HW3

Octi Zhang

November 15, 2023

OCTIPUS

1

a. False.

Deep neural network's loss function are non convex. Gradient descent provide some local minima, but no guarantee the local minima will be global minima

b. False.

There is not preference what is the best weight, a randomly initialized value may work as good as zero initialization or even better. Though initialize to 0 is better than initialized to some highly variant, exploding values.

c. True.

Non-linear activation functions in a neural network's hidden layers enable the network to learn non-linear decision boundaries.

d. False

The time complexities of both forward and backward passes in a neural network are comparable. For each layer, computations involve matrix multiplications, resulting in $O(nm)$ complexity, where n and m are matrix dimensions. Additional gradient calculations in the backward pass involves in the same multiplication, and do not significantly alter this complexity.

e. False.

Neural Networks, though versatile, are not always the best choice; their suitability varies with factors like data complexity, computational resources, and interpret-ability needs, where simpler models may be preferable.

2

Part a

$$
\begin{aligned}
\phi(x) \cdot \phi(x_0) &= \sum_{i=0}^{\infty} \left(\frac{1}{\sqrt{i!}} e^{-\frac{x^2}{2}} x^i \right) \left(\frac{1}{\sqrt{i!}} e^{-\frac{x_0^2}{2}} x_0^i \right) && \text{Definition of } \phi(x) \\
&= \sum_{i=0}^{\infty} \frac{1}{i!} e^{-\frac{x^2+x_0^2}{2}} x^i x_0^i && \text{Simplification} \\
&= e^{-\frac{x^2+x_0^2}{2}} \sum_{i=0}^{\infty} \frac{(xx_0)^i}{i!} && \text{Factorization} \\
&= e^{-\frac{x^2}{2}} e^{-\frac{x_0^2}{2}} e^{xx_0} && \text{Taylor Series of } e^{xx_0} \text{ as } e^z = \sum_{i=0}^{\infty} \frac{z^i}{i!} \\
&= e^{-\frac{x^2}{2} - \frac{x_0^2}{2} + xx_0} \\
&= e^{-\frac{(x-x_0)^2}{2}}
\end{aligned}
$$

Therefore, $K(x, x_0) = e^{-\frac{(x-x_0)^2}{2}}$ is indeed a kernel function for $\phi(x)$.

3

Part a

poly kernel hyperparameters:

$$\lambda = 2.7825 \times 10^{-5}$$

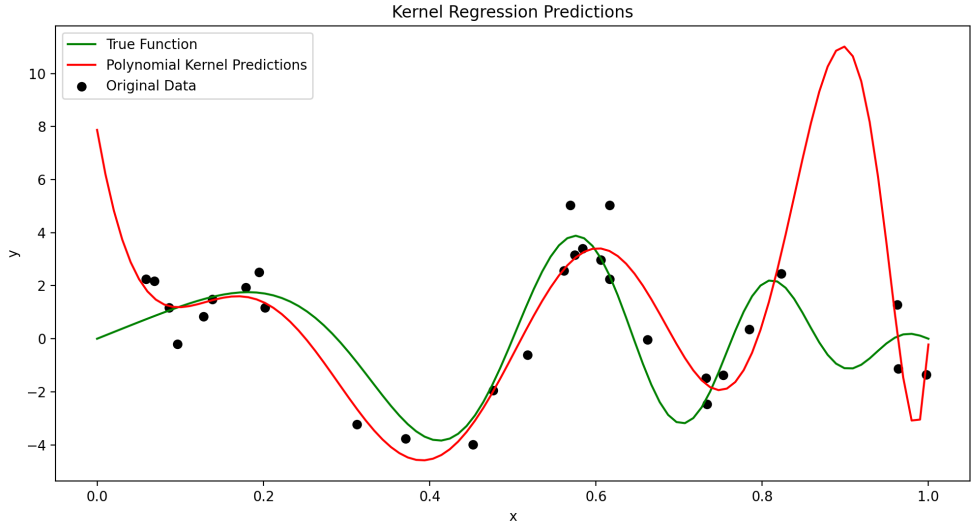
$$\text{degree} = 19.0$$

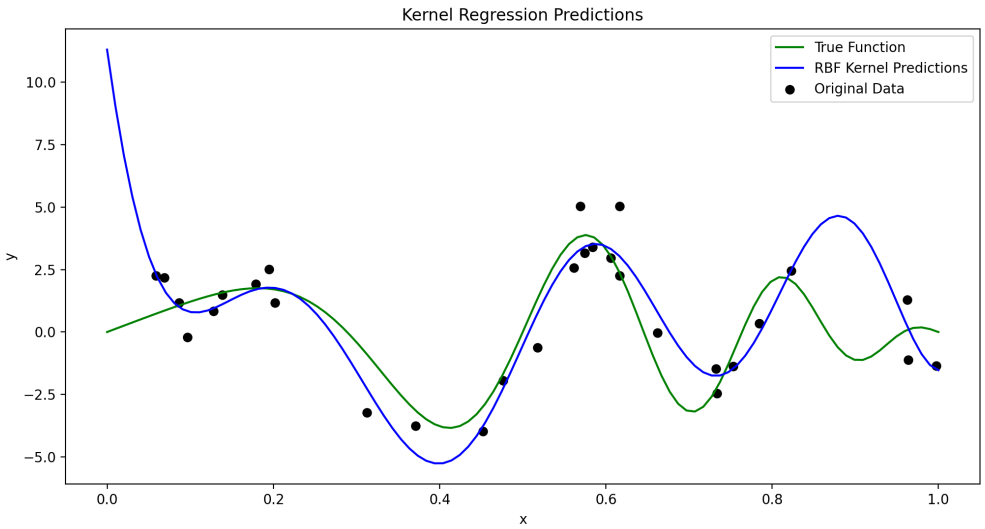
rbf kernel hyperparameters:

$$\lambda = 10^{-0.5}$$

$$\gamma = 11.2019$$

Part b





```python
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:
    """True function, which was used to generate data.
    Should be used for plotting.

    Args:
        x (np.ndarray): A (n,) array. Input.

    Returns:
        np.ndarray: A (n,) array.
    """
    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:
    """Polynomial kernel.

    Given two indices a and b it should calculate:
    K[a, b] = (x_i[a] * x_j[b] + 1)^d

    Args:
        x_i (np.ndarray): An (n,) array. Observations (Might be different from x_j).
        x_j (np.ndarray): An (m,) array. Observations (Might be different from x_i).
        d (int): Degree of polynomial.

    Returns:
        np.ndarray: A (n, m) matrix, where each element is as described above (see equation for K[a,

    Note:
        - It is crucial for this function to be vectorized, and not contain for-loops.
            It will be called a lot, and it has to be fast for reasonable run-time.
        - You might find .outer functions useful for this function.
            They apply an operation similar to xx^T (if x is a vector), but not necessarily with mult
            To use it simply append .outer to function. For example: np.add.outer, np.divide.outer
    """
    # raise NotImplementedError("Your Code Goes Here")
    poly_ker = (np.multiply.outer(x_i, x_j) + 1)**d
    return poly_ker

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

    Given two indices a and b it should calculate:
    K[a, b] = exp(-gamma*(x_i[a] - x_j[b])^2)

    Args:
        x_i (np.ndarray): An (n,) array. Observations (Might be different from x_j).
```

```
        x_j (np.ndarray): An (m,) array. Observations (Might be different from x_i).
        gamma (float): Gamma parameter for RBF kernel. (Inverse of standard deviation)

    Returns:
        np.ndarray: A (n, m) matrix, where each element is as described above (see equation for K[a,

    Note:
        - It is crucial for this function to be vectorized, and not contain for-loops.
            It will be called a lot, and it has to be fast for reasonable run-time.
        - You might find .outer functions useful for this function.
            They apply an operation similar to xx^T (if x is a vector), but not necessarily with mult
            To use it simply append .outer to function. For example: np.add.outer, np.divide.outer
    """

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


@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:
    """Trains and returns an alpha vector, that can be used to make predictions.

    Args:
        x (np.ndarray): Array of shape (n,). Observations.
        y (np.ndarray): Array of shape (n,). Targets.
        kernel_function (Union[poly_kernel, rbf_kernel]): Either poly_kernel or rbf_kernel functions.
        kernel_param (Union[int, float]): Gamma (if kernel_function is rbf_kernel) or d (if kernel_fu
        _lambda (float): Regularization constant.

    Returns:
        np.ndarray: Array of shape (n,) containing alpha hat as described in the pdf.
    """
    # raise NotImplementedError("Your Code Goes Here")
    k = kernel_function(x, x, kernel_param)
    a = np.linalg.solve(k + _lambda * np.eye(len(k)), y)
    return a


@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:
    """Performs cross validation.

    In a for loop over folds:
```

```
        1. Set current fold to be validation, and set all other folds as training set.
        2, Train a function on training set, and then get mean squared error on current fold (validat
    Return validation loss averaged over all folds.

    Args:
        x (np.ndarray): Array of shape (n,). Observations.
        y (np.ndarray): Array of shape (n,). Targets.
        kernel_function (Union[poly_kernel, rbf_kernel]): Either poly_kernel or rbf_kernel functions.
        kernel_param (Union[int, float]): Gamma (if kernel_function is rbf_kernel) or d (if kernel_fu
        _lambda (float): Regularization constant.
        num_folds (int): Number of folds. It should be either len(x) for LOO, or 10 for 10-fold CV.

    Returns:
        float: Average loss of trained function on validation sets across folds.
    """
    total_mse = 0
    fold_size = len(x) // num_folds
    for i in range(num_folds):
        start, end = i * fold_size, (i + 1) * fold_size

        x_train = np.concatenate([x[:start], x[end:]])
        y_train = np.concatenate([y[:start], y[end:]])
        x_test = x[start:end]
        y_test = y[start:end]

        a = train(x_train, y_train, kernel_function, kernel_param, _lambda)
        y_predict = a @ kernel_function(x_train, x_test, kernel_param)

        total_mse += np.mean((y_test - y_predict) ** 2)

    return total_mse / num_folds


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

    There are two possible approaches:
        - Grid Search - Fix possible values for lambda, loop over them and record value with the lowe
        - Random Search - Fix number of iterations, during each iteration sample lambda from some dis

    Args:
        x (np.ndarray): Array of shape (n,). Observations.
        y (np.ndarray): Array of shape (n,). Targets.
        num_folds (int): Number of folds. It should be len(x) for LOO.

    Returns:
        Tuple[float, float]: Tuple containing best performing lambda and gamma pair.

    Note:
        - You do not really need to search over gamma. 1 / (median(dist(x_i, x_j)^2) for all unique p
            should be sufficient for this problem. That being said you are more than welcome to do so
        - If using random search we recommend sampling lambda from distribution 10**i, where i~Unif(-
```

```
        - If using grid search we recommend choosing possible lambdas to 10**i, where i=linspace(-5,
    """
    # raise NotImplementedError("Your Code Goes Here")

    lambda_values= 10 ** np.linspace(-5, -1, 10)
    gamma = 1/np.median(np.subtract.outer(x, x) ** 2)
    lambda_mse = [(_lambda, gamma, cross_validation(x, y, rbf_kernel, gamma, _lambda, num_folds)) for
    return min(lambda_mse, key= lambda item : item[1])[0:2]


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

    There are two possible approaches:
        - Grid Search - Fix possible values for lambdas and ds.
            Have nested loop over all possibilities and record value with the lowest loss.
        - Random Search - Fix number of iterations, during each iteration sample lambda, d from some

    Args:
        x (np.ndarray): Array of shape (n,). Observations.
        y (np.ndarray): Array of shape (n,). Targets.
        num_folds (int): Number of folds. It should be either len(x) for LOO, or 10 for 10-fold CV.

    Returns:
        Tuple[float, int]: Tuple containing best performing lambda and d pair.

    Note:
        - You can use gamma = 1 / median((x_i - x_j)^2) for all unique pairs x_i, x_j in x) for this
          However, if you would like to search over other possible values of gamma, you are welcome t
        - If using random search we recommend sampling lambda from distribution 10**i, where i~Unif(-
            and d from distribution [5, 6, ..., 24, 25]
        - If using grid search we recommend choosing possible lambdas to 10**i, where i=linspace(-5,
            and possible ds to [5, 6, ..., 24, 25]
    """
    # raise NotImplementedError("Your Code Goes Here")
    lambda_values= 10 ** np.linspace(-5, -1, 10)
    print(lambda_values)
    degree_values = np.linspace(5, 25, 21)
    lambda_grid, degree_grid = np.meshgrid(lambda_values, degree_values)
    combos = list(zip(lambda_grid.flatten(), degree_grid.flatten()))
    lambda_mse = [(lambda_value, degree_value, cross_validation(x, y, poly_kernel, degree_value, lamb
    return min(lambda_mse, key=lambda item : item[2])[0:2]

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

    It should:
        A. Using x_30, y_30, rbf_param_search and poly_param_search report optimal values for lambda
            Note that x_30, y_30 has been loaded in for you. You do not need to use (x_300, y_300) or
        B. For both rbf and poly kernels, train a function using x_30, y_30 and plot predictions on d
```

```
    Note:
        - In part b fine grid can be defined as np.linspace(0, 1, num=100)
        - When plotting you might find that your predictions go into hundreds, causing majority of th
            To avoid this call plt.ylim(-6, 6).
    """
    (x_30, y_30), (x_300, y_300), (x_1000, y_1000) = load_dataset("kernel_bootstrap")
    poly_lambda, poly_degree = poly_param_search(x_30, y_30, num_folds=10)
    rbf_lambda, rbf_gamma = rbf_param_search(x_30, y_30, num_folds=10)

    print ((poly_lambda, poly_degree), (rbf_lambda, rbf_gamma))

def plot_data():
    (x_30, y_30), (x_300, y_300), (x_1000, y_1000) = load_dataset("kernel_bootstrap")
    # rbf_lambda, rbf_gamma = rbf_param_search(x_30, y_30, num_folds=10)
    poly_a = train(x_30, y_30, poly_kernel, 19, 2.782559402207126e-05)
    rbf_a = train(x_30, y_30, rbf_kernel, 11.201924992299844, 1e-05)

    f_rbf = lambda x : rbf_a @ rbf_kernel(x_30, x, 11.201924992299844)
    f_poly = lambda x : poly_a @ poly_kernel(x_30, x, 19)


    # Fine grid for plotting
    fine_grid = np.linspace(0, 1, 100)

    # Evaluating functions on the grid
    rbf_predictions = f_rbf(fine_grid)
    poly_predictions = f_poly(fine_grid)
    true_values = f_true(fine_grid)

    # Plotting
    plt.figure(figsize=(12, 6))

    # Original data
    plt.scatter(x_30, y_30, label='Original Data', color='black')

    # True function
    plt.plot(fine_grid, true_values, label='True Function', color='green')

    # RBF predictions
    plt.plot(fine_grid, rbf_predictions, label='RBF Kernel Predictions', color='blue')

    # Polynomial predictions
    # plt.plot(fine_grid, poly_predictions, label='Polynomial Kernel Predictions', color='red')

    plt.title('Kernel Regression Predictions')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend()
    plt.show()

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

4

Part a

```python
from typing import Optional

import torch
from torch import nn

from utils import problem


class LinearLayer(nn.Module):
    @problem.tag("hw3-A", start_line=1)
    def __init__(
        self, dim_in: int, dim_out: int, generator: Optional[torch.Generator] = None
    ):
        """Linear Layer, which performs calculation of: x @ weight + bias

        In constructor you should initialize weight and bias according to dimensions provided.
        You should use torch.randn function to initialize them by normal distribution, and provide th

        Both weight and bias should be of torch's type float.
        Additionally, for optimizer to work properly you will want to wrap both weight and bias in nn

        Args:
            dim_in (int): Number of features in data input.
            dim_out (int): Number of features output data should have.
            generator (Optional[torch.Generator], optional): Generator to use when creating weight an
                If defined it should be passed into torch.randn function.
                Defaults to None.

        Note:
            - YOU ARE NOT ALLOWED to use torch.nn.Linear (or it's functional counterparts) in this cl
            - Make use of pytorch documentation: https://pytorch.org/docs/stable/index.html
        """
        super().__init__()
        if generator is None:
            self.weight = nn.Parameter(torch.randn(dim_in, dim_out, dtype=torch.float64))
            self.bias = nn.Parameter(torch.randn(dim_out, dtype=torch.float64))
        else:
            self.weight = nn.Parameter(torch.randn(dim_in, dim_out, generator=generator, dtype=torch.
            self.bias = nn.Parameter(torch.randn(dim_out, generator=generator, dtype=torch.float64))

    @problem.tag("hw3-A")
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """Actually perform multiplication x @ weight + bias

        Args:
            x (torch.Tensor): More specifically a torch.FloatTensor, with shape of (n, dim_in).
                Input data.

        Returns:
            torch.Tensor: More specifically a torch.FloatTensor, with shape of (n, dim_out).
                Output data.
        """
        return x @ self.weight + self.bias
```

```python
import torch
from torch import nn

from utils import problem


class ReLULayer(nn.Module):
    @problem.tag("hw3-A")
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """Performs a Rectified Linear Unit calculation (ReLU):
        Element-wise:
            - if x > 0: return x
            - else: return 0

        Args:
            x (torch.Tensor): More specifically a torch.FloatTensor, with some shape.
                Input data.

        Returns:
            torch.Tensor: More specifically a torch.FloatTensor, with the same shape as x.
                Every negative element should be substituted with 0.
                Output data.

        Note:
            - YOU ARE NOT ALLOWED to use torch.nn.ReLU (or it's functional counterparts) in this clas
            - Make use of pytorch documentation: https://pytorch.org/docs/stable/index.html
        """
        return torch.where(x > 0, x, torch.tensor(0.0, dtype=x.dtype))
```

```python
import torch
from torch import nn

from utils import problem


class SigmoidLayer(nn.Module):
    @problem.tag("hw3-A")
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """Performs a sigmoid calculation:
        Element-wise given x return 1 / (1 + e^(-x))

        Args:
            x (torch.Tensor): More specifically a torch.FloatTensor, with some shape.
                Input data.

        Returns:
            torch.Tensor: More specifically a torch.FloatTensor, with the same shape as x.
                Every negative element should be substituted with sigmoid of that element.
                Output data.

        Note:
            - YOU ARE NOT ALLOWED to use torch.nn.Sigmoid (or torch.nn.functional.sigmoid) in this cl
                YOU CAN however use other aliases of sigmoid function in PyTorch if you are able to f
            - Make use of pytorch documentation: https://pytorch.org/docs/stable/index.html
        """
        return 1 / (1 + torch.exp(-x))
```

```python
import torch
```

```python
from torch import nn

from utils import problem


class SoftmaxLayer(nn.Module):
    @problem.tag("hw3-A")
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """Performs a softmax calculation.
        Given a matrix x (n, d) on each element performs:

        softmax(x) = exp(x_ij) / sum_k=0^d exp(x_ik)

        i.e. it first takes an exponential of each element,
            and that normalizes rows so that their L-1 norm is equal to 1.

        Args:
            x (torch.Tensor): More specifically a torch.FloatTensor, with shape (n, d).
                Input data.

        Returns:
            torch.Tensor: More specifically a torch.FloatTensor, also with shape (n, d).
                Each row has L-1 norm of 1, and each element is in [0, 1] (i.e. each row is a probabi
                Output data.

        Note:
            - For a numerically stable approach to softmax (needed for the problem),
                first subtract max of x from data (no need for dim argument, torch.max(x) suffices).
                This causes exponent to not blow up, and arrives to exactly the same answer.
            - YOU ARE NOT ALLOWED to use torch.nn.Softmax (or it's functional counterparts) in this c
            - Make use of pytorch documentation: https://pytorch.org/docs/stable/index.html
        """
        x_ = x - torch.max(x)
        return torch.exp(x_)/ torch.sum(torch.exp(x_), dim=1, keepdim=True)

import torch
from torch import nn

from utils import problem


class CrossEntropyLossLayer(nn.Module):
    @problem.tag("hw3-A")
    def forward(self, y_pred: torch.Tensor, y_true: torch.Tensor) -> torch.Tensor:
        """Calculate Crossentropy loss based on (normalized) predictions and true categories/classes.

        For a single example (x is vector of predictions, y is correct class):

        cross_entropy(x, y) = -log(x[y])

        Args:
            y_pred (torch.Tensor): More specifically a torch.FloatTensor, with shape (n, c).
                Predictions of classes. Each row is normalized so that L-1 norm is 1 (Each row is pro
                Input data.
            y_true (torch.Tensor): More specifically a torch.LongTensor, with shape (n,).
                Each element is an integer in range [0, c).
                Input data.
```

```
        Returns:
            torch.Tensor: More specifically a SINGLE VALUE torch.FloatTensor (i.e. with shape (1,)).
                Should be a mean over all examples.
                Result.

        Note:
            - YOU ARE NOT ALLOWED to use torch.nn.CrossEntropyLoss / torch.nn.NLLLoss (or their funct
            - Make use of pytorch documentation: https://pytorch.org/docs/stable/index.html
            - Not that this is different from torch.nn.CrossEntropyLoss, as it doesn't perform softma
        """
        individual_losses = -torch.log(y_pred[torch.arange(y_pred.size(0)), y_true])
        return individual_losses.mean()
```

```
import torch
from torch import nn

from utils import problem


class MSELossLayer(nn.Module):
    @problem.tag("hw3-A")
    def forward(self, y_pred: torch.Tensor, y_true: torch.Tensor) -> torch.Tensor:
        """Calculate MSE between predictions and truth values.

        Args:
            y_pred (torch.Tensor): More specifically a torch.FloatTensor, with some shape.
                Input data.
            y_true (torch.Tensor): More specifically a torch.FloatTensor, with the same shape as y_pr
                Input data.

        Returns:
            torch.Tensor: More specifically a SINGLE VALUE torch.FloatTensor (i.e. with shape (1,)).
                Result.

        Note:
            - YOU ARE NOT ALLOWED to use torch.nn.MSELoss (or it's functional counterparts) in this c
            - Make use of pytorch documentation: https://pytorch.org/docs/stable/index.html
        """
        return torch.mean((y_pred - y_true)**2)
```

```
import torch

from utils import problem


class SGDOptimizer(torch.optim.Optimizer):
    @problem.tag("hw3-A", start_line=1)
    def __init__(self, params, lr: float) -> None:
        """Constructor for Stochastic Gradient Descent (SGD) Optimizer.

        Provided code contains call to super class, which will initialize paramaters properly (see st
        This class will only update the parameters provided to it, based on their (already calculated

        Args:
            params: Parameters to update each step. You don't need to do anything with them.
                They are properly initialize through the super call.
```

```
            lr (float): Learning Rate of the gradient descent.

        Note:
            - YOU ARE NOT ALLOWED to use torch.optim.SGD in this class
            - While you are not allowed to use the class above, it might be extremely beneficial to l
            - Make use of pytorch documentation: https://pytorch.org/docs/stable/index.html
        """
        super().__init__(params, {"lr": lr})

    @problem.tag("hw3-A")
    def step(self, closure=None):  # noqa: E251
        """
        Performs a step of gradient descent. You should loop through each parameter, and update it's

        Args:
            closure (optional): Ignore this. We will not use in this class, but it is required for su
                Defaults to None.

        Hint:
            - Superclass stores parameters in self.param_groups (you will have to discover in what fo
        """
        for group in self.param_groups:
            for p in group['params']:
                p.data -= group['lr'] * p.grad.data

from typing import Dict, List, Optional

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

from utils import problem


@problem.tag("hw3-A")
def train(
    train_loader: DataLoader,
    model: nn.Module,
    criterion: nn.Module,
    optimizer: optim.Optimizer,
    val_loader: Optional[DataLoader] = None,
    epochs: int = 100,
) -> Dict[str, List[float]]:
    """Performs training of a provided model and provided dataset.

    Args:
        train_loader (DataLoader): DataLoader for training set.
        model (nn.Module): Model to train.
        criterion (nn.Module): Callable instance of loss function, that can be used to calculate loss
        optimizer (optim.Optimizer): Optimizer used for updating parameters of the model.
        val_loader (Optional[DataLoader], optional): DataLoader for validation set.
            If defined, if should be used to calculate loss on validation set, after each epoch.
            Defaults to None.
        epochs (int, optional): Number of epochs (passes through dataset/dataloader) to train for.
```

```
            Defaults to 100.

    Returns:
        Dict[str, List[float]]: Dictionary with history of training.
            It should have have two keys: "train" and "val",
            each pointing to a list of floats representing loss at each epoch for corresponding datas
            If val_loader is undefined, "val" can point at an empty list.

    Note:
        - Calculating training loss might expensive if you do it seperately from training a model.
            Using a running loss approach is advised.
            In this case you will just use the loss that you called .backward() on add sum them up ac
            Then you can divide by length of train_loader, and you will have an average loss for each
        - You will be iterating over multiple models in main function.
            Make sure the optimizer is defined for proper model.
        - Make use of pytorch documentation: https://pytorch.org/docs/stable/index.html
            You might find some examples/tutorials useful.
            Also make sure to check out torch.no_grad function. It might be useful!
        - Make sure to load the model parameters corresponding to model with the best validation loss
            You might want to look into state_dict: https://pytorch.org/tutorials/beginner/saving_loa
    """
    # raise NotImplementedError("Your Code Goes Here")
    model.train()
    history = {"train": [], "val": []}
    for epoch in tqdm(range(epochs)):
        total_loss = 0
        for _x, _y in train_loader:
            optimizer.zero_grad()
            y_pred = model(_x)
            loss = criterion(y_pred, _y)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()

        avg_train_loss = total_loss / len(train_loader)
        history["train"].append(avg_train_loss)

        if val_loader:
            total_val_loss = 0
            with torch.no_grad():
                for _x, _y in val_loader:
                    y_pred = model(_x)
                    loss = criterion(y_pred, _y)
                    total_val_loss += loss.item()
            ave_val_loss = total_val_loss / len(val_loader)
            history["val"].append(ave_val_loss)
            model.train()

    print(f'train_loss: {history["train"][-1]}' + f'val_loss: {history["val"][-1]}' if val_loader els


    return history


def plot_model_guesses(
    dataloader: DataLoader, model: nn.Module, title: Optional[str] = None
```

```
):
    """Helper function!
    Given data and model plots model predictions, and groups them into:
        - True positives
        - False positives
        - True negatives
        - False negatives

    Args:
        dataloader (DataLoader): Data to plot.
        model (nn.Module): Model to make predictions.
        title (Optional[str], optional): Optional title of the plot.
            Might be useful for distinguishing between MSE and CrossEntropy.
            Defaults to None.
    """
    with torch.no_grad():
        list_xs = []
        list_ys_pred = []
        list_ys_batch = []
        for x_batch, y_batch in dataloader:
            y_pred = model(x_batch)
            list_xs.extend(x_batch.numpy())
            list_ys_batch.extend(y_batch.numpy())
            list_ys_pred.extend(torch.argmax(y_pred, dim=1).numpy())

        xs = np.array(list_xs)
        ys_pred = np.array(list_ys_pred)
        ys_batch = np.array(list_ys_batch)

        # True positive
        if len(ys_batch.shape) == 2 and ys_batch.shape[1] == 2:
            # MSE fix
            ys_batch = np.argmax(ys_batch, axis=1)
        idxs = np.logical_and(ys_batch, ys_pred)
        plt.scatter(
            xs[idxs, 0], xs[idxs, 1], marker="o", c="green", label="True Positive"
        )
        # False positive
        idxs = np.logical_and(1 - ys_batch, ys_pred)
        plt.scatter(
            xs[idxs, 0], xs[idxs, 1], marker="o", c="red", label="False Positive"
        )
        # True negative
        idxs = np.logical_and(1 - ys_batch, 1 - ys_pred)
        plt.scatter(
            xs[idxs, 0], xs[idxs, 1], marker="x", c="green", label="True Negative"
        )
        # False negative
        idxs = np.logical_and(ys_batch, 1 - ys_pred)
        plt.scatter(
            xs[idxs, 0], xs[idxs, 1], marker="x", c="red", label="False Negative"
        )

        if title:
            plt.title(title)
        plt.xlabel("x0")
```

```
        plt.ylabel("x1")
        plt.legend()
        plt.show()

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]:
    """
    Main subroutine of the CrossEntropy problem.
    It's goal is to perform a search over hyperparameters, and return a dictionary containing trainin

    Models to check (please try them in this order):
        - Linear Regression Model
        - Network with one hidden layer of size 2 and sigmoid activation function after the hidden la
        - Network with one hidden layer of size 2 and ReLU activation function after the hidden layer
        - Network with two hidden layers (each with size 2)
            and Sigmoid, ReLU activation function after corresponding hidden layers
        - Network with two hidden layers (each with size 2)
            and ReLU, Sigmoid activation function after corresponding hidden layers
    NOTE: Each model should end with a Softmax layer due to CrossEntropyLossLayer requirement.

    Notes:
        - Try using learning rate between 1e-5 and 1e-3.
        - When choosing the number of epochs, consider effect of other hyperparameters on it.
            For example as learning rate gets smaller you will need more epochs to converge.
        - When searching over batch_size using powers of 2 (starting at around 32) is typically a goo
            Make sure it is not too big as you can end up with standard (or almost) gradient descent!

    Args:
        dataset_train (TensorDataset): Dataset for training.
        dataset_val (TensorDataset): Dataset for validation.
```

```
    Returns:
        Dict[str, Any]: Dictionary/Map containing history of training of all models.
            You are free to employ any structure of this dictionary, but we suggest the following:
            {
                name_of_model: {
                    "train": Per epoch losses of model on train set,
                    "val": Per epoch losses of model on validation set,
                    "model": Actual PyTorch model (type: nn.Module),
                }
            }
    """
    # raise NotImplementedError("Your Code Goes Here")
    input_sample, output_sample = dataset_train[0]
    # Determine the shapes
    class LinearModel(nn.Module):
        def __init__(self, input_size, output_size):
            super().__init__()
            self.linear = LinearLayer(input_size, output_size)
            self.softmax = SoftmaxLayer()

        def forward(self, inputs):
            x = self.linear(inputs)
            return self.softmax(x)

    class OneHiddenLayerModel(nn.Module):
        def __init__(self, input_size, output_size, hidden_size, activation_func):
            super().__init__()
            self.linear0 = LinearLayer(input_size, hidden_size)
            self.activation = activation_func
            self.linear1 = LinearLayer(hidden_size, output_size)
            self.softmax = SoftmaxLayer()

        def forward(self, inputs):
            x = self.activation(self.linear0(inputs))
            x = self.softmax(self.linear1(x))
            return x

    class TwoHiddenLayerModel(nn.Module):
        def __init__(self, input_size, output_size, hidden_size, activation_func1, activation_func2):
            super().__init__()
            self.linear0 = LinearLayer(input_size, hidden_size)
            self.activation0 = activation_func1
            self.linear1 = LinearLayer(hidden_size, hidden_size)
            self.activation1 = activation_func2
            self.linear2 = LinearLayer(hidden_size, output_size)
            self.softmax = SoftmaxLayer()

        def forward(self, inputs):
            x = self.activation0(self.linear0(inputs))
            x = self.activation1(self.linear1(x))
            x = self.softmax(self.linear2(x))
            return x

    input_feature_size = input_sample.shape[0]
    output_size = 2
```

```
    import itertools
    lr= 10 ** np.linspace(-5, -3, 3)
    batch_sizes = 2 ** np.linspace(5, 7, 3)
    models = {
        "Linear": LinearModel(input_feature_size, output_size),
        "OneHidden_Sigmoid": OneHiddenLayerModel(input_feature_size, output_size, 2, SigmoidLayer()),
        "OneHidden_ReLU": OneHiddenLayerModel(input_feature_size, output_size, 2, ReLULayer()),
        "TwoHidden_SigmoidReLU": TwoHiddenLayerModel(input_feature_size, output_size, 2, SigmoidLayer
        "TwoHidden_ReLUSigmoid": TwoHiddenLayerModel(input_feature_size, output_size, 2, ReLULayer(),
    }
    result = {}

    combos = list(itertools.product(lr, batch_sizes, models.items()))
    count = 0
    for lr, batch_size, [model_name, model] in combos:
        train_loader = DataLoader(dataset_train, batch_size=int(batch_size), shuffle=True)
        val_loader = DataLoader(dataset_val, batch_size=int(batch_size), shuffle=False)
        history = train(train_loader, model, CrossEntropyLossLayer(), SGDOptimizer(model.parameters()
        model_name_ = f'{model_name}_lr{lr}_batch{int(batch_size)}'
        print(f"{count}: {model_name_}")
        count +=1
        result[model_name_] = {}
        result[model_name_]['train'] = history['train']
        result[model_name_]['val'] = history['val']
        result[model_name_]["model"] = model
        result[model_name_]['lr'] = lr
        result[model_name_]['batch_size'] = int(batch_size)
    return result


@problem.tag("hw3-A")
def accuracy_score(model, dataloader) -> float:
    """Calculates accuracy of model on dataloader. Returns it as a fraction.

    Args:
        model (nn.Module): Model to evaluate.
        dataloader (DataLoader): Dataloader for CrossEntropy.
            Each example is a tuple consiting of (observation, target).
            Observation is a 2-d vector of floats.
            Target is an integer representing a correct class to a corresponding observation.

    Returns:
        float: Vanilla python float resprenting accuracy of the model on given dataset/dataloader.
            In range [0, 1].

    Note:
        - For a single-element tensor you can use .item() to cast it to a float.
        - This is similar to MSE accuracy_score function,
            but there will be differences due to slightly different targets in dataloaders.
    """
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for observation, target in dataloader:
```

```
            _predict = model(observation)
            _, predict_ = torch.max(_predict.data, 1)
            total += target.size(0)
            correct += (predict_ == target).sum().item()
    accuracy = correct / total
    return accuracy


@problem.tag("hw3-A", start_line=7)
def main():
    """
    Main function of the Crossentropy problem.
    It should:
        1. Call crossentropy_parameter_search routine and get dictionary for each model architecture/
        2. Plot Train and Validation losses for each model all on single plot (it should be 10 lines
            x-axis should be epochs, y-axis should me Crossentropy loss, REMEMBER to add legend
        3. Choose and report the best model configuration based on validation losses.
            In particular you should choose a model that achieved the lowest validation loss at ANY p
        4. Plot best model guesses on test set (using plot_model_guesses function from train file)
        5. Report accuracy of the model on test set.

    Starter code loads dataset, converts it into PyTorch Datasets, and those into DataLoaders.
    You should use these dataloaders, for the best experience with PyTorch.
    """
    (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))

    result = crossentropy_parameter_search(dataset_train, dataset_val)

    # Plotting Training and Validation Losses
    plot_and_save_graphs(result, "crossentropy")

    # Choosing and Reporting the Best Model
    best_model_name = min(result, key=lambda k: min(result[k]['val']))
    print(f"Best Model: {best_model_name}")

    # Plot Best Model Guesses on Test Set
    test_loader = DataLoader(dataset_test, batch_size=32)
    best_model = result[best_model_name]["model"]
    plot_model_guesses(test_loader, best_model, "best model")

    # Report Accuracy on Test Set
    # best score: 0.98
    accuracy = accuracy_score(best_model, test_loader)
    print(f"Accuracy on Test Set: {accuracy:.2f}")

def plot_and_save_graphs(results, filename_prefix):
    lr_batch_combinations = set((value['lr'], value['batch_size']) for value in results.values())
    unique_batch_sizes = set(batch_size for _, batch_size in lr_batch_combinations)
    unique_lrs = set(lr for lr, _ in lr_batch_combinations)

    # For each batch size, plot a figure with subplots for each learning rate
    for batch_size in unique_batch_sizes:
```

```
        plt.figure(figsize=(15, 6 * len(unique_lrs)))  # Adjust the figure size as needed

        for i, lr in enumerate(unique_lrs, start=1):
            plt.subplot(len(unique_lrs), 1, i)  # Create a subplot for each learning rate

            # Filter and plot models with the current lr and batch_size combination
            for model_name in results:
                if f"_lr{lr}_batch{batch_size}" in model_name:
                    plt.plot(results[model_name]['train'], label=f"{model_name} - Train")
                    plt.plot(results[model_name]['val'], label=f"{model_name} - Val")

            plt.xlabel('Epochs')
            plt.ylabel('MSE Loss')
            plt.title(f'LR: {lr}')
            plt.legend()

        plt.tight_layout()
        plt.savefig(f"hw3_written/{filename_prefix}_batch{batch_size}.png")
        plt.close()  # Close the figure to free memory

if __name__ == "__main__":
    main()

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

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:
    """Calculates accuracy of model on dataloader. Returns it as a fraction.

    Args:
        model (nn.Module): Model to evaluate.
        dataloader (DataLoader): Dataloader for MSE.
            Each example is a tuple consiting of (observation, target).
```

```
            Observation is a 2-d vector of floats.
            Target is also a 2-d vector of floats, but specifically with one being 1.0, while other i
            Index of 1.0 in target corresponds to the true class.

    Returns:
        float: Vanilla python float resprenting accuracy of the model on given dataset/dataloader.
            In range [0, 1].

    Note:
        - For a single-element tensor you can use .item() to cast it to a float.
        - This is similar to CrossEntropy accuracy_score function,
            but there will be differences due to slightly different targets in dataloaders.
    """
    # raise NotImplementedError("Your Code Goes Here")
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for observation, target in dataloader:
            _predict = model(observation)
            _, predict_ = torch.max(_predict.data, 1)
            _, target_ = torch.max(target, 1)

            total += target.size(0)
            correct += (predict_ == target_).sum().item()
    accuracy = correct / total
    return accuracy


@problem.tag("hw3-A")
def mse_parameter_search(
    dataset_train: TensorDataset, dataset_val: TensorDataset
) -> Dict[str, Any]:
    """
    Main subroutine of the MSE problem.
    It's goal is to perform a search over hyperparameters, and return a dictionary containing trainin

    Models to check (please try them in this order):
        - Linear Regression Model
        - Network with one hidden layer of size 2 and sigmoid activation function after the hidden la
        - Network with one hidden layer of size 2 and ReLU activation function after the hidden layer
        - Network with two hidden layers (each with size 2)
            and Sigmoid, ReLU activation function after corresponding hidden layers
        - Network with two hidden layers (each with size 2)
            and ReLU, Sigmoid activation function after corresponding hidden layers

    Notes:
        - Try using learning rate between 1e-5 and 1e-3.
        - When choosing the number of epochs, consider effect of other hyperparameters on it.
            For example as learning rate gets smaller you will need more epochs to converge.
        - When searching over batch_size using powers of 2 (starting at around 32) is typically a goo
            Make sure it is not too big as you can end up with standard (or almost) gradient descent!

    Args:
        dataset_train (TensorDataset): Training dataset.
        dataset_val (TensorDataset): Validation dataset.
```

```
    Returns:
        Dict[str, Any]: Dictionary/Map containing history of training of all models.
            You are free to employ any structure of this dictionary, but we suggest the following:
            {
                name_of_model: {
                    "train": Per epoch losses of model on train set,
                    "val": Per epoch losses of model on validation set,
                    "model": Actual PyTorch model (type: nn.Module),
                }
            }
    """
    # raise NotImplementedError("Your Code Goes Here")
    input_sample, output_sample = dataset_train[0]
    # Determine the shapes
    class LinearModel(nn.Module):
        def __init__(self, input_size, output_size):
            super().__init__()
            self.linear = LinearLayer(input_size, output_size)

        def forward(self, inputs):
            x = self.linear(inputs)
            return x

    class OneHiddenLayerModel(nn.Module):
        def __init__(self, input_size, output_size, hidden_size, activation_func):
            super().__init__()
            self.linear0 = LinearLayer(input_size, hidden_size)
            self.activation = activation_func
            self.linear1 = LinearLayer(hidden_size, output_size)

        def forward(self, inputs):
            x = self.activation(self.linear0(inputs))
            x = self.linear1(x)
            return x

    class TwoHiddenLayerModel(nn.Module):
        def __init__(self, input_size, output_size, hidden_size, activation_func1, activation_func2):
            super().__init__()
            self.linear0 = LinearLayer(input_size, hidden_size)
            self.activation0 = activation_func1
            self.linear1 = LinearLayer(hidden_size, hidden_size)
            self.activation1 = activation_func2
            self.linear2 = LinearLayer(hidden_size, output_size)

        def forward(self, inputs):
            x = self.activation0(self.linear0(inputs))
            x = self.activation1(self.linear1(x))
            x = self.linear2(x)
            return x

    input_feature_size = input_sample.shape[0]
    output_size = 2
    import itertools
    lr= 10 ** np.linspace(-5, -3, 3)
    batch_size = (2 ** np.linspace(5, 7, 3))
```

```
    models = {
        "Linear": LinearModel(input_feature_size, output_size),
        "OneHidden_Sigmoid": OneHiddenLayerModel(input_feature_size, output_size, 2, SigmoidLayer()),
        "OneHidden_ReLU": OneHiddenLayerModel(input_feature_size, output_size, 2, ReLULayer()),
        "TwoHidden_SigmoidReLU": TwoHiddenLayerModel(input_feature_size, output_size, 2, SigmoidLayer
        "TwoHidden_ReLUSigmoid": TwoHiddenLayerModel(input_feature_size, output_size, 2, ReLULayer(),
    }
    combos = list(itertools.product(lr, batch_size, models.items()))
    result = {}
    count = 0
    for lr, batch_size, [model_name, model] in combos:
        train_loader = DataLoader(dataset_train, batch_size=int(batch_size), shuffle=True)
        val_loader = DataLoader(dataset_val, batch_size=int(batch_size), shuffle=False)
        history = train(train_loader, model, MSELossLayer(), SGDOptimizer(model.parameters(), lr=lr),
        model_name_ = f'{model_name}_lr{lr}_batch{int(batch_size)}'
        print(f"{count}: {model_name_}")
        count +=1
        result[model_name_] = {}
        result[model_name_]['train'] = history['train']
        result[model_name_]['val'] = history['val']
        result[model_name_]["model"] = model
        result[model_name_]['lr'] = lr
        result[model_name_]['batch_size'] = int(batch_size)
    return result


@problem.tag("hw3-A", start_line=11)
def main():
    """
    Main function of the MSE problem.
    It should:
        1. Call mse_parameter_search routine and get dictionary for each model architecture/configura
        2. Plot Train and Validation losses for each model all on single plot (it should be 10 lines
            x-axis should be epochs, y-axis should me MSE loss, REMEMBER to add legend
        3. Choose and report the best model configuration based on validation losses.
            In particular you should choose a model that achieved the lowest validation loss at ANY p
        4. Plot best model guesses on test set (using plot_model_guesses function from train file)
        5. Report accuracy of the model on test set.

    Starter code loads dataset, converts it into PyTorch Datasets, and those into DataLoaders.
    You should use these dataloaders, for the best experience with PyTorch.
    """
    (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))
    )

    result = mse_parameter_search(dataset_train, dataset_val)
    # Plotting Training and Validation Losses
    plot_and_save_graphs(result, "mse_model_losses")
```

```
    # Choosing and Reporting the Best Model
    best_model_name = min(result, key=lambda k: min(result[k]['val']))
    print(f"Best Model: {best_model_name}")

    # Plot Best Model Guesses on Test Set
    test_loader = DataLoader(dataset_test, batch_size=32)
    best_model = result[best_model_name]["model"]
    plot_model_guesses(test_loader, best_model, "best model")

    # Report Accuracy on Test Set
    #0.7
    accuracy = accuracy_score(best_model, test_loader)
    print(f"Accuracy on Test Set: {accuracy:.2f}")


def to_one_hot(a: np.ndarray) -> np.ndarray:
    """Helper function. Converts data from categorical to one-hot encoded.

    Args:
        a (np.ndarray): Input array of integers with shape (n,).

    Returns:
        np.ndarray: Array with shape (n, c), where c is maximal element of a.
            Each element of a, has a corresponding one-hot encoded vector of length c.
    """
    r = np.zeros((len(a), 2))
    r[np.arange(len(a)), a] = 1
    return r

def plot_and_save_graphs(results, filename_prefix):
    lr_batch_combinations = set((value['lr'], value['batch_size']) for value in results.values())
    unique_batch_sizes = set(batch_size for _, batch_size in lr_batch_combinations)
    unique_lrs = set(lr for lr, _ in lr_batch_combinations)

    # For each batch size, plot a figure with subplots for each learning rate
    for batch_size in unique_batch_sizes:
        plt.figure(figsize=(15, 6 * len(unique_lrs)))  # Adjust the figure size as needed

        for i, lr in enumerate(unique_lrs, start=1):
            plt.subplot(len(unique_lrs), 1, i)  # Create a subplot for each learning rate

            # Filter and plot models with the current lr and batch_size combination
            for model_name in results:
                if f"_lr{lr}_batch{batch_size}" in model_name:
                    plt.plot(results[model_name]['train'], label=f"{model_name} - Train")
                    plt.plot(results[model_name]['val'], label=f"{model_name} - Val")

            plt.xlabel('Epochs')
            plt.ylabel('MSE Loss')
            plt.title(f'LR: {lr}')
            plt.legend()

        plt.tight_layout()
        plt.savefig(f"hw3_written/{filename_prefix}_batch{batch_size}.png")
        plt.close()  # Close the figure to free memory
```

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

Part b

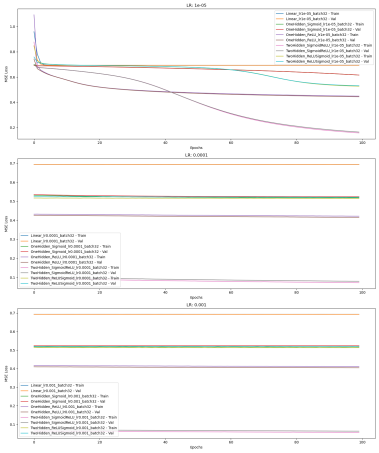


Figure 1: Batch Size 32

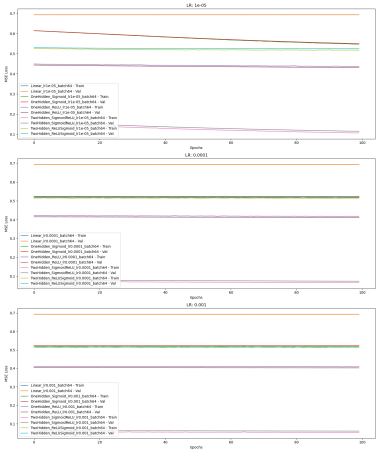


Figure 2: Batch Size 64

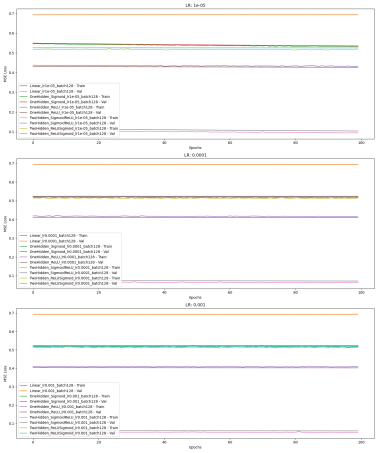


Figure 3: Batch Size 128

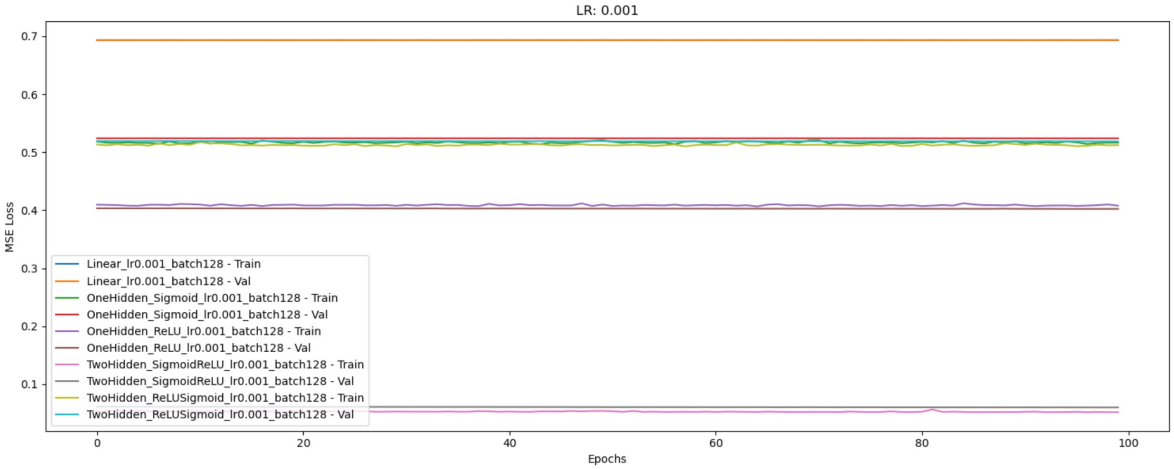


Figure 4: best hyperparameter lr: 0.001, batch size:128, model: Two Hidden Layer Sigmoid ReLU

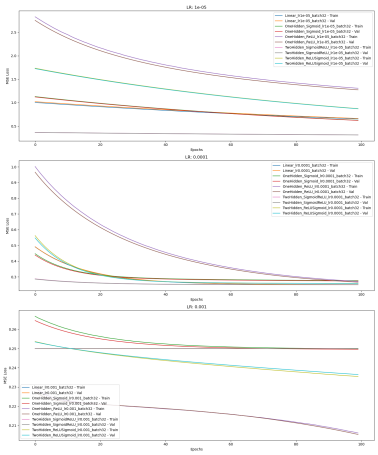


Figure 5: Batch Size 32

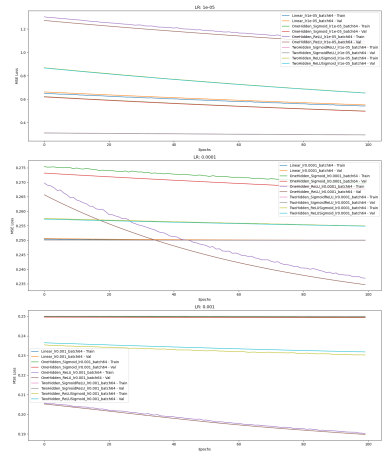


Figure 6: Batch Size 64

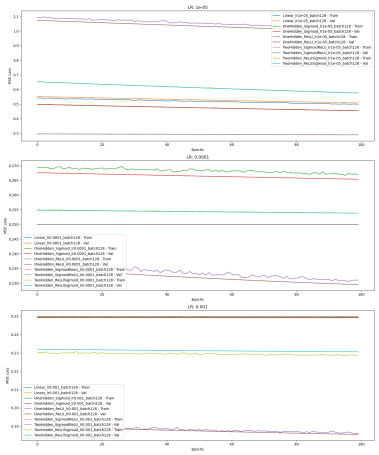


Figure 7: Batch Size 128

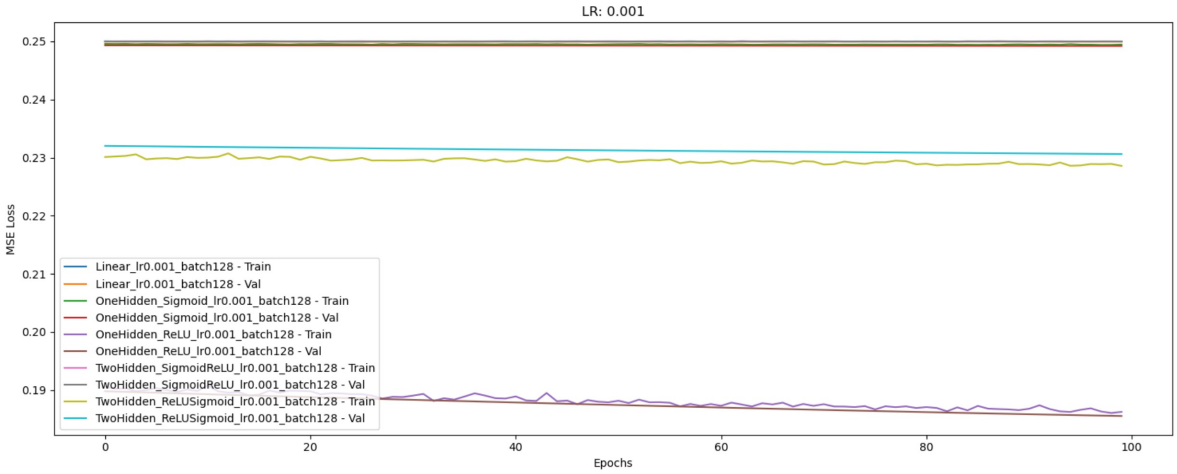


Figure 8: best hyperparameter lr: 0.001, batch size:128, model: Two Hidden Layer Sigmoid ReLU

Part c

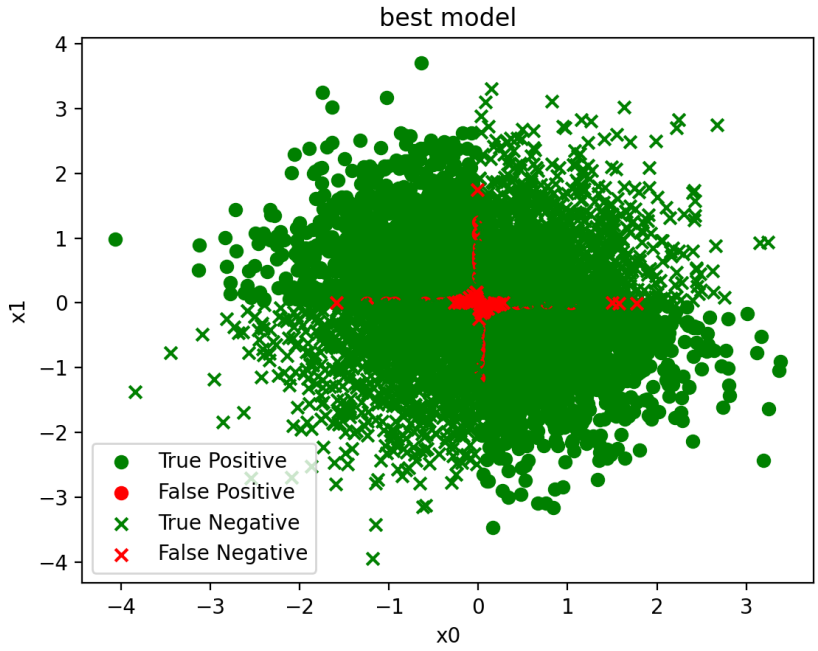


Figure 9: Crossentropy best model accuracy: 0.98

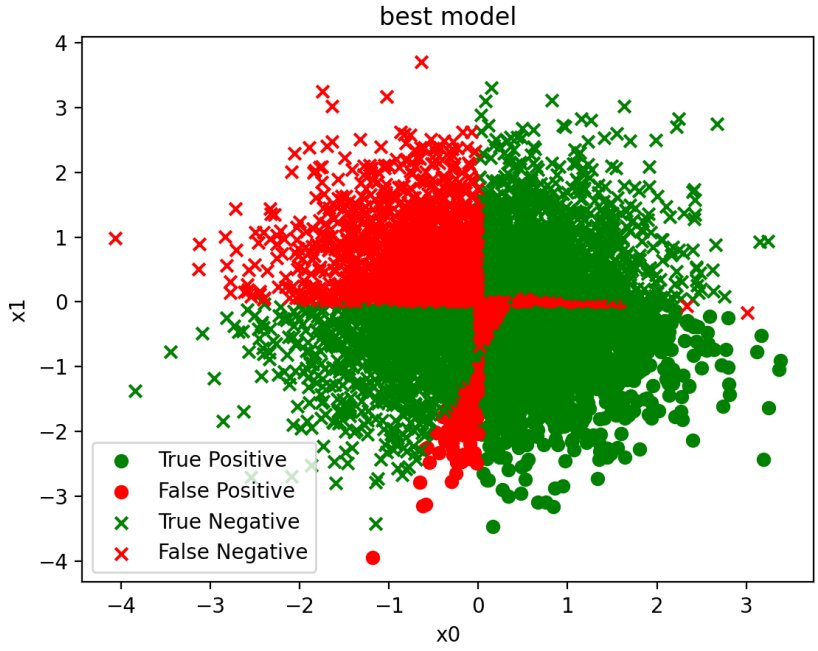


Figure 10: MSE best model accuracy: 0.7

5