Homework #3

Hyungseok Choi
Student No. : 2250166

June 30, 2022

Collaborated with Yann Hyunh, Kevin Tong.

Conceptual Questions

A1. The answers to these questions should be answerable without referring to external materials. Briefly justify your answers with a few words.

a. Decrease σ. σ plays a role in setting the range of affecting each other. Which means, with larger σ, each point affects others leads to underfitting.

b. True. Since loss functions are not convex, it might reach the local minima.

c. False. If we initialize all weights to zero, the neural network boils down to just having a single hidden unit. By making all $z_i, W_i^{(}l), andb_i^{(}l)$ same.

d. True. If we only use a linear activation function, no matter how many layers neural networks had, it would behave just like a single-layer perceptron.

e. False. With using chain rule, both requires $O(L)$ time complexity.

f. False. As mentioned in class, Neural Networks are not always the best choice for any circumstance.

Support Vector Machines

A2.

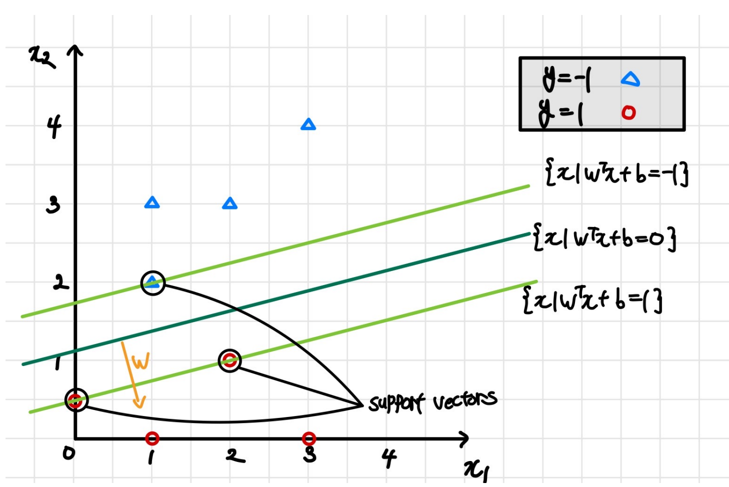


Figure 1: A2-a

a.

b.

$$0w_1 + 0.5w_2 + b = 1 \tag{1}$$
$$2w_1 + 1w_2 + b = 1 \tag{2}$$
$$1w_1 + 2w_2 + b = -1 \tag{3}$$

$$\text{by (2)-2(3)}, \; -3w_2 - b = 3 \tag{4}$$
$$\text{by (1)+(4)}, \; , -\frac{5}{2}w_2 = 4 \tag{5}$$

$$\therefore w_1 = \frac{2}{5}, w_2 = -\frac{8}{5}, b = \frac{9}{5}$$

c. Let the point x_0 on the hyperplane $\{x|w^T x + b = -1\}$ and the point x_1 on the hyperplane $\{x|w^T x + b = 1\}$ Then, the distance between two hyperplane is the same as the size of the projection vector $\overrightarrow{x_0 x_1}$ onto unit vector w

$\therefore distance = |(\vec{x_1} - \vec{x_0}) \cdot \frac{w}{|w|}| = |\frac{(w \cdot \vec{x_1} - w \cdot \vec{x_0})}{||w||}| = |\frac{((1-b)-(b-1)}{||w||}| = |\frac{2}{||w||}| = \frac{2}{||w||}$

Kernels and Bootstrap

A3.

$$\begin{aligned}
K(x, x') \quad & = \exp -\tfrac{(x-x')^2}{2} \\
& = \exp -\tfrac{(x-x')\cdot(x-x')}{2} \\
& = \exp -\tfrac{x\cdot(x-x') - x'\cdot(x-x')}{2} \\
& = \exp -\tfrac{x\cdot x - x\cdot x' - x'\cdot x - x'\cdot x'}{2} \\
& = \exp -\tfrac{||x||_2^2 + ||x'||_2^2 - 2x\cdot x'}{2} \\
& = \exp -\tfrac{||x||_2^2 + ||x'||_2^2}{2} \exp x \cdot x' \\
& = \exp -\tfrac{||x||_2^2 + ||x'||_2^2}{2} \textstyle\sum_{n=0}^{\infty} \tfrac{(x\cdot x')^n}{n!} \\
& = \phi(x) \cdot \phi(x')
\end{aligned}$$

A4.

a. (a) Poly kernal - lambda: 2.53536e-05, d: 19

(b) RBF kernal - lambda: 1e-05 , gamma: 13.04124

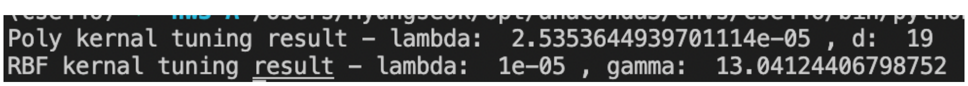


Figure 2: A4-a result

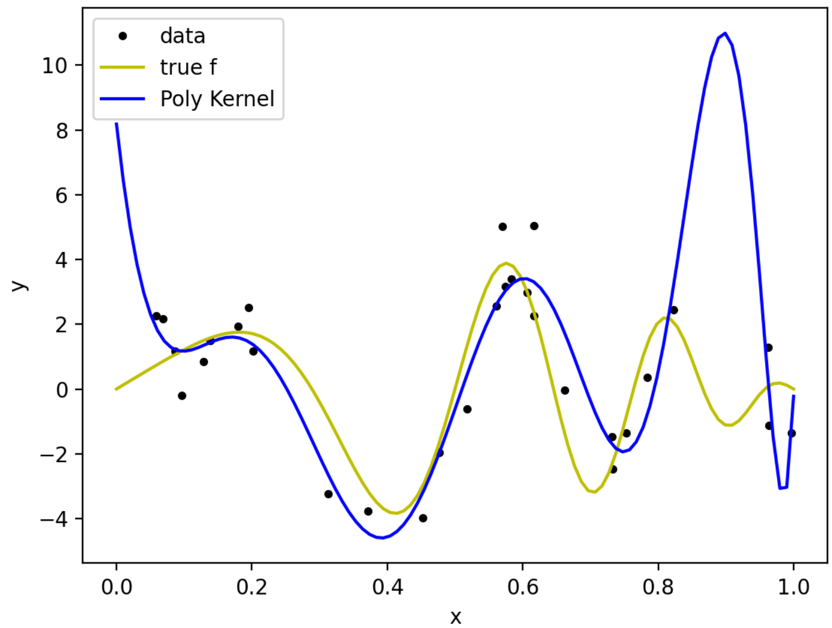


Figure 3: Poly Kernel plot

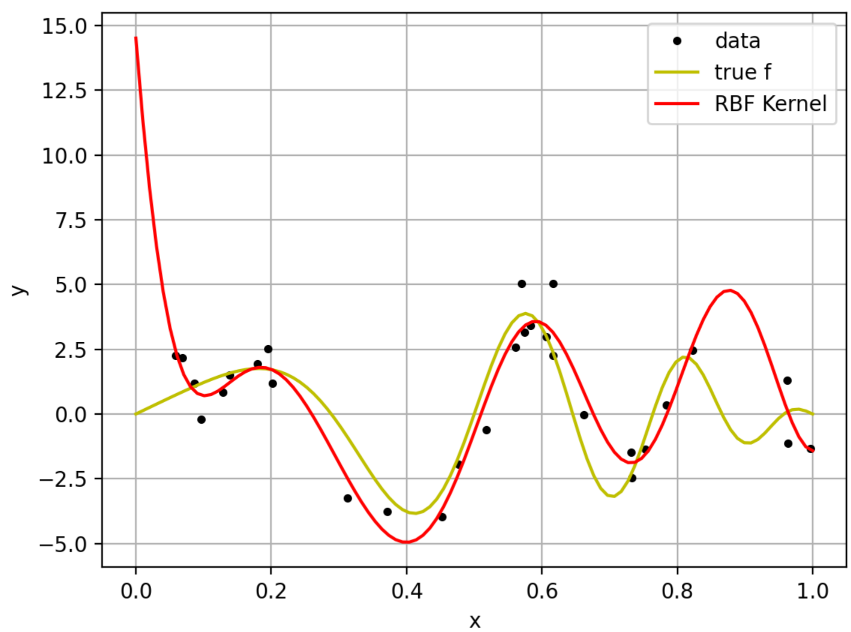


Figure 4: RBF Kernel plot

b.

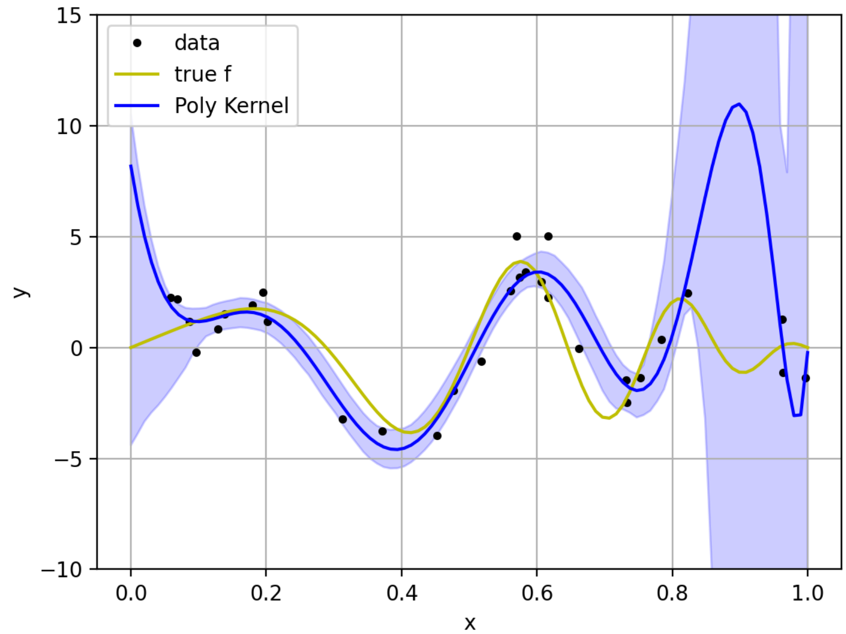


Figure 5: Poly Kernel bootstrap

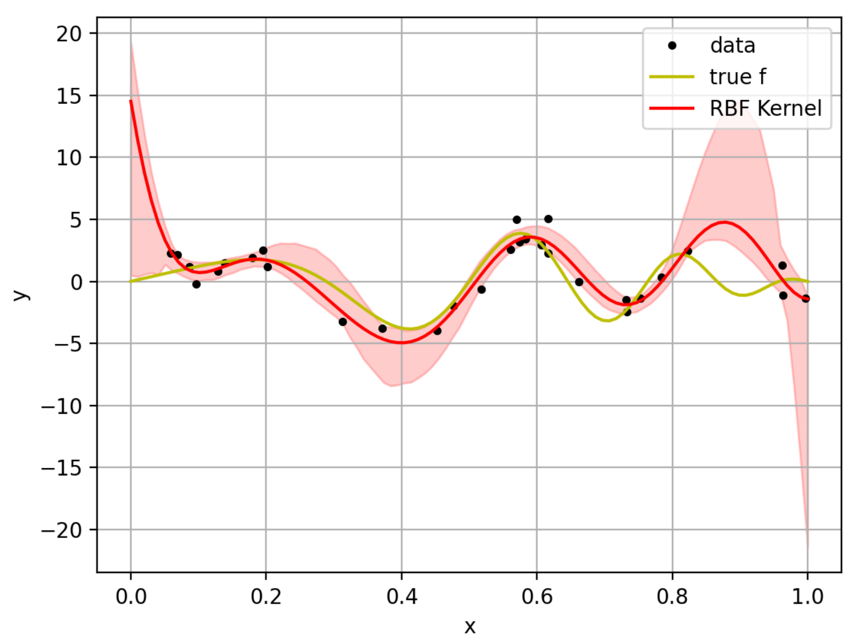


Figure 6: RBF Kernel bootstrap

c.

d. (a) Poly kernal - lambda: 1e-05, d: 25

(b) RBF kernal - lambda: 1e-05 , gamma: 15.0558

```
Poly kernal tuning result - lambda:  1e-05 , d:  25
RBF kernal tuning result - lambda:  1e-05 , gamma:  15.055886219769825
```

Figure 7: A4-d result

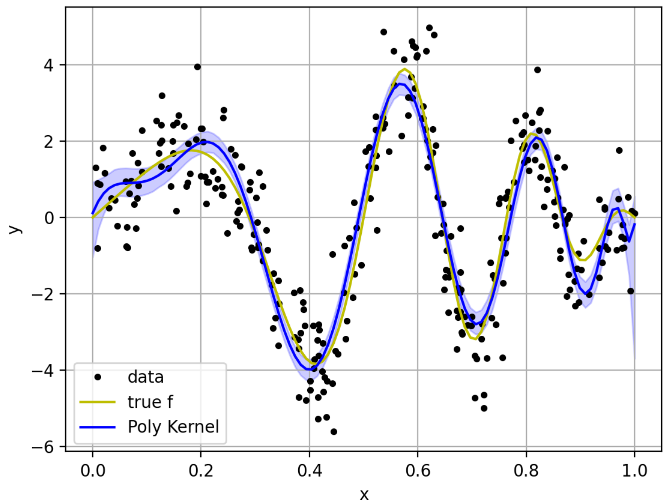


Figure 8: Poly Kernel plot

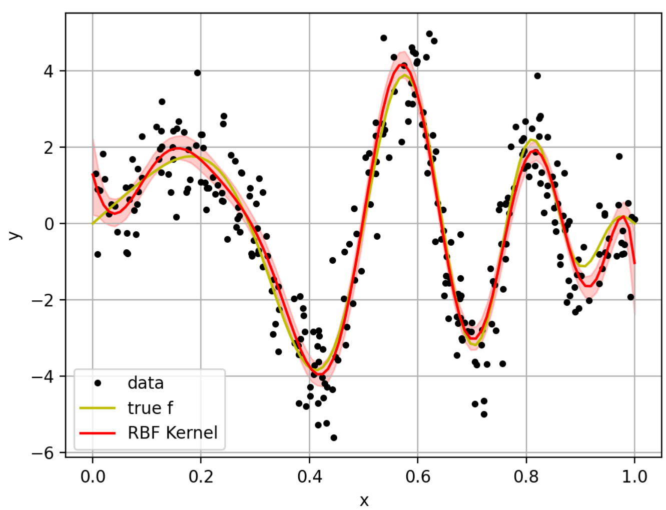


Figure 9: RBF Kernel plot

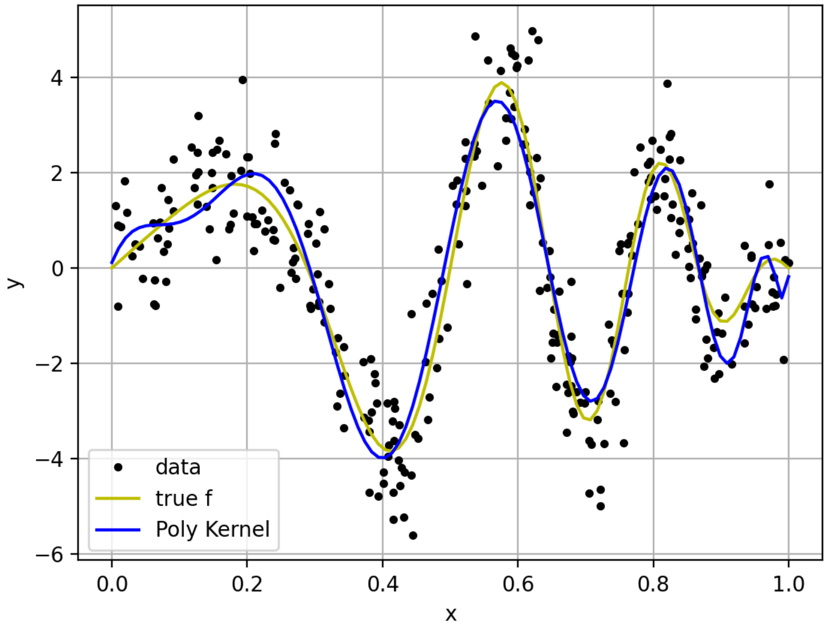


Figure 10: Poly Kernel bootstrap

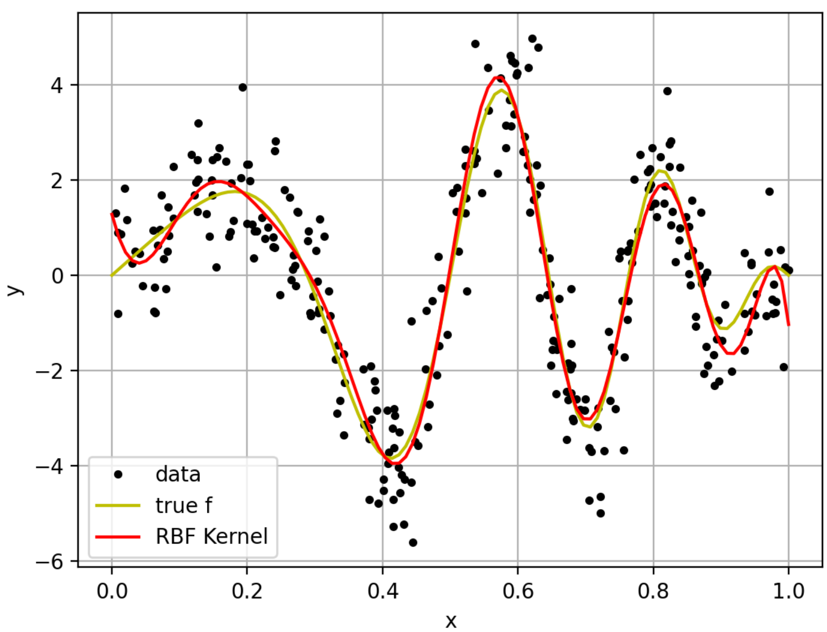


Figure 11: RBF Kernel bootstrap

e. 5% Percentile: 0.04266
95% Percentile: 0.09897
Since both values, especially 5% percentile value is positive, we know that there is statistically significant evidence to suggest that the RBF kernel is superior in prediction to the Poly kernel.

```
kernel_bootstrap/main.py
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)


@problem.tag("hw3-A")
def poly_kernel(x_i: np.ndarray, x_j: np.ndarray, d: int) -> np.ndarray:

    return (np.multiply.outer(x_i, x_j)+1)**d



@problem.tag("hw3-A")
def rbf_kernel(x_i: np.ndarray, x_j: np.ndarray, gamma: float) -> np.ndarray:

    return np.exp(-gamma*((np.subtract.outer(x_i, x_j)**2)))


@problem.tag("hw3-A")
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:

    K = kernel_function(x, x, kernel_param)

    return np.linalg.solve(K+_lambda*np.eye(K.shape[0]), y)


@problem.tag("hw3-A", start_line=1)
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:
```

```python
    fold_size = len(x) // num_folds
    errors = []

    for i in range(num_folds):
        current_start, current_end = i * fold_size, (i+1)*fold_size

        x_train, y_train = np.append(x[:current_start], x[current_end:]), np.append(y[:curre
        x_test, y_test = x[current_start: current_end], y[current_start: current_end]

        alpha = train(x_train, y_train, kernel_function, kernel_param, _lambda)
        K = kernel_function(x_train, x_test, kernel_param)

        predict = alpha@K
        errors.append(np.mean((predict-y_test)**2))

    return np.mean(errors)

@problem.tag("hw3-A")
def rbf_param_search(
    x: np.ndarray, y: np.ndarray, num_folds: int
) -> Tuple[float, float]:

    n = len(x)
    dists = []

    for i in range(n):
        for j in range(i+1, n):
            dists.append((x[i]-x[j])**2)

    gamma = 1 / np.median(dists)
    gamma_candidate = np.random.normal(gamma, 1, 50)

    min_errors = float("inf")
    min_lambda = None
    min_gamma = None

    lambda_candidate = 10 ** np.linspace(-5, -1, num=100)

    for l in lambda_candidate:
        for g in gamma_candidate:
            e = cross_validation(x, y, rbf_kernel, g, l, num_folds)
            if e < min_errors:
                min_gamma = g
                min_errors = e
                min_lambda = l

    return (min_lambda, min_gamma)


@problem.tag("hw3-A")
def poly_param_search(
    x: np.ndarray, y: np.ndarray, num_folds: int
) -> Tuple[float, int]:
```

```
    n = len(x)

    min_errors = float("inf")
    min_lambda = None
    min_d = 0

    lambda_candidate = 10 ** np.linspace(-5, -1, num=100)
    poly_candidate = np.arange(5, 26)

    for l in lambda_candidate:
        for d in poly_candidate:
            e = cross_validation(x, y, poly_kernel, d, l, num_folds)
            if e < min_errors:
                min_d = d
                min_errors = e
                min_lambda = l

    return (min_lambda, min_d)


@problem.tag("hw3-A", start_line=1)
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)
    result = None

    for i in range(bootstrap_iters):
        idices = np.random.choice(len(x), len(x))
        x_iter = np.array([x[i] for i in idices])
        y_iter = np.array([y[i] for i in idices])

        # train, predict
        alpha = train(x_iter, y_iter, kernel_function, kernel_param, _lambda)
        K = kernel_function(x_iter, x_fine_grid, kernel_param)
        predict = (alpha@K).reshape((1, -1))

        if result is None:
            result = predict
        else:
            result = np.append(result, predict, axis=0)

    return np.percentile(result, [5, 95], axis=0)


@problem.tag("hw3-A", start_line=1)
def main():

    run_for = "d"
```

```python
(x_30, y_30), (x_300, y_300), (x_1000, y_1000) = load_dataset("kernel_bootstrap")

if run_for in ["abc", "a", "b", "c"]:
    (poly_opt_lambda, poly_opt_dim) = poly_param_search(x_30, y_30, len(x_30))
    print("Poly kernal tuning result -- lambda: ", poly_opt_lambda ,", d: ", poly_opt_dim
    (RBF_opt_lambda, RBF_opt_gamma) = rbf_param_search(x_30, y_30, len(x_30))
    print("RBF kernal tuning result -- lambda: ", RBF_opt_lambda, ", gamma: ", RBF_opt_ga

if run_for in ["abc", "b", "c"]:
    x = np.linspace(0, 1, 100)
    true_y = f_true(x)

    poly_alpha = train(x_30, y_30, poly_kernel, poly_opt_dim, poly_opt_lambda)
    poly_K = poly_kernel(x_30, x, poly_opt_dim)
    poly_y = poly_alpha@poly_K

    RBF_alpha = train(x_30, y_30, rbf_kernel, RBF_opt_gamma, RBF_opt_lambda)
    RBF_K = rbf_kernel(x_30, x, RBF_opt_gamma)
    RBF_y = RBF_alpha@RBF_K

    if run_for in ["abc", "b"]:
        plt.plot(x_30, y_30, "ko", label="data", markersize=3)
        plt.plot(x, true_y, "y--", label="true f")
        plt.plot(x, poly_y, "b--", label="Poly Kernel")
        plt.xlabel("x")
        plt.ylabel("y")
        plt.legend()
        plt.show()

        plt.plot(x_30, y_30, "ko", label="data", markersize=3)
        plt.plot(x, true_y, "y--", label="true f")
        plt.plot(x, RBF_y, "r--", label="RBF Kernel")
        plt.xlabel("x")
        plt.ylabel("y")
        plt.grid()
        plt.legend()
        plt.show()

if run_for in ["abc", "c"]:
    poly_boot = bootstrap(x_30, y_30, poly_kernel, poly_opt_dim, poly_opt_lambda, 300)
    RBF_boot = bootstrap(x_30, y_30, rbf_kernel, RBF_opt_gamma, RBF_opt_lambda, 300)

    plt.plot(x_30, y_30, "ko", label="data", markersize=3)
    plt.plot(x, true_y, "y--", label="true f")
    plt.plot(x, poly_y, "b--", label="Poly Kernel")
    plt.fill_between(x, poly_boot[0], poly_boot[1], color = "b", alpha=0.2)
    plt.xlabel("x")
    plt.ylabel("y")
    plt.ylim((-10, 15))
    plt.grid()
    plt.legend()
    plt.show()

    plt.plot(x_30, y_30, "ko", label="data", markersize=3)
```

```
    plt.plot(x, true_y, "y-", label="true f")
    plt.plot(x, RBF_y, "r-", label="RBF Kernel")
    plt.fill_between(x, RBF_boot[0], RBF_boot[1], color = "r", alpha=0.2)
    plt.xlabel("x")
    plt.ylabel("y")
    plt.grid()
    plt.legend()
    plt.show()

if run_for in ["d", "e"]:
    (poly_opt_lambda, poly_opt_dim) = poly_param_search(x_300, y_300, 10)
    print("Poly kernal tuning result - lambda: ", poly_opt_lambda ,", d: ", poly_opt_dim
    (RBF_opt_lambda, RBF_opt_gamma) = rbf_param_search(x_300, y_300, 10)
    print("RBF kernal tuning result - lambda: ", RBF_opt_lambda, ", gamma: ", RBF_opt_gam

    if run_for == "d":
        poly_alpha = train(x_300, y_300, poly_kernel, poly_opt_dim, poly_opt_lambda)
        RBF_alpha = train(x_300, y_300, rbf_kernel, RBF_opt_gamma, RBF_opt_lambda)

        x = np.linspace(0, 1, 100)
        true_y = f_true(x)

        poly_K = poly_kernel(x_300, x, poly_opt_dim)
        poly_y = poly_alpha@poly_K

        RBF_K = rbf_kernel(x_300, x, RBF_opt_gamma)
        RBF_y = RBF_alpha@RBF_K

        poly_boot = bootstrap(x_300, y_300, poly_kernel, poly_opt_dim, poly_opt_lambda,
        RBF_boot = bootstrap(x_300, y_300, rbf_kernel, RBF_opt_gamma, RBF_opt_lambda, 3

        plt.plot(x_300, y_300, "ko", label="data", markersize=3)
        plt.plot(x, true_y, "y-", label="true f")
        plt.plot(x, poly_y, "b-", label="Poly Kernel")
        plt.xlabel("x")
        plt.ylabel("y")
        plt.grid()
        plt.legend()
        plt.show()

        plt.plot(x_300, y_300, "ko", label="data", markersize=3)
        plt.plot(x, true_y, "y-", label="true f")
        plt.plot(x, RBF_y, "r-", label="RBF Kernel")
        plt.xlabel("x")
        plt.ylabel("y")
        plt.grid()
        plt.legend()
        plt.show()

        plt.plot(x_300, y_300, "ko", label="data", markersize=3)
        plt.plot(x, true_y, "y-", label="true f")
        plt.plot(x, poly_y, "b-", label="Poly Kernel")
        plt.fill_between(x, poly_boot[0], poly_boot[1], color = "b", alpha=0.2)
        plt.xlabel("x")
        plt.ylabel("y")
```

```
            plt.grid()
            plt.legend()
            plt.show()

            plt.plot(x_300, y_300, "ko", label="data", markersize=3)
            plt.plot(x, true_y, "y-", label="true f")
            plt.plot(x, RBF_y, "r-", label="RBF Kernel")
            plt.fill_between(x, RBF_boot[0], RBF_boot[1], color = "r", alpha=0.2)
            plt.xlabel("x")
            plt.ylabel("y")
            plt.grid()
            plt.legend()
            plt.show()

    if run_for == "e":
        result = []

        for j in range(300):
            idices = np.random.choice(len(x_1000), 1000)
            x_iter = np.array([x_1000[i] for i in idices])
            y_iter = np.array([y_1000[i] for i in idices])

            poly_alpha = train(x_iter, y_iter, poly_kernel, poly_opt_dim, poly_opt_lambda)
            RBF_alpha = train(x_iter, y_iter, rbf_kernel, RBF_opt_gamma, RBF_opt_lambda)

            poly_K = poly_kernel(x_300, x_iter, poly_opt_dim)
            RBF_K = rbf_kernel(x_300, x_iter, RBF_opt_gamma)

            poly_predict = (poly_alpha@poly_K).reshape((1,-1))
            RBF_predict = (RBF_alpha@RBF_K).reshape((1,-1))

            result.append(np.mean((poly_predict-y_iter)**2 - (RBF_predict-y_iter)**2))

        ci5, ci95 = np.percentile(np.array(result), [5, 95])

        print(ci5, ci95)

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

Introduction to PyTorch

A5.

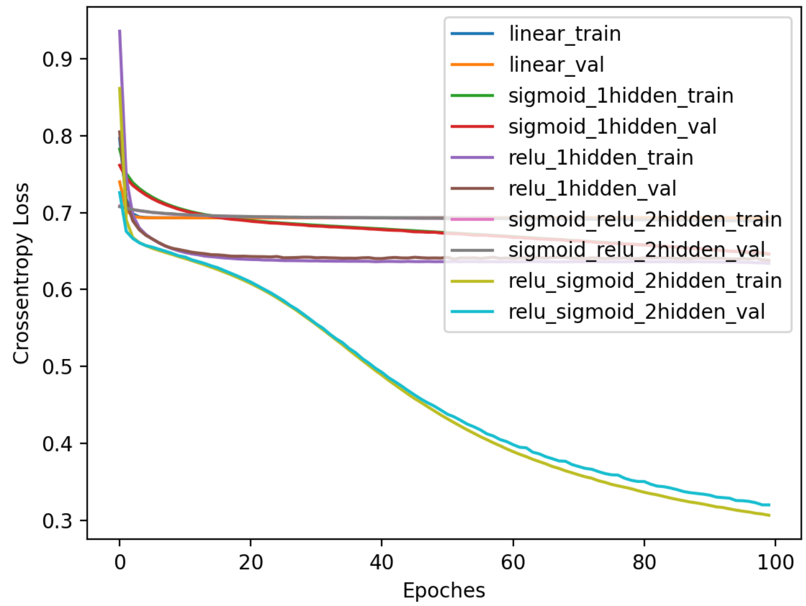


Figure 12: Cross Entropy Losses (learning rate = 0.005)

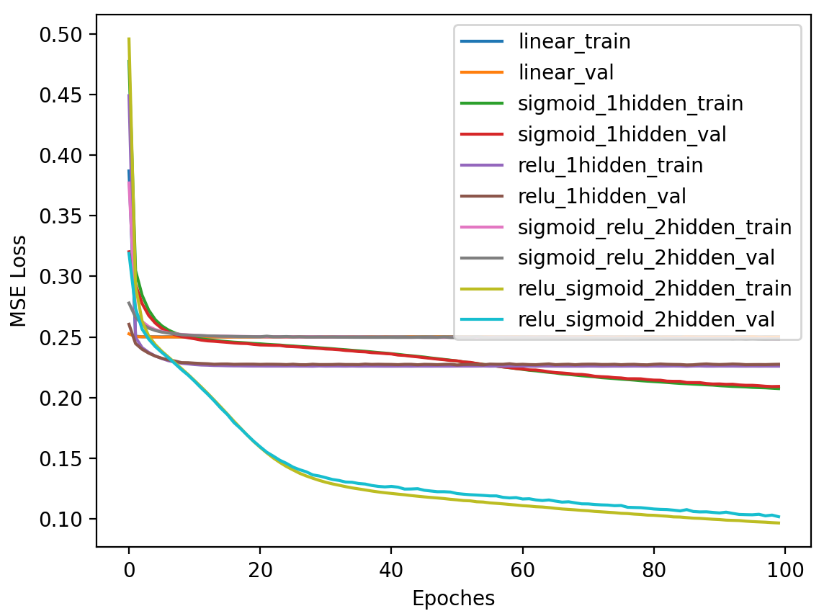


Figure 13: MSE Losses (learning rate = 0.009)

b.

c. (a) Cross Entropy Loss - relu sigmoid 2hidden (0.9078)

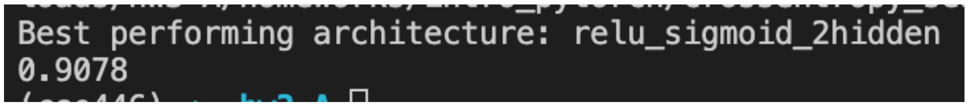


Figure 14: Cross Entropy Best Model and Accuracy

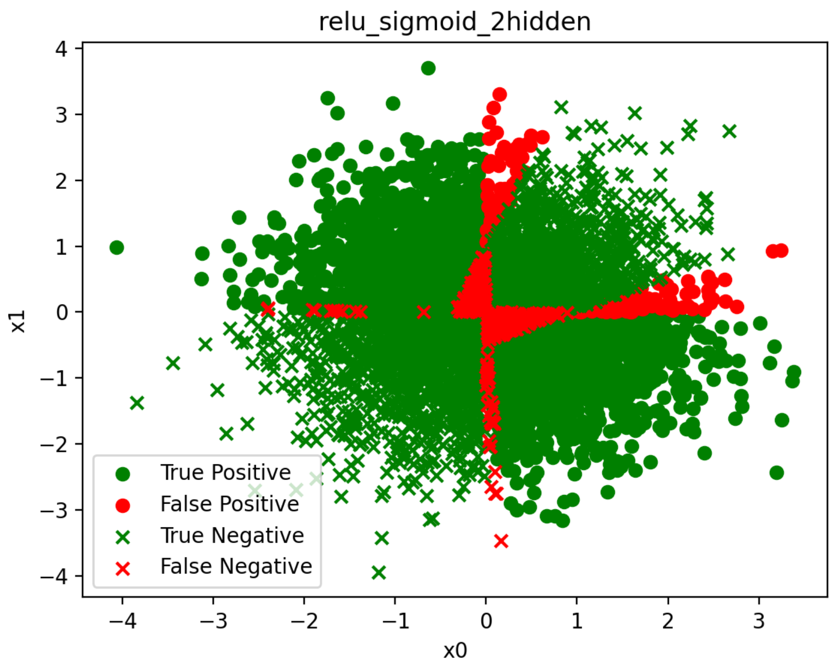
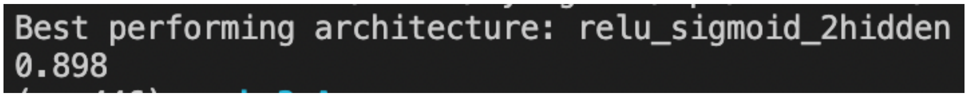


Figure 15: relu sigmoid 2hidden Model Prediction

(b) MSE Loss - relu sigmoid 2hidden (0.6562)



Best performing architecture: relu_sigmoid_2hidden
0.898

Figure 16: MSE Best Model and Accuracy

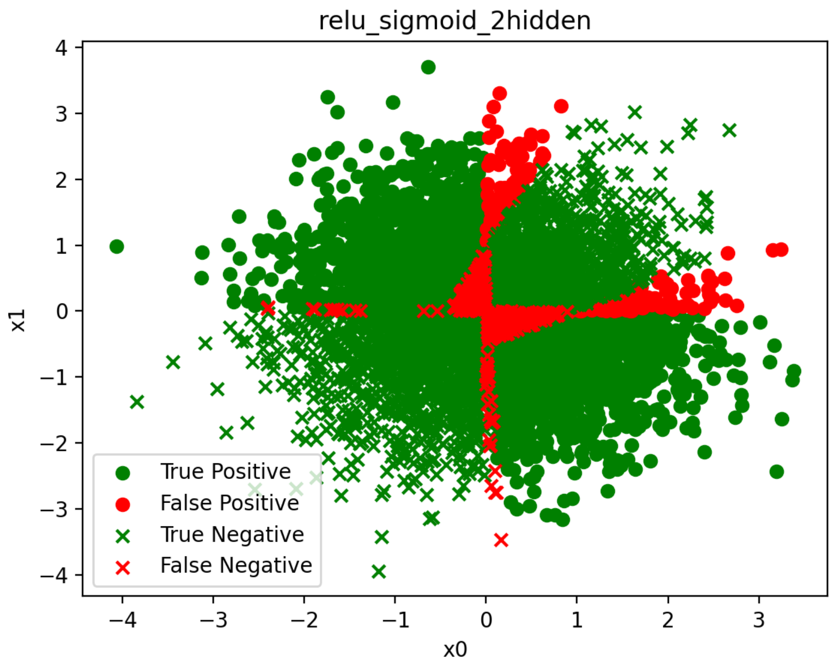


Figure 17: relu sigmoid 2hidden Model Prediction

intro_pytorch/crossentropy_search.py

```
if __name__ == "__main__":
    from layers import LinearLayer, ReLULayer, SigmoidLayer, SoftmaxLayer
    from losses import CrossEntropyLossLayer
    from optimizers import SGDOptimizer
    from train import plot_model_guesses, train
else:
    from .layers import LinearLayer, ReLULayer, SigmoidLayer, SoftmaxLayer
    from .optimizers import SGDOptimizer
    from .losses import CrossEntropyLossLayer
    from .train import plot_model_guesses, train

from typing import Any, Dict

import numpy as np
import torch
from matplotlib import pyplot as plt
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

from utils import load_dataset, problem

RNG = torch.Generator()
RNG.manual_seed(446)


@problem.tag("hw3-A")
def crossentropy_parameter_search(
    dataset_train: TensorDataset, dataset_val: TensorDataset
) -> Dict[str, Any]:

    train_loader = DataLoader(dataset_train, batch_size=32, shuffle=True)
    val_loader = DataLoader(dataset_val, batch_size=32, shuffle=True)
    learning_rate = 0.005

    ret = dict()
    criterion = CrossEntropyLossLayer()

    linearlayer = nn.Sequential(
        LinearLayer(2, 2, generator=RNG),
        SoftmaxLayer()
    )
    optimizer = SGDOptimizer(params=linearlayer.parameters(), lr=learning_rate)
    train_result = train(train_loader, linearlayer, criterion, optimizer, val_loader, 100)
    ret['linear'] = { "train": train_result["train"], "val": train_result["val"], "model": l

    sigmoid_1hiddenlayer = nn.Sequential(
        LinearLayer(2,2, generator=RNG),
        SigmoidLayer(),
        LinearLayer(2,2, generator=RNG),
        SoftmaxLayer()
    )
    optimizer = SGDOptimizer(params=sigmoid_1hiddenlayer.parameters(), lr=learning_rate)
    train_result = train(train_loader, sigmoid_1hiddenlayer, criterion, optimizer, val_loade
```

```
    ret['sigmoid_1hidden'] = { "train": train_result["train"], "val": train_result["val"], "

    relu_1hiddenlayer = nn.Sequential(
        LinearLayer(2,2, generator=RNG),
        ReLULayer(),
        LinearLayer(2,2, generator=RNG),
        SoftmaxLayer()
    )
    optimizer = SGDOptimizer(params=relu_1hiddenlayer.parameters(), lr=learning_rate)
    train_result = train(train_loader, relu_1hiddenlayer, criterion, optimizer, val_loader,
    ret['relu_1hidden'] = { "train": train_result["train"], "val": train_result["val"], "mo

    sigmoid_relu_2hiddenlayer = nn.Sequential(
        LinearLayer(2,2, generator=RNG),
        SigmoidLayer(),
        LinearLayer(2,2, generator=RNG),
        ReLULayer(),
        LinearLayer(2,2, generator=RNG),
        SoftmaxLayer()
    )
    optimizer = SGDOptimizer(params=sigmoid_relu_2hiddenlayer.parameters(), lr=learning_rate
    train_result = train(train_loader, sigmoid_relu_2hiddenlayer, criterion, optimizer, val_
    ret['sigmoid_relu_2hidden'] = { "train": train_result["train"], "val": train_result["val

    relu_sigmoid_2hiddenlayer = nn.Sequential(
        LinearLayer(2,2, generator=RNG),
        ReLULayer(),
        LinearLayer(2,2, generator=RNG),
        SigmoidLayer(),
        LinearLayer(2,2, generator=RNG),
        SoftmaxLayer()
    )
    optimizer = SGDOptimizer(params=relu_sigmoid_2hiddenlayer.parameters(), lr=learning_rate
    train_result = train(train_loader, relu_sigmoid_2hiddenlayer, criterion, optimizer, val_
    ret['relu_sigmoid_2hidden'] = { "train": train_result["train"], "val": train_result["val

    return ret

@problem.tag("hw3-A")
def accuracy_score(model, dataloader) -> float:
    correct = 0
    total = 0
    with torch.no_grad():
        for data in dataloader:
            obs, target = data
            outputs = model(obs)
            _, predicted = torch.max(outputs.data, 1)
            total += target.size(0)
            correct += (predicted == target).sum().item()
    return correct / total


@problem.tag("hw3-A", start_line=7)
def main():
    (x, y), (x_val, y_val), (x_test, y_test) = load_dataset("xor")
```

```
    dataset_train = TensorDataset(torch.from_numpy(x), torch.from_numpy(y))
    dataset_val = TensorDataset(torch.from_numpy(x_val), torch.from_numpy(y_val))
    dataset_test = TensorDataset(torch.from_numpy(x_test), torch.from_numpy(y_test))

    ce_configs = crossentropy_parameter_search(dataset_train, dataset_val)

    min_loss = float("inf")
    min_model_name = None
    min_model = None

    for i in ce_configs.items():
        x = range(100)
        train = i[1]['train']
        val = i[1]['val']
        model_name = i[0]
        model = i[1]["model"]
        plt.plot(x , train, label = model_name + "_train")
        plt.plot(x , val, label = model_name + "_val")
        m_loss = min(val)
        if m_loss < min_loss:
            min_loss = m_loss
            min_model_name = model_name
            min_model = model


    plt.ylabel("Crossentropy Loss")
    plt.xlabel("Epoches")
    plt.legend()
    plt.show()

    print("Best performing architecture: " + min_model_name)

    plot_model_guesses(DataLoader(dataset_test), min_model, min_model_name)
    ac = accuracy_score(min_model, DataLoader(dataset_test))
    print(ac)

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

intro_pytorch/mean_squared_error_search.py

```
if __name__ == "__main__":
    from layers import LinearLayer, ReLULayer, SigmoidLayer
    from losses import MSELossLayer
    from optimizers import SGDOptimizer
    from train import plot_model_guesses, train
else:
    from .layers import LinearLayer, ReLULayer, SigmoidLayer
    from .optimizers import SGDOptimizer
    from .losses import MSELossLayer
    from .train import plot_model_guesses, train


from typing import Any, Dict
```

```python
import numpy as np
import torch
from matplotlib import pyplot as plt
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

from utils import load_dataset, problem

RNG = torch.Generator()
RNG.manual_seed(446)


@problem.tag("hw3-A")
def accuracy_score(model: nn.Module, dataloader: DataLoader) -> float:

    correct = 0
    total = 0
    with torch.no_grad():
        for data in dataloader:
            obs, target = data
            outputs = model(obs)
            _, predicted = torch.max(outputs.data, 1)
            _, target = torch.max(target.data, 1)
            total += target.size(0)
            correct += (predicted == target).sum().item()
    return correct / total


@problem.tag("hw3-A")
def mse_parameter_search(
    dataset_train: TensorDataset, dataset_val: TensorDataset
) -> Dict[str, Any]:

    train_loader = DataLoader(dataset_train, batch_size=32, shuffle=True)
    val_loader = DataLoader(dataset_val, batch_size=32, shuffle=True)


    ret = dict()
    criterion = MSELossLayer()
    learning_rate = 0.009

    linearlayer = nn.Sequential(
        LinearLayer(2, 2, generator=RNG)
    )
    optimizer = SGDOptimizer(params=linearlayer.parameters(), lr=learning_rate)
    train_result = train(train_loader, linearlayer, criterion, optimizer, val_loader, 100)
    ret['linear'] = { "train": train_result["train"], "val": train_result["val"], "model": l

    sigmoid_1hiddenlayer = nn.Sequential(
        LinearLayer(2,2, generator=RNG),
        SigmoidLayer(),
        LinearLayer(2,2, generator=RNG)
    )
    optimizer = SGDOptimizer(params=sigmoid_1hiddenlayer.parameters(), lr=learning_rate)
    train_result = train(train_loader, sigmoid_1hiddenlayer, criterion, optimizer, val_loade
```

```
    ret['sigmoid_1hidden'] = { "train": train_result["train"], "val": train_result["val"], "

    relu_1hiddenlayer = nn.Sequential(
        LinearLayer(2,2, generator=RNG),
        ReLULayer(),
        LinearLayer(2,2, generator=RNG)
    )
    optimizer = SGDOptimizer(params=relu_1hiddenlayer.parameters(), lr=learning_rate)
    train_result = train(train_loader, relu_1hiddenlayer, criterion, optimizer, val_loader,
    ret['relu_1hidden'] = { "train": train_result["train"], "val": train_result["val"], "mo

    sigmoid_relu_2hiddenlayer = nn.Sequential(
        LinearLayer(2,2, generator=RNG),
        SigmoidLayer(),
        LinearLayer(2,2, generator=RNG),
        ReLULayer(),
        LinearLayer(2,2, generator=RNG)
    )
    optimizer = SGDOptimizer(params=sigmoid_relu_2hiddenlayer.parameters(), lr=learning_rate
    train_result = train(train_loader, sigmoid_relu_2hiddenlayer, criterion, optimizer, val_
    ret['sigmoid_relu_2hidden'] = { "train": train_result["train"], "val": train_result["val

    relu_sigmoid_2hiddenlayer = nn.Sequential(
        LinearLayer(2,2, generator=RNG),
        ReLULayer(),
        LinearLayer(2,2, generator=RNG),
        SigmoidLayer(),
        LinearLayer(2,2, generator=RNG)
    )
    optimizer = SGDOptimizer(params=relu_sigmoid_2hiddenlayer.parameters(), lr=learning_rate
    train_result = train(train_loader, relu_sigmoid_2hiddenlayer, criterion, optimizer, val_
    ret['relu_sigmoid_2hidden'] = { "train": train_result["train"], "val": train_result["val

    return ret



@problem.tag("hw3-A", start_line=11)
def main():

    (x, y), (x_val, y_val), (x_test, y_test) = load_dataset("xor")

    dataset_train = TensorDataset(torch.from_numpy(x), torch.from_numpy(to_one_hot(y)))
    dataset_val = TensorDataset(
        torch.from_numpy(x_val), torch.from_numpy(to_one_hot(y_val))
    )
    dataset_test = TensorDataset(
        torch.from_numpy(x_test), torch.from_numpy(to_one_hot(y_test))
    )

    mse_configs = mse_parameter_search(dataset_train, dataset_val)

    min_loss = float("inf")
    min_model_name = None
    min_model = None
```

```python
    for i in mse_configs.items():
        x = range(100)
        train = i[1]['train']
        val = i[1]['val']
        model_name = i[0]
        model = i[1]["model"]
        plt.plot(x , train, label = model_name + " train")
        plt.plot(x , val, label = model_name + " val")
        m_loss = min(val)
        if m_loss < min_loss:
            min_loss = m_loss
            min_model_name = model_name
            min_model = model


    plt.ylabel("MSE Loss")
    plt.xlabel("Epoches")
    plt.legend()
    plt.show()

    print("Best performing architecture: " + min_model_name)

    plot_model_guesses(DataLoader(dataset_test), min_model, min_model_name)
    ac = accuracy_score(min_model, DataLoader(dataset_test))
    print(ac)


def to_one_hot(a: np.ndarray) -> np.ndarray:

    r = np.zeros((len(a), 2))
    r[np.arange(len(a)), a] = 1
    return r


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

Administrative

A6.

a. 25 hours