Homework #3

CSE 446/546: Machine Learning
Profs. Kevin Jamieson and Ludwig Schmidt
Due: Friday 20th May, 2022 11:59pm
82 points(A)/15 points(B)

Please review all homework guidance posted on the website before submitting to GradeScope. Reminders:

- Make sure to read the "What to Submit" section following each question and include all items.
- Please provide succinct answers and supporting reasoning for each question. Similarly, when discussing experimental results, concisely create tables and/or figures when appropriate to organize the experimental results. All explanations, tables, and figures for any particular part of a question must be grouped together.
- For every problem involving generating plots, please include the plots as part of your PDF submission.
- When submitting to Gradescope, please link each question from the homework in Gradescope to the location of its answer in your homework PDF. **Failure to do so may result in deductions of up to 10% of the points for each problem improperly tagged**. For instructions, see `https://www.gradescope.com/get_started#student-submission`.
- If you collaborate on this homework with others, you must indicate who you worked with on your homework. Failure to do so may result in accusations of plagiarism.
- **The coding problems are available in a .zip file on the course website**, with some starter code. All coding questions in this class will have starter code. **Before attempting these problems make sure your coding environment is working**. See instructions in README file in the .zip file.
- For every problem involving code, please include the code as part of your PDF for the PDF submission *in addition to* submitting your code to the separate assignment on Gradescope created for code.

Not adhering to these reminders may result in point deductions.

Note on Homework Length — This homework may take slightly longer than previous homeworks due to coding implementations (kernel ridge regression and neural networks). Plan accordingly: **please start early** (look over all problems so you can work efficiently), ask questions on Ed, come to office hours, and ask your TAs/instructor if you need any help!

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

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

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

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

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

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

What to Submit:

- a). Decrease σ, that will make the γ larger, so more distance between points.
- b). True, When minimizing a non-convex loss function, the gradient descent might reach the local optimal solution. The gradient can either blow up or go to zero.
- c). False, initial weights should use a random generator.
- d). False, non-linear activation functions introduce non-linearity into the neural network's hidden layers.
- e). False, The time complexities of the backward pass step and the forward pass step are the same.
- f). False, to predicate house price is better to use Linear Regression rather than NN.

Support Vector Machines

A2. Recall that solving the SVM problem amounts to solving the following constrained optimization problem:

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

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

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

Consider the following labeled data points:

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

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

   Note: You do not need to use graphing software to create the drawing. A clear picture of a hand-drawing suffices.

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

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

c. *[4 points]* Show that for any solvable SVM problem, the distance between the two separating hyperplanes is $\frac{2}{||w||_2}$.

   Hint 1: The distance between two hyperplanes is the distance between any point x_0 on one of the hyperplanes and its projection on the other hyperplane.
   Hint 2: A direction w and an offset c define the hyperplane: $H = \{x \in \mathbb{R}^n | w^T x = c\}$. The projection of a vector y onto H is given by $P_H(y) = y - \frac{w^T y - c}{||w||_2^2} w$.

What to Submit:

- a).

- b). $w_1 + 2w_2 + b = -1$, $0 + 0.5w_2 + b = 1$, $w_1 + 0 + b = 1$
  $w^T = [-0.5, -1]$, $b = 1.5$
- c). Proof.

Kernels and Bootstrap

A3. *[5 points]* Suppose that our inputs x are one-dimensional and that our feature map is infinite-dimensional: $\phi(x)$ is a vector whose ith component is:

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

for all nonnegative integers i. (Thus, ϕ is an infinite-dimensional vector.) Show that $K(x, x') = \mathrm{e}^{-\frac{(x-x')^2}{2}}$ is a kernel function for this feature map, i.e.,

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

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

What to Submit:

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

A4. This problem will get you familiar with kernel ridge regression using the polynomial and RBF kernels. First, let's generate some data. Let $n = 30$ and $f_*(x) = 4\sin(\pi x)\cos(6\pi x^2)$. For $i = 1, \ldots, n$ let each x_i be drawn uniformly at random from $[0, 1]$, and let $y_i = f_*(x_i) + \epsilon_i$ where $\epsilon_i \sim \mathcal{N}(0, 1)$. For any function f, the true error and the train error are respectively defined as:

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

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

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

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

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

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

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

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

c. *[5 points]* We wish to build bootstrap percentile confidence intervals for $\widehat{f}_{\text{poly}}(x)$ and $\widehat{f}_{\text{rbf}}(x)$ for all $x \in [0, 1]$ from part b.[2] Use the non-parametric bootstrap with $B = 300$ bootstrap iterations to find 5% and 95% percentiles at each point x on a fine grid over $[0, 1]$.

Specifically, for each bootstrap sample $b \in \{1, \ldots, B\}$, draw uniformly at randomly with replacement, n samples from $\{(x_i, y_i)\}_{i=1}^n$, train an $\widehat{f}_b$ using the b-th resampled dataset, compute $\widehat{f}_b(x)$ for each x in your fine grid; let the 5% percentile at point x be the largest value ν such that $\frac{1}{B}\sum_{b=1}^{B}\mathbf{1}\{\widehat{f}_b(x) \leq \nu\} \leq .05$, define the 95% percentile analogously.

Plot the 5 and 95 percentile curves on the plots from part b.

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

e. *[5 points]* For this problem, use the $\widehat{f}_{\text{poly}}(x)$ and $\widehat{f}_{\text{rbf}}(x)$ learned in part d. Suppose $m = 1000$ additional samples $(x'_1, y'_1), \ldots, (x'_m, y'_m)$ are drawn i.i.d. the same way the first n samples were drawn.

Use the non-parametric bootstrap with $B = 300$ to construct a confidence interval on

$$\mathbb{E}\left[\left(Y - \widehat{f}_{\text{poly}}(X)\right)^2 - \left(Y - \widehat{f}_{\text{rbf}}(X)\right)^2\right]$$

(i.e. randomly draw with replacement m samples denoted as $\{(\widetilde{x}'_i, \widetilde{y}'_i)\}_{i=1}^m$ from $\{(x'_i, y'_i)\}_{i=1}^m$ and compute $\frac{1}{m}\sum_{i=1}^{m}\left[\left(\widetilde{y}'_i - \widehat{f}_{\text{poly}}(\widetilde{x}'_i)\right)^2 - \left(\widetilde{y}'_i - \widehat{f}_{\text{rbf}}(\widetilde{x}'_i)\right)^2\right]$, repeat this B times) and find 5% and 95% percentiles. Report these values.

Using this confidence interval, is there statistically significant 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? (Hint: does the confidence interval contain 0?)

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

[2] See Hastie, Tibshirani, Friedman Ch. 8.2 for a review of the bootstrap procedure.

What to Submit:

- a). best poly lam, best d = 0.000026, 19
  best rbf lam, best gam = 0.002212 , 43.11

- b)

- c)

- d)

- e) The 5% and 95% percentiles = -4756.83615935224 and -4725.732430692829
  It's seems like the poly kernel is too much better rbf.....

Listing 1: A4 code

```python
import math
from typing import Tuple, Union

import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import KFold

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

    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
            To use it simply append .outer to function. For example: np.add.outer, np.divide
    """
    return (1 + x_i.reshape(1, -1).T.dot(x_j.reshape(1, -1))) ** d



@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

    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
            To use it simply append .outer to function. For example: np.add.outer, np.divide
    """
    n = x_i.shape[0]
    m = x_j.shape[0]
    K = np.ndarray(shape=(n, m))
    for a in range(n):
        for b in range(m):
            K[a][b] = 2.718281828459045**(-gamma*pow(x_i[a] - x_j[b], 2))

    return K


@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
        kernel_param (Union[int, float]): Gamma (if kernel_function is rbf_kernel) or d (if
        _lambda (float): Regularization constant.

    Returns:
        np.ndarray: Array of shape (n,) containing alpha hat as described in the pdf.
    """

    return np.linalg.solve(kernel_function(x, x, kernel_param) + (_lambda * np.eye(x.shape[0


@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
    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 f
        kernel_param (Union[int, float]): Gamma (if kernel_function is rbf_kernel) or d (if
        _lambda (float): Regularization constant.
        num_folds (int): Number of folds. It should be either len(x) for LOO, or 10 for 10-f

    Returns:
        float: Average loss of trained function on validation sets across folds.
    """
    fold_size = len(x) // num_folds
    errors = 0

    for fold in range(num_folds):
        start_index = fold*fold_size
        end_index = (fold+1)*fold_size
        X_test = x[start_index:end_index]
        X_train = np.delete(x, np.arange(start_index, end_index, 1))
        Y_text = y[start_index:end_index]
        Y_train = np.delete(y, np.arange(start_index, end_index, 1))
        alpha_h = train(X_train, Y_train, kernel_function, kernel_param, _lambda)
        errors += np.sum(((np.dot(alpha_h, kernel_function(X_train, X_test, kernel_param)) -

    return errors/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
        - Random Search - Fix number of iterations, during each iteration sample lambda from

    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-f
```

```
    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 
            should be sufficient for this problem. That being said you are more than welcome
        - If using random search we recommend sampling lambda from distribution 10**i, where
        - If using grid search we recommend choosing possible lambdas to 10**i, where i=lins
    """
    #raise NotImplementedError("Your Code Goes Here")
    gamma_median = []
    for i in range(len(x)):
        for j in range (i+1, len(x)):
            gamma_median.append(pow(x[i]-x[j],2))
    gamma_median.sort
    gam_max = (gamma_median[len(x) // 2] + gamma_median[~(len(x) // 2)]) / 2

    lambda_array = 10 ** np.linspace(-5, -1, 30)
    gamma_array = np.linspace(0.0001, 1/gam_max, 30)

    error = 10000000
    best_lam = -10000
    best_gam = -10000
    for lam in lambda_array:
        for gam in gamma_array:
            cur_error = cross_validation(x, y, rbf_kernel, gam, lam, num_folds)
            if error > cur_error:
                error = cur_error
                best_lam = lam
                best_gam = gam

    print("rbf_param_search, best_lam: " , best_lam, ", best_gam: ", best_gam, ", error: ",
    return best_lam, best_gam

@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 f

    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-f

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

    Note:
```

```
        - You do not really need to search over gamma. 1 / median((x_i - x_j) for all unique
          should be sufficient for this problem. That being said you are more than welcome
        - If using random search we recommend sampling lambda from distribution 10**i, where
          and d from distribution {7, 8, ..., 20, 21}
        - If using grid search we recommend choosing possible lambdas to 10**i, where i=lins
          and possible ds to [7, 8, ..., 20, 21]
    """

    lambda_array = 10 ** np.linspace(-5, -1, 30)

    error = 10000000
    best_lam = -10000
    best_d = -10000
    for lam in lambda_array:
        for d in range(7, 26):
            cur_error = cross_validation(x, y, poly_kernel, d, lam, num_folds)
            if error > cur_error:
                error = cur_error
                best_lam = lam
                best_d = d

    print("poly_param_search, best_lam: " , best_lam, ", best_d: ", best_d, ", error: ", err
    return best_lam, best_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:
    """Bootstrap function simulation empirical confidence interval of function class.

    For each iteration of bootstrap:
        1. Sample len(x) many of (x, y) pairs with replacement
        2. Train model on these sampled points
        3. Predict values on x_fine_grid (see provided code)

    Lastly after all iterations, calculated 5th and 95th percentiles of predictions for each

    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
        kernel_param (Union[int, float]): Gamma (if kernel_function is rbf_kernel) or d (if
        _lambda (float): Regularization constant.
        bootstrap_iters (int, optional): [description]. Defaults to 300.

    Returns:
        np.ndarray: A (2, 100) numpy array, where each row contains 5 and 95 percentile of

    Note:
```

```
        - See np.percentile function.
            It can take two percentiles at the same time, and take percentiles along specifi
    """
    x_fine_grid = np.linspace(0, 1, 100)
    predictions = np.zeros(shape=(bootstrap_iters, 100))
    res = np.zeros(shape=(2, 100))

    for i in range(bootstrap_iters):
        index = np.random.choice(np.arange(len(x)), len(x), replace=True)
        x_boot = x[index]
        y_boot = y[index]
        alpha_h = train(x_boot, y_boot, kernel_function, kernel_param, _lambda)
        predictions[i, :] = np.dot(alpha_h, kernel_function(x_boot, x_fine_grid, kernel_para
    res[0] = np.array(np.percentile(predictions, 5, axis=0))
    res[1] = np.array(np.percentile(predictions, 95, axis=0))
    return res

@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 fo
        B. For both rbf and poly kernels, train a function using x_30, y_30 and plot predic
        C. For both rbf and poly kernels, plot 5th and 95th percentiles from bootstrap using
        D. Repeat A, B, C with x_300, y_300
        E. Compare rbf and poly kernels using bootstrap as described in the pdf. Report 5 an

    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 majo
            To avoid this call plt.ylim(-6, 6).
    """
    (x_30, y_30), (x_300, y_300), (x_1000, y_1000) = load_dataset("kernel_bootstrap")

    # 4.a ------------------------------------------------------------------------
    #best_poly_lam, best_d = poly_param_search(x_30, y_30, 30)
    #best_rbf_lam, best_gam = rbf_param_search(x_30, y_30, 30)
    # Above code is very slow, I recorded the output, so don't need to run again.
    best_poly_lam, best_d =  2.592943797404667e-05, 19
    best_rbf_lam, best_gam = 0.002212216291070448 , 43.110969187684965


    # 4.b ------------------------------------------------------------------------
    x_fine_grid = np.linspace(0, 1, 100)
    f_poly = train(x_30, y_30, poly_kernel, best_d, best_poly_lam)
    f_poly_predictions = np.dot(f_poly, poly_kernel(x_30, x_fine_grid, best_d))
    plt.figure(1)
    plt.title('4.b: f_poly')
    plt.plot(x_30, y_30, 'ko')
    plt.plot(x_fine_grid, f_true(x_fine_grid))
    plt.plot(x_fine_grid, f_poly_predictions)
    plt.legend(['Data', 'True', 'poly_predictions'])
    plt.show()
```

```
f_rbf = train(x_30, y_30, rbf_kernel, best_gam, best_rbf_lam)
f_rbf_predictions = np.dot(f_rbf, rbf_kernel(x_30, x_fine_grid, best_gam))
plt.figure(2)
plt.title('4.b: f_rbf')
plt.plot(x_30, y_30, 'ko')
plt.plot(x_fine_grid, f_true(x_fine_grid))
plt.plot(x_fine_grid, f_rbf_predictions)
plt.legend(['Data', 'True', 'rbf_predictions'])
plt.show()


# 4.c ----------------------------------------------------------------
x_fine_grid = np.linspace(0, 1, 100)
boot_poly = bootstrap(x_30, y_30, poly_kernel, best_d, best_poly_lam)
plt.figure(3)
plt.title('4.c: boot_poly')
plt.ylim(-15, 20)
plt.plot(x_30, y_30, 'ko')
plt.plot(x_fine_grid, f_true(x_fine_grid))
plt.plot(x_fine_grid, f_poly_predictions)
plt.plot(x_fine_grid, boot_poly[0])
plt.plot(x_fine_grid, boot_poly[1])
plt.legend(['Data', 'True', 'poly_predictions', '5%', '95%'])
plt.show()

rbf_boot = bootstrap(x_30, y_30, rbf_kernel, best_gam, best_rbf_lam)
plt.figure(4)
plt.title('4.c: boot_rbf')
plt.plot(x_30, y_30, 'ko')
plt.plot(x_fine_grid, f_true(x_fine_grid))
plt.plot(x_fine_grid, f_rbf_predictions)
plt.plot(x_fine_grid, rbf_boot[0])
plt.plot(x_fine_grid, rbf_boot[1])
plt.legend(['Data', 'True', 'rbf_predictions', '5%', '95%'])
plt.show()



# 4.d ----------------------------------------------------------------
#best_poly_lam, best_d = poly_param_search(x_300, y_300, 10)
#best_rbf_lam, best_gam = rbf_param_search(x_300, y_300, 10)
# Above code is very very slow, I recorded the output, so don't need to run again.
best_poly_lam, best_d = 1e-05 , 25
best_rbf_lam, best_gam = 1e-05 , 8.534305850573523

x_fine_grid300 = np.linspace(0, 1, 300)
x_fine_grid = np.linspace(0, 1, 100)
f_poly = train(x_300, y_300, poly_kernel, best_d, best_poly_lam)
f_poly_predictions = np.dot(f_poly, poly_kernel(x_300, x_fine_grid300, best_d))
plt.figure(5)
plt.title('4.d: f_poly')
plt.plot(x_300, y_300, 'ko')
plt.plot(x_fine_grid, f_true(x_fine_grid))
plt.plot(x_fine_grid300, f_poly_predictions)
```

```
plt.legend(['Data', 'True', 'poly_predictions'])
plt.show()

f_rbf = train(x_300, y_300, rbf_kernel, best_gam, best_rbf_lam)
f_rbf_predictions = np.dot(f_rbf, rbf_kernel(x_300, x_fine_grid300, best_gam))
plt.figure(6)
plt.title('4.d: f_rbf')
plt.plot(x_300, y_300, 'ko')
plt.plot(x_fine_grid, f_true(x_fine_grid))
plt.plot(x_fine_grid300, f_rbf_predictions)
plt.legend(['Data', 'True', 'rbf_predictions'])
plt.show()


boot_poly = bootstrap(x_300, y_300, poly_kernel, best_d, best_poly_lam)
plt.figure(7)
plt.title('4.d: boot_poly')
plt.plot(x_300, y_300, 'ko')
plt.plot(x_fine_grid, f_true(x_fine_grid))
plt.plot(x_fine_grid300, f_poly_predictions)
plt.plot(x_fine_grid, boot_poly[0])
plt.plot(x_fine_grid, boot_poly[1])
plt.legend(['Data', 'True', 'poly_predictions', '5%', '95%'])
plt.show()

rbf_boot = bootstrap(x_300, y_300, rbf_kernel, best_gam, best_rbf_lam)
plt.figure(8)
plt.title('4.d: boot_rbf')
plt.plot(x_300, y_300, 'ko')
plt.plot(x_fine_grid, f_true(x_fine_grid))
plt.plot(x_fine_grid300, f_rbf_predictions)
plt.plot(x_fine_grid, rbf_boot[0])
plt.plot(x_fine_grid, rbf_boot[1])
plt.legend(['Data', 'True', 'rbf_predictions', '5%', '95%'])
plt.show()



# 4.e ----------------------------------------------------------------
x_fine_grid = np.linspace(0, 1, 100)
bootstrap_iters = 300
errors = np.zeros(bootstrap_iters)

for i in range(bootstrap_iters):
    index = np.random.choice(np.arange(len(x_1000)), len(x_1000), replace=True)
    x_boot = x_1000[index]
    y_boot = y_1000[index]
    poly_error = 0
    rbf_error = 0
    for j in range(1000):
        v = np.zeros(shape=(1,))
        v[0] = x_boot[j]
        poly_error += (y_boot[j] - np.dot(f_poly, poly_kernel(x_300, v, best_poly_lam)))
        rbf_error += (y_boot[j] - np.dot(f_rbf, rbf_kernel(x_300, v, best_rbf_lam)))**2/
    errors[i] = (poly_error - rbf_error) / 1000
```

```
    boot_5 = np.array(np.percentile(errors, 5))
    boot_95 = np.array(np.percentile(errors, 95))
    print("4.e: boot_error")
    print("boot_5: ", boot_5, ", boot_95: ", boot_95)

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

Introduction to PyTorch

A5. PyTorch is a great tool for developing, deploying and researching neural networks and other gradient-based algorithms. In this problem we will explore how this package is built, and re-implement some of its core components. Firstly start by reading `README.md` file provided in `intro_pytorch` subfolder. More detailed specification is provided in problem README and docstring comments for each function.

a. *[10 points]* You will start by implementing components of our own PyTorch modules. You can find these in folders: `layers`, `losses` and `optimizers`. Almost each file there should contain at least one problem function, including exact directions for what to achieve in this problem. Lastly, you should implement functions in `train.py` file.

b. *[5 points]* Next we will use the above module to perform hyperparameter search. Here we will also treat loss function as a hyper-parameter. However, because cross-entropy and MSE require different shapes we are going to use two different files: `crossentropy_search.py` and `mean_squared_error_search.py`. For each you will need to build and train (in provided order) 5 models:

   - Linear neural network (Single layer, no activation function)
   - NN with one hidden layer (2 units) and sigmoid activation function after the hidden layer
   - NN with one hidden layer (2 units) and ReLU activation function after the hidden layer
   - NN with two hidden layer (each with 2 units) and Sigmoid, ReLU activation functions after first and second hidden layers, respectively
   - NN with two hidden layer (each with 2 units) and ReLU, Sigmoid activation functions after first and second hidden layers, respectively

   For each loss function, submit a plot of losses from training and validation sets. All models should be on the same plot (10 lines per plot), with two plots total (1 for MSE, 1 for cross-entropy).

c. *[5 points]* For each loss function, report the best performing architecture (best performing is defined here as achieving the lowest validation loss at any point during the training), and plot its guesses on test set. You should use function `plot_model_guesses` from `train.py` file. Lastly, report accuracy of that model on a test set.

On Softmax function

One of the activation functions we ask you to implement is softmax. For a prediction $\hat{y} \in \mathbb{R}^k$ corresponding to single datapoint (in a problem with k classes):

$$\text{softmax}(\hat{y}_i) = \frac{\exp(\hat{y}_i)}{\sum_j \exp(\hat{y}_j)}$$

What to Submit:

- b)

- c) The best MSE performing model is NN with two hidden layer (each with 2 units) and ReLU, Sigmoid. Accuracy is 0.9018

The best CE performing model is NN with two hidden layer (each with 2 units) and Sigmoid, ReLU. Accuracy is 0.6258

Listing 2: A4 code

```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 provide
        You should use torch.randn function to initialize them by normal distribution, and p

        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 b

        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
                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) i
            - Make use of pytorch documentation: https://pytorch.org/docs/stable/index.html
        """
        super().__init__()
        self.weight = torch.nn.Parameter(torch.randn(dim_in, dim_out, generator=generator))
        self.bias = torch.nn.Parameter(torch.randn(dim_out, generator=generator))

    @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_i
                Input data.

        Returns:
            torch.Tensor: More specifically a torch.FloatTensor, with shape of (n, dim_out).
                Output data.
        """
        return x.float() @ 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
            - Make use of pytorch documentation: https://pytorch.org/docs/stable/index.html
        """
        return(x * (x > 0))
```

```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) i
                YOU CAN however use other aliases of sigmoid function in PyTorch if you are
```

```
        - Make use of pytorch documentation: https://pytorch.org/docs/stable/index.html
    """
    return(1 / (1. + torch.exp(-x)))


import torch
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
                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) s
                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)
            - Make use of pytorch documentation: https://pytorch.org/docs/stable/index.html
        """
        nonmaxed_x = x - torch.max(x)
        e = torch.exp(nonmaxed_x)
        e_sum = torch.sum(e, 1, keepdim=True)

        return e/e_sum

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
```

```
        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 r
                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 shap
                Should be a mean over all examples.
                Result.

        Note:
            - YOU ARE NOT ALLOWED to use torch.nn.CrossEntropyLoss / torch.nn.NLLLoss (or th
            - 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 perfo
        """
        return(torch.mean(-torch.log(y_pred.gather(1, y_true.view(-1,1)))))


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 shap
                Input data.

        Returns:
            torch.Tensor: More specifically a SINGLE VALUE torch.FloatTensor (i.e. with shap
                Result.

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

```python
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 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 calcu
        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 ep
            Defaults to None.
        epochs (int, optional): Number of epochs (passes through dataset/dataloader) to trai
            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 correspond
            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
            Using a running loss approach is advised.
            In this case you will just use the loss that you called .backward() on add sum t
            Then you can divide by length of train_loader, and you will have an average loss
        - 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 valida
            You might want to look into state_dict: https://pytorch.org/tutorials/beginner/s
    """
    #raise NotImplementedError("Your Code Goes Here")
    train_list = []
    val_list = []
```

```
    for epoch in range(epochs):
        train_loss = 0
        for X_train, y_train in train_loader:
            optimizer.zero_grad()
            prediction = model(X_train)
            loss = criterion(prediction, y_train)
            train_loss += loss.item()
            loss.backward()
            optimizer.step()
        avg_loss = train_loss/len(train_loader)
        train_list.append(avg_loss)

        if val_loader is not None:
            val_loss = 0
            with torch.no_grad():
                for X_val, y_val in val_loader:
                    prediction = model(X_val)
                    loss = criterion(prediction, y_val)
                    val_loss += loss.item()
            avg_loss = val_loss/len(val_loader)
            val_list.append(avg_loss)

    return {"train": train_list, "val": val_list}

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 containin

    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
        - Network with one hidden layer of size 2 and ReLU activation function after the hid
        - 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 requiremen

    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 typic
            Make sure it is not too big as you can end up with standard (or almost) gradient

    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 foll
            {
                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")
    dataloader_train = DataLoader(
        dataset_train, batch_size=32, shuffle=True, generator=RNG,
    )
    dataloader_val = DataLoader(
        dataset_val, batch_size=32, shuffle=True,
    )

    lin = nn.Sequential(
        LinearLayer(2, 2, generator = RNG),
        SoftmaxLayer()
    )
    nn1_sig = nn.Sequential(
```

```
        LinearLayer(2, 2, generator = RNG),
        SigmoidLayer(),
        LinearLayer(2, 2, generator = RNG),
        SoftmaxLayer()
    )
    nn1_relu = nn.Sequential(
        LinearLayer(2, 2, generator = RNG),
        ReLULayer(),
        LinearLayer(2, 2, generator = RNG),
        SoftmaxLayer()
    )
    nn2_sig_relu = nn.Sequential(
        LinearLayer(2, 2, generator = RNG),
        SigmoidLayer(),
        LinearLayer(2, 2, generator = RNG),
        ReLULayer(),
        LinearLayer(2, 2, generator = RNG),
        SoftmaxLayer()
    )
    nn2_relu_sig = nn.Sequential(
        LinearLayer(2, 2, generator = RNG),
        ReLULayer(),
        LinearLayer(2, 2, generator = RNG),
        SigmoidLayer(),
        LinearLayer(2, 2, generator = RNG),
        SoftmaxLayer()
    )
    models = [["lin", lin],
              ["nn1_sig", nn1_sig],
              ["nn1_relu", nn1_relu],
              ["nn2_sig_relu", nn2_sig_relu],
              ["nn2_relu_sig", nn2_relu_sig]]
    res = {}

    for model in models:
        print("model: ", model[0])
        loss = CrossEntropyLossLayer()
        optimizer = SGDOptimizer(model[1].parameters(), lr=1e-2)
        res[model[0]] = [train(dataloader_train, model[1], loss, optimizer, dataloader_val),
    return res


@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/datal
```

```
        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.
    """
    #raise NotImplementedError("Your Code Goes Here")
    accuracy = 0
    for X, y in dataloader:
        prediction = model(X)
        y_prediction = prediction.argmax(dim=1)
        accuracy += torch.sum(y_prediction == y).item()

    return accuracy/len(dataloader.dataset)


@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 arc
        2. Plot Train and Validation losses for each model all on single plot (it should be
            x-axis should be epochs, y-axis should me Crossentropy loss, REMEMBER to add leg
        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
        4. Plot best model guesses on test set (using plot_model_guesses function from train
        5. Report accuracy of the model on test set.

    Starter code loads dataset, converts it into PyTorch Datasets, and those into DataLoade
    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))

    ce_configs = crossentropy_parameter_search(dataset_train, dataset_val)
    #raise NotImplementedError("Your Code Goes Here")

    dataloader_test = DataLoader(
        dataset_test, batch_size=32, shuffle=True,
    )

    for model in ce_configs.keys():
        accuracy = accuracy_score(ce_configs[model][1], dataloader_test)
        print("Accuracy for", model, "is:", accuracy)

        plt.plot(ce_configs[model][0]["train"], label=model + "_train")
        plt.plot(ce_configs[model][0]["val"], label=model + "_val")


    plt.title("Losses for CE Search")
```

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

    plot_model_guesses(dataloader_test, ce_configs["nn2_relu_sig"][1], model)


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, whi
            Index of 1.0 in target corresponds to the true class.

    Returns:
        float: Vanilla python float respreting accuracy of the model on given dataset/datal
            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")
    accuracy = 0
    for X, y in dataloader:
        prediction = model(X)
        y_prediction = prediction.argmax(dim=1)
        y_label = y.argmax(dim=1)
        accuracy += torch.sum(y_prediction == y_label).item()
    return accuracy/len(dataloader.dataset)


@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 containin

    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
        - Network with one hidden layer of size 2 and ReLU activation function after the hid
        - 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 typic
            Make sure it is not too big as you can end up with standard (or almost) gradient

    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 foll
            {
                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")
```

```
    dataloader_train = DataLoader(
        dataset_train, batch_size=32, shuffle=True, generator=RNG,
    )
    dataloader_val = DataLoader(
        dataset_val, batch_size=32, shuffle=True,
    )

    lin = nn.Sequential(
        LinearLayer(2, 2, generator = RNG)
    )
    nn1_sig = nn.Sequential(
        LinearLayer(2, 2, generator = RNG),
        SigmoidLayer(),
        LinearLayer(2, 2, generator = RNG)
    )
    nn1_relu = nn.Sequential(
        LinearLayer(2, 2, generator = RNG),
        ReLULayer(),
        LinearLayer(2, 2, generator = RNG)
    )
    nn2_sig_relu = nn.Sequential(
        LinearLayer(2, 2, generator = RNG),
        SigmoidLayer(),
        LinearLayer(2, 2, generator = RNG),
        ReLULayer(),
        LinearLayer(2, 2, generator = RNG)
    )
    nn2_relu_sig = nn.Sequential(
        LinearLayer(2, 2, generator = RNG),
        ReLULayer(),
        LinearLayer(2, 2, generator = RNG),
        SigmoidLayer(),
        LinearLayer(2, 2, generator = RNG)
    )
    models = [["lin", lin],
              ["nn1_sig", nn1_sig],
              ["nn1_relu", nn1_relu],
              ["nn2_sig_relu", nn2_sig_relu],
              ["nn2_relu_sig", nn2_relu_sig]]
    res = {}


    for model in models:
        print("model: ", model[0])
        loss = MSELossLayer()
        optimizer = SGDOptimizer(model[1].parameters(), lr=1e-2)
        res[model[0]] = [train(dataloader_train, model[1], loss, optimizer, dataloader_val),
    return res



@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/
        2. Plot Train and Validation losses for each model all on single plot (it should be
            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
        4. Plot best model guesses on test set (using plot_model_guesses function from train
        5. Report accuracy of the model on test set.

    Starter code loads dataset, converts it into PyTorch Datasets, and those into DataLoade
    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))
    )

    mse_configs = mse_parameter_search(dataset_train, dataset_val)
    #raise NotImplementedError("Your Code Goes Here")
    dataloader_test = DataLoader(
        dataset_test, batch_size=32, shuffle=True,
    )


    for model in mse_configs.keys():
        accuracy = accuracy_score(mse_configs[model][1], dataloader_test)
        print("Accuracy for", model, "is:", accuracy)
        plt.plot(mse_configs[model][0]["train"], label=model + "_train")
        plt.plot(mse_configs[model][0]["val"], label=model + "_val")

    plt.title("Losses for MSE Search")
    plt.xlabel("Epoches")
    plt.ylabel("Loss")
    plt.legend()
    plt.show()

    plot_model_guesses(dataloader_test, mse_configs["nn2_relu_sig"][1], model)




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


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

Administrative

A6.

a. *[2 points]* About how many hours did you spend on this homework? There is no right or wrong answer :)
27 hours