CSE546 - Homework # 3 - Solutions

Cheng-Yen Yang

May 22, 2021

A.1

A.1.a

When we trained an Support Vector Machine classifier with an RBF kernel:

$$K(u,v) = \exp\left(-\frac{\|u-v\|_2^2}{2\sigma^2}\right) = \exp\left(-\gamma \|u-v\|_2^2\right), \tag{1}$$

and found it under-fitting the training set, we will want to tighten the margins by increasing the value of γ (**decreasing the value of σ**).

A.1.b

True, gradient descent might not reach the globally-optimal solution but the locally-optimal solution instead when minimizing a non-convex loss function.

A.1.c

False, initializing all weights to zero is overall not an ideal practice. For example, consider a neural network with sigmoid activation function, the derivative with respect to the loss function will be identical across all weights resulting in undesirable training situation.

A.1.d

False, non-linear activation functions introduce non-linearity into the neural network's hidden layers or else it will be nothing other than a linear combination of the inputs.

A.1.e

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.

A.2

By definition of $\phi(x)$, we have $\phi(x)\phi(x')$:

$$\begin{aligned}\phi(x)\cdot\phi(x') &= \sum_{i=0}^{\infty}\frac{1}{\sqrt{i!}}e^{-\frac{x^2}{2}}x^i\cdot\frac{1}{\sqrt{i!}}e^{-\frac{x'^2}{2}}x'^i && (2)\\ &= e^{-\frac{x^2+x'^2}{2}}\sum_{i=0}^{\infty}\frac{(xx')^i}{i!}, && (3)\end{aligned}$$

then by using the Taylor expansion $e^z = \sum_{n=0}^{\infty}\frac{z^n}{n!}$:

$$\begin{aligned}\phi(x)\cdot\phi(x') &= e^{-\frac{x^2+x'^2}{2}}e^{xx'} && (4)\\ &= e^{-\frac{(x-x')^2}{2}}. && (5)\end{aligned}$$

A.3

A.3.a

(Polynomial Kernel) $d = 11$ and $\lambda = 0.1$
(RBF Kernel) $\gamma = 3$ and $\lambda = 0.00001$

A.3.b

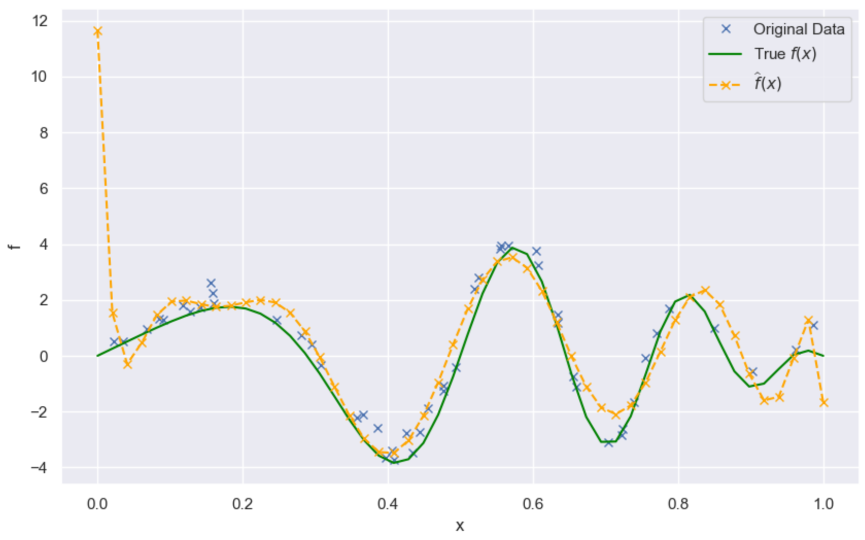

Figure 1: Plot of original data $\{(x_i, y_i)\}_{i=1}^n$, true $f(x)$ and $\widehat{f}_{poly}(x)$ using leave-one-out cross validation.

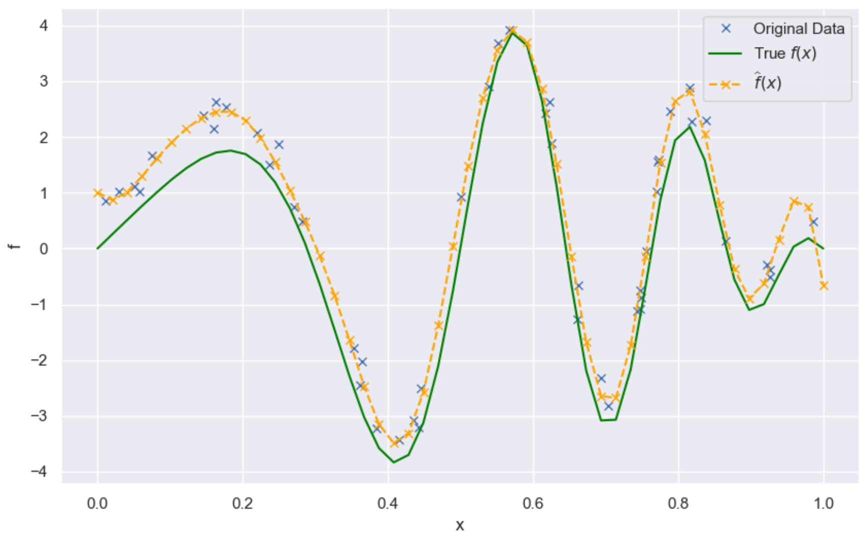

Figure 2: Plot of original data $\{(x_i, y_i)\}_{i=1}^n$, true $f(x)$ and $\widehat{f}_{rbf}(x)$ using leave-one-out cross validation.

A.3.c

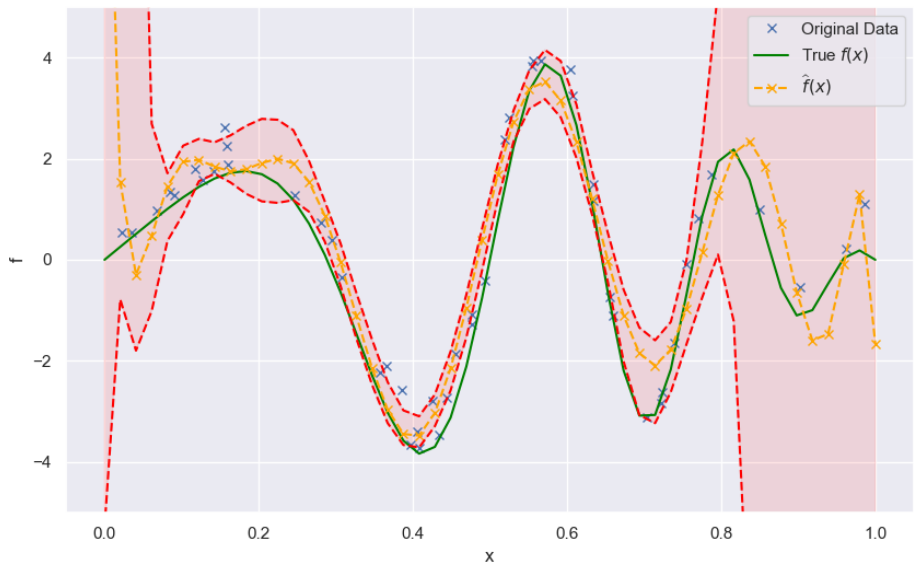

Figure 3: Plot of original data $\{(x_i, y_i)\}_{i=1}^n$, true $f(x)$, $\widehat{f}_{poly}(x)$ and its 5 and 95 percentile curves using leave-one-out cross validation.

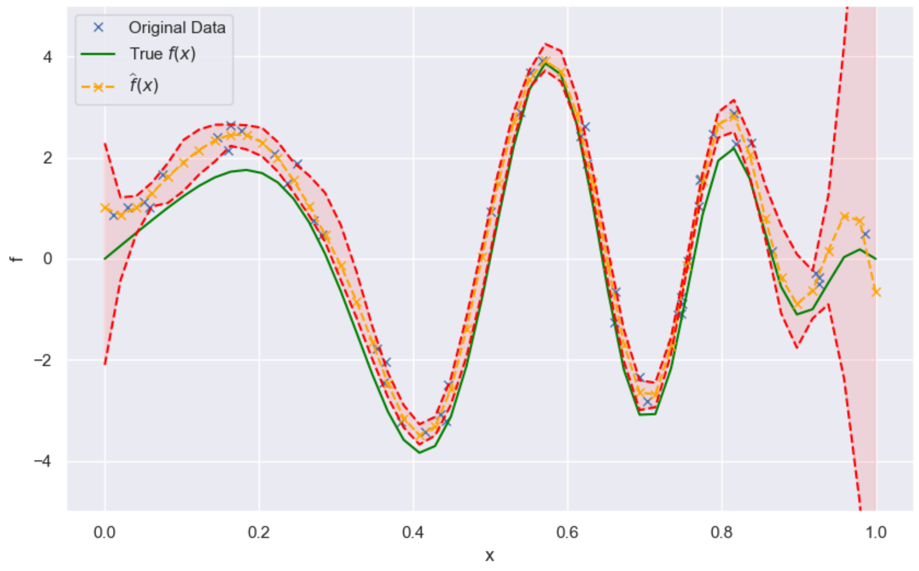

Figure 4: Plot of original data $\{(x_i, y_i)\}_{i=1}^n$, true $f(x)$, $\widehat{f}_{rbf}(x)$ and its 5 and 95 percentile curves using leave-one-out cross validation.

A.3.d

(Polynomial Kernel) $d = 14$ and $\lambda = 0.01$
(RBF Kernel) $\gamma = 2$ and $\lambda = 0.0000001$

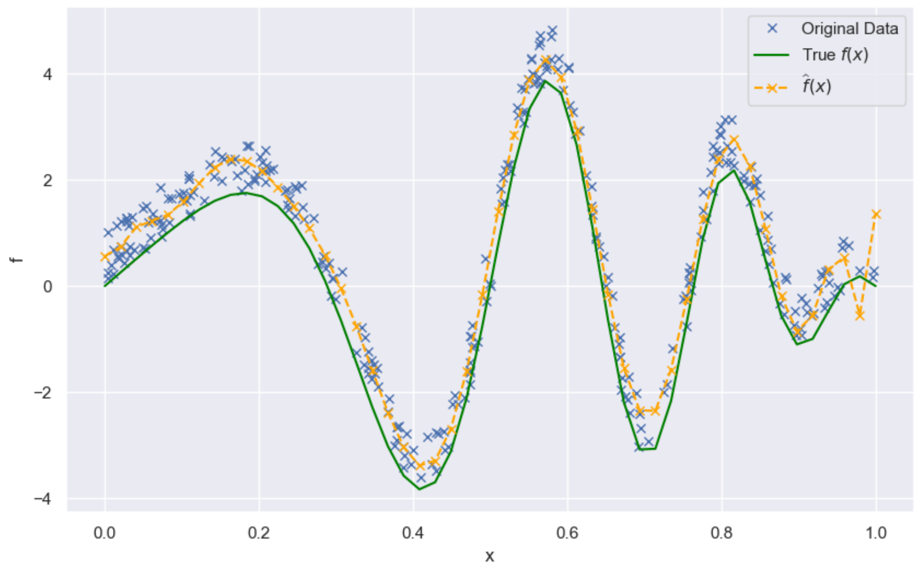

Figure 5: Plot of original data $\{(x_i, y_i)\}_{i=1}^{n}$, true $f(x)$ and $\widehat{f}_{poly}(x)$ using k-fold cross validation with $k = 30$.

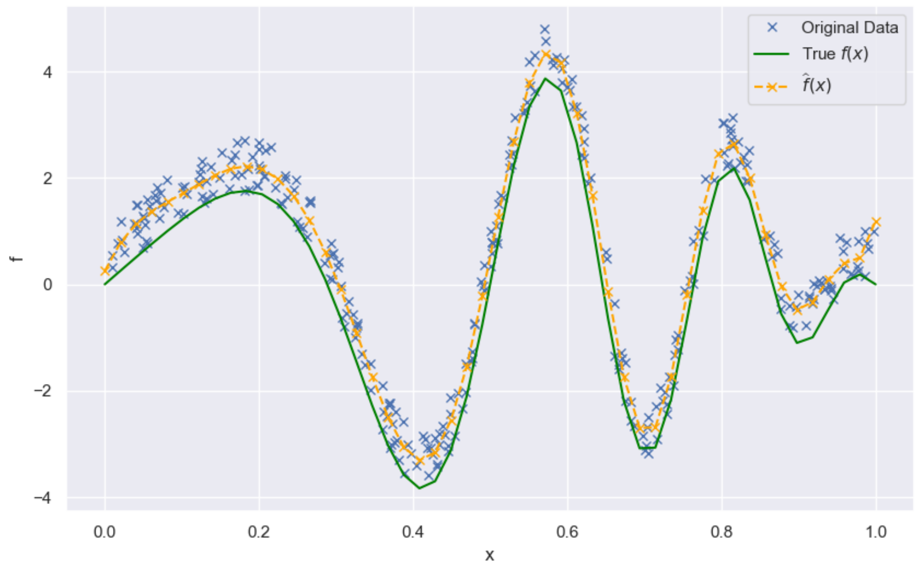

Figure 6: Plot of original data $\{(x_i, y_i)\}_{i=1}^{n}$, true $f(x)$ and $\widehat{f}_{rbf}(x)$ using k-fold cross validation with $k = 30$.

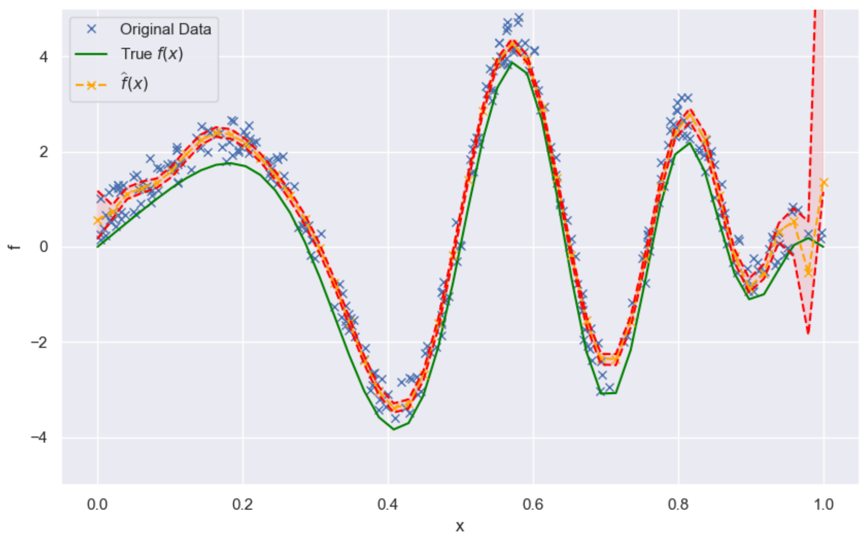

Figure 7: Plot of original data $\{(x_i, y_i)\}_{i=1}^n$, true $f(x)$, $\widehat{f}_{poly}(x)$ and its 5 and 95 percentile curves using k-fold cross validation with $k = 10$.

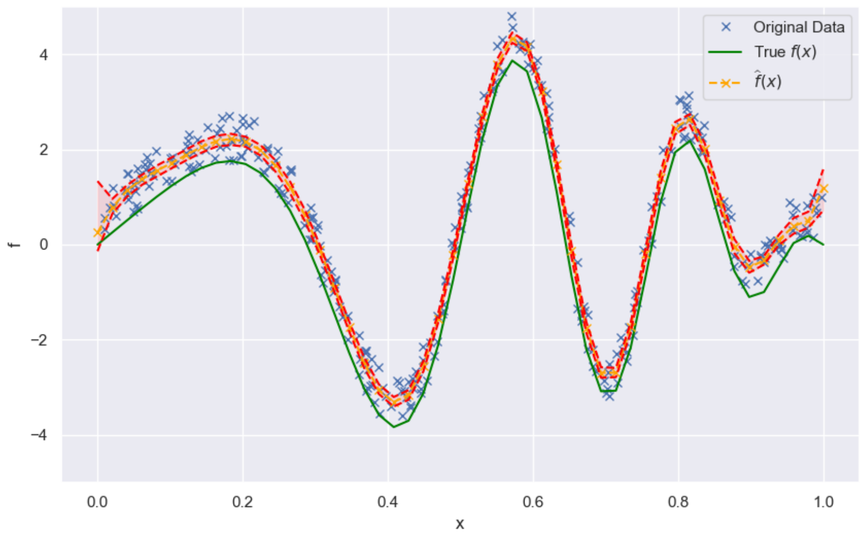

Figure 8: Plot of original data $\{(x_i, y_i)\}_{i=1}^n$, true $f(x)$, $\widehat{f}_{rbf}(x)$ and its 5 and 95 percentile curves using k-fold cross validation with $k = 10$.

A.3.e

The 5 and 95 percentile for $\frac{1}{m}\sum_{i=1}^{m}\left[\left(\widetilde{y}_i' - \widehat{f}_{\text{poly}}(\widetilde{x}_i')\right)^2 - \left(\widetilde{y}_i' - \widehat{f}_{\text{rbf}}(\widetilde{x}_i')\right)^2\right]$ are $[0.024328, 0.035018]$ which is an evidence to suggest that one of $\widehat{f}_{\text{rbf}}$ and $\widehat{f}_{\text{poly}}$ is better than the other at predicting Y from X as the interval did not contain 0.

A.3.code

```
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from sklearn.model_selection import KFold, LeaveOneOut

sns.set()

np.random.seed(1968990 + 2021051)


# Polynomial kernel function
def k_poly(X, Z, d):
    return (1 + np.outer(X, Z)) ** d


# RBF kernel function
def k_rbf(X, Z, gamma):
    return np.exp(-gamma * np.subtract.outer(X, Z) ** 2)


class KRR:
    def __init__(self, type='poly', reg_lambda=1e-3, d=1, gamma=1):
        self.reg_lambda = reg_lambda
        if type == 'poly':
            self.kernel_fn = k_poly
            self.param = d
        if type == 'rbf':
            self.kernel_fn = k_rbf
            self.param = gamma
        self.X = None
        self.X_mean = None
        self.X_std = None
        self.alpha = None

    def fit(self, X, Y):
        self.X_mean, self.X_std = np.mean(X, axis=0), np.std(X, axis=0)
        self.X = (X - self.X_mean) / self.X_std
        # https://stackoverflow.com/questions/54771760/closed-form-ridge-
            regression
        # https://people.eecs.berkeley.edu/~wainwrig/stat241b/lec6.pdf
        self.alpha = np.linalg.solve(self.kernel_fn(self.X, self.X, self.param)
           + self.reg_lambda * np.eye(X.shape[0]), Y)

    def predict(self, Z):
        Z = (Z - self.X_mean) / self.X_std
        return np.dot(self.alpha, self.kernel_fn(self.X, Z, self.param))


# https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.
    LeaveOneOut.html
def leave_one_out_cross_validation(X, Y, model):
    loo = LeaveOneOut()
    error = []
    for train_index, test_index in loo.split(X):
        X_train, X_test = X[train_index], X[test_index]
        Y_train, Y_test = Y[train_index], Y[test_index]
        model.fit(X_train, Y_train)
        Y_pred = model.predict(X_test)
        error.append(np.mean((Y_pred - Y_test) ** 2))
    return np.mean(error)


def k_fold_cross_validation(X, Y, model, k):
    if k == X.shape[0]:
        return leave_one_out_cross_validation(X, Y, model)
    kf = KFold(n_splits=k)
    error = []
    for train_index, test_index in kf.split(X):
        X_train, X_test = X[train_index], X[test_index]
        Y_train, Y_test = Y[train_index], Y[test_index]
        model.fit(X_train, Y_train)
        Y_pred = model.predict(X_test)
        error.append(np.mean((Y_pred - Y_test) ** 2))
    return np.mean(error)


# https://ogrisel.github.io/scikit-learn.org/sklearn-tutorial/modules/generated/
    sklearn.cross_validation.Bootstrap.html
def bootstrap(X, Y, B, model):
    x = np.linspace(0, 1, 50)
    preds = []
    indices = np.arange(X.shape[0])

    for i in range(B):
        boot_index = np.random.choice(indices, size=X.shape[0], replace=True)
        X_boot, Y_boot = X[boot_index], Y[boot_index]
        model.fit(X_boot, Y_boot)
        preds.append(model.predict(x))

    return preds


def bootstrap_two_model(X, Y, B, poly_model, rbf_model, x):
    poly_preds, rbf_preds = [], []
    square_error = []
    indices = np.arange(X.shape[0])

    for i in range(B):
        boot_index = np.random.choice(indices, size=X.shape[0], replace=True)
        X_boot, Y_boot = X[boot_index], Y[boot_index]
        # model.fit(X_boot, Y_boot)
        poly_preds.append(poly_model.predict(X_boot))
        rbf_preds.append(rbf_model.predict(X_boot))
        square_error.append(np.mean((Y_boot - poly_preds[-1]) ** 2 - (Y_boot -
            rbf_preds[-1]) ** 2))

    print(np.percentile(square_error, 5, axis=0), np.percentile(square_error,
        95, axis=0))
```

```
def main_poly(n, B):
    X = np.random.uniform(size=n)
    Y = 4 * np.sin(np.pi * X) * np.cos(6 * np.pi * X ** 2) + np.random.rand(n)

    model = KRR('poly')

    lambdas = 10. ** (-1 * np.arange(0, 10))
    ds = np.arange(2, 15)
    best_err = np.float('inf')
    best_lambda = None
    best_d = None

    for reg_lambda in lambdas:
        for d in ds:
            # print('Fitting Polynomial Kernel(lambda={}, d={})'.format(
                reg_lambda, d))
            model.reg_lambda = reg_lambda
            model.param = d
            error = leave_one_out_cross_validation(X, Y, model)
            if error < best_err:
                best_lambda = reg_lambda
                best_d = d
                best_err = error
    print('Best Hyper-Parameters for Polynomial Kernel: lambda={}, d={}'.format(
        best_lambda, best_d))

    # Plot
    model.reg_lambda = best_lambda
    model.param = best_d
    model.fit(X, Y)

    x = np.linspace(0.0, 1.0, 50)
    f_true = 4 * np.sin(np.pi * x) * np.cos(6 * np.pi * x ** 2)
    f_hat = model.predict(x)

    plt.figure(figsize=(10, 6))
    plt.plot(X[np.argsort(X)], Y[np.argsort(X)], 'x', label='Original Data')
    plt.plot(x, f_true, label='True $f(x)$', color='green')
    plt.plot(x, f_hat, '--x', label='$\widehat f(x)$', color='orange')
    plt.xlabel('x')
    plt.ylabel('f')
    plt.legend()
    plt.savefig('A3b_poly.png')
    plt.show()

    bootstrap_preds = bootstrap(X, Y, B, model)
    plt.figure(figsize=(10, 6))
    plt.plot(X[np.argsort(X)], Y[np.argsort(X)], 'x', label='Original Data')
    plt.plot(x, f_true, label='True $f(x)$', color='green')
    plt.plot(x, f_hat, '--x', label='$\widehat f(x)$', color='orange')
    plt.plot(x, np.percentile(bootstrap_preds, 5, axis=0), '--', color='red')
    plt.plot(x, np.percentile(bootstrap_preds, 95, axis=0), '--', color='red')
    plt.fill_between(x, np.percentile(bootstrap_preds, 5, axis=0), np.percentile
        (bootstrap_preds, 95, axis=0),
                     alpha=0.1, color="red")
    plt.ylim((-5, 5))
    plt.xlabel('x')
    plt.ylabel('f')
    plt.legend()
    plt.savefig('A3c_poly.png')
    plt.show()


def main_rbf(n, B):
    X = np.random.uniform(size=n)
    Y = 4 * np.sin(np.pi * X) * np.cos(6 * np.pi * X ** 2) + np.random.rand(n)

    model = KRR('rbf')

    lambdas = 10. ** (-1 * np.arange(0, 10))
    gammas = np.arange(2, 15, 0.5)
    best_err = np.float('inf')
    best_lambda = None
    best_gamma = None

    for reg_lambda in lambdas:
        for gamma in gammas:
            # print('Fitting Polynomial Kernel(lambda={}, gamma={})'.format(
                reg_lambda, gamma))
            model.reg_lambda = reg_lambda
            model.param = gamma
            error = leave_one_out_cross_validation(X, Y, model)
            if error < best_err:
                best_lambda = reg_lambda
                best_gamma = gamma
                best_err = error
    print('Best Hyper-Parameters for RBF kernel: lambda={}, gamma={}'.format(
        best_lambda, best_gamma))

    # Plot
    model.reg_lambda = best_lambda
    model.param = best_gamma
    model.fit(X, Y)

    x = np.linspace(0.0, 1.0, 50)
    f_true = 4 * np.sin(np.pi * x) * np.cos(6 * np.pi * x ** 2)
    f_hat = model.predict(x)

    plt.figure(figsize=(10, 6))
    plt.plot(X[np.argsort(X)], Y[np.argsort(X)], 'x', label='Original Data')
    plt.plot(x, f_true, label='True $f(x)$', color='green')
    plt.plot(x, f_hat, '--x', label='$\widehat f(x)$', color='orange')
    plt.xlabel('x')
    plt.ylabel('f')
    plt.legend()
    plt.savefig('A3b_rbf.png')
    bootstrap_preds = bootstrap(X, Y, B, model)
    plt.figure(figsize=(10, 6))
    plt.plot(X[np.argsort(X)], Y[np.argsort(X)], 'x', label='Original Data')
    plt.plot(x, f_true, label='True $f(x)$', color='green')
    plt.plot(x, f_hat, '--x', label='$\widehat f(x)$', color='orange')
    plt.plot(x, np.percentile(bootstrap_preds, 5, axis=0), '--', color='red')
    plt.plot(x, np.percentile(bootstrap_preds, 95, axis=0), '--', color='red')
    plt.fill_between(x, np.percentile(bootstrap_preds, 5, axis=0), np.percentile
        (bootstrap_preds, 95, axis=0),
                     alpha=0.1, color="red")
    plt.ylim((-5, 5))
    plt.xlabel('x')
    plt.ylabel('f')
    plt.legend()
    plt.savefig('A3c_rbf.png')
    plt.show()


def main_poly_kf(n, k, B):
    X = np.random.uniform(size=n)
    Y = 4 * np.sin(np.pi * X) * np.cos(6 * np.pi * X ** 2) + np.random.rand(n)

    model = KRR('poly')

    lambdas = 10. ** (-1 * np.arange(0, 10))
    ds = np.arange(2, 15)
    best_err = np.float('inf')
    best_lambda = None
    best_d = None

    for reg_lambda in lambdas:
        for d in ds:
            # print('Fitting Polynomial Kernel(lambda={}, d={})'.format(
                reg_lambda, d))
            model.reg_lambda = reg_lambda
            model.param = d
            error = k_fold_cross_validation(X, Y, model, k)
            if error < best_err:
                best_lambda = reg_lambda
                best_d = d
                best_err = error
    print('Best Hyper-Parameters for Polynomial Kernel: lambda={}, d={}'.format(
        best_lambda, best_d))

    # Plot
    model.reg_lambda = best_lambda
    model.param = best_d
    model.fit(X, Y)

    x = np.linspace(0.0, 1.0, 50)
    f_true = 4 * np.sin(np.pi * x) * np.cos(6 * np.pi * x ** 2)
    f_hat = model.predict(x)

    plt.figure(figsize=(10, 6))
    plt.plot(X[np.argsort(X)], Y[np.argsort(X)], 'x', label='Original Data')
    plt.plot(x, f_true, label='True $f(x)$', color='green')
    plt.plot(x, f_hat, '--x', label='$\widehat f(x)$', color='orange')
    plt.xlabel('x')
    plt.ylabel('f')
    plt.legend()
    plt.savefig('A3d_poly_1.png')

    bootstrap_preds = bootstrap(X, Y, B, model)
    plt.figure(figsize=(10, 6))
    plt.plot(X[np.argsort(X)], Y[np.argsort(X)], 'x', label='Original Data')
    plt.plot(x, f_true, label='True $f(x)$', color='green')
    plt.plot(x, f_hat, '--x', label='$\widehat f(x)$', color='orange')
    plt.plot(x, np.percentile(bootstrap_preds, 5, axis=0), '--', color='red')
    plt.plot(x, np.percentile(bootstrap_preds, 95, axis=0), '--', color='red')
    plt.fill_between(x, np.percentile(bootstrap_preds, 5, axis=0), np.percentile
        (bootstrap_preds, 95, axis=0),
                    alpha=0.1, color="red")
    plt.ylim((-5, 5))
    plt.xlabel('x')
    plt.ylabel('f')
    plt.legend()
    plt.savefig('A3d_poly_2.png')

    return model  # return best Poly model


def main_rbf_kf(n, B, k):
    X = np.random.uniform(size=n)
    Y = 4 * np.sin(np.pi * X) * np.cos(6 * np.pi * X ** 2) + np.random.rand(n)

    model = KRR('rbf')

    lambdas = 10. ** (-1 * np.arange(0, 10))
    gammas = np.arange(2, 15, 0.5)
    best_err = np.float('inf')
    best_lambda = None
    best_gamma = None

    for reg_lambda in lambdas:
        for gamma in gammas:
            # print('Fitting RBF Kernel(lambda={}, gamma={})'.format(reg_lambda,
                gamma))
            model.reg_lambda = reg_lambda
            model.param = gamma
            error = k_fold_cross_validation(X, Y, model, k)
            if error < best_err:
                best_lambda = reg_lambda
                best_gamma = gamma
                best_err = error
    print('Best Hyper-Parameters for RBF kernel: lambda={}, gamma={}'.format(
        best_lambda, best_gamma))

    # Plot
    model.reg_lambda = best_lambda
    model.param = best_gamma
    model.fit(X, Y)

    x = np.linspace(0.0, 1.0, 50)
    f_true = 4 * np.sin(np.pi * x) * np.cos(6 * np.pi * x ** 2)
    f_hat = model.predict(x)

    plt.figure(figsize=(10, 6))
    plt.plot(X[np.argsort(X)], Y[np.argsort(X)], 'x', label='Original Data')
    plt.plot(x, f_true, label='True $f(x)$', color='green')
    plt.plot(x, f_hat, '--x', label='$\widehat f(x)$', color='orange')
    plt.xlabel('x')
    plt.ylabel('f')
    plt.legend()
    plt.savefig('A3d_rbf_1.png')

    bootstrap_preds = bootstrap(X, Y, B, model)
    plt.figure(figsize=(10, 6))
    plt.plot(X[np.argsort(X)], Y[np.argsort(X)], 'x', label='Original Data')
    plt.plot(x, f_true, label='True $f(x)$', color='green')
    plt.plot(x, f_hat, '--x', label='$\widehat f(x)$', color='orange')
    plt.plot(x, np.percentile(bootstrap_preds, 5, axis=0), '--', color='red')
    plt.plot(x, np.percentile(bootstrap_preds, 95, axis=0), '--', color='red')
    plt.fill_between(x, np.percentile(bootstrap_preds, 5, axis=0), np.percentile
        (bootstrap_preds, 95, axis=0),
                     alpha=0.1, color="red")
    plt.ylim((-5, 5))
    plt.xlabel('x')
    plt.ylabel('f')
    plt.legend()
    plt.savefig('A3d_rbf_2.png')

    return model  # return best RBF model


def main_e(poly_model, rbf_model, n=1000, B=300):
    X = np.random.uniform(size=n)
    Y = 4 * np.sin(np.pi * X) * np.cos(6 * np.pi * X ** 2) + np.random.rand(n)

    poly_model.fit(X, Y)
    rbf_model.fit(X, Y)

    bootstrap_two_model(X, Y, B, poly_model, rbf_model, X)


if __name__ == '__main__':
    main_poly(n=50, B=300)
    main_rbf(n=50, B=300)
    poly_model = main_poly_kf(n=300, k=10, B=300)
    rbf_model = main_rbf_kf(n=300, k=10, B=300)
    main_e(poly_model, rbf_model)
```

A.4

A.4.a

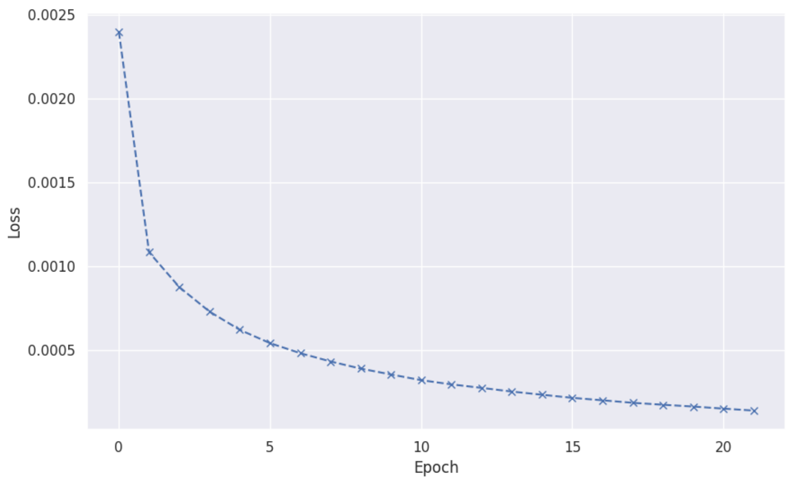

Figure 9: Plot of the training loss versus epoch for the two-layer model with test accuracy being 97.5% and test loss being 0.08623.

A.4.b

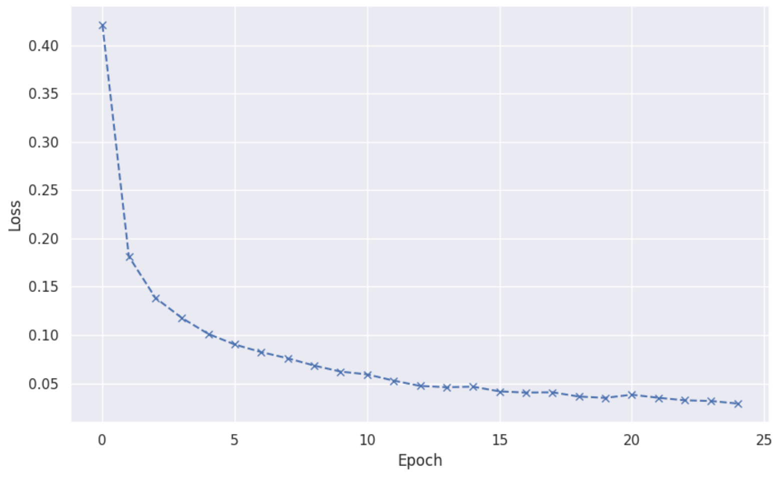

Figure 10: Plot of the training loss versus epoch for the three-layer model with test accuracy being 96.7% and test loss being 0.126628.

A.4.c

The two-layer model have 50,890 trainable parameters in total while the three-layer model have 26,506 trainable parameters. We can observe that both models give very similar results in term of the training process and testing evaluation. As both model converges around 25 epochs, the three-layer model takes half the training time in comparison to the two-layer model due to its smaller parameter numbers.

A.4.code

```
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
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

sns.set()


np.random.seed(1968990 + 20210517)
torch.manual_seed(1968990 + 20210517)

# https://courses.cs.washington.edu/courses/cse446/21sp/sections/07/
    Pytorch_Neural_Networks.html

def preapare_dataset():

    mnist_trainset = datasets.MNIST(root='./data', train=True, download=True,
        transform=transforms.ToTensor())
    mnist_testset = datasets.MNIST(root='./data', train=False, download=True,
        transform=transforms.ToTensor())
    train_loader = torch.utils.data.DataLoader(mnist_trainset, batch_size=256,
        shuffle=True)
    test_loader = torch.utils.data.DataLoader(mnist_testset, batch_size=256,
        shuffle=True)

    return train_loader, test_loader


# # https://discuss.pytorch.org/t/how-are-layer-weights-and-biases-initialized-
    by-default/13073
# # stdv = 1. / math.sqrt(self.weight.size(1))
# # self.weight.data.uniform_(-stdv, stdv)
# # if self.bias is not None:
# #     self.bias.data.uniform_(-stdv, stdv)
# class MNISTNetwork_A4a(nn.Module):
#     def __init__(self, hidden_size):
#         super().__init__()
#         self.hidden_size = hidden_size
#         self.linear_0 = nn.Linear(784, self.hidden_size)
#         self.linear_1 = nn.Linear(self.hidden_size, 10)
#
#     def forward(self, inputs):
#         x = self.linear_0(inputs)
#         x = F.relu(x)
#            x = self.linear_1(x)
#            return F.log_softmax(x, dim=1)
#
#
# class MNISTNetwork_A4b(nn.Module):
#       def __init__(self, hidden_size):
#            super().__init__()
#            self.hidden_size = hidden_size
#            self.linear_0 = nn.Linear(784, self.hidden_size)
#            self.linear_1 = nn.Linear(self.hidden_size, self.hidden_size)
#            self.linear_2 = nn.Linear(self.hidden_size, 10)
#
#       def forward(self, inputs):
#            x = self.linear_0(inputs)
#            x = F.relu(x)
#            x = self.linear_1(x)
#            x = F.relu(x)
#            x = self.linear_2(x)
#            return F.log_softmax(x, dim=1)


class A4a_net():
    def __init__(self, input_dim=784, hidden_size=64, output_dim=10):
        self.alpha0 = 1 / np.sqrt(input_dim)
        self.alpha1 = 1 / np.sqrt(hidden_size)
        self.w0 = torch.FloatTensor(input_dim, hidden_size).uniform_(-self.
            alpha0, self.alpha0)
        self.b0 = torch.FloatTensor(1, hidden_size).uniform_(-self.alpha0, self.
            alpha0)
        self.w1 = torch.FloatTensor(hidden_size, output_dim).uniform_(-self.
            alpha1, self.alpha1)
        self.b1 = torch.FloatTensor(1, output_dim).uniform_(-self.alpha1, self.
            alpha1)

        self.params = [self.w0, self.b0, self.w1, self.b1]
        for param in self.params:
            param.requires_grad = True

    def forward(self, inputs):
        x = torch.matmul(inputs, self.w0) + self.b0
        x = F.relu(x)
        x = torch.matmul(x, self.w1) + self.b1
        return x

class A4b_net():
    def __init__(self, input_dim=784, hidden_size=32, output_dim=10):
        self.alpha0 = 1 / np.sqrt(input_dim)
        self.alpha1 = 1 / np.sqrt(hidden_size)
        self.w0 = torch.FloatTensor(input_dim, hidden_size).uniform_(-self.
            alpha0, self.alpha0)
        self.b0 = torch.FloatTensor(1, hidden_size).uniform_(-self.alpha0, self.
            alpha0)
        self.w1 = torch.FloatTensor(hidden_size, hidden_size).uniform_(-self.
            alpha1, self.alpha1)
        self.b1 = torch.FloatTensor(1, hidden_size).uniform_(-self.alpha1, self.
            alpha1)
        self.w2 = torch.FloatTensor(hidden_size, output_dim).uniform_(-self.
            alpha1, self.alpha1)
        self.b2 = torch.FloatTensor(1, output_dim).uniform_(-self.alpha1, self.
            alpha1)

        self.params = [self.w0, self.b0, self.w1, self.b1, self.w2, self.b2]
        for param in self.params:
            param.requires_grad = True

    def forward(self, inputs):
        x = torch.matmul(inputs, self.w0) + self.b0
        x = F.relu(x)
        x = torch.matmul(x, self.w1) + self.b1
        x = F.relu(x)
        x = torch.matmul(x, self.w2) + self.b2
        return x


def main_A4a():
    n = 28
    input_size = 784
    hidden_size = 64
    k = 10
    epochs = 32

    train_loader, test_loader = preapare_dataset()

    # model = MNISTNetwork_A4a(hidden_size).to("cuda")
    # optimizer = optim.Adam(model.parameters(), lr=5e-3)

    model = A4a_net(input_size, hidden_size, k)
    optimizer = optim.Adam(model.params, lr=5e-3)


    losses = []
    for i in range(epochs):
        loss_epoch = 0
        acc = 0
        for batch in tqdm(train_loader):
            images, labels = batch
            images, labels = images, labels
            images = images.view(-1, 784)
            optimizer.zero_grad()
            logits = model.forward(images)
            preds = torch.argmax(logits, 1)
            acc += torch.sum(preds == labels).item()
            loss = F.cross_entropy(logits, labels)
            loss_epoch += loss.item()
            loss.backward()
            optimizer.step()

        print("Epoch ", i)
        print("Loss:", loss_epoch / len(train_loader.dataset))
        print("Acc:", acc / len(train_loader.dataset))
        losses.append(loss_epoch / len(train_loader.dataset))
        if acc / len(train_loader.dataset) > 0.99:
            break

    loss_epoch = 0
    acc = 0
    for batch in tqdm(test_loader):
        images, labels = batch
        images, labels = images, labels
        images = images.view(-1, 784)

        logits = model.forward(images)
        preds = torch.argmax(logits, 1)
        acc += torch.sum(preds == labels).item()
        loss = F.cross_entropy(logits, labels)
        loss_epoch += loss.item()
    print('Testing dataset')
    print("Loss:", loss_epoch / len(test_loader))
    print("Acc:", acc / len(test_loader.dataset))

    plt.figure(figsize=(10, 6))
    plt.plot(np.arange(len(losses)), losses, '--x', label='Training Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.savefig('A4a.png')

def main_A4b():
    n = 28
    input_size = 784
    hidden_size = 32
    k = 10
    epochs = 64

    train_loader, test_loader = preapare_dataset()

    # model = MNISTNetwork_A4b(hidden_size).to("cuda")
    # optimizer = optim.Adam(model.parameters(), lr=5e-3)

    model = A4b_net(input_size, hidden_size, k)
    optimizer = optim.Adam(model.params, lr=5e-3)

    # print(sum(p.numel() for p in model.parameters()))

    losses = []
    for i in range(epochs):
        loss_epoch = 0
        acc = 0
        for batch in tqdm(train_loader):
            images, labels = batch
            images, labels = images, labels
            images = images.view(-1, 784)
            optimizer.zero_grad()
            logits = model.forward(images)
            preds = torch.argmax(logits, 1)
            acc += torch.sum(preds == labels).item()
            loss = F.cross_entropy(logits, labels)
            loss_epoch += loss.item()
            loss.backward()
            optimizer.step()

        print("Epoch:", i)
        print("Loss:", loss_epoch / len(train_loader))
        print("Acc:", acc / len(train_loader.dataset))
        losses.append(loss_epoch / len(train_loader))
        if acc / len(train_loader.dataset) > 0.99:
            break

    loss_epoch = 0
    acc = 0
    for batch in tqdm(test_loader):
        images, labels = batch
        images, labels = images, labels
        images = images.view(-1, 784)

        logits = model.forward(images)
        preds = torch.argmax(logits, 1)
        acc += torch.sum(preds == labels).item()
        loss = F.cross_entropy(logits, labels)
        loss_epoch += loss.item()
    print('Testing dataset')
    print("Loss:", loss_epoch / len(test_loader))
    print("Acc:", acc / len(test_loader.dataset))

    plt.figure(figsize=(10, 6))
    plt.plot(np.arange(len(losses)), losses, '--x', label='Training Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.savefig('A4b.png')


if __name__ == '__main__':
    main_A4a()
    main_A4b()
```

A.5

A.5.a

For the model using pretrained AlextNet as a fixed feature extractor, we use the SGD optimizer with a learning rate of 0.005 and momentum of 0.9, the batch size is 256 and the training-validation split ratio is 0.9.

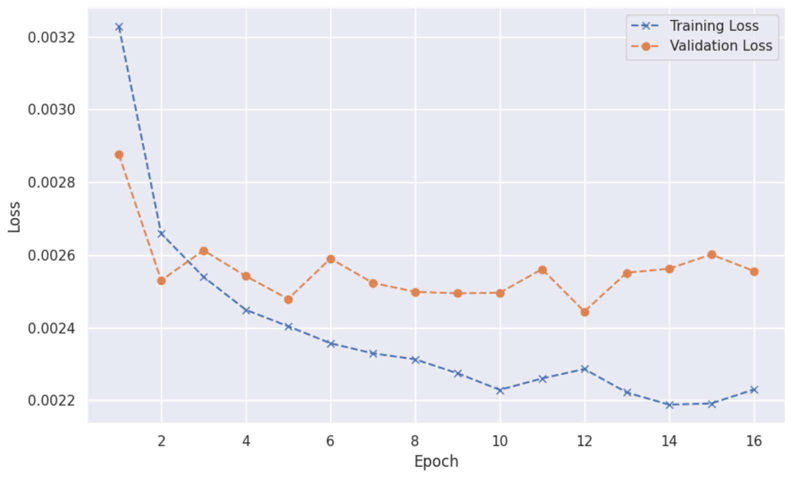

Figure 11: Plot of the training loss versus epoch for the AlexNet model in A.5.a.

We trained the model for 16 epochs and the highest validation accuracy achieved is 78.14000% while the validation loss is 0.00256. The final testing accuracy is 77.27000% and testing loss is 0.00258.

A.5.b

For the model using pretrained AlextNet, we use the SGD optimizer with a learning rate of 0.002 and momentum of 0.9, the batch size is 256 and the training-validation split ratio is 0.9.

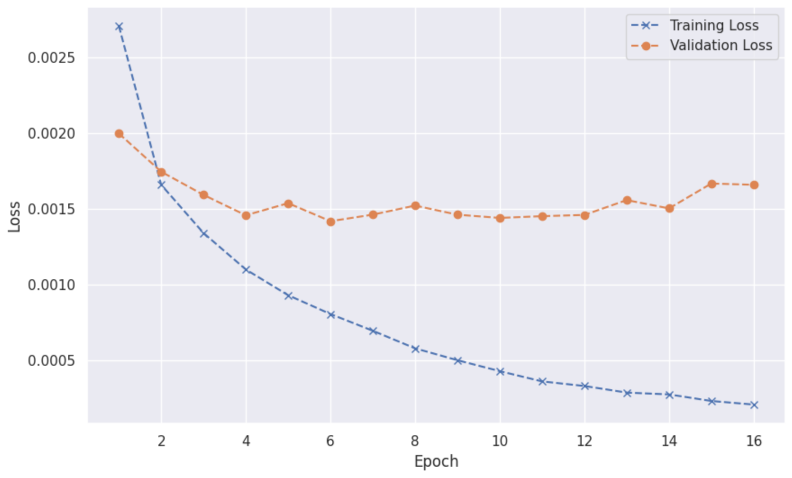

Figure 12: Plot of the training loss versus epoch for the AlexNet model in A.5.b.

We trained the model for 16 epochs and the highest validation accuracy achieved is 89.60000% while the validation loss is 0.00166. The final testing accuracy is 88.99000% and testing loss is 0.00167.

A.5.code

```
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
import torchvision.datasets as datasets
import torchvision.transforms as transforms
from tqdm import tqdm

sns.set()

np.random.seed(1968990 + 20210518)
torch.manual_seed(1968990 + 20210518)

transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

batch_size = 256


def prepare_dataset():
    cifar10_set = datasets.CIFAR10(root='./data', train=True, download=True,
       transform=transform)
    train_size = int(len(cifar10_set) * 0.9)
    val_size = len(cifar10_set) - train_size
    cifar10_trainset, cifar10_valset = torch.utils.data.random_split(cifar10_set
       , [train_size, val_size])
    cifar10_testset = datasets.CIFAR10(root='./data', train=False, download=True
       , transform=transform)

    train_loader = torch.utils.data.DataLoader(cifar10_trainset, batch_size=
       batch_size, shuffle=True)
    val_loader = torch.utils.data.DataLoader(cifar10_valset, batch_size=
       batch_size, shuffle=True)
    test_loader = torch.utils.data.DataLoader(cifar10_testset, batch_size=
       batch_size, shuffle=True)

    return train_loader, val_loader, test_loader

def eval(model, dataloader, epoch):
    correct = 0
    total = 0
    criterion = nn.CrossEntropyLoss()
    with torch.no_grad():
        running_loss = 0.0
        for data in dataloader:
            images, labels = data[0].to("cuda"), data[1].to("cuda")
            outputs = model(images)
            loss = criterion(outputs, labels)
            _, predicted = torch.max(outputs.data, 1)

            total += labels.size(0)
            correct += (predicted == labels).sum().item()
            running_loss += loss.item()

    print('[%d] Val Accuracy: %.5f %% Val Loss: %.5f' % (epoch + 1, 100 *
        correct / total, running_loss / total))
    return running_loss / total

def run(model, exp='A5a'):
    train_loader, val_loader, test_loader = prepare_dataset()

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=5e-3, momentum=0.9,
        weight_decay=1e-5)
    epochs = 16

    train_losses, val_losses = [], []
    for epoch in range(epochs):
        running_loss = 0.0
        acc = 0
        model.train()
        for i, data in enumerate(tqdm(train_loader)):
            inputs, labels = data[0].to("cuda"), data[1].to("cuda")
            optimizer.zero_grad()

            output = model(inputs)
            loss = criterion(output, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()

        print('[%d] Train Loss: %.5f' % (epoch + 1, running_loss / (len(
            train_loader) * batch_size)))
        train_losses.append(running_loss / (len(train_loader) * batch_size))
        running_loss = 0.0
        val_losses.append(eval(model, val_loader, epoch))

    eval(model, test_loader, -2)

    plt.figure(figsize=(10, 6))
    plt.plot(np.arange(1, epochs + 1), train_losses, '--x', label='Training Loss
        ')
    plt.plot(np.arange(1, epochs + 1), val_losses, '--o', label='Validation Loss
        ')
    plt.legend()
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.savefig(exp + '.png')
```

```
def main():
    model = torchvision.models.alexnet(pretrained=True)
    for param in model.parameters():
        param.requires_grad = False
    model.classifier[6] = nn.Linear(4096, 10)
    model.to('cuda')
    run(model, 'A5a')
    model = torchvision.models.alexnet(pretrained=True)
    model.classifier[6] = nn.Linear(4096, 10)
    model.to('cuda')
    run(model, 'A5b')


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

A.6

Note that for better visualization and comparison purpose we only plotted 6 set of the hyper-parameters for each models.

A.6.a

Fully-connected output, 0 hidden layers (logistic regression): this network has no hidden layers and linearly maps the input layer to the output layer. This can be written as

$$x^{\text{out}} = W(x^{\text{in}}) + b$$

where $x^{\text{out}} \in \mathbb{R}^{10}$, $x^{\text{in}} \in \mathbb{R}^{32\times32\times3}$, $W \in \mathbb{R}^{10\times3072}$, $b \in \mathbb{R}^{10}$ since $3072 = 32 \cdot 32 \cdot 3$. The hyper-parameters we try to tune in this subsection is batch size and learning rate, the experiments are plotted below:

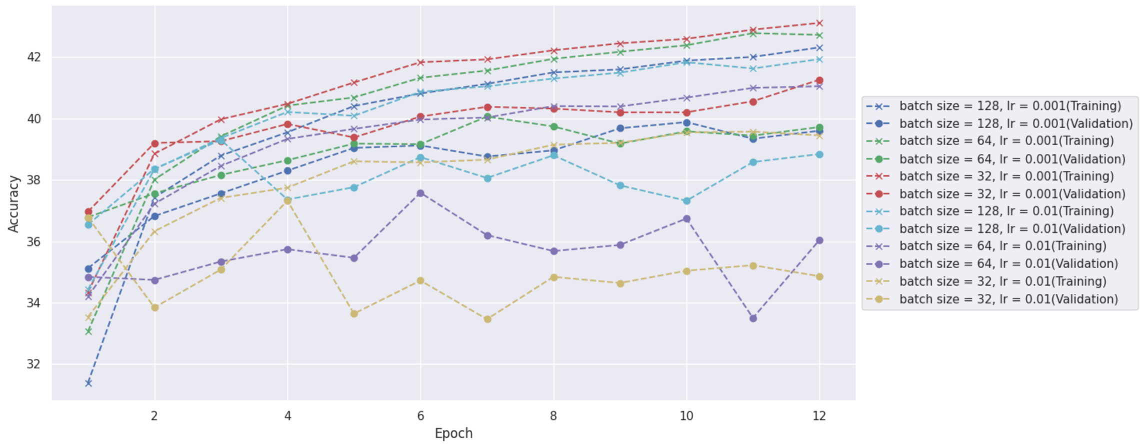

Figure 13: Plot of the training accuracies and validation accuracies versus epoch.

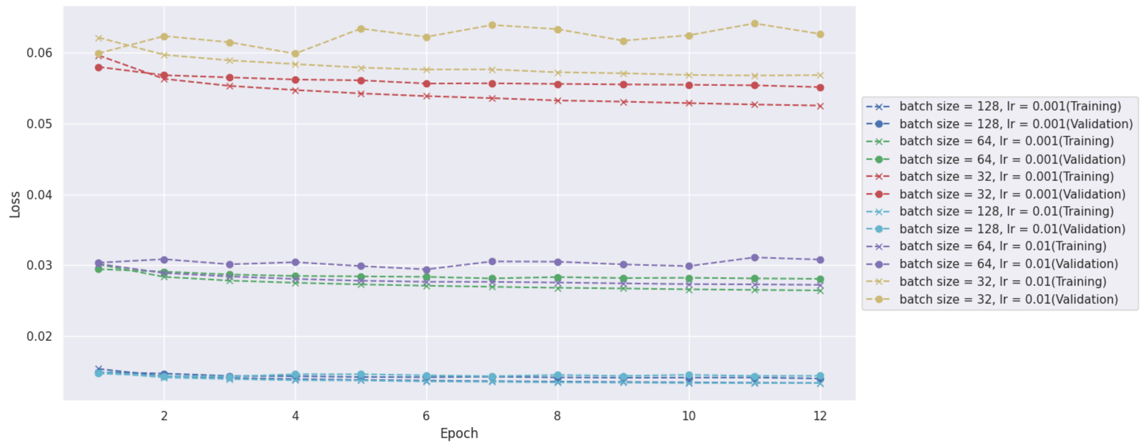

Figure 14: Plot of the training losses and validation losses versus epoch.

The best model from our experiments is the one with batch size of 32 and learning rate of 0.001 reaching a training accuracies of 43.153% and testing accuracies of 39.710% after 12 epochs.

A.6.b

Fully-connected output, 1 fully-connected hidden layer: this network has one hidden layer denoted as $x^{\text{hidden}} \in \mathbb{R}^M$ where M will be a hyperparameter you choose (M could be in the hundreds). The non-linearity applied to the hidden layer will be the `relu` (relu$(x) = \max\{0, x\}$. This network can be written as

$$x^{\text{out}} = W_2 \text{relu}(W_1(x^{\text{in}}) + b_1) + b_2$$

where $W_1 \in \mathbb{R}^{M \times 3072}$, $b_1 \in \mathbb{R}^M$, $W_2 \in \mathbb{R}^{10 \times M}$, $b_2 \in \mathbb{R}^{10}$. The hyper-parameters we try to tune in this subsection is batch size, learning rate and hidden size, the experiments are plotted below:

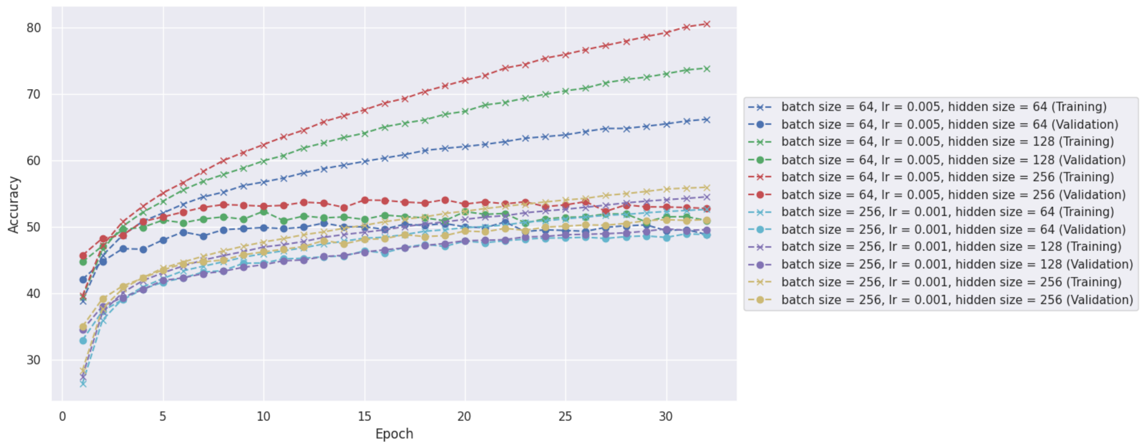

Figure 15: Plot of the training accuracies and validation accuracies versus epoch.

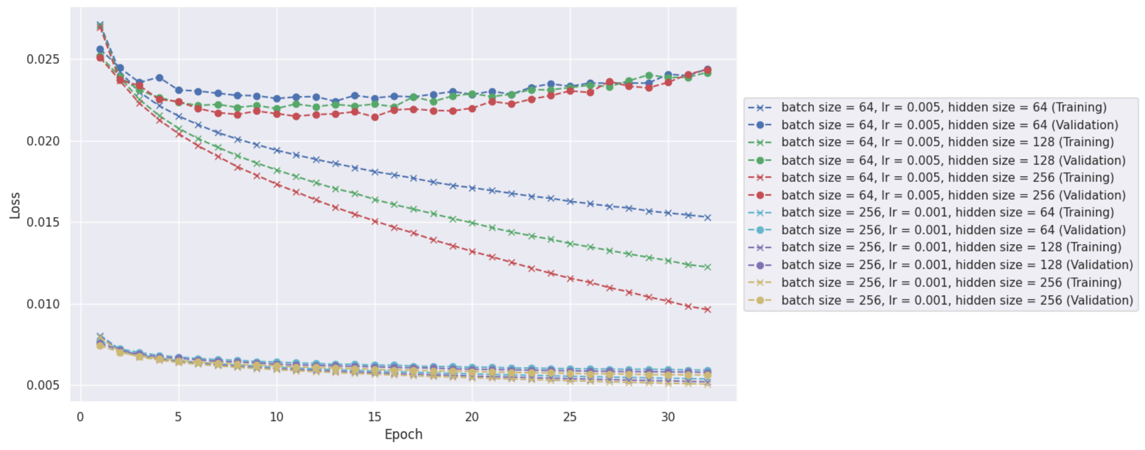

Figure 16: Plot of the training losses and validation losses versus epoch.

The best model from our experiments is the one with batch size of 64, learning rate of 0.005 and hidden size of 256 reaching a training accuracies of 80.787% and testing accuracies of 52.000% after 32 epochs.

A.6.c

Convolutional layer with max-pool and fully-connected output: for a convolutional layer W_1 with filters of size $k \times k \times 3$, and M filters (reasonable choices are $M = 100$, $k = 5$), we have that $\text{Conv2d}(x^{\text{in}}, W_1) \in \mathbb{R}^{(33-k)\times(33-k)\times M}$. This network can be written as

$$x^{\text{output}} = W_2(\text{MaxPool}(\text{relu}(\text{Conv2d}(x^{\text{input}}, W_1) + b_1))) + b_2$$

where $W_2 \in \mathbb{R}^{10\times M(\lfloor\frac{33-k}{N}\rfloor)^2}$, $b_2 \in \mathbb{R}^{10}$. The hyper-parameters we try to tune in this subsection is batch size, learning rat, hidden size and kernel size, the experiments are plotted below:

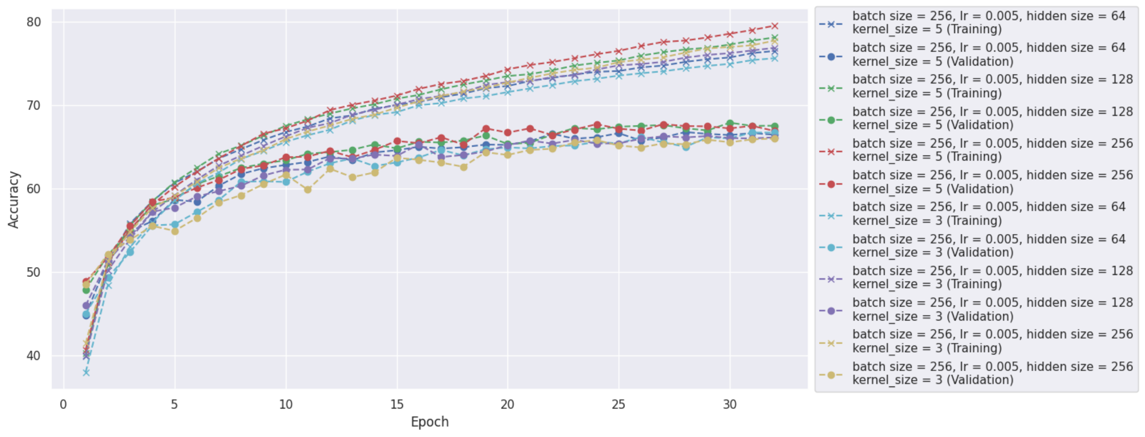

Figure 17: Plot of the training accuracies and validation accuracies versus epoch.

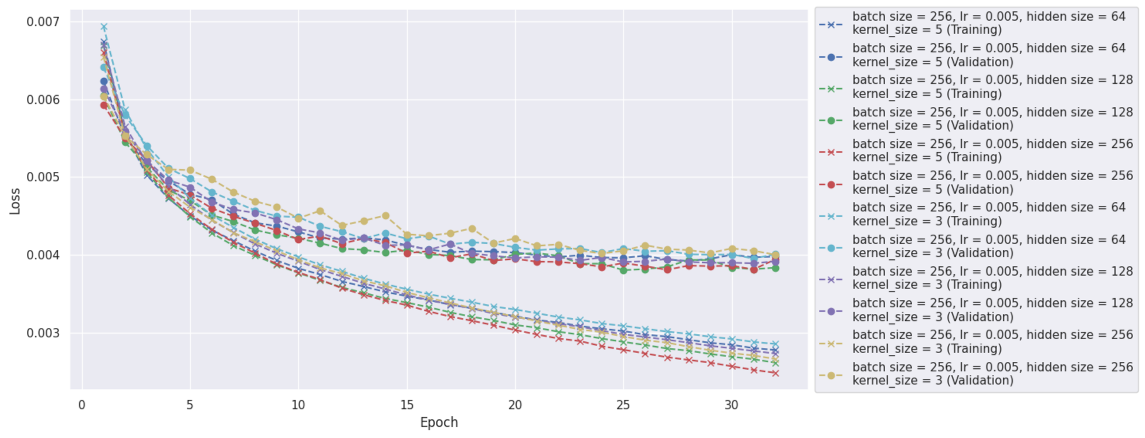

Figure 18: Plot of the training losses and validation losses versus epoch.

The best model from our experiments is the one with batch size of 256, learning rate of 0.005, hidden size of 256 and kernel size of 5 reaching a training accuracies of 79.524% and testing accuracies of 66.170% after 32 epochs.

A.6.d

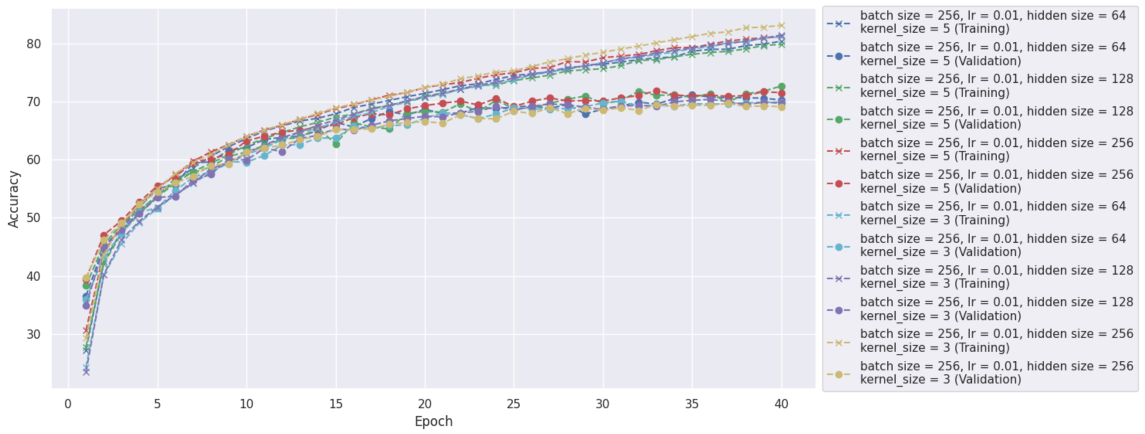

Figure 19: Plot of the training accuracies and validation accuracies versus epoch.

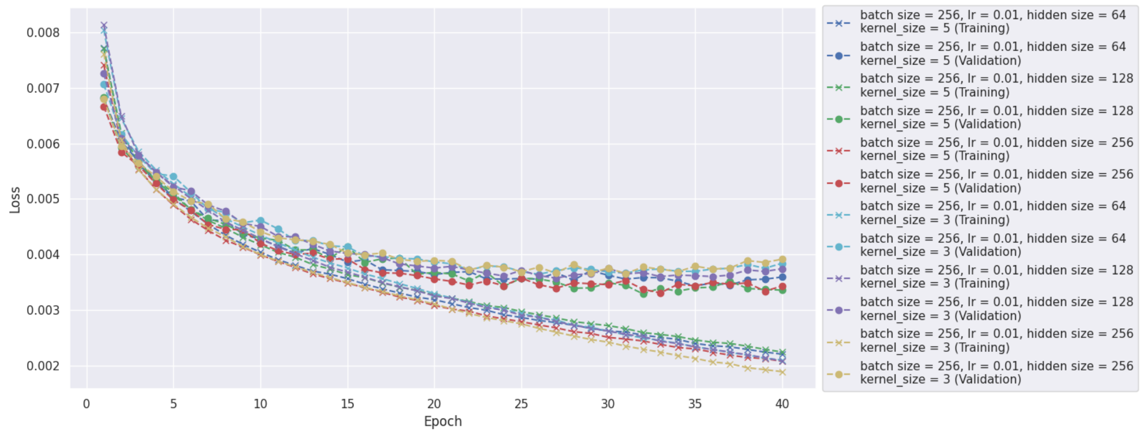

Figure 20: Plot of the training losses and validation losses versus epoch.

The best model from our experiments is the one with batch size of 256, learning rate of 0.01, hidden size of 256 and kernel size of 5 reaching a training accuracies of 79.822% and testing accuracies of 71.220% after 40 epochs.

A.6.model

```
import torch
import torch.nn as nn
import torch.nn.functional as F


class A6a(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(32 * 32 * 3, 10)

    def forward(self, x):
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = self.fc1(x)
        return F.log_softmax(x, dim=1)


class A6b(nn.Module):
    def __init__(self, hidden_size=64):
        super().__init__()
        self.hidden_size = hidden_size
        self.fc1 = nn.Linear(32 * 32 * 3, self.hidden_size)
        self.fc2 = nn.Linear(self.hidden_size, 10)

    def forward(self, x):
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)


class A6c(nn.Module):
    def __init__(self, hidden_size=64, kernel_size=5):
        super().__init__()
        self.hidden_size = hidden_size
        self.kernel_size = kernel_size
        self.conv1 = nn.Conv2d(3, self.hidden_size, self.kernel_size)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(self.hidden_size * ((33 - self.kernel_size)//2) **
            2, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.pool(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)

        return F.log_softmax(x, dim=1)

class A6d(nn.Module):
    def __init__(self, hidden_size=128, kernel_size=5):
        super().__init__()
        self.hidden_size = hidden_size
        self.kernel_size = kernel_size
        self.conv1 = nn.Conv2d(3, self.hidden_size, self.kernel_size)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(self.hidden_size, 16, self.kernel_size)
        self.fc1 = nn.Linear(16 * ((33 - self.kernel_size)//5) ** 2, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1)  # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

A.6.code

```
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
import torchvision.datasets as datasets
import torchvision.transforms as transforms
from tqdm import tqdm

from A6_model import A6a, A6b, A6c, A6d

sns.set()


np.random.seed(1968990 + 20210518)
torch.manual_seed(1968990 + 20210518)


def prepare_dataset(batch_size=64, train_val_split_ratio=0.9):

    transform = transforms.Compose([
        # transforms.Resize(256),
        # transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
            0.225]),
    ])

    cifar10_set = datasets.CIFAR10(root='./data', train=True, download=False,
        transform=transform)
    train_size = int(len(cifar10_set) * train_val_split_ratio)
    val_size = len(cifar10_set) - train_size
    cifar10_trainset, cifar10_valset = torch.utils.data.random_split(cifar10_set
        , [train_size, val_size])
    cifar10_testset = datasets.CIFAR10(root='./data', train=False, download=
        False, transform=transform)

    train_loader = torch.utils.data.DataLoader(cifar10_trainset, batch_size=
        batch_size, shuffle=True)
    val_loader = torch.utils.data.DataLoader(cifar10_valset, batch_size=
        batch_size, shuffle=True)
    test_loader = torch.utils.data.DataLoader(cifar10_testset, batch_size=
        batch_size, shuffle=True)

    return train_loader, val_loader, test_loader


def train(epochs, model, train_loader, val_loader, criterion, optimizer,
        batch_size):
    train_losses = []
    val_losses = []
    train_accs = []
    val_accs =[]
    model.train()
    for epoch in range(epochs):
        running_loss = 0.0
        correct = 0
        total = 0
        for i, data in enumerate(train_loader):
            inputs, labels = data[0].to("cuda"), data[1].to("cuda")
            optimizer.zero_grad()

            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1)
            optimizer.step()
            running_loss += loss.item()

            _, predicted = torch.max(outputs.data, 1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)

        train_accs.append(100 * correct / total)
        train_losses.append(running_loss / total)
        print('[%d] Train Accuracy: %.3f %% Train Loss: %.3f' % (epoch + 1, 100
            * correct / total, running_loss / total))
        running_loss = 0.0
        val_acc, val_loss = eval(epoch, model, val_loader, criterion)
        val_accs.append(val_acc)
        val_losses.append(val_loss)

    return train_accs, val_accs, train_losses, val_losses
```

```
def eval(epoch, model, eval_loader, criterion):
    running_loss = 0.0
    correct = 0
    total = 0
    for i, data in enumerate(eval_loader):
        inputs, labels = data[0].to("cuda"), data[1].to("cuda")

        outputs = model(inputs)
        loss = criterion(outputs, labels)
        running_loss += loss.item()

        _, predicted = torch.max(outputs.data, 1)
        correct += (predicted == labels).sum().item()
        total += labels.size(0)

    print('[%d] Val Accuracy: %.3f %% Val Loss: %.3f' % (epoch + 1, 100 *
        correct / total, running_loss / total))
    return 100 * correct / total, running_loss / total


def plot_acc(train_acc, val_acc, labels, figname):
    epochsx = [int(x) for x in np.arange(1, len(train_acc[0])+1)]
    plt.figure(figsize=(15, 6))
    COLORS = ['b', 'g', 'r', 'c', 'm', 'y']
    for tacc, vacc, label, color in zip(train_acc, val_acc, labels, COLORS):
        plt.plot(epochsx, tacc, '--x', label=label + ' (Training)', color=color)
        plt.plot(epochsx, vacc, '--o', label=label + ' (Validation)', color=
            color)
    plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.tight_layout()
    plt.savefig(figname)


def plot_loss(train_loss, val_loss, labels, figname):
    epochsx = [int(x) for x in np.arange(1, len(train_loss[0])+1)]
    plt.figure(figsize=(15, 6))
    COLORS = ['b', 'g', 'r', 'c', 'm', 'y']
    for tloss, vloss, label, color in zip(train_loss, val_loss, labels, COLORS):
        plt.plot(epochsx, tloss, '--x', label=label + ' (Training)', color=color
            )
        plt.plot(epochsx, vloss, '--o', label=label + ' (Validation)', color=
            color)
    plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.tight_layout()
    plt.savefig(figname)


def run_A6a(epochs=12):

    hyper_params = [
        # {'batch_size': 128, 'lr': 1e-3},
        # {'batch_size': 64, 'lr': 1e-3},
        {'batch_size': 32, 'lr': 1e-3},
        {'batch_size': 128, 'lr': 1e-2},
        {'batch_size': 64, 'lr': 1e-2},
        {'batch_size': 32, 'lr': 1e-2}
    ]

    all_train_accuracies = []
    all_val_accuracies = []
    all_train_losses = []
    all_val_losses = []
    labels = []
    for param in hyper_params:
        batch_size = param['batch_size']
        lr = param['lr']

        train_loader, val_loader, test_loader = prepare_dataset(batch_size=
            batch_size)

        model = A6a()
        criterion = nn.CrossEntropyLoss()
        optimizer = optim.SGD(model.parameters(), lr=lr, momentum=0.9)
        model.to("cuda")

        train_accuracies, val_accuracies, train_losses, val_losses = train(
            epochs, model, train_loader, val_loader, criterion, optimizer,
            batch_size)

        all_train_accuracies.append(train_accuracies)
        all_val_accuracies.append(val_accuracies)
        all_train_losses.append(train_losses)
        all_val_losses.append(val_losses)
        labels.append('batch size = {}, lr = {}'.format(batch_size, lr))
        eval(-1, model, test_loader, criterion)

    plot_acc(all_train_accuracies, all_val_accuracies, labels, 'A6a_acc.png')
    plot_loss(all_train_losses, all_val_losses, labels, 'A6a_loss.png')


def run_A6b(epochs=32):

    hyper_params = [
        {'batch_size': 64, 'lr': 5e-3, 'hidden_size': 64},
        {'batch_size': 64, 'lr': 5e-3, 'hidden_size': 128},
        {'batch_size': 64, 'lr': 5e-3, 'hidden_size': 256},
        {'batch_size': 256, 'lr': 1e-3, 'hidden_size': 64},
        {'batch_size': 256, 'lr': 1e-3, 'hidden_size': 128},
        {'batch_size': 256, 'lr': 1e-3, 'hidden_size': 256},
    ]

    all_train_accuracies = []
    all_val_accuracies = []
    all_train_losses = []
    all_val_losses = []
    labels = []
    for param in hyper_params:
        batch_size = param['batch_size']
        lr = param['lr']
        hidden_size = param['hidden_size']

        train_loader, val_loader, test_loader = prepare_dataset(batch_size=
            batch_size)

        model = A6b(hidden_size=hidden_size)
        criterion = nn.CrossEntropyLoss()
        optimizer = optim.SGD(model.parameters(), lr=lr, momentum=0.9)
        model.to("cuda")

        train_accuracies, val_accuracies, train_losses, val_losses = train(
            epochs, model, train_loader, val_loader, criterion, optimizer,
            batch_size)

        all_train_accuracies.append(train_accuracies)
        all_val_accuracies.append(val_accuracies)
        all_train_losses.append(train_losses)
        all_val_losses.append(val_losses)
        labels.append('batch size = {}, lr = {}, hidden size = {}'.format(
            batch_size, lr, hidden_size))
        eval(-1, model, test_loader, criterion)

    plot_acc(all_train_accuracies, all_val_accuracies, labels, 'A6b_acc.png')
    plot_loss(all_train_losses, all_val_losses, labels, 'A6b_loss.png')


def run_A6c(epochs=32):

    hyper_params = [
        {'batch_size': 256, 'lr': 5e-3, 'hidden_size': 64, 'kernel_size': 5},
        {'batch_size': 256, 'lr': 5e-3, 'hidden_size': 128, 'kernel_size': 5},
        {'batch_size': 256, 'lr': 5e-3, 'hidden_size': 256, 'kernel_size': 5},
        {'batch_size': 256, 'lr': 5e-3, 'hidden_size': 64, 'kernel_size': 3},
        {'batch_size': 256, 'lr': 5e-3, 'hidden_size': 128, 'kernel_size': 3},
        {'batch_size': 256, 'lr': 5e-3, 'hidden_size': 256, 'kernel_size': 3},
    ]
    best_param = [
        {'batch_size': 256, 'lr': 5e-3, 'hidden_size': 256, 'kernel_size': 5}
    ]

    all_train_accuracies = []
    all_val_accuracies = []
    all_train_losses = []
    all_val_losses = []
    labels = []
    for param in hyper_params:
        print(param)
        batch_size = param['batch_size']
        lr = param['lr']
        hidden_size = param['hidden_size']
        kernel_size = param['kernel_size']
        train_loader, val_loader, test_loader = prepare_dataset(batch_size=
            batch_size)

        model = A6c(hidden_size=hidden_size, kernel_size=kernel_size)
        criterion = nn.CrossEntropyLoss()
        optimizer = optim.SGD(model.parameters(), lr=lr, momentum=0.9,
            weight_decay=1e-5)
        model.to("cuda")

        train_accuracies, val_accuracies, train_losses, val_losses = train(
            epochs, model, train_loader, val_loader, criterion, optimizer,
            batch_size)

        all_train_accuracies.append(train_accuracies)
        all_val_accuracies.append(val_accuracies)
        all_train_losses.append(train_losses)
        all_val_losses.append(val_losses)
        labels.append('batch size = {}, lr = {}, hidden size = {}\nkernel_size =
             {}'.format(batch_size, lr, hidden_size, kernel_size))
        eval(-1, model, test_loader, criterion)

    plot_acc(all_train_accuracies, all_val_accuracies, labels, 'A6c_acc.png')
    plot_loss(all_train_losses, all_val_losses, labels, 'A6c_loss.png')


def run_A6d(epochs=40, batch_size=128, lr=1e-2, momentum=0.9, hidden_size=256):
    hyper_params = [
        {'batch_size': 256, 'lr': 1e-2, 'hidden_size': 64, 'kernel_size': 5},
        {'batch_size': 256, 'lr': 1e-2, 'hidden_size': 128, 'kernel_size': 5},
        {'batch_size': 256, 'lr': 1e-2, 'hidden_size': 256, 'kernel_size': 5},
        {'batch_size': 256, 'lr': 1e-2, 'hidden_size': 64, 'kernel_size': 3},
        {'batch_size': 256, 'lr': 1e-2, 'hidden_size': 128, 'kernel_size': 3},
        {'batch_size': 256, 'lr': 1e-2, 'hidden_size': 256, 'kernel_size': 3},
    ]
    best_param = [
        {'batch_size': 256, 'lr': 5e-3, 'hidden_size': 256, 'kernel_size': 5}
    ]

    all_train_accuracies = []
    all_val_accuracies = []
    all_train_losses = []
    all_val_losses = []
    labels = []
    for param in hyper_params:
        print(param)
        batch_size = param['batch_size']
        lr = param['lr']
        hidden_size = param['hidden_size']
        kernel_size = param['kernel_size']

        train_loader, val_loader, test_loader = prepare_dataset(batch_size=
            batch_size)

        model = A6d(hidden_size=hidden_size, kernel_size=kernel_size)
        criterion = nn.CrossEntropyLoss()
        optimizer = optim.SGD(model.parameters(), lr=lr, momentum=0.9,
            weight_decay=1e-5)
        model.to("cuda")

        train_accuracies, val_accuracies, train_losses, val_losses = train(
            epochs, model, train_loader, val_loader,
                                                                                    criterion
                                                                                        ,
                                                                                        optimizer
                                                                                        ,
                                                                                        batch_size
                                                                                        )

        all_train_accuracies.append(train_accuracies)
        all_val_accuracies.append(val_accuracies)
        all_train_losses.append(train_losses)
        all_val_losses.append(val_losses)
        labels.append('batch size = {}, lr = {}, hidden size = {}\nkernel_size =
             {}'.format(batch_size, lr, hidden_size,




        eval(-1, model, test_loader, criterion)

    plot_acc(all_train_accuracies, all_val_accuracies, labels, 'A6d_acc.png')
    plot_loss(all_train_losses, all_val_losses, labels, 'A6d_loss.png')



def main():
    # run_A6a()
    # run_A6b()
    run_A6c()
    # run_A6d()

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

B1

B.1.a

$$\begin{aligned}
\mathbb{P}\left[\hat{R}_n(f)=0\right] &= \mathbb{P}\left[\frac{1}{n}\sum_{i=1}^{n}\mathbf{1}(f(x_i)\neq y_i)=0\right] && (6)\\
&= \prod_{i=1}^{n}\mathbb{P}[]f(x_i)=y_i] && (7)\\
&= \prod_{i=1}^{n}1-\mathbb{P}[f(x_i)\neq y_i] && (8)\\
&= \left(1-\mathbb{P}[f(x_k)\neq y_k]\right)^n, k\in[1,\dots,n] && (9)\\
&= \left(1-\mathbb{E}_{X,Y}[\mathbf{1}(f(X)\neq Y)]\right)^n && (10)\\
&= (1-R(f))^n \le (1-\epsilon)^n \le e^{-n\epsilon} && (11)
\end{aligned}$$

B.1.b

For every $f\in\mathcal{F}$, define $A_f=\{R(f)>\epsilon \text{ and } \hat{R}_n(f)=0\}$, then we can use the union bound inequality:

$$\begin{aligned}
\mathbb{P}\left[\exists f\in\mathcal{F}: R(f)>\epsilon \text{ and } \hat{R}_n(f)=0\right] &\le \mathbb{P}\left[A_1\cup\ldots\cup A_k\right], A_k\in A_f\ \forall\ k\in[1,\dots,n] && (12)\\
&\le \sum_{f\in\mathcal{F}}\mathbb{P}(A_f) && (13)\\
&\le |\mathcal{F}|e^{-\epsilon n} && (14)
\end{aligned}$$

B.1.c

$$\begin{aligned}
&|\mathcal{F}|e^{-\epsilon n}\le\delta && (15)\\
&\Rightarrow e^{-\epsilon n}\le\frac{\delta}{|\mathcal{F}|} && (16)\\
&\Rightarrow -\epsilon n\le log\frac{\delta}{|\mathcal{F}|} && (17)\\
&\Rightarrow \epsilon\ge\frac{1}{n}\log\frac{|\mathcal{F}|}{\delta} && (18)\\
&\Rightarrow \epsilon^*=\frac{1}{n}\log\frac{|\mathcal{F}|}{\delta} && (19)
\end{aligned}$$

B.1.d

$$\begin{aligned}
\mathbb{P}(\hat{R}_n(f)=0\Rightarrow R(\widehat{f})-R(f^*)\le\frac{1}{n}\log\frac{|\mathcal{F}|}{\delta}) &= \mathbb{P}(\hat{R}_n(f)=0 \text{ and } R(\widehat{f})-R(f^*)>\frac{1}{n}\log\frac{|\mathcal{F}|}{\delta}) && (20)\\
&= \mathbb{P}(\hat{R}_n(f)=0 \text{ and } R(f)-R(f^*)>\epsilon^*) && (21)\\
&\ge 1-\mathbb{P}\left[\hat{R}_n(f)=0 \text{ and } R(f)>\epsilon^*\right] && (22)\\
&\ge 1-\mathbb{P}\left[\exists f\in\mathcal{F}: R(f)>\epsilon^* \text{ and } \hat{R}_n(f)=0\right] && (23)\\
&\ge 1-|\mathcal{F}|e^{-\epsilon^* n} && (24)\\
&\ge 1-\delta && (25)
\end{aligned}$$

B2

B.2.a

Given the current iterate $\widehat{\mathbf{w}} = \mathbf{w}_k$, the expression of next iterate $\mathbf{w}_{k+1}$ will be:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta \left(\frac{1}{n} \sum_{i=1}^{n} \delta_p(l(x_i, y_i), w) \right), \tag{26}$$

where the step function $\delta_p(\cdot)$ is defined as:

$$\delta_p(l(x_i, y_i), w) = \begin{cases} -y_i x_i, & \text{if } -y_i w^T x_i \geq 0 \\ 0, & \text{if } -y_i w^T x_i < 0 \end{cases}. \tag{27}$$

B.2.b

If we choose $\eta = 1$ and run the Perceptron algorithm with batch size of 1, we will have the update rule expressed as:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta \cdot \delta_p(l(x_i, y_i), w). \tag{28}$$

When we incorrectly predict the label, we will be adding $y_i x_i$ term to the original weights. On the other hand, we will do nothing to the weights when we correctly predict the label. Therefore, by choosing η as 1 and using a batch size of 1, the Perceptron algorithm can be simply viewed as SGD.

B.2.c

A simple example to illustrate why hinge loss is generally preferred over the loss given in the problem statement will be that since the loss given above will do nothing to the weight if we correctly predict the label even though the prediction is only slightly above the linear decision boundaries. In this case, the hinge loss can still be able to update the weights when the prediction is correct causing the overall model to be more robust by increasing the margin of the decision boundaries.