Отчет по лабораторной работе №4 по курсу "Нейроинформатика".

Выполнил Пищик Е.С. М8О-406Б-19.

Цель работы.

Исследование свойств некоторых видов сетей с радиальными базисными элементами, алгоритмов обучения, а также применение сетей в задачах классификации и аппроксимации функции.

Часть 1.

Файл run.py.

Импортируем нужные библиотеки, класс RBF из файла model.py, класс Dataset из файла dataset.py.

In [ ]:

```
import torch
from torch import nn
from tqdm import tqdm
import matplotlib.pyplot as plt
import os

from model import RBF
from dataset import Dataset

import warnings
warnings.filterwarnings('ignore')
```

Создаем класс Pipeline, где происходит инициализация гиперпараметров, датасета, модели, запуск цикла обучения, отрисовка результатов.

In [ ]:

```
class Pipeline:
    def __init__(self,
                 epochs=1000,
                 lr=0.01,
                 wd=0.0001,
                 save_path='checkpoints/',
                 save_every=500,
                 save_weights=True,
                 logs_path='logs/'):
        self.epochs = epochs
        self.lr = lr
        self.wd = wd
        self.save_path = save_path
        self.logs_path = logs_path
        self.save_every = save_every
        self.save_weights = save_weights

        if not os.path.exists(save_path):
            os.makedirs(save_path, mode=0o777)

        if not os.path.exists(logs_path):
            os.makedirs(logs_path, mode=0o777)

        self.init_dataset()
        self.init_model()

        self.train()
        self.test()

        self.plot()

    def init_weights(self, m):
        if isinstance(m, nn.Linear):
            nn.init.xavier_uniform_(m.weight)
            m.bias.data.fill_(0.0)

    def init_dataset(self):
        params_elips1 = {'a': 0.4,
                         'b': 0.4,
                         'alpha': 0,
                         'x0': 0.1,
                         'y0': -0.15}

        params_elips2 = {'a': 0.7,
                         'b': 0.7,
                         'alpha': 0,
                         'x0': 0,
                         'y0': 0}

        params_parabola = {'p': -1,
                           'alpha': 0,
                           'x0': 0.8,
                           'y0': 0}

        params_dct = {'elipsis1': params_elips1,
                      'elipsis2': params_elips2,
                      'parabola': params_parabola}

        dataset = Dataset(params_dct)
```

```
        train_size = int(0.7 * len(dataset))
        valid_size = int(0.2 * len(dataset))

        self.train_dataset = dataset[:train_size]
        self.valid_dataset = dataset[train_size:train_size + valid_size]
        self.test_dataset = dataset[train_size + valid_size:]
        self.plot_dataset = dataset[:]

    def init_model(self):
        self.model = RBF(rbf_features=32, in_features=2, out_features=3)
        self.model.apply(self.init_weights)

        self.optimizer = torch.optim.Adam(self.model.parameters(),
                                          lr=self.lr,
                                          weight_decay=self.wd)

        self.loss_fn = nn.CrossEntropyLoss()

    def train(self):
        tqdm_iter = tqdm(range(self.epochs))

        for epoch in tqdm_iter:
            self.model.train()

            train_loss = 0.0
            valid_loss = 0.0

            pred = self.model(self.train_dataset[0])
            target = self.train_dataset[1]

            loss = self.loss_fn(pred, target)

            self.optimizer.zero_grad()
            loss.backward()
            self.optimizer.step()

            train_loss += loss.item()

            if self.save_weights:
                if epoch % self.save_every == 0 or epoch == self.epochs - 1:
                    self.save(epoch)

            self.model.eval()

            with torch.no_grad():
                pred = self.model(self.valid_dataset[0])
                target = self.valid_dataset[1]

                loss = self.loss_fn(pred, target)
                valid_loss += loss.item()

            tqdm_iter.set_postfix(
                {'epoch:': f'{epoch + 1}/{self.epochs}',
                 'train loss:': train_loss,
                 'valid loss:': valid_loss})

            tqdm_iter.refresh()

    def test(self):
        self.model.eval()
```

```
        test_loss = 0.0

        with torch.no_grad():
            pred = self.model(self.test_dataset[0])
            target = self.test_dataset[1]

            loss = self.loss_fn(pred, target)
            test_loss += loss.item()

        print('test loss:', test_loss)

    def save(self, epoch):
        state_dict = self.model.state_dict()

        torch.save(state_dict, self.save_path +
                   f'epoch-{epoch}.pth')

    def plot(self):
        self.model.eval()

        x_arr, y_arr, color_pred, color_true = [], [], [], []

        pred = self.model(self.plot_dataset[0])
        pred = torch.argmax(pred, dim=1)

        target = self.plot_dataset[1]

        x_arr = self.plot_dataset[0][:, 0]
        y_arr = self.plot_dataset[0][:, 1]

        for pred_val in pred:
            if pred_val == 0:
                color_pred.append('b')
            elif pred_val == 1:
                color_pred.append('g')
            elif pred_val == 2:
                color_pred.append('r')

        for true_val in target:
            if true_val == 0:
                color_true.append('b')
            elif true_val == 1:
                color_true.append('g')
            elif true_val == 2:
                color_true.append('r')

        fig = plt.figure(figsize=(15, 10), dpi=300)

        ax1 = fig.add_subplot(211)
        ax1.grid(True)
        ax1.scatter(x_arr, y_arr, c=color_pred)
        ax1.set_xlabel('x')
        ax1.set_ylabel('y')
        ax1.set_title('pred')

        ax2 = fig.add_subplot(212)
        ax2.grid(True)
        ax2.scatter(x_arr, y_arr, c=color_true)
        ax2.set_xlabel('x')
        ax2.set_ylabel('y')
```

```
        ax2.set_title('true')

        plt.savefig(f'{self.logs_path}classification.png')


def main():
    Pipeline()


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

Файл model.py.

Модель - сеть с радиальными базисными элементами, написанная при помощи библиотеки PyTorch.

In [ ]:

```
from torch import nn
import torch


class RBF(nn.Module):
    def __init__(self, rbf_features, in_features, out_features):
        super().__init__()
        self.in_features = in_features
        self.rbf_features = rbf_features
        self.out_features = out_features
        self.centres = nn.Parameter(torch.Tensor(rbf_features, in_features))
        self.sigmas = nn.Parameter(torch.Tensor(rbf_features))
        self.linear = nn.Linear(rbf_features, out_features)
        self.reset_parameters()

    def reset_parameters(self):
        nn.init.normal_(self.centres, 0, 1)
        nn.init.constant_(self.sigmas, 0)

    def forward(self, input):
        size = (input.size(0), self.rbf_features, self.in_features)
        x = input.unsqueeze(1).expand(size)
        c = self.centres.unsqueeze(0).expand(size)
        exp = torch.exp(self.sigmas).unsqueeze(0)
        l2 = (x - c).pow(2).sum(-1).pow(0.5)
        distances = l2 / exp
        return self.linear(distances)
```

Файл dataset.py.

Класс Dataset - унаследованный класс от torch.utils.data.Dataset, где хранятся наши данные.

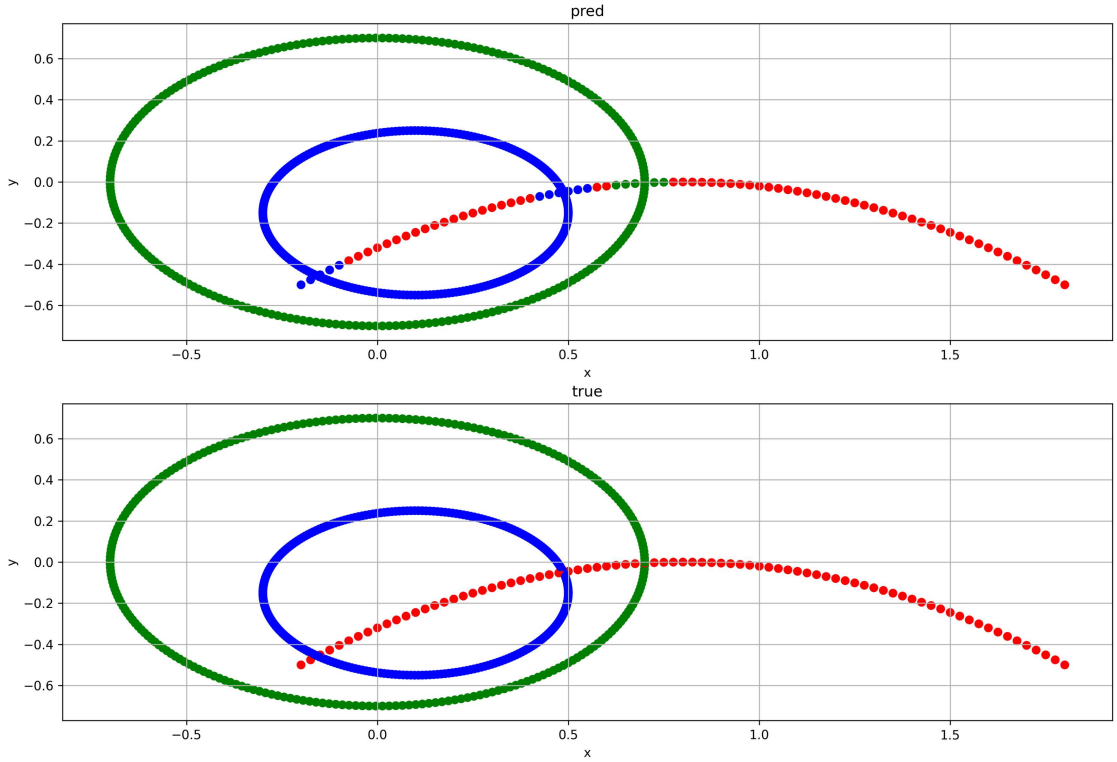
In [ ]:

```
from torch.utils.data import Dataset as Base
import torch
import numpy as np


class Dataset(Base):
    def __init__(self, params_dct):
        super().__init__()
        params_elips1 = params_dct['elipsis1']
        params_elips2 = params_dct['elipsis2']
        params_parabola = params_dct['parabola']

        self.elips1 = self.gen_elipsis(params_elips1['a'],
                                       params_elips1['b'],
                                       params_elips1['alpha'],
                                       params_elips1['x0'],
                                       params_elips1['y0'])

        self.elips2 = self.gen_elipsis(params_elips2['a'],
                                       params_elips2['b'],
                                       params_elips2['alpha'],
                                       params_elips2['x0'],
                                       params_elips2['y0'])

        self.parab = self.gen_parabola(params_parabola['p'],
                                       params_parabola['alpha'],
                                       params_parabola['x0'],
                                       params_parabola['y0'])

        self.samples, self.labels = self.concat_tensors(self.elips1,
                                                        self.elips2,
                                                        self.parab)

        self.samples, self.labels = self.permute(self.samples, self.labels)

    def elipsis(self, a, b):
        start = 0.0
        end = 2 * np.pi
        step = 0.025
        vals = np.linspace(start, end, int((end - start) / step) + 1)
        x = np.array([a * np.cos(t) for t in vals])
        y = np.array([b * np.sin(t) for t in vals])
        return x, y

    def parabola(self, p):
        start = -1.0
        end = 1.0
        step = 0.025
        x = np.linspace(start, end, int((end - start) / step) + 1)
        y = np.array([(el ** 2) / (2 * p) for el in x])
        return x, y

    def func1(self, x, y, alpha):
        return np.cos(alpha) * x + np.sin(alpha) * y

    def func2(self, x, y, alpha):
        return -np.sin(alpha) * x + np.cos(alpha) * y

    def rotate_coords(self, coords, alpha, x0, y0):
```

```
        res_x, res_y = [], []

        for x, y in zip(coords[0], coords[1]):
            res_x.append(self.func1(x, y, alpha) + x0)
            res_y.append(self.func2(x, y, alpha) + y0)

        x = torch.FloatTensor(np.array(res_x))
        y = torch.FloatTensor(np.array(res_y))

        return x, y

    def gen_elipsis(self, a, b, alpha, x0, y0):
        elipsis = self.elipsis(a, b)
        x, y = self.rotate_coords(elipsis, alpha, x0, y0)
        return x, y

    def gen_parabola(self, p, alpha, x0, y0):
        parabola = self.parabola(p)
        x, y = self.rotate_coords(parabola, alpha, x0, y0)
        return x, y

    def concat_tensors(self, t1, t2, t3):
        x = torch.cat((t1[0], t2[0], t3[0]), dim=0)
        y = torch.cat((t1[1], t2[1], t3[1]), dim=0)
        classes = []

        samples = torch.FloatTensor(np.vstack((np.array(x), np.array(y))))

        first_class = len(t1[0])
        second_class = len(t1[0]) + len(t2[0])
        third_class = len(t1[0]) + len(t2[0]) + len(t3[0])

        for i in range(samples.shape[1]):
            if i < first_class:
                classes.append(0)
            elif i < second_class:
                classes.append(1)
            elif i < third_class:
                classes.append(2)

        samples = torch.permute(samples, (1, 0))
        targets = torch.LongTensor(classes)
        return samples, targets

    def permute(self, samples, labels):
        permut = np.random.permutation(samples.shape[0])
        return samples[permut], labels[permut]

    def __len__(self):
        return self.samples.shape[0]

    def __getitem__(self, index):
        return self.samples[index], self.labels[index]
```

Результаты.

Классификация.



Часть 2.

Файл run.py.

Импортируем нужные библиотеки, класс RBF из файла model.py, класс Dataset из файла dataset.py.

In [ ]:

```
import numpy as np
import torch
from torch import nn
from tqdm import tqdm
import matplotlib.pyplot as plt
import os

from model import RBF
from dataset import Dataset

import warnings
warnings.filterwarnings('ignore')
```

Создаем класс Pipeline, где происходит инициализация гиперпараметров, датасета, модели, запуск цикла обучения, отрисовка результатов.

In [ ]:

```
class Pipeline:
    def __init__(self,
                 epochs=1000,
                 lr=0.01,
                 wd=0.0001,
                 save_path='checkpoints/',
                 save_every=500,
                 save_weights=True,
                 logs_path='logs/'):
        self.epochs = epochs
        self.lr = lr
        self.wd = wd
        self.save_path = save_path
        self.logs_path = logs_path
        self.save_every = save_every
        self.save_weights = save_weights

        if not os.path.exists(save_path):
            os.makedirs(save_path, mode=0o777)

        if not os.path.exists(logs_path):
            os.makedirs(logs_path, mode=0o777)

        self.init_dataset()
        self.init_model()

        self.train()
        self.test()

        self.plot()

    def init_weights(self, m):
        if isinstance(m, nn.Linear):
            nn.init.xavier_uniform_(m.weight)
            m.bias.data.fill_(0.0)

    def func(self, t):
        return np.sin(-5 * t**2 + 10 * t - 5)

    def init_dataset(self):
        start, end, h = 0, 2.5, 0.01

        t = np.linspace(start, end, int((end - start) / h) + 1)
        self.times = t

        dataset = Dataset(t, self.func)

        train_size = int(0.7 * len(dataset))
        valid_size = int(0.2 * len(dataset))

        self.train_dataset = dataset[:train_size]
        self.valid_dataset = dataset[train_size:train_size + valid_size]
        self.test_dataset = dataset[train_size + valid_size:]
        self.plot_dataset = dataset[:]

    def init_model(self):
        self.model = RBF(rbf_features=32, in_features=1, out_features=1)
        self.model.apply(self.init_weights)
```

```
        self.optimizer = torch.optim.Adam(self.model.parameters(),
                                          lr=self.lr,
                                          weight_decay=self.wd)

        self.loss_fn = nn.MSELoss()

    def train(self):
        tqdm_iter = tqdm(range(self.epochs))

        for epoch in tqdm_iter:
            self.model.train()

            train_loss = 0.0
            valid_loss = 0.0

            pred = self.model(self.train_dataset[0].unsqueeze(1))
            target = self.train_dataset[1].unsqueeze(1)

            loss = self.loss_fn(pred, target)

            self.optimizer.zero_grad()
            loss.backward()
            self.optimizer.step()

            train_loss += loss.item()

            if self.save_weights:
                if epoch % self.save_every == 0 or epoch == self.epochs - 1:
                    self.save(epoch)

            self.model.eval()

            with torch.no_grad():
                pred = self.model(self.valid_dataset[0].unsqueeze(1))
                target = self.valid_dataset[1].unsqueeze(1)

                loss = self.loss_fn(pred, target)
                valid_loss += loss.item()

            tqdm_iter.set_postfix(
                {'epoch:': f'{epoch + 1}/{self.epochs}',
                 'train loss:': train_loss,
                 'valid loss:': valid_loss})

            tqdm_iter.refresh()

    def test(self):
        self.model.eval()

        test_loss = 0.0

        with torch.no_grad():
            pred = self.model(self.test_dataset[0].unsqueeze(1))
            target = self.test_dataset[1].unsqueeze(1)

            loss = self.loss_fn(pred, target)
            test_loss += loss.item()

        print('test loss:', test_loss)

    def save(self, epoch):
```

```
        state_dict = self.model.state_dict()

        torch.save(state_dict, self.save_path +
                   f'epoch-{epoch}.pth')

    def plot(self):
        self.model.eval()

        pred = self.model(self.plot_dataset[0].unsqueeze(1))
        target = self.plot_dataset[1]
        x_arr = self.plot_dataset[0]

        pred = pred[:, 0].detach().numpy()

        fig = plt.figure(figsize=(15, 10), dpi=300)

        ax = fig.add_subplot(211)
        ax.grid(True)
        ax.plot(x_arr, target, label='real', color='#0056d6')
        ax.plot(x_arr, pred, label='approx', color='#73d925')
        ax.set_xlabel('time')
        ax.set_ylabel('x')
        ax.legend(loc='upper right')

        plt.savefig(f'{self.logs_path}function.png')


def main():
    Pipeline()


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

Файл model.py.

Модель - сеть с радиальными базисными элементами, написанная при помощи библиотеки PyTorch.

In [ ]:

```
from torch import nn
import torch


class RBF(nn.Module):
    def __init__(self, rbf_features, in_features, out_features):
        super().__init__()
        self.in_features = in_features
        self.rbf_features = rbf_features
        self.out_features = out_features
        self.centres = nn.Parameter(torch.Tensor(rbf_features, in_features))
        self.sigmas = nn.Parameter(torch.Tensor(rbf_features))
        self.sw = nn.Linear(rbf_features, rbf_features)
        self.linear = nn.Linear(rbf_features, out_features)
        self.reset_parameters()

    def reset_parameters(self):
        nn.init.normal_(self.centres, 0, 1)
        nn.init.constant_(self.sigmas, 0)

    def forward(self, input):
        size = (input.size(0), self.rbf_features, self.in_features)
        x = input.unsqueeze(1).expand(size)
        c = self.centres.unsqueeze(0).expand(size)
        exp = torch.exp(self.sigmas).unsqueeze(0)
        l2 = (x - c).pow(2).sum(-1).pow(0.5)
        distances = l2 / exp
        out = self.sw(distances)
        out = self.linear(out)
        return out
```

Файл dataset.py.

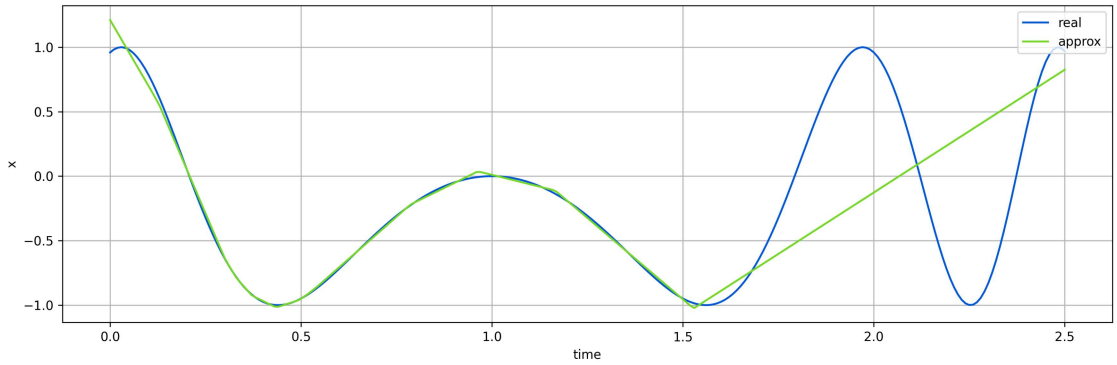
Класс Dataset - унаследованный класс от torch.utils.data.Dataset, где хранятся наши данные.

In [ ]:

```
from torch.utils.data import Dataset as Base
import torch


class Dataset(Base):
    def __init__(self, x, func):
        super().__init__()
        self.x = torch.FloatTensor(x)
        self.y = torch.FloatTensor([func(el) for el in x])

    def __len__(self):
        return len(self.x)

    def __getitem__(self, index):
        return self.x[index], self.y[index]
```

Результаты.

Аппроксимация.



Выводы.

В данной лабораторной работе мы научились работать с моделью RBF, решили задачу нелинейной классификации и аппроксимации.

In [ ]: