```python
import random

import numba

import minitorch

datasets = minitorch.datasets
FastTensorBackend = minitorch.TensorBackend(minitorch.FastOps)
if numba.cuda.is_available():
    GPUBackend = minitorch.TensorBackend(minitorch.CudaOps)


def default_log_fn(epoch, total_loss, correct, losses):
    print("Epoch ", epoch, " loss ", total_loss, "correct", correct)


def RParam(*shape, backend):
    r = minitorch.rand(shape, backend=backend) - 0.5
    return minitorch.Parameter(r)


class Network(minitorch.Module):
    def __init__(self, hidden, backend):
        super().__init__()

        # Submodules
        self.layer1 = Linear(2, hidden, backend)
        self.layer2 = Linear(hidden, hidden, backend)
        self.layer3 = Linear(hidden, 1, backend)

    def forward(self, x):
        # ASSIGN3.5
        h = self.layer1.forward(x).relu()
        h = self.layer2.forward(h).relu()
        return self.layer3.forward(h).sigmoid()
        # END ASSIGN3.5


class Linear(minitorch.Module):
    def __init__(self, in_size, out_size, backend):
        super().__init__()
        self.weights = RParam(in_size, out_size, backend=backend)
        s = minitorch.zeros((out_size,), backend=backend)
        s = s + 0.1
        self.bias = minitorch.Parameter(s)
        self.out_size = out_size

    def forward(self, x):
        # ASSIGN3.5
        batch, in_size = x.shape
        return x.view(batch, in_size) @ self.weights.value + self.bias.value
        # END ASSIGN3.5


class FastTrain:
    def __init__(self, hidden_layers, backend=FastTensorBackend):
        self.hidden_layers = hidden_layers
        self.model = Network(hidden_layers, backend)
        self.backend = backend

    def run_one(self, x):
        return self.model.forward(minitorch.tensor([x], backend=self.backend))

    def run_many(self, X):
        return self.model.forward(minitorch.tensor(X, backend=self.backend))

    def train(self, data, learning_rate, max_epochs=500, log_fn=default_log_fn):

        self.model = Network(self.hidden_layers, self.backend)
        optim = minitorch.SGD(self.model.parameters(), learning_rate)
        BATCH = 10
        losses = []

        for epoch in range(max_epochs):
            total_loss = 0.0
            c = list(zip(data.X, data.y))
            random.shuffle(c)
            X_shuf, y_shuf = zip(*c)

            for i in range(0, len(X_shuf), BATCH):
                optim.zero_grad()
                X = minitorch.tensor(X_shuf[i : i + BATCH], backend=self.backend)
                y = minitorch.tensor(y_shuf[i : i + BATCH], backend=self.backend)
                # Forward

                out = self.model.forward(X).view(y.shape[0])
                prob = (out * y) + (out - 1.0) * (y - 1.0)
                loss = -prob.log()
                (loss / y.shape[0]).sum().view(1).backward()

                total_loss = loss.sum().view(1)[0]

                # Update
                optim.step()

            losses.append(total_loss)
            # Logging
            if epoch % 10 == 0 or epoch == max_epochs:
                X = minitorch.tensor(data.X, backend=self.backend)
                y = minitorch.tensor(data.y, backend=self.backend)
                out = self.model.forward(X).view(y.shape[0])
                y2 = minitorch.tensor(data.y)
                correct = int(((out.detach() > 0.5) == y2).sum()[0])
                log_fn(epoch, total_loss, correct, losses)


if __name__ == "__main__":
    import argparse

    parser = argparse.ArgumentParser()
    parser.add_argument("--PTS", type=int, default=50, help="number of points")
    parser.add_argument("--HIDDEN", type=int, default=10, help="number of hiddens")
    parser.add_argument("--RATE", type=float, default=0.05, help="learning rate")
    parser.add_argument("--BACKEND", default="cpu", help="backend mode")
    parser.add_argument("--DATASET", default="simple", help="dataset")
    parser.add_argument("--PLOT", default=False, help="dataset")

    args = parser.parse_args()

    PTS = args.PTS

    if args.DATASET == "xor":
        data = minitorch.datasets["Xor"](PTS)
    elif args.DATASET == "simple":
        data = minitorch.datasets["Simple"].simple(PTS)
    elif args.DATASET == "split":
        data = minitorch.datasets["Split"](PTS)

    HIDDEN = int(args.HIDDEN)
    RATE = args.RATE

    FastTrain(
        HIDDEN, backend=FastTensorBackend if args.BACKEND != "gpu" else GPUBackend
    ).train(data, RATE)
```