000-FunctionDataset

June 9, 2024

[1]:
```
import numpy as np
import torch
from torch.utils.data import Dataset, DataLoader
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

class FunctionDataset(Dataset):
    def __init__(self, num_samples, input_dim, function_class='linear',␣
↪noise_std=0.0):
        """
        Initialize the dataset with the given parameters.

        Args:
            num_samples (int): Number of samples to generate.
            input_dim (int): Dimensionality of the input data.
            function_class (str): Type of function to generate ('linear',␣
↪'two_layer_nn', 'k_sparse_linear', 'decision_tree').
            noise_std (float): Standard deviation of the noise to add to the␣
↪function outputs.
        """
        self.num_samples = num_samples
        self.input_dim = input_dim
        self.function_class = function_class
        self.noise_std = noise_std
        self.data, self.weights = self._generate_data()

    def _generate_data(self):
        """
        Generate data based on the specified function class.

        Returns:
            data (list): List of (x, y) pairs.
            weights (ndarray or tuple): Weights or parameters of the generated␣
↪function.
        """
        data = []
```

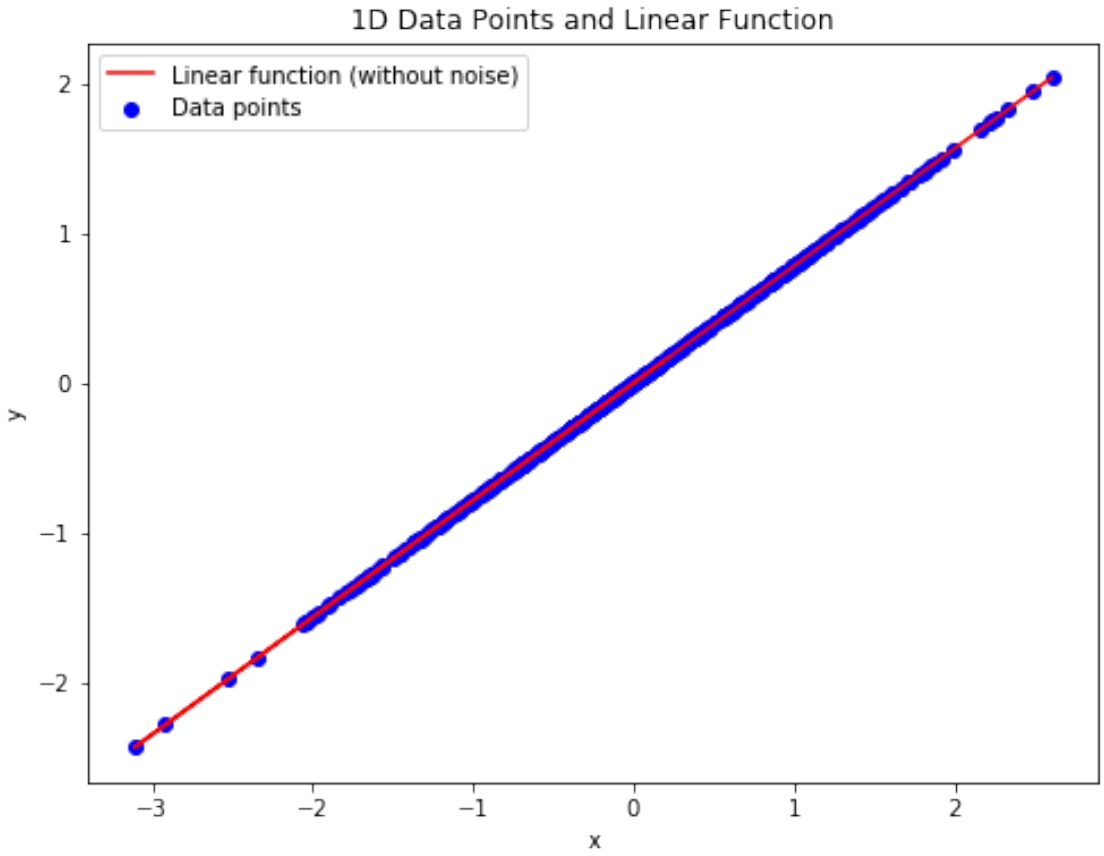
```
        if self.function_class == 'linear':
            # Generate weights for a linear function and define the function
            self.weights = np.random.normal(0, 1, self.input_dim)
            func = lambda x: self.weights @ x

        elif self.function_class == 'two_layer_nn':
            # Generate weights for a two-layer neural network and define the␣
↪function
            hidden_units = 100
            W1 = np.random.normal(0, 1, (hidden_units, self.input_dim))
            W2 = np.random.normal(0, 1, hidden_units)
            self.weights = (W1, W2)
            func = lambda x: W2 @ np.maximum(0, W1 @ x)

        elif self.function_class == 'k_sparse_linear':
            # Generate weights for a k-sparse linear function and define the␣
↪function
            k = 3
            self.weights = np.random.normal(0, 1, self.input_dim)
            sparse_indices = np.random.choice(self.input_dim, k, replace=False)
            sparse_w = np.zeros_like(self.weights)
            sparse_w[sparse_indices] = self.weights[sparse_indices]
            self.weights = sparse_w
            func = lambda x: sparse_w @ x

        elif self.function_class == 'decision_tree':
            # Generate a decision tree and define the function
            def generate_tree(depth):
                if depth == 0:
                    return np.random.normal(0, 1)
                feature = np.random.randint(0, self.input_dim)
                threshold = np.random.normal(0, 1)
                left = generate_tree(depth-1)
                right = generate_tree(depth-1)
                return feature, threshold, left, right

            tree = generate_tree(3)
            self.weights = tree

            def evaluate_tree(tree, x):
                if not isinstance(tree, tuple):
                    return tree
                feature, threshold, left, right = tree
                if x[feature] > threshold:
                    return evaluate_tree(right, x)
                else:
                    return evaluate_tree(left, x)
```

```
            func = lambda x: evaluate_tree(tree, x)

        else:
            raise NotImplementedError(f"Function class {self.function_class} is␣
↪not implemented")

        # Generate the data samples
        for _ in range(self.num_samples):
            x = np.random.normal(0, 1, self.input_dim)
            y = func(x) + np.random.normal(0, self.noise_std)
            data.append((x, y))

        return data, self.weights

    def __len__(self):
        """
        Return the total number of samples.

        Returns:
            int: Number of samples in the dataset.
        """
        return len(self.data)

    def __getitem__(self, idx):
        """
        Retrieve a single sample from the dataset.

        Args:
            idx (int): Index of the sample to retrieve.

        Returns:
            tuple: (x, y) where x is the input tensor and y is the output␣
↪tensor.
        """
        x, y = self.data[idx]
        return torch.tensor(x, dtype=torch.float32), torch.tensor(y,␣
↪dtype=torch.float32)

if __name__ == "__main__":
    # Example with 1 dimension
    dataset_1d = FunctionDataset(num_samples=500, input_dim=1,␣
↪function_class='linear', noise_std=0.0)
    dataloader_1d = DataLoader(dataset_1d, batch_size=1, shuffle=False)

    print("1D example prompts:")
    print("Weights:", dataset_1d.weights)
```

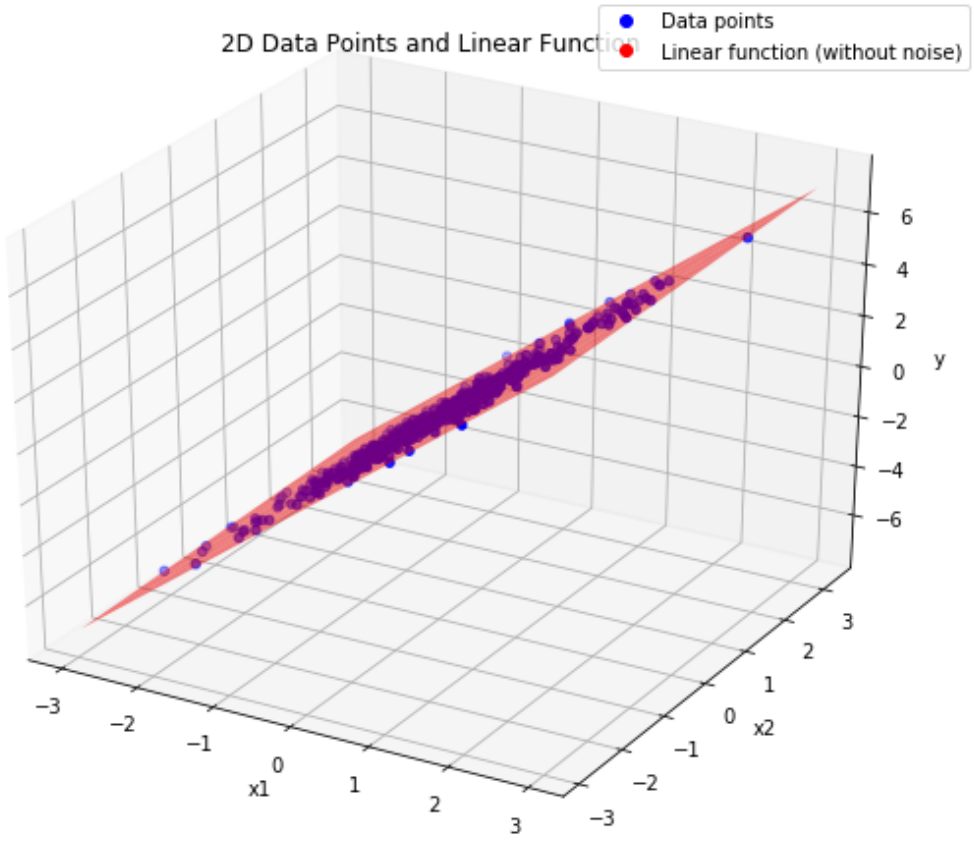
```
    for i, (x, y) in enumerate(dataloader_1d):
        if i >= 3:
            break
        print(f"Prompt {i+1}: x = {x.numpy().flatten()}, y = {y.item()}")
        computed_y = dataset_1d.weights @ x.numpy().flatten()
        print(f"Computed y (without noise): {computed_y}")

    # Plotting 1D example
    x_data_1d = np.array([x.numpy().flatten() for x, y in dataloader_1d])
    y_data_1d = np.array([y.item() for x, y in dataloader_1d])
    weights_1d = dataset_1d.weights

    plt.figure(figsize=(8, 6))
    plt.scatter(x_data_1d, y_data_1d, c='blue', label='Data points')
    plt.plot(x_data_1d, x_data_1d * weights_1d[0], color='red', label='Linear␣
↪function (without noise)')

    plt.xlabel('x')
    plt.ylabel('y')
    plt.title('1D Data Points and Linear Function')
    plt.legend()
    plt.show()

    # Example with 2 dimensions
    dataset_2d = FunctionDataset(num_samples=500, input_dim=2,␣
↪function_class='linear', noise_std=0.0)
    dataloader_2d = DataLoader(dataset_2d, batch_size=1, shuffle=False)

    print("\n2D example prompts:")
    print("Weights:", dataset_2d.weights)
    for i, (x, y) in enumerate(dataloader_2d):
        if i >= 3:
            break
        print(f"Prompt {i+1}: x = {x.numpy().flatten()}, y = {y.item()}")
        computed_y = dataset_2d.weights @ x.numpy().flatten()
        print(f"Computed y (without noise): {computed_y}")

    # Plotting 2D example
    x_data_2d = np.array([x.numpy().flatten() for x, y in dataloader_2d])
    y_data_2d = np.array([y.item() for x, y in dataloader_2d])
    weights_2d = dataset_2d.weights

    fig = plt.figure(figsize=(10, 8))
    ax = fig.add_subplot(111, projection='3d')

    ax.scatter(x_data_2d[:, 0], x_data_2d[:, 1], y_data_2d, c='blue',␣
↪label='Data points')
```

```
    # Create a grid of points
    x1_range = np.linspace(min(x_data_2d[:, 0]), max(x_data_2d[:, 0]), 10)
    x2_range = np.linspace(min(x_data_2d[:, 1]), max(x_data_2d[:, 1]), 10)
    x1_grid, x2_grid = np.meshgrid(x1_range, x2_range)
    y_grid = weights_2d[0] * x1_grid + weights_2d[1] * x2_grid

    ax.plot_surface(x1_grid, x2_grid, y_grid, color='red', alpha=0.5)

    ax.set_xlabel('x1')
    ax.set_ylabel('x2')
    ax.set_zlabel('y')
    ax.set_title('2D Data Points and Linear Function')

    # Manually add legend
    scatter_proxy = plt.Line2D([0], [0], linestyle="none", c='blue', marker='o')
    surface_proxy = plt.Line2D([0], [0], linestyle="none", c='red', marker='o')
    ax.legend([scatter_proxy, surface_proxy], ['Data points', 'Linear function␣
↪(without noise)'])

    plt.show()
```

```
1D example prompts:
Weights: [0.78176963]
Prompt 1: x = [0.21461792], y = 0.1677817702293396
Computed y (without noise): 0.16778177413242068
Prompt 2: x = [-1.5935533], y = -1.2457915544509888
Computed y (without noise): -1.2457915770363899
Prompt 3: x = [1.0617096], y = 0.830012321472168
Computed y (without noise): 0.8300123540863203
```




```
2D example prompts:
Weights: [2.32734572 0.04548628]
Prompt 1: x = [-0.8279254  2.0867457], y = -1.831950306892395
Computed y (without noise): -1.8319502911629186
Prompt 2: x = [-0.6176249 -1.6522378], y = -1.5125807523727417
Computed y (without noise): -1.5125807701371203
Prompt 3: x = [-1.6134185  -0.04637234], y = -3.757091760635376
Computed y (without noise): -3.757091847607
```


2D Data Points and Linear Function
Data points
Linear function (without noise)
x1
x2
y

[ ]: