```python
import torch
from torch import nn
from torch import optim
import numpy as np
import data
import matplotlib.pyplot as plt
from typing import List, Callable
from sklearn.model_selection import train_test_split
from copy import deepcopy
import math
import time

class PTDeep(nn.Module):

    def __init__(self, dimensions: List[int], activations: List[Callable], param_lambda=0.0, use_cuda=False):
        super(PTDeep, self).__init__()
        device = "cuda" if use_cuda is True else "cpu"
        D = dimensions[0]
        C = dimensions[-1]
        weights = []
        biases = []
        for i in range(len(dimensions) - 1):
            dim_in = dimensions[i]
            dim_out = dimensions[i+1]
            W = np.random.randn(dim_in, dim_out).astype(np.float64)
            b = np.random.randn(1, dim_out).astype(np.float64)
            weights.append(nn.Parameter(data=torch.from_numpy(W).to(device=device), requires_grad=True))
            biases.append(nn.Parameter(data=torch.from_numpy(b).to(device=device), requires_grad=True))

        self.weights = nn.ParameterList(weights)
        self.biases = nn.ParameterList(biases)
        self.activations = activations
        self.param_lambda = param_lambda
        self.loss_trace = None

    def forward(self, X):
        h = X
        for i in range(len(self.weights)):
            scores = torch.mm(h, self.weights[i]) + self.biases[i]
            h = self.activations[i](scores)

        return h

    def get_loss(self, X, Yoh_):
        N = len(X)
        probs = self.forward(X)
        logprobs = torch.log(probs)
        loss = - 1 / N * torch.sum(logprobs * Yoh_)
        return loss

    def train(self, X, Yoh_, param_niter=10_000, param_delta=0.05, print_frequency=200,
              epsilon=1e-3, trace=True, early_stopping=False):
        optimizer = optim.SGD(self.parameters(), lr=param_delta, weight_decay=self.param_lambda)
        best_validation_loss = None
        if trace is True:
            self.loss_trace = []
        if early_stopping is True:
            best_weights = None
            best_biases = None
            x_train, x_validate, y_train, y_validate = train_test_split(X, Yoh_, test_size=0.2)

        start = time.time()
        for i in range(param_niter):
            # calculate loss
            loss = self.get_loss(x_train, y_train)
            # if trace is true, remember this loss
```

```python
            if trace is True:
                self.loss_trace.append(loss.detach().numpy())
            # periodically print status
            if i % print_frequency == 0:
                end = time.time()
                print(f"iteration {i} loss {loss} duration {end-start}")
            # calculate the gradients and adjust the parameters
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()
            # if early stopping is on, validate and remember best models parameters
            if early_stopping is True:
                validation_loss = self.get_loss(x_validate, y_validate)
                optimizer.zero_grad()
                if best_validation_loss is None or validation_loss < best_validation_loss:
                    best_validation_loss = validation_loss
                    best_weights = deepcopy(self.weights)
                    best_biases = deepcopy(self.biases)

        if early_stopping is True:
            self.weights = best_weights
            self.biases = best_biases

        return self

    def train_mb(self, X, Yoh_,
             param_niter=10_000, param_delta=0.05, print_frequency=200, batch_size=128,
             trace=True, early_stopping=False, optimizer=None, scheduler=None):
        if optimizer is None:
            optimizer = optim.SGD(self.parameters(), lr=param_delta, weight_decay=self.param_lambda)

        best_validation_loss = None
        if trace is True:
            self.loss_trace = []
        if early_stopping is True:
            best_weights = None
            best_biases = None
            x_train, x_validate, y_train, y_validate = train_test_split(X, Yoh_, test_size=0.2)
        else:
            x_train, y_train = X, Yoh_

        start = time.time()
        for i in range(param_niter):
            # shuffle and create batches
            p = np.random.permutation(len(x_train))
            x_train = x_train[p]
            y_train = y_train[p]
            number_of_batches = int(math.ceil(len(x_train) / batch_size))
            # iterate over all batches
            for j in range(number_of_batches):
                x_batch = x_train[(j*batch_size):((j+1)*batch_size)]
                y_batch = y_train[(j*batch_size):((j+1)*batch_size)]
                loss = self.get_loss(x_batch, y_batch)
                # if trace is true, remember this loss
                if trace is True:
                    self.loss_trace.append(loss.detach().numpy())
                # periodically print status
                if i % print_frequency == 0 and j % print_frequency == 0:
                    end = time.time()
                    print(f"iteration {i} batch {j} loss {loss} duration {end-start}")
                # calculate the gradients and adjust the parameters
                loss.backward()
                optimizer.step()
                optimizer.zero_grad()

            # if scheduler is given, step it up
```

```python
            if scheduler is not None:
                scheduler.step()
            # if early stopping is on, validate and remember best models parameters
            if early_stopping is True:
                validation_loss = self.get_loss(x_validate, y_validate)
                optimizer.zero_grad()
                if best_validation_loss is None or validation_loss < best_validation_loss:
                    best_validation_loss = validation_loss
                elif validation_loss > best_validation_loss:
                    print("Validation loss is going up! Ending training.")
                    break

        return self


    def eval(self, X):
        X = torch.from_numpy(X)
        probs = self.forward(X)
        return probs.detach().numpy()


def my_softmax(X):
    return torch.softmax(X, 1)


def count_params(model: nn.Module):
    count = 0
    for name, param in model.named_parameters():
        print(f"{name}: {param.shape}")
        count += param.shape[0] * param.shape[1]
    print("number of params:", count)


if __name__ == "__main__":
    np.random.seed(101)
    X, Y_ = data.sample_gmm_2d(4, 2, 40)
    Yoh_ = data.class_to_onehot(Y_)

    model = PTDeep([2, 10, 2], [torch.sigmoid, my_softmax])
    model.train(torch.from_numpy(X), torch.from_numpy(Yoh_), 10_000, 0.05, 200)
    probs = model.eval(X)
    # predicted classes
    Y = np.hstack([np.argmax(probs[i][:]) for i in range(probs.shape[0])])
    # reshaping for other methods purposes
    Y_ = np.hstack(Y_)
    accuracy, pr, M = data.eval_perf_multi(Y, Y_)
    print("Accuracy: ", accuracy)
    print("Precision / Recall: ", pr)
    print("Confussion Matrix: ", M)
    print("All parameters:")
    count_params(model)

    def decfun(X):
        X = torch.from_numpy(X)
        to_return = model.forward(X)
        return to_return.detach().numpy().argmax(axis=1)

    # decfun za binlogreg
    decfun = lambda X: model.forward(torch.from_numpy(X)).detach().numpy()[:, 1]
    bbox = (np.min(X, axis=0), np.max(X, axis=0))
    data.graph_surface(decfun, bbox, offset=0.5)
    # graph the data points
    data.graph_data(X, Y_, Y, special=[])
    # show the plot
    plt.show()
```