Отчет по лабораторной работе №2 по курсу "Нейроинформатика".

Выполнил Пищик Е.С. М8О-406Б-19.

Цель работы.

Исследование свойств линейной нейронной сети и алгоритмов ее обучения, применение сети в задачах аппроксимации и фильтрации.

Файл run.py.

Импортируем нужные библиотеки, класс LinearModel из файла model.py.

In [ ]:

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

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

In [ ]:

```
class Pipeline:
    def __init__(self,
                 mode,
                 epochs=1000,
                 lr=0.01,
                 wd=0.0001,
                 delays=5,
                 save_path='checkpoints/',
                 save_every=500,
                 save_weights=True,
                 logs_path='logs/'):
        self.epochs = epochs
        self.lr = lr
        self.wd = wd
        self.mode = mode
        self.delays = delays
        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(num=1)
        self.train(num=2)

        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 _input_signal1(self, t):
        return np.sin(-2 * np.sin(t) * t**2 + 7)

    def _input_signal2(self, t):
        return np.cos(t**2 - 2*t + 3)

    def _output_signal(self, t):
        return np.cos(t**2 - 2*t - np.pi) / 3

    def init_dataset(self):
        start1, end1, h1 = 0, 3.5, 0.01
        start2, end2, h2 = 0, 6.0, 0.025

        t1 = np.linspace(start1, end1, int((end1 - start1) / h1) + 1)
        t2 = np.linspace(start2, end2, int((end2 - start2) / h2) + 1)

        self.times1, self.times2 = t1, t2

        x1, x2 = self._input_signal1(t1), self._input_signal2(t2)
```

```
        y1, y2 = self._output_signal(t1), self._output_signal(t2)

        range1 = range(len(x1) - self.delays)
        range2 = range(len(x2) - self.delays)

        data1 = np.array([np.hstack(x1[i:i+self.delays]) for i in range1])
        data2 = np.array([np.hstack(x2[i:i+self.delays]) for i in range2])

        if self.mode == 'inp-inp':
            self.train_x1 = torch.FloatTensor(data1)
            self.train_y1 = torch.FloatTensor(x1[self.delays:])
            self.train_x2 = torch.FloatTensor(data2)
            self.train_y2 = torch.FloatTensor(x2[self.delays:])
        elif self.mode == 'inp-outp':
            self.train_x1 = torch.FloatTensor(data1)
            self.train_y1 = torch.FloatTensor(y1[self.delays:])
            self.train_x2 = torch.FloatTensor(data2)
            self.train_y2 = torch.FloatTensor(y2[self.delays:])

    def init_model(self):
        self.model1 = LinearModel(inp=self.delays, outp=1)
        self.model2 = LinearModel(inp=self.delays, outp=1)

        self.model1.apply(self.init_weights)
        self.model2.apply(self.init_weights)

        self.optimizer1 = torch.optim.Adam(self.model1.parameters(),
                                           lr=self.lr,
                                           weight_decay=self.wd)
        self.optimizer2 = torch.optim.Adam(self.model2.parameters(),
                                           lr=self.lr,
                                           weight_decay=self.wd)
        self.loss_fn1 = nn.MSELoss()
        self.loss_fn2 = nn.MSELoss()

    def train(self, num):
        if num == 1:
            model = self.model1
            optimizer = self.optimizer1
            loss_fn = self.loss_fn1
            x, y = self.train_x1, self.train_y1
        elif num == 2:
            model = self.model2
            optimizer = self.optimizer2
            loss_fn = self.loss_fn2
            x, y = self.train_x2, self.train_y2

        tqdm_iter = tqdm(range(self.epochs))

        for epoch in tqdm_iter:
            pred = model(x)
            target = y.unsqueeze(1)

            loss = loss_fn(pred, target)

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

            if self.save_weights:
                if epoch % self.save_every == 0 or epoch == self.epochs - 1:
```

```
                    self.save(epoch, num, self.mode)

            tqdm_iter.set_postfix(
                {'epoch:': f'{epoch + 1}/{self.epochs}', 'loss:': loss.item()})
            tqdm_iter.refresh()

    def save(self, epoch, num, mode):
        if num == 1:
            state_dict = self.model1.state_dict()
        elif num == 2:
            state_dict = self.model2.state_dict()

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

    def plot(self):
        self.model1 = self.model1.eval()
        self.model2 = self.model2.eval()

        pred1 = self.model1(self.train_x1).detach().numpy()
        pred2 = self.model2(self.train_x2).detach().numpy()

        target1 = self.train_y1.numpy()
        target2 = self.train_y2.numpy()

        times1 = self.times1[self.delays:]
        times2 = self.times2[self.delays:]

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

        ax1 = fig.add_subplot(211)
        ax1.grid(True)
        ax1.plot(times1, pred1, label='approx', color='#73d925')
        ax1.plot(times1, target1, label='real', color='#0056d6')
        ax1.set_xlabel('time')
        ax1.set_ylabel('x')
        ax1.legend(loc='upper right')

        ax2 = fig.add_subplot(212)
        ax2.grid(True)
        ax2.plot(times2, pred2, label='approx', color='#73d925')
        ax2.plot(times2, target2, label='real', color='#0056d6')
        ax2.set_xlabel('time')
        ax2.set_ylabel('x')
        ax2.legend(loc='upper right')

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


def main():
    Pipeline(mode='inp-inp')
    Pipeline(mode='inp-outp')


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

Файл model.py.

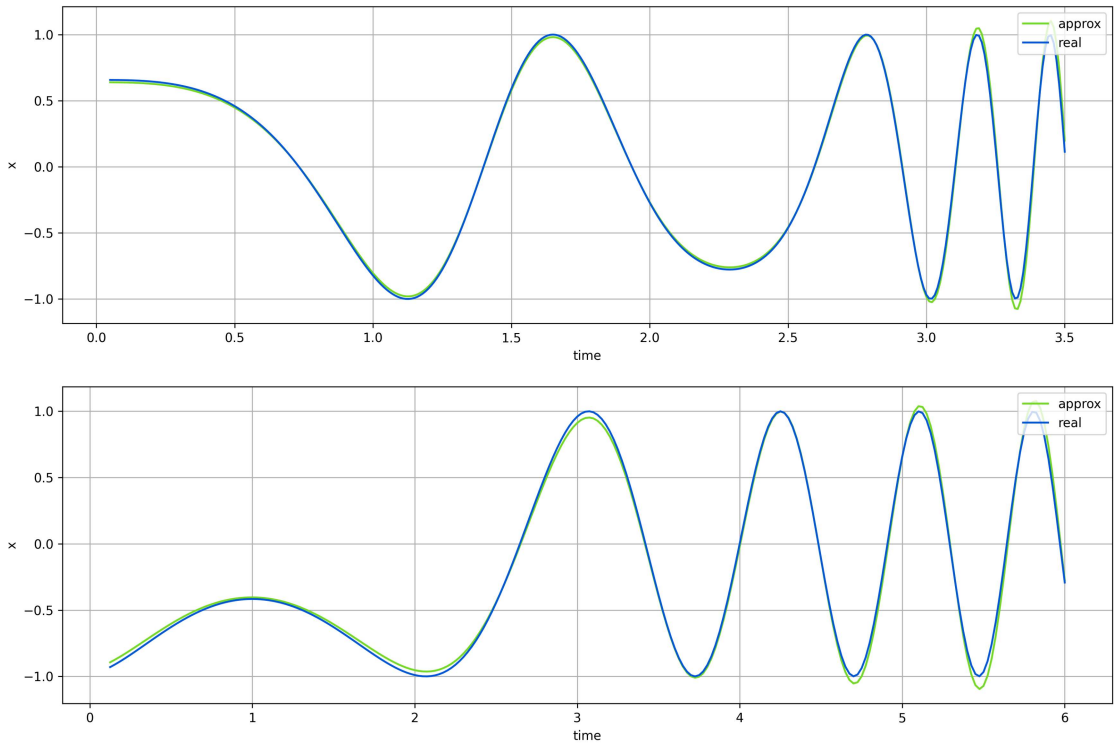
Модель - линейная сеть, написанная при помощи библиотеки PyTorch.

In [ ]:

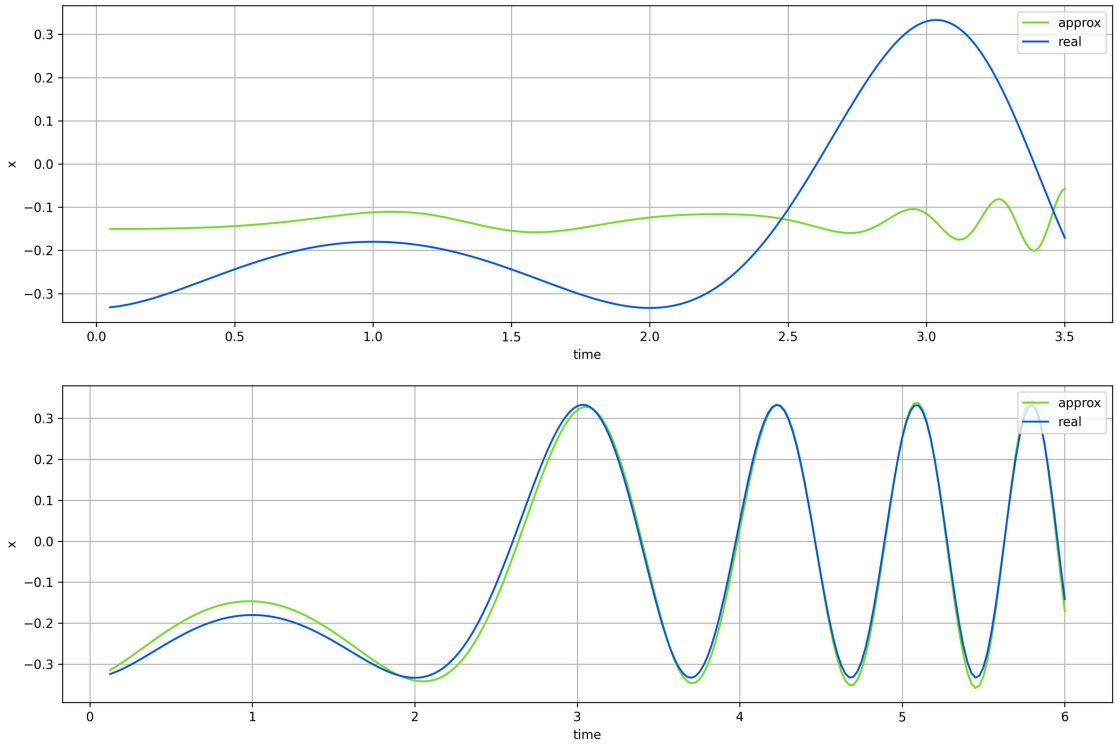
```
from torch import nn


class LinearModel(nn.Module):
    def __init__(self, inp, outp):
        super().__init__()
        self.fc = nn.Linear(inp, outp)

    def forward(self, x):
        return self.fc(x)
```

Результаты.

Предсказание входного сигнала по предыдущим значениям входного сигнала.



Предсказание выходного сигнала по предыдущим значениям входного сигнала.



Выводы.

В данной лабораторной работе мы научились работать с линейной моделью с задержками, решили задачу предсказания следующего значения входного/выходного сигнала по последовательности предыдущих значений входного сигнала.

In [ ]: