net is not convex, and if we start near all zeros, that means the first few iterations of the Gradient descent will aways get the same gradient, hugely limiting weight configuration of the model.

A.1.d

Flase, the what gives non linear decision boundary is the number of layers and the number of neurons in the layers, both ReLU and the Sigmoid functions is using linear decision boundary under the hood, **it's a lot of linear decision bounary combined together**.

A.1.e

A.2: Kernels And Bootstrap

Give na vector whose n th components paramterized by n is given by:

$$\frac{1}{\sqrt{n!}}\exp\left(\frac{-x^2}{2}\right)x^n$$

where x is one dimensional, and the feature mapping function $\phi(x)$ is an infinite dimensional function. Then:

$$\begin{aligned}
\langle\phi(x),\phi(y)\rangle &= \sum_{n=1}^{\infty}\phi_n(x)\phi_n y \qquad \text{(A.2.1)}\\
&= \sum_{n=1}^{\infty}\frac{1}{\sqrt{n!}}\exp\left(-\frac{x^2}{2}\right)x^n\frac{1}{\sqrt{n!}}\exp\left(-\frac{y^2}{2}\right)y^n\\
&= \sum_{n=1}^{\infty}\frac{(xy)^n}{n!}\exp\left(-\frac{x^2+y^2}{2}\right)\\
&= \exp\left(-\frac{x^2+y^2}{2}\right)\sum_{n=1}^{\infty}\frac{(xy)^n}{n!}\\
&= \exp\left(-\frac{x^2+y^2}{2}\right)\exp\left(xy\right)\\
&= \exp\left(-\frac{x^2+y^2}{2}+\frac{2xy}{2}\right)\\
&= \exp\left(\frac{-(x-y)^2}{2}\right)
\end{aligned}$$

And this is the RBF kernel for a scalar, in the 1d case.

A.3: Kernel Ridge Regression

A.3.a

To implement, we will need to take care of the process of solving for the best parameter α, and we will also need to careful about the offset. So what we are going to train is on the zero mean data, and then the prediction made from the model will have to add back the offset from the training set of course.
Here is basically what we had from the sections:

$$\begin{aligned} &\frac{1}{2}\nabla_x[\|K\alpha - y\|_2^2 + \lambda\alpha^T K\alpha] = 0 \qquad \text{(A.3.a)} \\ \implies\ & K^T(K\alpha - y) + \lambda K\alpha = 0 \\ & K(K\alpha - y) + \lambda K\alpha = 0 \\ & KK\alpha - Ky + \lambda K\alpha = 0 \\ & KK\alpha + \lambda K\alpha = Ky \\ & K\alpha + \lambda\alpha = y \\ & \alpha = (K + \lambda I)^{-1}y \end{aligned}$$

Where, K and y are from the training set. And in this case, the predictor can be computed via: $K_{\text{test,train}}\alpha$ where the $K_{\text{test,train}}$ is computed via:

$$K_{i,j} = \langle \phi(X_{\text{test}}[i,:]), \phi(X_{\text{train}}[:,j]) \rangle$$

And this is the code I implemented for the Kenrel Ridge Regression:

```
### This is a script for CSE 546 SPRING 2021, HW3, A.3
### Implementing the kernel ridge regression and visualize some stuff.
### Author: Hongda Li

import numpy as np
from scipy import linalg
import matplotlib.pyplot as plt

linspace = np.linspace
randn = np.random.randn
pinvh = linalg.pinvh
inv = linalg.inv
eye = np.eye
mean = np.mean
std = np.std

class KernelRidge:

    def __init__(this, regularizer_lambda, kernelfunc: callable):
        """

        :param regularizer_lambda:
        :param kernelfunc:
            Takes in the WHOLE traning matrix and compute the kenrnel matrix K.

        """
        this.Lambda = regularizer_lambda
        this.KernelMatrix = None
        this.X = None
        this.Kernel = kernelfunc
        this.Alpha = None
        this.Bias = None
    @property
    def w(this):
```

```
        if this.X is None: return None
        return this.X.T@this.Alpha

    def fit(this, X, y):
        """

        :param x:
        :param y:
        :return:
        """
        assert type(X) is np.ndarray and type(y) is np.ndarray, "X, y, must be numpy array"
        assert X.ndim == 2 and y.ndim <= 2
        Warn = "X, y dimension problem"
        if y.ndim == 2:
            assert y.shape[0] == X.shape[0], Warn
            assert y.shape[1] == 1, Warn
        else:
            assert y.shape[0] == X.shape[0], Warn
            y = y[:, np.newaxis]
        assert X.shape[0] >= 1, "Need more than just one sample. "
        # Standardized.
        this.X = X
        n, d = X.shape
        Lambda = this.Lambda
        K = this.Kernel(this.X, this.X)
        assert K.ndim == 2 and K.shape[0] == K.shape[1] and K.shape[0] == n, \
            "your kernel function implemention is wrong, kernel matrix is having the wrong shape"
        assert np.all(np.abs(K-K.T) < 1e-9), "kernel matrix is not symmetric."
        this.KernelMatrix = K
        # get the bias

        # get the alpha.
        this.Alpha = pinvh(K + Lambda*eye(n))@y


    def predict(this, Xtest):
        assert this.X is not None, "Can't predict when not trained yet. "
        Xtrain = this.X
        return this.Kernel(Xtest, Xtrain)@this.Alpha



def main():
    def SimpleTest():
        N = 100
        w, b = 1, 0
        x = linspace(-1, 1, N)
        eps = randn(N)*0.1
        y = w*x + b + eps
        X = x[:, np.newaxis]
        def KernelFunc(X, Y):
            return X@Y.T
        Model = KernelRidge(regularizer_lambda=0.01, kernelfunc=KernelFunc)
        Model.fit(X, y)
        Yhat = Model.predict(X)
        plt.plot(x, y)
        plt.plot(x, Yhat)
        plt.show()
    SimpleTest()


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

A.3.b

To train the model and get the best hyperparameter, here are some of my strategies:

1. Use "sklearn.metrics.pairwise" to compute the kernel matrix. Cause I am lazy and stupid so I just want smart people to write the code for me.
2. Use "sklearn.modelselection" to do cross val cause I am lazy.
3. For each parameter γ, λ we want to use, we draw a new set of 30 sample to test it.
4. Linear search on the degree of the poly kenel, bruteforcing out the one with minimal error across the validation set.
5. Section search on Gamma for the gaussian kernel. Partition a range into fine little sections, and use scipy optimizer to look for the minimal on all tiny little section. And then bruteforce out the optimal by comparing the best results from each section. Here we make the assumption that the crossval error function is at least, smooth and all of its derivatives are bonded.

Becase of the uncertainty of drawing the samples from the populations, best hyper parameters are just an estimate, which is not gurantee to be the same each time when we run the program. But I think it's close enough for our perposes.
This is the output from my code:

```
guassian kernel best is: [gamma, lambda] [172.91284562   0.21884506]
Poly kernel best is: [deg, lambda] (39, 0.030a18816458288159)
```

I think there is a balance between the regularization parameters and the kernel parameters, so the solution might not be unique.
Note: I tried out sone of the hyperparameter and found the best looking one. Also take note that the polynomial kernel doesn't extrapolate well. Because the larter the degree, the larger the regularization, to flatten it out, but when it goes out of the zoom of the training data, it loses its predictive ability.
And here is the code for the hyperparameter search:

```
### This is the script that produce plots and data for A3
### This is for CSE 546 SPRING 2021, HW3.
### Author: Hongda Alto Li
### Requries: kernel_ridge_regression.py


import numpy as np
cos, sin, pi = np.cos, np.sin, np.pi
rand, randn = np.random.rand, np.random.randn
norm = np.linalg.norm
zeros = np.zeros
mean = np.mean
sum = np.sum
min = np.min
max = np.max
linspace = np.linspace
logspace = np.logspace
from kernel_ridge_regression import KernelRidge
from sklearn.metrics.pairwise import rbf_kernel, polynomial_kernel
from sklearn.model_selection import KFold
import matplotlib.pyplot as plt
from scipy.optimize import minimize

def RBFKernel(X, Y, gamma):
    """
        X is the row data matrix.
    :param x:
    :return:
    """

    return rbf_kernel(X, Y, gamma=gamma)
```

```
def MyPolyKernel(X, Y, d):
    """
        X is the row data matrix.
    :param x:
    :return:
    """
    if Y is None: X = Y
    return polynomial_kernel(X, Y, gamma=1, degree=d)


def main():
    n = 30w # Global
    f = lambda x: 4 * sin(pi * x) * cos(6 * pi * x ** 2)

    def GenerateXY():
        x = rand(n)
        y = f(x) + randn(n)
        return x[:, np.newaxis], y

    def CrossValErrorEstimate(X, y, regualarizer, kernelfunc, param_norm=False):
        Errors = []
        AlphaNorm = []
        kf = KFold(n_splits=n, shuffle=False)
        for TrainIdx, TestIdx in kf.split(X):
            Model = KernelRidge(regularizer_lambda=regualarizer, kernelfunc=kernelfunc)
            Model.fit(X[TrainIdx], y[TrainIdx])
            yhat = Model.predict(X[TestIdx])
            Error = sum(yhat - y[TestIdx])**2
            Errors.append(Error)
            AlphaNorm.append(norm(Model.w, np.inf))
        if param_norm:
            return mean(Errors), min(AlphaNorm)
        return mean(Errors)

    def PolyKernlExample():
        X, y = GenerateXY()

        Model = KernelRidge(regularizer_lambda=0,
                            kernelfunc=lambda X, Y: MyPolyKernel(X, Y, 20))
        Model.fit(X, y)
        x = np.linspace(0, 1, 1000)

        yhat = Model.predict(x[:, np.newaxis])
        plt.scatter(X.reshape(-1), y, c="red")
        plt.plot(x, yhat)
        plt.show()

    PolyKernlExample()

    def PolyKernelHypertune():
        MaxDegree = 40
        BestParameter = None
        SmallestError = float("inf")
        Errors = {}
        for degree in range(1, MaxDegree + 1):
            Lambda = 2**(-10)
            while True:
                X, y = GenerateXY()
                Error, AlphaNorm = CrossValErrorEstimate(
                    X, y, Lambda, lambda X, Y: MyPolyKernel(X, Y, degree), True
                )
                Errors[Lambda, degree] = Error
                if Error < SmallestError:
                    SmallestError = Error
                    BestParameter = (degree, Lambda)
                    print(f"Best param updated [deg, lambda]:{BestParameter} ")
                if AlphaNorm < 1e-4:
                    break
```

```
                Lambda *= 1.1
        print(f"Best hyperparam Seems to be: {BestParameter}")
        return BestParameter

    def GuassianKernelHypertune():
        # Grid search, Fix the training sample
        X, y = GenerateXY()
        def GetError(gamma, l):
            l, gamma = abs(l), abs(gamma)
            Kernelfun = lambda x, y: RBFKernel(x,y, gamma=gamma)
            Error = CrossValErrorEstimate(X, y, regualarizer=l, kernelfunc=Kernelfun)
            return Error
        # A bunch of guesses!
        Gammas, Lambdas = linspace(0, 30, 10), logspace(-10, 0, num=2, base=2)
        BestError = float("inf")
        BestParams = None
        for gammastart, gammaend in zip(Gammas[:-1], Gammas[1:]):
            GammaGuess = (gammastart + gammaend)/2
            for lstart, lend in zip(Lambdas[:-1], Lambdas[1:]):
                LambdaGuess = (lstart + lend)/2
                Res = minimize(lambda x: GetError(x[0], x[1]),
                               np.array([GammaGuess, LambdaGuess]))
                Xmin = Res.x
                Fval = Res.fun
                if Fval < BestError:
                    BestError = Fval
                    BestParams = np.abs(Xmin)
                    print(f"Best parem updated, [gamma, lambda] {BestParams}")
        print(f"Guassian Bestparams: {BestParams}")
        return BestParams
    GaussianBest = GuassianKernelHypertune()
    PolyBest = PolyKernelHypertune()
    print(f"guassian kernel best is: [gamma, lambda] {GaussianBest}")
    print(f"Poly kernel best is: [deg, lambda] {PolyBest}")


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

A.3.c

A.3.d

A.3.e