Homework #3

CSE 546: Machine Learning
Cassia Cai
November 24, 2022

Collaborators: Apoorva Kalaskar and Madeleine Grunde-McLaughlin

Conceptual Questions

A1.

a. *[2 points]* Say you trained an SVM classifier with an RBF kernel ($K(u, v) = \exp\left(-\frac{\|u-v\|_2^2}{2\sigma^2}\right)$). It seems to underfit the training set: should you increase or decrease σ?

DECREASE σ. We know that $\gamma = \frac{1}{2\sigma^2}$ and decreasing γ increases the margin whereas increasing γ tightens the margin. We want to decrease the margin (so increase γ) and therefore decrease σ.

b. *[2 points]* True or False: Training deep neural networks requires minimizing a non-convex loss function, and therefore gradient descent might not reach the globally-optimal solution.

TRUE. Training deep neural nets means minimizing a non-convex loss function, so gradient descent may not necessarily reach the globally optimal solution.

c. *[2 points]* True or False: It is a good practice to initialize all weights to zero when training a deep neural network.

FALSE. It is not a good practice to initialize all weights to zero when training a deep neural network because if we did, then neurons would learn the same features during training.

d. *[2 points]* True or False: We use non-linear activation functions in a neural network's hidden layers so that the network learns non-linear decision boundaries.

TRUE. We do so because we want the network to learn non-linear decision boundaries. Non-linear activation functions thus introduce non-linearity into the hidden layers. Without these non-linear activation functions, the network will be a linear combination of inputs.

e. *[2 points]* True or False: Given a neural network, the time complexity of the backward pass step in the backpropagation algorithm can be prohibitively larger compared to the relatively low time complexity of the forward pass step.

FALSE. The time complexity of the backward pass step is the same as the time complexity of the forward pass step in a neural network.

f. *[2 points]* True or False: Neural Networks are the most extensible model and therefore the best choice for any circumstance.

FALSE. Neural networks are not the best choice for all circumstances because they require a lot of data. If we had smaller amounts of data, we may find that other learning algorithms before better. For example, simple linear regression will best learn a linear trend.

Logistic Regression

A2. Here we consider the MNIST dataset, but for binary classification. Specifically, the task is to determine whether a digit is a 2 or 7. Here, let $Y = 1$ for all the "7" digits in the dataset, and use $Y = -1$ for "2". We will use regularized logistic regression. Given a binary classification dataset $\{(x_i, y_i)\}_{i=1}^n$ for $x_i \in \mathbb{R}^d$ and $y_i \in \{-1, 1\}$ we showed in class that the regularized negative log likelihood objective function can be written as

$$J(w, b) = \frac{1}{n} \sum_{i=1}^{n} \log(1 + \exp(-y_i(b + x_i^T w))) + \lambda ||w||_2^2$$

Note that the offset term b is not regularized. For all experiments, use $\lambda = 10^{-1}$. Let $\mu_i(w, b) = \frac{1}{1+\exp(-y_i(b+x_i^T w))}$.

a. *[8 points]* Derive the gradients $\nabla_w J(w, b)$, $\nabla_b J(w, b)$ and give your answers in terms of $\mu_i(w, b)$ (your answers should not contain exponentials).

$$\nabla_w J(w, b) = \nabla_w \left(\frac{1}{n} \sum_{i=1}^{n} \log(1 + \exp(-y_i(b + x_i^T w))) + \lambda ||w||_2^2 \right)$$

$$= \frac{1}{n} \sum_{i=1}^{n} \nabla_w \left(\log(1 + \exp(-y_i(b + x_i^T w))) \right) + \nabla_w \left(\lambda ||w||_2^2 \right)$$

$$= -\frac{1}{n} \sum_{i=1}^{n} \nabla_w \left(-\log(\mu_i(w, b)) \right) + 2\lambda w$$

$$= -\frac{1}{n} \sum_{i=1}^{n} \frac{\nabla_w \mu_i(w, b)}{\mu_i(w, b)} + 2\lambda w$$

$$= -\frac{1}{n} \sum_{i=1}^{n} \nabla_w \left(\frac{1}{1 + \exp(-y_i(b + x_i^T w))} \right) \frac{1}{\mu_i(w, b)} + 2\lambda w$$

$$= -\frac{1}{n} \sum_{i=1}^{n} \left(\frac{-y_i x_i \frac{1-\mu_i(w,b)}{\mu_i(w,b)}}{(1 + \exp(-y_i(b + x_i^T w)))^2} \right) \frac{1}{\mu_i(w, b)} + 2\lambda w$$

$$= -\frac{1}{n} \sum_{i=1}^{n} (-y_i x_i (1 - \mu_i(w, b))) + 2\lambda w$$

$$\nabla_w J(w, b) = \nabla_b \left(\frac{1}{n} \sum_{i=1}^{n} \log(1 + \exp(-y_i(b + x_i^T w))) + \lambda ||w||_2^2 \right)$$

$$= \frac{1}{n} \sum_{i=1}^{n} \nabla_b \left(\log(1 + \exp(-y_i(b + x_i^T w))) \right)$$

$$= -\frac{1}{n} \sum_{i=1}^{n} \nabla_b \left(-\log(\mu_i(w, b)) \right)$$

$$= -\frac{1}{n} \sum_{i=1}^{n} \frac{\nabla_b \mu_i(w, b)}{\mu_i(w, b)}$$

$$= -\frac{1}{n} \sum_{i=1}^{n} \nabla_b \left(\frac{1}{1 + \exp(-y_i(b + x_i^T w))} \right) \frac{1}{\mu_i(w, b)}$$

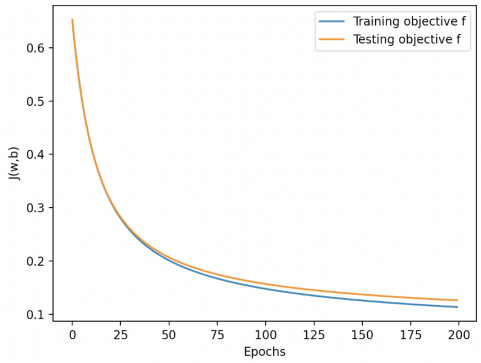
$$= -\frac{1}{n} \sum_{i=1}^{n} \left(\frac{-y_i \frac{1-\mu_i(w,b)}{\mu_i(w,b)}}{(1 + \exp(-y_i(b + x_i^T w)))^2} \right) \frac{1}{\mu_i(w, b)}$$

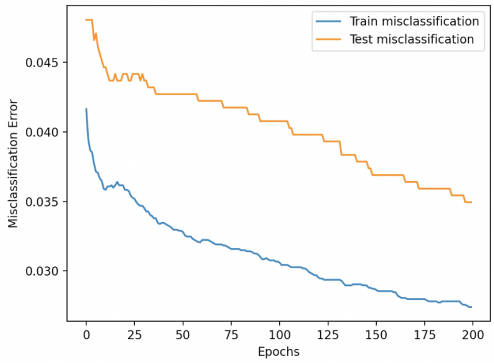
$$= -\frac{1}{n} \sum_{i=1}^{n} (-y_i (1 - \mu_i(w, b)))$$

b. *[8 points]* Implement gradient descent with an initial iterate of all zeros. Try several values of step sizes to find one that appears to make convergence on the training set as fast as possible. Run until you feel you are near to convergence.

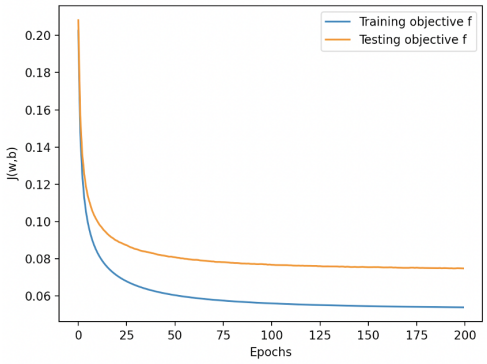
(i) For both the training set and the test, plot $J(w, b)$ as a function of the iteration number (and show both curves on the same plot).

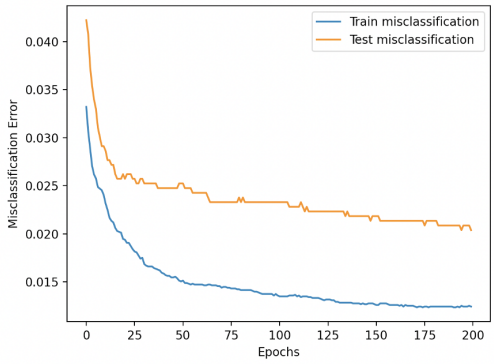
(ii) For both the training set and the test, classify the points according to the rule $\text{sign}(b + x_i^T w)$ and plot the misclassification error as a function of the iteration number (and show both curves on the same plot).



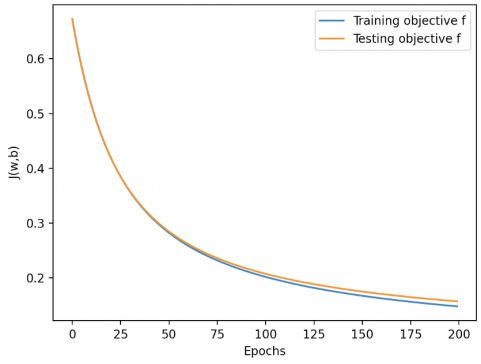


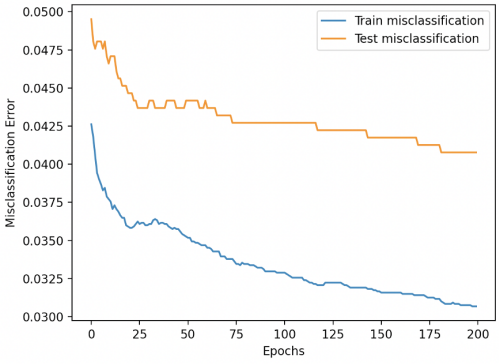
c. *[7 points]* Repeat (b) using stochastic gradient descent with a batch size of 1. Note, the expected gradient with respect to the random selection should be equal to the gradient found in part (a). Show both plots described in (b) when using batch size 1. Take careful note of how to scale the regularizer.





d. *[7 points]* Repeat (b) using stochastic gradient descent with batch size of 100. That is, instead of approximating the gradient with a single example, use 100. Note, the expected gradient with respect to the random selection should be equal to the gradient found in part (a).





```
from typing import Dict, List, Tuple
import matplotlib.pyplot as plt
import numpy as np
from utils import load_dataset, problem

RNG = np.random.RandomState(seed=446)
Dataset = Tuple[Tuple[np.ndarray, np.ndarray], Tuple[np.ndarray, np.ndarray]]

def load_2_7_mnist() -> Dataset:
    (x_train, y_train), (x_test, y_test) = load_dataset("mnist")
    train_idxs = np.logical_or(y_train == 2, y_train == 7)
    test_idxs = np.logical_or(y_test == 2, y_test == 7)
    y_train_2_7 = y_train[train_idxs]
    y_train_2_7 = np.where(y_train_2_7 == 7, 1, -1)
    y_test_2_7 = y_test[test_idxs]
    y_test_2_7 = np.where(y_test_2_7 == 7, 1, -1)
    return (x_train[train_idxs], y_train_2_7), (x_test[test_idxs], y_test_2_7)

class BinaryLogReg:
    def __init__(self, _lambda: float = 1e-3):
        self._lambda: float = _lambda
        self.weight: np.ndarray = None
        self.bias: float = 0.0

    def mu(self, X: np.ndarray, y: np.ndarray) -> np.ndarray:
        mu_array = 1/(1+np.exp(-y*(self.bias + X.dot(self.weight))))
        return mu_array

    def loss(self, X: np.ndarray, y: np.ndarray) -> float:
        n, d = X.shape
        regularizer = self._lambda * np.sum(np.square(self.weight))
        return (1/n) * np.sum(np.log(1 + np.exp(np.negative(y)*(self.bias + X @ self.weight)

    def gradient_J_weight(self, X: np.ndarray, y: np.ndarray) -> np.ndarray:
        new_mu = self.mu(X,y)
        n, d = X.shape
        return (1/n) * (y * (new_mu - 1) @ X) + 2 * (self._lambda * self.weight)

    def gradient_J_bias(self, X: np.ndarray, y: np.ndarray) -> float:
        return np.mean((self.mu(X, y) - 1)*y, axis = 0)

    def predict(self, X: np.ndarray) -> np.ndarray:
        return np.sign(self.bias + X.dot(self.weight))

    def misclassification_error(self, X: np.ndarray, y: np.ndarray) -> float:
        return 1 - np.mean(y == self.predict(X))

    def step(self, X: np.ndarray, y: np.ndarray, learning_rate: float = 1e-4):
        self.weight = self.weight - learning_rate * self.gradient_J_weight(X, y)
        self.bias = self.bias - learning_rate * self.gradient_J_bias(X, y)

    def train(self, X_train: np.ndarray, y_train: np.ndarray, X_test: np.ndarray, y_test: np
    ) -> Dict[str, List[float]]:
        num_batches = int(np.ceil(len(X_train) // batch_size))
        result: Dict[str, List[float]] = {
```

```
            "train_losses": [], "train_errors": [],
            "test_losses": [], "test_errors": [],}
        ind_train_arr = np.arange(X_train.shape[0])
        if self.weight is None:
            self.weight = np.zeros(X_train.shape[1])
        if self.bias is None:
            self.bias = 0.
        counter = 0
        for i in range(epochs):
            RNG.shuffle(ind_train_arr)
            X_train_reshuffled = X_train[ind_train_arr]
            y_train_reshuffled = y_train[ind_train_arr]
            x_batch = np.array_split(X_train_reshuffled, num_batches)
            y_batch = np.array_split(y_train_reshuffled, num_batches)
            train_loss, train_err, test_loss, test_err = 0.,0.,0.,0.
            for j in range(num_batches):
                self.step(x_batch[j], y_batch[j])
            train_loss = self.loss(X_train, y_train)
            train_err = self.misclassification_error(X_train, y_train)
            test_loss = self.loss(X_test, y_test)
            test_err = self.misclassification_error(X_test, y_test)
            result["train_losses"].append(train_loss)
            result["train_errors"].append(train_err)
            result["test_losses"].append(test_loss)
            result["test_errors"].append(test_err)
            counter += 1
        return result

if __name__ == "__main__":
    model = BinaryLogReg()
    (x_train, y_train), (x_test, y_test) = load_2_7_mnist()
    history = model.train(x_train, y_train, x_test, y_test)

    # Plot losses
    plt.plot(history["train_losses"], label="Training objective f")
    plt.plot(history["test_losses"], label="Testing objective f")
    plt.xlabel("Epochs")
    plt.ylabel("J(w,b)")
    plt.legend()
    plt.show()

    # Plot error
    plt.plot(history["train_errors"], label="Train misclassification")
    plt.plot(history["test_errors"], label="Test misclassification")
    plt.xlabel("Epochs")
    plt.ylabel("Misclassification Error")
    plt.legend()
    plt.show()
```

Support Vector Machines

A3. Recall that solving the SVM problem amounts to solving the following constrained optimization problem:

Given data points $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$ find

$$\min_{w,b} ||w||_2 \text{ subject to } y_i(x_i^T w - b) \geq 1 \text{ for } i \in \{1, \ldots, n\}$$

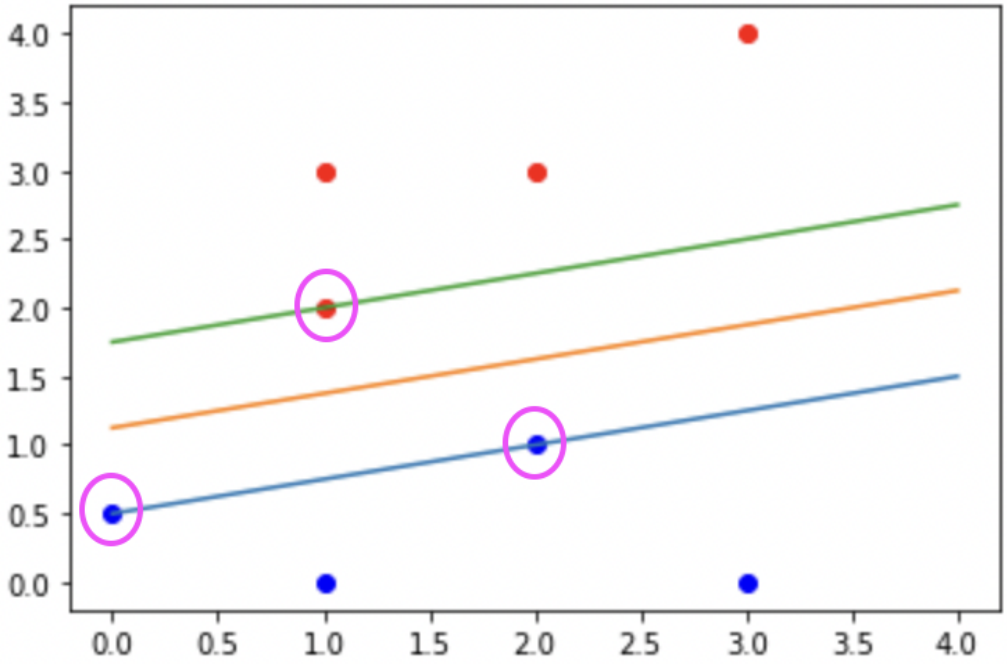
where $x_i \in \mathbb{R}^d$, $y_i \in \{-1, 1\}$, and $w \in \mathbb{R}^d$.

Consider the following labeled data points:

$$\begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 2 & 3 \\ 3 & 4 \end{bmatrix} \text{ with label } y = -1 \text{ and } \begin{bmatrix} 0 & 0.5 \\ 1 & 0 \\ 2 & 1 \\ 3 & 0 \end{bmatrix} \text{ with label } y = 1$$

a. *[2 points]* Graph the data points above. Highlight the support vectors and write their coordinates. Draw the two parallel hyperplanes separating the two classes of data such that the distance between them is as large as possible. Draw the maximum-margin hyperplane. Write the equations describing these three hyperplanes using only x, w, b(that is without using any specific values). Draw w(it doesn't have to have the exact magnitude, but it should have the correct orientation).

Our support vectors are (0,0.5), (2,1), and (1,2). The hyper-planes are green and blue. The equations are $0w_1 + 0.5w_2 + b = 1$, $2w_1 + 1w_2 + b = 1$, and $1w_1 + 2w_2 + b = -1$.



b. *[2 points]* For the data points above, find w and b.

$$0w_1 + 0.5w_2 + b = -1 \rightarrow w_2 = -2 - 2b$$
$$w_1 + 2w_2 + b = 1 \rightarrow w_1 = 1 - 2w_2 - b$$
$$2w_1 + w_2 + b = -1$$

Via substitution:

$$2(1 - 2w_2 - b) + w_2 + b = -1 \rightarrow 3w_2 + b = 3$$
$$3(-2 - 2b) + b = 3 \rightarrow -6 - 5b = 3 \rightarrow b = 9/5$$
$$0.5w_2 + 9/5 = -1 \rightarrow 10w_2 + 9 = -5 \rightarrow w_2 = -7/5$$

$$w_1 + 2w_2 + b = 1 \rightarrow w_1 = 1 - 2(-7/5) - 9/5 \rightarrow w_1 = 0$$

We have found that $b = 9/5$, $w_1 = 0$ and $w_2 = -7/5$. From our graph, we see that we have plotted several lines: $y = 0.25x + 7/4$, $y = 0.25x + 0.625$, and $y = 0.25x + 0.5$.

Hint: Use the support vectors and the values $\{-1, 1\}$ to create a linear system of equations where the unknowns are w_1, w_2 and b.

Kernels

A4. *[5 points]* Suppose that our inputs x are one-dimensional and that our feature map is infinite-dimensional: $\phi(x)$ is a vector whose ith component is:

$$\frac{1}{\sqrt{i!}} \mathrm{e}^{-x^2/2} x^i \ ,$$

for all nonnegative integers i. (Thus, ϕ is an infinite-dimensional vector.)

a. Show that $K(x, x') = \mathrm{e}^{-\frac{(x-x')^2}{2}}$ is a kernel function for this feature map, i.e.,

$$\phi(x) \cdot \phi(x') = \mathrm{e}^{-\frac{(x-x')^2}{2}} \ .$$

Hint: Use the Taylor expansion of $z \mapsto \mathrm{e}^z$. (This is the one dimensional version of the Gaussian (RBF) kernel).

$$\phi(x)\phi(x') = \left(\frac{1}{\sqrt{i!}} \mathrm{e}^{-x^2/2} x^i\right)\left(\frac{1}{\sqrt{i!}} \mathrm{e}^{-x'^2/2} x'^i\right) = \sum_{i=1}^{\infty} \frac{1}{i!} x^i x'^i \mathrm{e}^{\frac{-x^2-x'^2}{2}}$$

$$= \mathrm{e}^{\frac{-x^2-x'^2}{2}} \sum_{i=1}^{\infty} \frac{x^i x'^i}{i!} = \mathrm{e}^{\frac{-x^2-x'^2}{2}} \mathrm{e}^{-xx'} = \mathrm{e}^{-\frac{(x-x')^2}{2}}$$

A5. This problem will get you familiar with kernel ridge regression using the polynomial and RBF kernels. First, let's generate some data. Let $n = 30$ and $f_*(x) = 4\sin(\pi x)\cos(6\pi x^2)$. For $i = 1, \ldots, n$ let each x_i be drawn uniformly at random from $[0, 1]$, and let $y_i = f_*(x_i) + \epsilon_i$ where $\epsilon_i \sim \mathcal{N}(0, 1)$. For any function f, the true error and the train error are respectively defined as:

$$\mathcal{E}_{\text{true}}(f) = \mathbb{E}_{X,Y}\left[(f(X) - Y)^2\right], \qquad \widehat{\mathcal{E}}_{\text{train}}(f) = \frac{1}{n}\sum_{i=1}^{n} (f(x_i) - y_i)^2 \ .$$

Now, our goal is, using kernel ridge regression, to construct a predictor:

$$\widehat{\alpha} = \arg\min_{\alpha} \|K\alpha - y\|_2^2 + \lambda \alpha^\top K \alpha \ , \qquad \widehat{f}(x) = \sum_{i=1}^{n} \widehat{\alpha}_i k(x_i, x)$$

where $K \in \mathbb{R}^{n \times n}$ is the kernel matrix such that $K_{i,j} = k(x_i, x_j)$, and $\lambda \geq 0$ is the regularization constant.

a. *[10 points]* Using leave-one-out cross validation, find a good λ and hyperparameter settings for the following kernels:

- $k_{\text{poly}}(x, z) = (1 + x^\top z)^d$ where $d \in \mathbb{N}$ is a hyperparameter,
- $k_{\text{rbf}}(x, z) = \exp(-\gamma \|x - z\|_2^2)$ where $\gamma > 0$ is a hyperparameter[1].

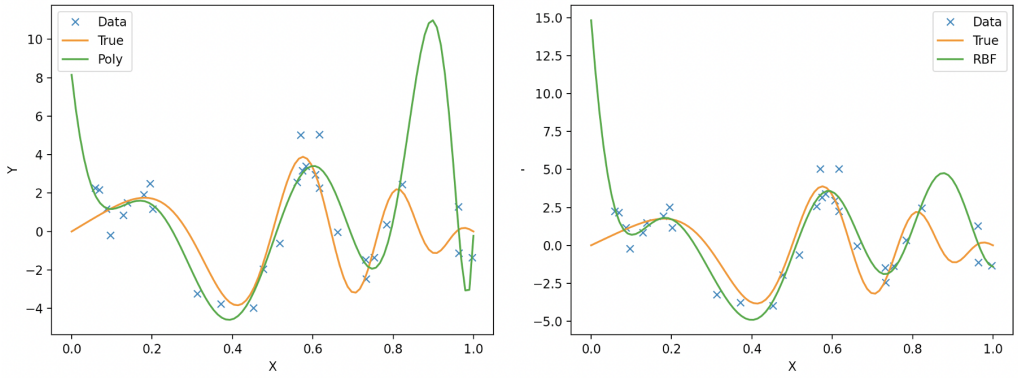
[1]Given a dataset $x_1, \ldots, x_n \in \mathbb{R}^d$, a heuristic for choosing a range of γ in the right ballpark is the inverse of the median of all $\binom{n}{2}$ squared distances $\|x_i - x_j\|_2^2$.

We strongly recommend implementing either grid search or random search. **Do not use sklearn**, but actually implement of these algorithms. Reasonable values to look through in this problem are: $\lambda \in 10^{[-5,-1]}$, $d \in [5, 25]$, γ sampled from a narrow gaussian distribution centered at value described in the footnote.
Report the values of d, γ, and the λ values for both kernels.

RBF kernel: $\lambda = 0.0013$ and $\gamma = 13.0972$.

Poly kernel: $\lambda = 2.5595$ and $d = 19$.

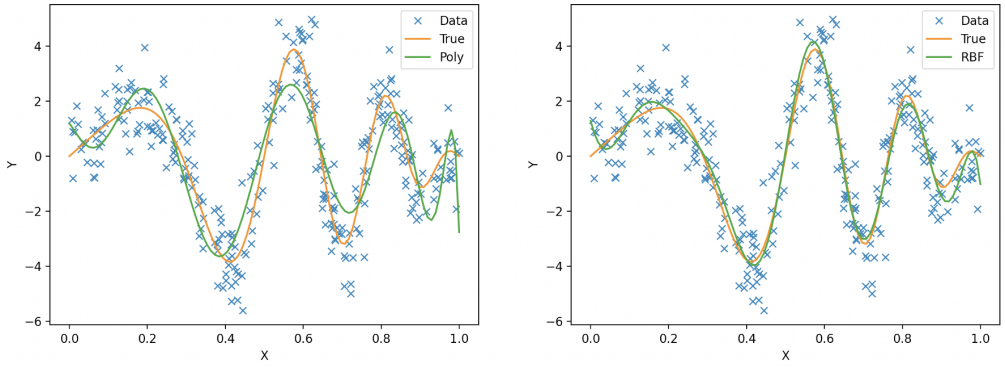
b. *[10 points]* Let $\widehat{f}_{\text{poly}}(x)$ and $\widehat{f}_{\text{rbf}}(x)$ be the functions learned using the hyperparameters you found in part a. For a single plot per function $\widehat{f} \in \left\{\widehat{f}_{\text{poly}}(x), \widehat{f}_{\text{rbf}}(x)\right\}$, plot the original data $\{(x_i, y_i)\}_{i=1}^n$, the true $f(x)$, and $\widehat{f}(x)$ (i.e., define a fine grid on $[0, 1]$ to plot the functions).



c. *[5 points]* Repeat parts a and b with $n = 300$, but use 10-fold CV instead of leave-one-out for part a.

RBF kernel: $\lambda = 1e-05$ and $\gamma = 14.9527$.

Poly kernel: $\lambda = 1e-05$ and $d = 19$.




```
from typing import Tuple, Union
import matplotlib.pyplot as plt
import numpy as np
from utils import load_dataset, problem

def f_true(x: np.ndarray) -> np.ndarray:
    return 4 * np.sin(np.pi * x) * np.cos(6 * np.pi * x ** 2)

def poly_kernel(x_i: np.ndarray, x_j: np.ndarray, d: int) -> np.ndarray:
    K = (np.outer(x_i, x_j) + 1)**d
    return K
```

```
def rbf_kernel(x_i: np.ndarray, x_j: np.ndarray, gamma: float) -> np.ndarray:
    K = np.exp(-gamma * (np.subtract.outer(x_i, x_j)) ** 2)
    return K

def train(
    x: np.ndarray,
    y: np.ndarray,
    kernel_function: Union[poly_kernel, rbf_kernel],  # type: ignore
    kernel_param: Union[int, float],
    _lambda: float,
) -> np.ndarray:
    n = len(x)
    K = kernel_function(x,x, kernel_param)
    return np.linalg.inv((K + _lambda * np.eye(n,n))).dot(y)

def cross_validation(
    x: np.ndarray,
    y: np.ndarray,
    kernel_function: Union[poly_kernel, rbf_kernel],  # type: ignore
    kernel_param: Union[int, float],
    _lambda: float,
    num_folds: int,
) -> float:
    fold_size = len(x) // num_folds
    error = []
    start_index = 0
    end_index = fold_size
    for index in range(num_folds):
        x_current = x[ start_index : end_index ]
        y_current = y[ start_index : end_index ]
        index_array = np.array(range(start_index, end_index, 1))
        xtrain = np.delete(x, index_array, 0)
        ytrain = np.delete(y, index_array, 0)
        start_index += fold_size; end_index += fold_size
        alpha = train(xtrain,ytrain, kernel_function, kernel_param, _lambda)
        yp = alpha.dot((kernel_function(xtrain,x_current, kernel_param)))
        now_error = np.mean((np.subtract(y_current, yp))**2)
        error.append(now_error)
    return np.mean(error)

def rbf_param_search(
    x: np.ndarray, y: np.ndarray, num_folds: int
) -> Tuple[float, float]:
    lambdas = np.logspace(-5.0, -1.0)
    gamma_std = 0.01
    dist_sq = np.subtract.outer(x, x)**2
    gamma = 1/np.median(dist_sq[np.triu_indices(dist_sq.shape[0])])
    gammas = np.random.normal(gamma, gamma_std, size = 30)

    errors = np.zeros([len(lambdas),len(gammas)])
    counter = 0
    for j in range(len(lambdas)):
        for ii in range(len(gammas)):
            errors[j,ii] = cross_validation(x, y, rbf_kernel, gammas[ii], lambdas[j], n
            counter += 1
```

```
    index = np.argwhere(errors == np.min(errors))
    index = index[0]
    lambda_value = lambdas[index[0]]; gamma_value = gammas[index[1]]
    return [lambda_value, gamma_value]

def poly_param_search(
    x: np.ndarray, y: np.ndarray, num_folds: int
) -> Tuple[float, int]:
    lambdas = np.logspace(-5.0, -1.0)
    ds = np.arange(5,25,1)
    errors = np.zeros([len(lambdas), len(ds)])
    for i in range(len(lambdas)):
        for j in range(len(ds)):
            errors[i,j] = cross_validation(x,y,poly_kernel, ds[j], lambdas[i], num_folds

    index = np.argwhere(errors == np.min(errors))
    index = index[0]
    lambda_value= lambdas[index[0]]
    d_value = ds[index[1]]
    return [lambda_value, d_value]

def bootstrap(
    x: np.ndarray,
    y: np.ndarray,
    kernel_function: Union[poly_kernel, rbf_kernel],  # type: ignore
    kernel_param: Union[int, float],
    _lambda: float,
    bootstrap_iters: int = 300,
) -> np.ndarray:
    x_fine_grid = np.linspace(0, 1, 100)

def main():
    (x_30, y_30), (x_300, y_300), (x_1000, y_1000) = load_dataset("kernel_bootstrap")
    rbfparams30= rbf_param_search(x_30, y_30,len(x_30))
    print(rbfparams30)
    polyparams30 = poly_param_search(x_30, y_30,10)
    print(polyparams30)
    xfine = np.linspace(0, 1, num=100)
    alpha_rbf = train(x_30,y_30, rbf_kernel, rbfparams30[1], rbfparams30[0])
    ypred_rbf = alpha_rbf.dot(rbf_kernel(x_30,xfine, rbfparams30[1]))
    alpha_poly = train(x_30,y_30, poly_kernel, polyparams30[1], polyparams30[0])
    ypred_poly = alpha_poly.dot(poly_kernel(x_30,xfine, polyparams30[1]))
    ytrue = f_true(xfine)
    plt.scatter(x_30, y_30)
    plt.plot(xfine, ytrue)
    plt.plot(xfine, ypred_rbf)
    plt.plot(xfine ,ypred_poly)
    plt.xlabel("X"); plt.ylabel("Y")
    plt.show()

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

Neural Networks for MNIST

For questions A.6 you will use a lot of PyTorch.

A6. In Homework 1, we used ridge regression for training a classifier for the MNIST data set. In this problem, we will use PyTorch to build a simple neural network classifier for MNIST to further improve our accuracy.

We will implement two different architectures: a shallow but wide network, and a narrow but deeper network. For both architectures, we use d to refer to the number of input features (in MNIST, $d = 28^2 = 784$), h_i to refer to the dimension of the i-th hidden layer and k for the number of target classes (in MNIST, $k = 10$). For the non-linear activation, use ReLU. Recall from lecture that

$$\text{ReLU}(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases}.$$

Weight Initialization

Consider a weight matrix $W \in \mathbb{R}^{n \times m}$ and $b \in \mathbb{R}^n$. Note that here m refers to the input dimension and n to the output dimension of the transformation $x \mapsto Wx + b$. Define $\alpha = \frac{1}{\sqrt{m}}$. Initialize all your weight matrices and biases according to $\text{Unif}(-\alpha, \alpha)$.

Training

For this assignment, use the Adam optimizer from `torch.optim`. Adam is a more advanced form of gradient descent that combines momentum and learning rate scaling. It often converges faster than regular gradient descent in practice. You can use either Gradient Descent or any form of Stochastic Gradient Descent. Note that you are still using Adam, but might pass either the full data, a single datapoint or a batch of data to it. Use cross entropy for the loss function and ReLU for the non-linearity.

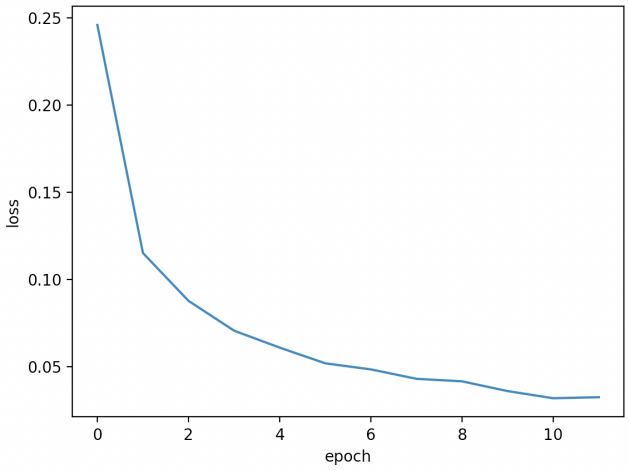
Implementing the Neural Networks

a. *[10 points]* Let $W_0 \in \mathbb{R}^{h \times d}$, $b_0 \in \mathbb{R}^h$, $W_1 \in \mathbb{R}^{k \times h}$, $b_1 \in \mathbb{R}^k$ and $\sigma(z) \colon \mathbb{R} \to \mathbb{R}$ some non-linear activation function applied element-wise. Given some $x \in \mathbb{R}^d$, the forward pass of the wide, shallow network can be formulated as:

$$\mathcal{F}_1(x) \coloneqq W_1 \sigma(W_0 x + b_0) + b_1$$

Use $h = 64$ for the number of hidden units and choose an appropriate learning rate. Train the network until it reaches 99% accuracy on the training data and provide a training plot (loss vs. epoch). Finally evaluate the model on the test data and report both the accuracy and the loss.

The accuracy on the test data is 97.12. The loss on the test data is 0.1368.

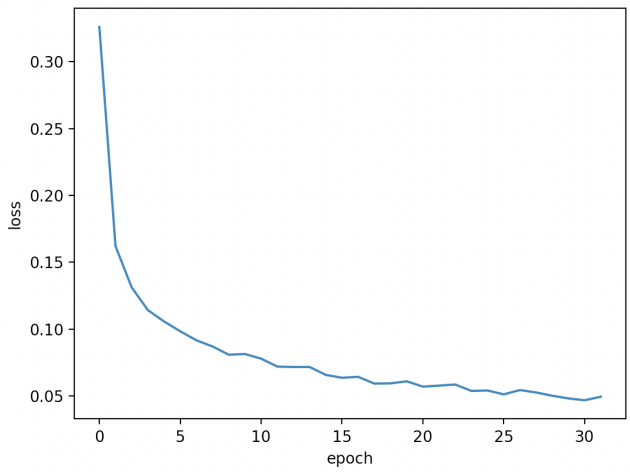


b. *[10 points]* Let $W_0 \in \mathbb{R}^{h_0 \times d}$, $b_0 \in \mathbb{R}^{h_0}$, $W_1 \in \mathbb{R}^{h_1 \times h_0}$, $b_1 \in \mathbb{R}^{h_1}$, $W_2 \in \mathbb{R}^{k \times h_1}$, $b_2 \in \mathbb{R}^k$ and $\sigma(z) : \mathbb{R} \to \mathbb{R}$ some non-linear activation function. Given some $x \in \mathbb{R}^d$, the forward pass of the network can be formulated as:

$$\mathcal{F}_2(x) \coloneqq W_2\sigma(W_1\sigma(W_0x + b_0) + b_1) + b_2$$

Use $h_0 = h_1 = 32$ and perform the same steps as in part a.

The accuracy on the test data is 96.42. The loss on the test data is 0.1794.



c. *[5 points]* Compute the total number of parameters of each network and report them. Then compare the number of parameters as well as the test accuracies the networks achieved. Is one of the approaches (wide, shallow vs. narrow, deeper) better than the other? Give an intuition for why or why not.

Our two-layer model has 50,890 parameters. Our three-layer model has 26,506 parameters. Both models output similar results when we compare our test evaluation. The three-layer model is faster (in terms of training time) due to the fact that we have fewer parameters.

```
import math
from typing import List
import matplotlib.pyplot as plt
import torch
from torch.distributions import Uniform
from torch.nn import Module
from torch.nn.functional import cross_entropy, relu
from torch.nn.parameter import Parameter
from torch.optim import Adam
from torch.utils.data import DataLoader, TensorDataset
import numpy as np
from utils import load_dataset, problem

class F1(Module):
    def __init__(self, h: int, d: int, k: int):
        super().__init__()
        self.alpha0 = 1 / np.sqrt(d)
        self.alpha1 = 1 / np.sqrt(h)
        self.w0 = torch.nn.Parameter(Uniform(-self.alpha0, self.alpha0).sample(sample_shape=
        self.w1 = torch.nn.Parameter(Uniform(-self.alpha1, self.alpha1).sample(sample_shape=
        self.b0 = torch.nn.Parameter(Uniform(-self.alpha0, self.alpha0).sample(sample_shape=
        self.b1 = torch.nn.Parameter(Uniform(-self.alpha1, self.alpha1).sample(sample_shape=
        self.params = [self.w0, self.w1, self.b0, self.b1]
```

```
        for param in self.params:
          param.requires_grad = True

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        n,_ = x.shape
        b0 = self.b0.repeat(n,1)
        b1 = self.b1.repeat(n,1)
        x = torch.matmul(x, self.w0.T) + b0
        x = torch.nn.functional.relu(x)
        x = torch.matmul(x, self.w1.T) + b1
        return x

class F2(Module):
    def __init__(self, h0: int, h1: int, d: int, k: int):
        super().__init__()
        self.alpha0 = 1 / np.sqrt(d)
        self.alpha1 = 1 / np.sqrt(h0)
        self.alpha2 = 1 / np.sqrt(h1)
        self.w0 = torch.nn.Parameter(Uniform(-self.alpha0, self.alpha0).sample(sample_shape=
        self.w1 = torch.nn.Parameter(Uniform(-self.alpha1, self.alpha1).sample(sample_shape=
        self.w2 = torch.nn.Parameter(Uniform(-self.alpha2, self.alpha2).sample(sample_shape=
        self.b0 = torch.nn.Parameter(Uniform(-self.alpha0, self.alpha0).sample(sample_shape=
        self.b1 = torch.nn.Parameter(Uniform(-self.alpha1, self.alpha1).sample(sample_shape=
        self.b2 = torch.nn.Parameter(Uniform(-self.alpha2, self.alpha2).sample(sample_shape=
        self.params = [self.w0, self.w1, self.w2, self.b0, self.b1, self.b2]
        for param in self.params:
          param.requires_grad = True

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        b0 = self.b0.repeat(x.shape[0],1)
        x = torch.matmul(x, self.w0.T) + b0
        x = torch.nn.functional.relu(x)
        b1 = self.b1.repeat(x.shape[0],1)
        x = torch.matmul(x, self.w1.T) + b1
        x = torch.nn.functional.relu(x)
        b2 = self.b2.repeat(x.shape[0],1)
        x = torch.matmul(x, self.w2.T) + b2
        return x

def train(model: Module, optimizer: Adam, train_loader: DataLoader) -> List[float]:
    epochs = 32
    losses = []
    accuracies = []
    for i in range(epochs):
        loss_epoch = 0
        acc = 0
        for images, labels in train_loader:
            x, y = images, labels
            y_preds = model.forward(x)
            loss_rand = cross_entropy(y_preds, y)
            optimizer.zero_grad()
            loss_rand.backward()
            optimizer.step()
            preds = torch.argmax(y_preds, 1)
            acc += torch.sum(preds == y)/len(preds)
```

```
            loss_epoch += loss_rand.item()
        acc = acc / len(train_loader)
        print(acc)
        if acc > 0.99:
            break
        losses.append(loss_epoch/len(train_loader))
        accuracies.append(acc)
    return losses

def main():
    (x, y), (x_test, y_test) = load_dataset("mnist")
    x = torch.from_numpy(x).float()
    y = torch.from_numpy(y).long()
    x_test = torch.from_numpy(x_test).float()
    y_test = torch.from_numpy(y_test).long()

    # Question 6a
    model = F1(h = 64, d = 784, k = 10)
    optimizer = Adam(model.params, lr = 5e-3)
    train_loader = DataLoader(TensorDataset(x,y), batch_size = 64, shuffle=True)
    losses = train(model, optimizer, train_loader)
    y_hat = model(x_test)
    test_prds = torch.argmax(y_hat,1)
    accuracy_val = torch.sum(test_prds == y_test)/len(test_prds)
    print(accuracy_val,'******')
    test_loss_val = cross_entropy(y_hat, y_test).item()
    print(test_loss_val,'******')
    plt.plot(np.arange(len(losses)), losses)
    plt.xlabel('epoch')
    plt.ylabel('loss')
    plt.show()
    total_params = sum(param.numel() for param in model.parameters())
    print(total_params,'****')

    # Question 6b
    model = F2(h0 = 32, h1 = 32, d = 784, k = 10)
    optimizer = Adam(model.params, lr = 5e-3)
    train_loader = DataLoader(TensorDataset(x,y), batch_size = 64, shuffle=True)
    losses = train(model, optimizer, train_loader)
    y_hat = model(x_test)
    test_prds = torch.argmax(y_hat,1)
    accuracy_val = torch.sum(test_prds == y_test)/len(test_prds)
    print(accuracy_val,'******')
    test_loss_val = cross_entropy(y_hat, y_test).item()
    print(test_loss_val,'******')
    plt.plot(np.arange(len(losses)), losses)
    plt.xlabel('epoch')
    plt.ylabel('loss')
    plt.show()
    total_params = sum(param.numel() for param in model.parameters())
    print(total_params,'****')

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