

A1: Conceptual Questions

A.1.a

Decrease 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 a non-convex objective function, assuming that 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 always get the same gradient, hugely limiting weight configuration of the model.

A.1.d

False, 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 boundary combined together**.

A.1.e

A.2: Kernels And Bootstrap

Give a vector whose n th components parameterized 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:

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

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

A.3: Kernel Ridge Regression

For problem A.3, this is the core routine I used for the whole problem, filename: “kernel_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_implementation_is_wrong,_kernel_matrix_is_having_the_wrong_shape"
    assert np.all(np.abs(K-K.T) < 1e-5), "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.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 be 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:

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

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

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

To implement the method, I used the following tricks:

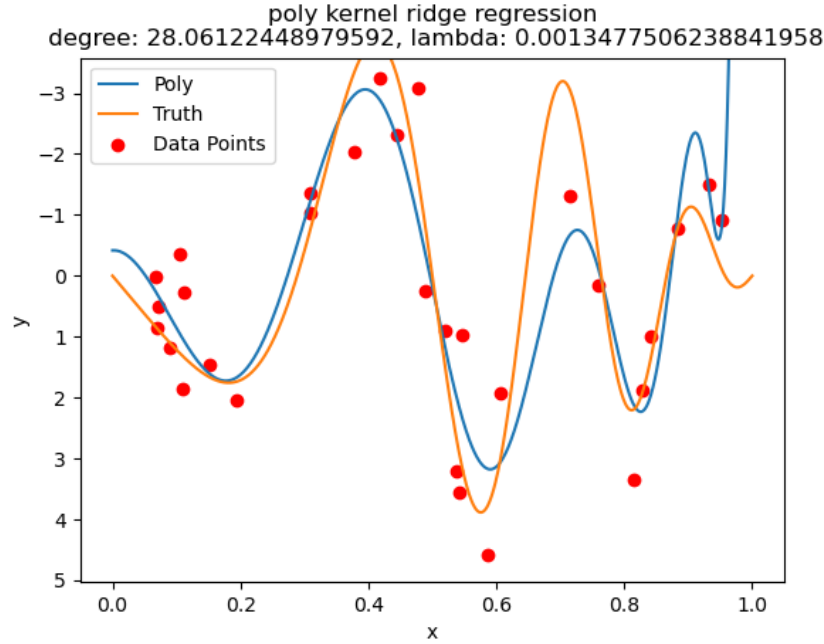
1. All the questions from a, b, c, d, e, are running on a fixed sample. When I do the black box optimization, and grid search for hyper parameter, I want to make the whole algorithm deterministic, given a certain sample set.
2. I used a black box optimization to improve the grid search algorithm, The algorithm is SHGO, Simplicial Homology Global Optimization. It's used as a bounded non-convex optimization algorithm as a subroutine for the grid search algorithm.
3. I used the footnote information to narrow down the search for the parameters of the Gaussian model.

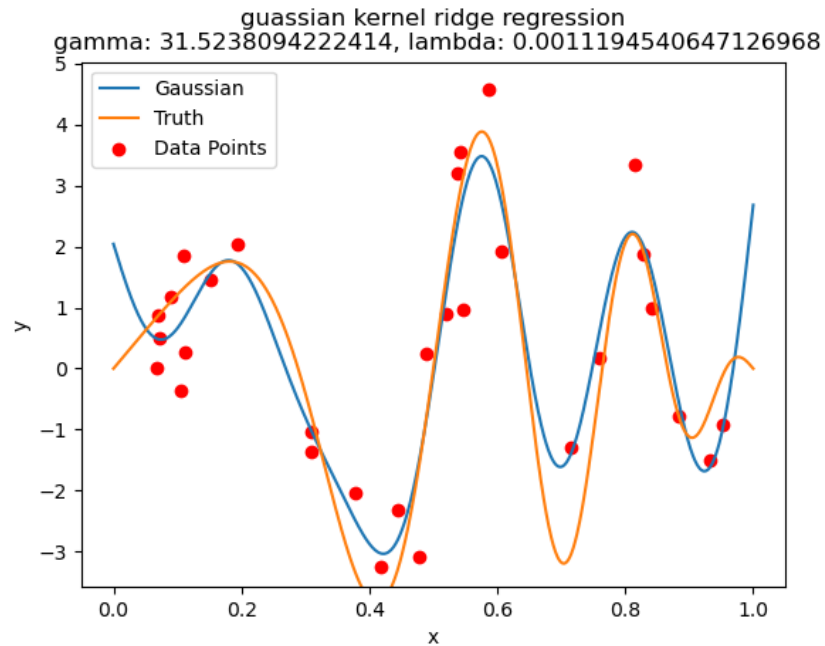
Note: That hyperparameter search algorithm I wrote runs forever. **Note:** For 30 samples, there are a lot of variance on the models, and here is one of the hyper parameter identified by my algorithm:

1. For poly kernel: $d = 28.06$, $\lambda = 0.0013477$.
2. For the KBF kernel: $\gamma = 31.5238$, $\lambda = 0.001119$.

A.3.b

This is the plot I had for the particular identified hyper parameters:



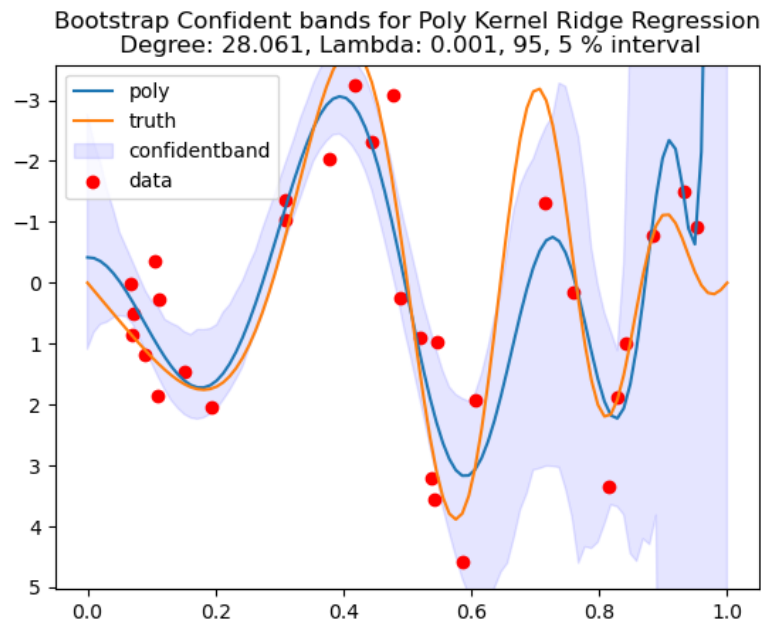


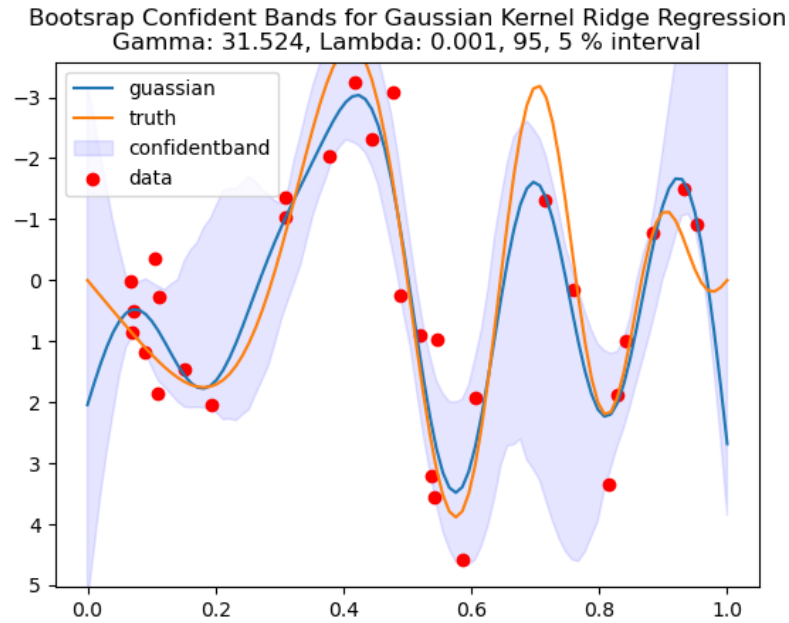
A.3.c

For building the bootstrap, this is something about the implementation that I think it's worth noting:

1. I am still using the same set of samples that produces the hyperparameters.
2. Because of huge variance for the poly model at the boundary of the data set, I have to scale the graph so that we can see what is happening, instead of zooming out like crazy.

Here is the graph for both the model:

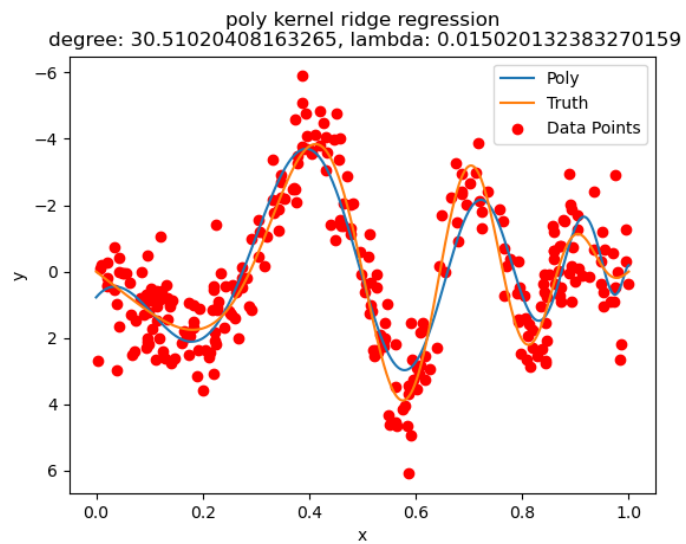


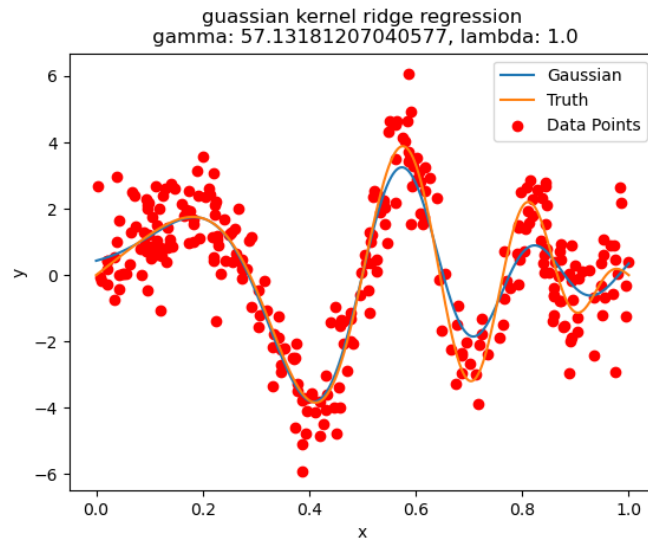


And there are a lot of variance. Very dependent on the sample too.

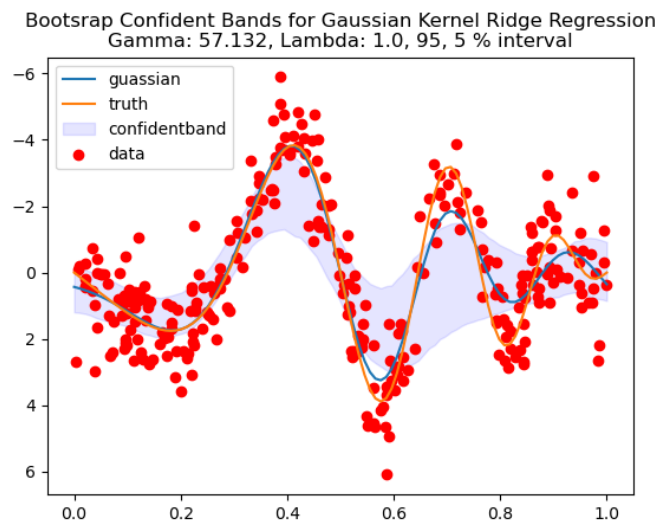
A.3.d

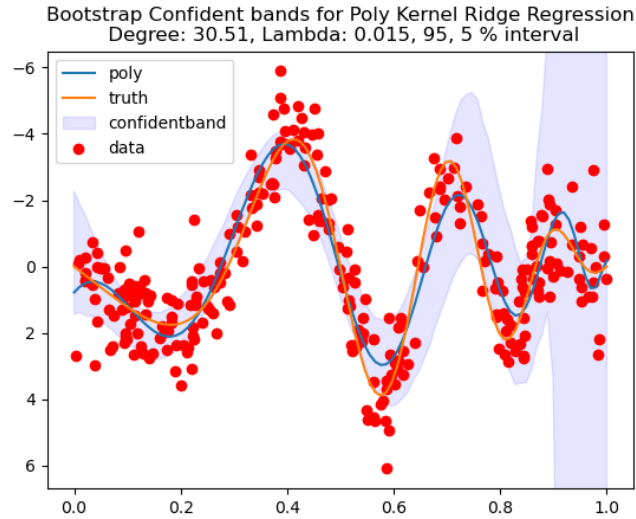
This is the repeated experiment with a sample size of 300 and a 10 Fold cross validation:





And this is the confident bands I placed on these models:





A.3.e

And for those 2 particular models I got on the previous part for the problem, I ran the algorithm and bootstrapped the difference, and it turns out that this is the confident interval for the L2 norm error difference:

$$(0.08477231069630578, 0.15565650293476457)$$

And it's larger than zero for both upper and lower bound. Therefore, the conclusion is that, the polynomial model is having a larger error.

A.3.Code

This is the main driver code I used for all the questions, it's quiet complicated so I have put it in the end of the section.

File name: "A3.py"

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

import numpy as np
cos, sin, pi = np.cos, np.sin, np.pi
rand, randn, randint = np.random.rand, np.random.randn, np.random.randint
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
percentile = np.percentile
var = np.var
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, shgo
from scipy.optimize import Bounds
```



```
### Some constant for the whole script:
```

```
def RBFKernel(X, Y, gamma):
    """
        Use this kernel to take the inner products between the columns of X, Y
    :param x:
    :return:
    """

    return rbf_kernel(X, Y, gamma=gamma)

def MyPolyKernel(X, Y, d):
    """
        Use this kernel to take the inner product between the columns of the X, Y
        matrix.
    :param x:
    :return:
    """
    if Y is None: X = Y
    return polynomial_kernel(X, Y, gamma=1, degree=d)

def CrossValErrorEstimate(X, y, regularizer, kernelfunc, split=None, param_norm=False):
    Errors = []
    AlphaNorm = []
    if split is None:
        split = X.shape[0]
    kf = KFold(n_splits=split, shuffle=False) # Make it deterministic given the same training
        sample.

    for TrainIdx, TestIdx in kf.split(X):
        Model = KernelRidge(regularizer_lambda=regularizer, kernelfunc=kernelfunc)
        Model.fit(X[TrainIdx], y[TrainIdx])
        yhat = Model.predict(X[TestIdx]).reshape(-1)
        Error = (sum(yhat - y[TestIdx])**2)/len(TestIdx)
        Errors.append(Error)
        AlphaNorm.append(norm(Model.w, np.inf))
    if param_norm:
        return mean(Errors), min(AlphaNorm)
    return mean(Errors)

def GenerateXY(n):
    f = lambda x: 4 * sin(pi * x) * cos(6 * pi * x ** 2)
    x = rand(n)
    x.sort()
    y = f(x) + randn(n)
    return x[:, np.newaxis], y

def main(n=30, KfoldSplit=30):
    X, y = GenerateXY(n) # THIS IS SHARED! FOR ALL
    f = lambda x: 4 * sin(pi * x) * cos(6 * pi * x ** 2)
    def PolyKernelHypertune():
        def GetError(deg, l):
            Kernefun = lambda x, y: MyPolyKernel(x, y, deg)
            Error = CrossValErrorEstimate(X,
                                          y,
                                          regularizer=l,
                                          kernelfunc=Kernefun,
                                          split=KfoldSplit)

            return Error
        BestError = float("inf")
        Best = None
```

```

for Deg in np.linspace(7, 31):
    Result = shgo(lambda x: GetError(Deg, x),
        bounds=[(0, 0.05)],
        n=100,
        sampling_method="simplicial",
        options={"f_tol": 1e-8}
    )
    if Result.fun < BestError:
        print(f"Poly_Kernel_Best_error_update_for_deg:_{Deg},_Lambda:_{Result.x},_Error:_{Result.fun}")
        BestError = Result.fun
        Best = (Deg, Result.x[0])
    else:
        print(f"failed_at:_deg:_{Deg},_Lambda:_{Result.x[0]},_Error:_{Result.fun}")

# Result = shgo(lambda x: GetError(x, 0),
#     bounds=[(1, 100)],
#     n=200, sampling_method="simplicial",
#     options={"f_tol": 1e-8, "disp": True})
# print(f"SHGO Optimization Results: {Result}")
return Best

def GaussianKernelHypertune():
    # Grid search, Fix the training sample
    # X, y = GenerateXY()
    Xs = X.reshape(-1)
    Distance = []
    for II in range(Xs.size):
        for JJ in range(II + 1, Xs.size):
            Distance.append(1/(norm(Xs[II] - Xs[JJ])**2))
    GammaLower, GammaHigher = \
        percentile(Distance, 25), percentile(Distance, 75)
    print(f"Hypertune_Gamma_kernel_search_range:_{GammaLower},_GammaHigher}")
    def GetError(gamma, l):
        l, gamma = abs(l), abs(gamma)
        Kernelfun = lambda x, y: RBFKernel(x,y, gamma=gamma)
        Error = CrossValErrorEstimate(X,
            Y,
            regularizer=l,
            kernelfunc=Kernelfun,
            split=KfoldSplit
        )
        return Error
    # GRID SEARCH INITIAL GUESS
    Result = shgo(lambda x: GetError(x[0], x[1]),
        bounds=[(GammaLower, GammaHigher), (0, 1)],
        n=500, sampling_method='sobol',
        options={"f_tol": 1e-4, "disp": True})
    print("Optimization_results:_")
    print(Result)
    print(f"Guassain_Bestparams:_{Result.x}")
    return Result.x

# ===== Hyper Param! =====
GaussianBest = [20, 0.058]
PolyBest = [19, 0.05]

GaussianBest = GaussianKernelHypertune()
PolyBest = PolyKernelHypertune()

print(f"guassain_kernel_best_is:_[gamma,_lambda]_{GaussianBest}")
print(f"Poly_kernel_best_is:_[deg,_lambda]_{PolyBest}")

def DrawPolyModel():
    x = linspace(0, 1, 1000)
    Model = KernelRidge(regularizer_lambda=PolyBest[1],
        kernelfunc=
            lambda X, Y: MyPolyKernel(X, Y, PolyBest[0]))
    Model.fit(X, y)

```

```

plt.ylim([max(y)*1.1, min(y)*1.1])
plt.plot(x, Model.predict(x[:, np.newaxis]).reshape(-1))
plt.plot(x, f(x))
plt.scatter(X.reshape(-1), y, c="red")
plt.title(f"poly_kernel_ridge_regression\n_\n"
          f"degree:_{PolyBest[0]},_lambda:_{PolyBest[1]}")
plt.xlabel("x")
plt.ylabel("y")
plt.legend(["Poly", "Truth", "Data_Points"])
plt.savefig("A3b-poly.png")
plt.show()
return Model

BestPolyModel = DrawPolyModel()

def DrawGuassianModel():
    x = linspace(0, 1, 1000)
    Model = KernelRidge(regularizer_lambda=GaussianBest[1],
                        kernelfunc=
                        lambda X, Y: RBFKernel(X, Y, GaussianBest[0])
                        )
    Model.fit(X, y)
    plt.ylim([min(y)*1.1, max(y)*1.1])
    plt.plot(x, Model.predict(x[:, np.newaxis]).reshape(-1))
    plt.plot(x, f(x))
    plt.scatter(X.reshape(-1), y, c="red")
    plt.title(f"gaussian_kernel_ridge_regression\n_\n"
              f"gamma:_{GaussianBest[0]},_lambda:_{GaussianBest[1]}")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.legend(["Gaussian", "Truth", "Data_Points"])
    plt.savefig("A3b-gauss.png")
    plt.show()
    return Model

BestGaussianModel = DrawGuassianModel()

# -----
# Bootstrap and estimating the confident interval.

def BoopStraping(Kernelfunc, Lambda, x):
    # Given the kernel func, produce the confidence interval.
    BagOfModels = []
    UpperPercentile = []
    LowerPercentile = []
    print("Boopstraping_fitting_the_model")
    for _ in range(300):
        Indices = randint(0, n, 30)
        XTild, Ytild = X[Indices], y[Indices]
        Model = KernelRidge(
            kernelfunc=Kernelfunc,
            regularizer_lambda=Lambda
        )
        Model.fit(XTild, Ytild)
        BagOfModels.append(Model)
    ModelPredictions = np.array(
        [Model.predict(x[:, np.newaxis]).reshape(-1) for Model in BagOfModels])

    for II in range(ModelPredictions.shape[1]):
        UpperPercentile.append(percentile(ModelPredictions[:, II], 95))
        LowerPercentile.append(percentile(ModelPredictions[:, II], 5))

    return UpperPercentile, LowerPercentile

def BoopStrapModelDifference(GaussModel, PolyModel):
    m = 1000
    X, y = GenerateXY(m)
    AllMeanDiff = []

```

```

print("solving_A3(e)")
for _ in range(300):
    Idx = randint(0, m, m)
    Idx.sort()
    Idx = np.unique(Idx)
    Xtild, ytild = X[Idx], y[Idx]
    yhat1 = GaussModel.predict(Xtild).reshape(-1)
    yhat2 = PolyModel.predict(Xtild).reshape(-1)
    Var1 = var(yhat1 - ytild)
    Var2 = var(yhat2 - ytild)
    AllMeanDiff.append(Var2 - Var1)
    print(f"Bootstrap_sample:{_}, The_difference_of_MSE_is:{AllMeanDiff[-1]}")
UpperPercentile, LowerPercentile = \
    percentile(AllMeanDiff, 95), percentile(AllMeanDiff, 5)
return LowerPercentile, UpperPercentile

Xgrid = linspace(0, 1, 100)
UpperPercentile, LowerPercentile = BoopStraping(
    lambda x, y: MyPolyKernel(x, y, PolyBest[0]),
    PolyBest[1], Xgrid
)
# Plot the polynomial Boop strap,
plt.title("Bootstrap_Confident_bands_for_Poly_Kernel_Ridge_Regression\n"
          f"Degree:{round(PolyBest[0], 3)}, Lambda:{round(PolyBest[1], 3)}, 95, 5%_interval")
# plt.plot(Xgrid, UpperPercentile)
# plt.plot(Xgrid, LowerPercentile)
plt.ylim([max(y) * 1.1, min(y) * 1.1])
plt.plot(Xgrid, BestPolyModel.predict(Xgrid[:, np.newaxis]))
plt.plot(Xgrid, f(Xgrid))
plt.fill_between(Xgrid, UpperPercentile, LowerPercentile, color='b', alpha=.1)
plt.scatter(X.reshape(-1), y, c="red")
plt.legend(["poly", "truth", "confidentband", "data"])
plt.savefig("Poly-boopstraped.png")
plt.show()
## Plot the Gaussian Boopstrap
UpperPercentile, LowerPercentile = BoopStraping(
    lambda x, y: RBFKernel(x, y, GaussianBest[0]),
    GaussianBest[1], Xgrid
)
plt.title("Bootsrap_Confident_Bands_for_Gaussian_Kernel_Ridge_Regression\n"
          f"Gamma:{round(GaussianBest[0], 3)}, Lambda:{round(GaussianBest[1], 3)}, 95, 5%_interval")
# plt.plot(Xgrid, UpperPercentile)
# plt.plot(Xgrid, LowerPercentile)
plt.ylim([max(y) * 1.1, min(y) * 1.1])
plt.plot(Xgrid, BestGaussianModel.predict(Xgrid[:, np.newaxis]))
plt.plot(Xgrid, f(Xgrid))
plt.fill_between(Xgrid, UpperPercentile, LowerPercentile, color='b', alpha=.1)
plt.scatter(X.reshape(-1), y, c="red")
plt.legend(["guassian", "truth", "confidentband", "data"])
plt.savefig("gaussian-boopstraped.png")
plt.show()

# A3 Part (e). Additional Boopstrap to compare the models.
if n == 300 and KfoldSplit == 10:
    print("For_A3(e)_we_are_going_to_use_the_idea_of_"
          "boopstrap_to_find_the_confidence_interval_of_the_"
          "_best_MSE_model_minus_the_Poly_Model")

    LowerPercentile, UpperPercentile = \
        BoopStrapModelDifference(BestGaussianModel, BestPolyModel)
    print(f"The_Upper_and_lower_bound_of_the_confidence_interval_is:"
          f"{LowerPercentile, UpperPercentile}")
    with open("goodplots/A3e.txt", "a") as file:
        file.write(f"Upper, Lower_bound:{LowerPercentile, UpperPercentile}")

```

```

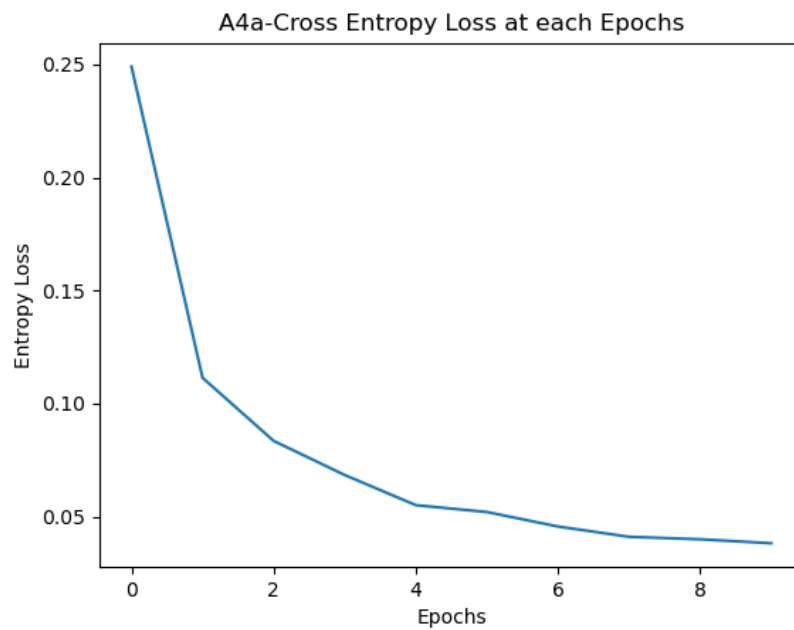
if __name__ == "__main__":
    import os
    print(f"cwd: {os.getcwd()}")
    main(n=300, KfoldSplit=10)
    #main()

```

A.4: Neural Networks for MNIST

A.4.a

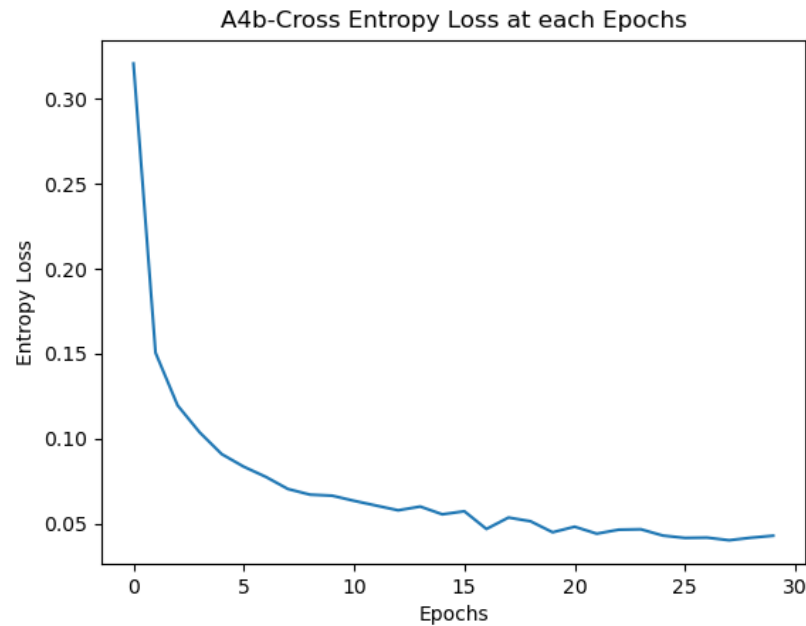
This is the graph I had for this the Entropy loss for each Epochs.



The code is in [A.4.code](#)

The score on the test set is 0.9704. (Portion of labels it got right)

A.4.b



The code is in [A.4.code](#)

The score on the test set is 0.9661. (Portion of labels it got right)

A.4.code

Here is my implementation of everything in this problem in one file:

File name: “neural_net_mnist.py”

```
# This is a code written for CSE 546 HW3 A4, in spring 2021
# We are using neural net to distinguish the digits 2, 7 in the MNIST dataset.
# Author: Hongda Li
# Don't copy my code it has my style in it.

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision.datasets as datasets
from torchvision import transforms
from tqdm import tqdm
from math import sqrt
from collections import Iterable
import matplotlib.pyplot as plt

tensor = torch.tensor
zeros = torch.zeros
rand = torch.rand

MNIST_TRAIN = datasets.MNIST(root="./data",
                             train=True,
                             download=True,
                             transform=transforms.ToTensor())
# MNIST_TRAIN = torch.utils.data.Subset(MNIST_TRAIN, range(1000))
MNIST_TEST = datasets.MNIST(root="./data",
                             train=False,
                             download=True,
                             transform=transforms.ToTensor())
```

```

class MyNN:

    def __init__(this, modelParameterMaker:callable):
        """
            Architecture:
            Direct linear stacking of weight and biases. with ReLU.
        :param modelParameterMaker:
            A function that makes the W, weight matrices and the weight vector
            b in the neural network.
        """
        this.Weights, this.Biases = modelParameterMaker()
        assert isinstance(this.Weights, Iterable)
        assert isinstance(this.Biases, Iterable)

    def feedforward(this, X, y):
        """
            X is a row data matrix.
        :param X:
        :param y:
        :return:
        """
        assert X.ndim == 2
        assert X.shape[0] == y.shape[0] or X.shape[1] == y.shape[0]
        a = X # output of the first layer
        for W, b in list(zip(this.Weights, this.Biases))[:-1]:
            a = F.relu(a@W + b)
        # direct out put from last layer into the loss function.
        a = a @ this.Weights[-1] + this.Biases[-1]
        return F.cross_entropy(a, y)

    def predict(this, X):
        a = X # output of the first layer
        for W, b in list(zip(this.Weights, this.Biases))[:-1]:
            a = F.relu(a @ W + b)
        a = a @ this.Weights[-1] + this.Biases[-1]
        Probability = F.softmax(a, dim=1)
        return torch.max(Probability, dim=1)[1]

    @property
    def parameters(this):
        # Weights and biases concat together.
        return list(this.Weights) + list(this.Biases)

    @staticmethod
    def A4a():
        """
            Get the parameters ready for A4a.
        :param gpu:
            Whether to use the GPU on the tensor.
        :return:
            2 iterables of the weights and biases.
        """
        W0 = zeros(28 ** 2, 64, requires_grad=True)
        alpha = 1/sqrt(W0.shape[1])
        W0.data += 2*alpha* rand(W0.shape) - alpha
        W1 = zeros(64, 10, requires_grad=True)
        alpha = 1 / sqrt(W1.shape[1])
        W1.data += 2*alpha*rand(W1.shape) - alpha
        b0 = zeros(1,64, requires_grad=True)
        b1 = zeros(1,10, requires_grad=True)
        return [W0, W1], [b0, b1]

    @staticmethod
    def A4b():
        W0 = zeros(28 ** 2, 32, requires_grad=True)

```

```

alpha = 1 / sqrt(W0.shape[1])
W0.data += 2 * alpha * rand(W0.shape) - alpha

W1 = zeros(32, 32, requires_grad=True)
alpha = 1 / sqrt(W1.shape[1])
W1.data += 2 * alpha * rand(W1.shape) - alpha

W2 = zeros(32, 10, requires_grad=True)
alpha = 1 / sqrt(W2.shape[1])
W2.data += 2 * alpha * rand(W2.shape) - alpha

b0 = zeros(1, 32, requires_grad=True)
b1 = zeros(1, 32, requires_grad=True)
b2 = zeros(1, 10, requires_grad=True)

return [W0, W1, W2], [b0, b1, b2]

def main():

    data_loader = torch.utils.data.DataLoader(MNIST_TRAIN,
                                              batch_size=250,
                                              shuffle=True)

    Epochs = 30

    def Accuracy(yhat, y):
        return sum(yhat == y)/yhat.numel()

    def RunMNIST(Model, Optimizer, part):
        EpochLosses = []
        for E in range(Epochs):
            EpochLoss = 0.0
            for X, y, in data_loader:
                X = X.view(-1, 784)
                Optimizer.zero_grad()
                Loss = Model.feedforward(X, y)
                EpochLoss += Loss.item()/X.shape[0]
                Loss.backward()
                Optimizer.step()
            EpochLosses.append(EpochLoss)
            X = torch.stack([D[0].reshape(-1) for D in MNIST_TRAIN], axis=0)
            y = torch.tensor([D[1] for D in MNIST_TRAIN])
            Rate = Accuracy(Model.predict(X), y)
            print(f"Epoch: {E}, Loss: {EpochLoss}")
            print(f"accuracy: {Rate}")
            if Rate > 0.99:
                print("Process terminated because 99% accuracy reached.")
                break
        X = torch.stack([D[0].reshape(-1) for D in MNIST_TEST], axis=0)
        y = torch.tensor([D[1] for D in MNIST_TEST])
        TestAccuracy = Accuracy(Model.predict(X), y)
        print(f"Test set accuracy is: {TestAccuracy}")
        plt.plot(EpochLosses)
        plt.title(f"A4{part}-Cross Entropy Loss at each Epochs")
        plt.xlabel("Epochs")
        plt.ylabel("Entropy Loss")
        plt.savefig(f"A4({part})-NN-mnist.png")
        plt.show()
        return TestAccuracy

    Model = MyNN(MyNN.A4a)
    Optimizer = optim.Adam(Model.parameters, lr=0.01)
    Rate = RunMNIST(Model, Optimizer, part="a")
    with open("A4a.txt", "w+") as f:
        f.write(str(Rate))

```



```

Model = MyNN(MyNN.A4b)
Optimizer = optim.Adam(Model.parameters, lr=0.01)
Rate = RunMNIST(Model, Optimizer, part="b")
with open("A4b.txt", "w+") as f:
    f.write(str(Rate))

if __name__ == "__main__":
    import os
    print(f"cwd:{os.getcwd()} ")
    main()

```

A.b.c

There are 26506 parameters for the deep model and there are 50890 for the shallow model. Usually, it will make sense to have deeper networks, and this is the central hypothesis for many of the existing, famous networks, like the case of the deep residual net. However, it must be clear that, it comes with the risk of overfitting, in addition, the performance will be limited by the width of the hidden layer. This is true because latent variables represent abstract features that network can work with.

I would say deeper, larger network is better, and the more parameters the better, if it overfits, we just regularize it. So the second network is better, it manages to achieve accuracy close to the first network all with much less parameters.

A.5: Using Pretrained Networks and Transfer Learning

⊃

A.6: Image Classification on CIFAR-10