Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №2 по курсу «Нейроинформатика»

Студент: М. А. Бронников
Преподаватель: Н. П. Аносова
Группа: М8О-407Б

Дата:
Оценка:
Подпись:

Москва, 2020

Линейная нейронная сеть. Правило обучения Уидроу-Хоффа

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

Основные этапы работы:

1. Использовать линейную нейронную сеть с задержками для аппроксимации функции. В качестве метода обучения использовать адаптацию.
2. Использовать линейную нейронную сеть с задержками для аппроксимации функции и выполнения многошагового прогноза.
3. Использовать линейную нейронную сеть в качестве адаптивного фильтра для подавления помех. Для настройки весовых коэффициентов использовать метод наименьших квадратов.

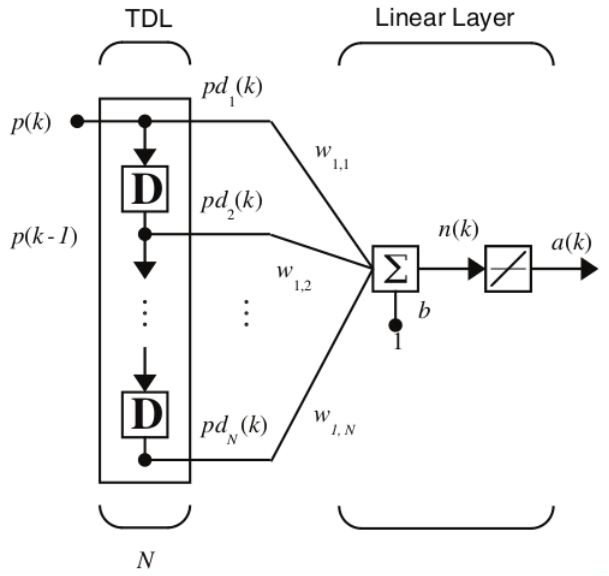
Вариант №4:
Входные сигналы:

1. $x = \sin(-3t^2 + 10t - 5),\ t \in [0.5, 4],\ h = 0.01$
2. $x = \sin(2t^2 - 6t + 3),\ t \in [0, 5],\ h = 0.02$

Выходной сигнал: $y = \frac{1}{2}\sin(2t^2 - 6t - \pi)$

1 Структура модели

Для решения этой задачи необходимо воспользоваться фльтратором, который состоит из *TDL блока* и *Линейного слоя* и имеет следующую структуру:



Чтобы реализовать линейный слой (хотя в данной работе можно ограничиться лишь одним нейроном) можно воспользоваться представлением весов и смещений перцептронов как марицу $(n+1) \times m$, где n - число входов, а m - число выходов. При этом в качестве выходов я использую функцию $net = \sum_{i=0}^{n} w_i x_i + b$, а ошибку измеряю с помощью метрики $MSE = \frac{\sum_{i=1}^{N}(t_i - a_i)^2}{N}$.

Реализация линейного слоя:

```
class LinearLayer:
    def __init__(self, steps = 50, lr = 0.0001, stop_err=0.0):
        self.steps = steps
        self.w = None
        self.rate = lr
        self.stop_err = stop_err

    def fit(self, X, y):
        # add column for bias and transpose data for comphort operations:
        X_t = np.append(X, np.ones((X.shape[0], 1)), axis = 1)
        y_t = np.array(y)
```

```
12
13          #init weights:
14          if self.w is None:
15              self.w = np.random.random((X_t.shape[1], y_t.shape[1]))
16
17          # main loop:
18          for step in tqdm(range(self.steps)):
19              for i in range(X_t.shape[0]):
20                  e = y_t[i] - X_t[i].dot(self.w) # compute error
21                  # change weights:
22                  self.w += self.rate * \
23                      X_t[i].reshape(
24                          X_t.shape[1], 1
25                      ).dot(
26                          e.reshape(1, y_t.shape[1])
27                      )
28              mse = ((y_t - X_t.dot(self.w))**2).mean()
29              if mse < self.stop_err:
30                  break
31
32          return self # return trained model
33
34      def set_steps(self, steps):
35          self.steps = steps
36
37      def set_learning_rate(self, rate):
38          self.rate = rate
39
40      # Predict answers
41      def predict(self, X):
42          X_t = np.append(X, np.ones((X.shape[0], 1)), axis = 1)
43          return X_t.dot(self.w)
44
45      def display(self):
46          ans = "Input(n," + str(self.w.shape[0] - 1) + ") --> "
47          ans += "Linear_Layer(" + str(self.w.shape[1]) + ") --> "
48          ans += "Output(n, " + str(self.w.shape[1]) + ") --> "
49          return ans
50
51      def weights(self):
52          return self.w[:-1]
53
54      def bias(self):
55          return self.w[-1]
56
57      # RMSE
58      def score(self, X, y):
59          X_t = np.append(X, np.ones((X.shape[0], 1)), axis = 1)
60          y_t = np.array(y)
```

```
61          return ((y_t - X_t.dot(self.w))**2).mean()**0.5
```

Как можно заметить, я использую правило *Уидроу-Хоффа* для обучения, которое практически идентично классическому стохачтическому градиентному спуску, за исключением того, что выбор объекта из выборки для корректировки значений в данном случае берется в определенной последовательности, а не случайным образом. Основная идея этого метода заключается в корректировке весов в сторону наискорейшего убывания функции ошибки, а именно в направлении антиградиента этой ошибки. Также при обучении предусмотрена остановка при достижении заданного значения ошибки.

Реализация TDL довольно банальна, за исключением интересного момента - наличия текущего состояния для генерации нового вектора. Также не стоит особого уделения внимания и класс самого фильтра. Поэтому просто приведу их реализации:

```
1  class TDL:
2      def __init__(self, D = 1, pad_zeros=True):
3          self.depth = D
4          self.padding = pad_zeros
5          self.queue = np.zeros(D)
6
7      def fit(self, X, Y = None):
8          # init train data such as in self.predict method
9          if self.padding:
10             in_arr = np.append(np.zeros(self.depth - 1), X)
11             result = np.zeros((len(X) - 1, self.depth))
12             if Y is None:
13                 Y = X[-len(X) + 1:]
14             else:
15                 Y = Y[-len(X) + 1:]
16         else:
17             if len(X) < self.depth:
18                 return None
19             in_arr = np.array(X)
20             result = np.zeros((len(X) - self.depth, self.depth))
21             if Y is None:
22                 Y = X[-len(X) + self.depth:]
23             else:
24                 Y = Y[-len(X) + self.depth:]
25
26         for i in range(in_arr.shape[0] - self.depth):
27             result[i] = in_arr[i:i + self.depth]
28
29         return result, Y
30
31     def tdl_init(self, values):
32         if values.shape[0] != self.depth - 1:
33             raise ValueError("You should give " + str(self.depth - 1) + " values for
```

```
                init")
34          self.queue = np.append(np.zeros(1), np.array(values))
35
36
37      def tdl_init_zeros(self):
38          self.queue = np.zeros(self.depth)
39
40
41      def predict(self, X):
42          # init delay line and alloc mem
43          in_arr = np.append(self.queue[1:], X)
44          result = np.zeros((len(X), self.depth))
45
46          # fill memory buffer by values of line
47          for i in range(in_arr.shape[0] - self.depth + 1):
48              result[i] = in_arr[i:i + self.depth]
49          # update queue
50          self.queue = in_arr[-self.depth:]
51          return result
52
53
54      def display(self):
55          ans = "TDL(" + str(self.depth) + ") --> "
56          return ans
57
58
59  class Filtrator:
60      def __init__(self, D = 1, pad_zeros = False, steps = 50, l_r=0.001, stop_err=0.0):
61          self.tdl = TDL(D, pad_zeros)
62          self.linlr = LinearLayer(steps, l_r, stop_err)
63          self.tld_initialized = pad_zeros
64          self.last_predict = None
65
66      def fit(self, X, Y = None):
67          X1, Y1 = self.tdl.fit(X, Y)
68          Y1 = np.array(Y1).reshape(len(Y1), 1)
69          self.linlr.fit(X1, Y1)
70          return self
71
72      def tdl_init(self, values):
73          self.tdl.tdl_init(values)
74          self.tld_initialized = True
75
76      def tdl_init_zeros(self):
77          self.tdl.tdl_init_zeros()
78
79      def predict(self, x):
80          if not self.tld_initialized:
81              raise ValueError("You should init input before predict")
```

```
82          ans = self.linlr.predict(self.tdl.predict(x)).ravel()
83          self.last_predict = ans[-1]
84          return ans
85
86      def display(self):
87          return self.tdl.display() + self.linlr.display()
88
89      def score_value(Y_t, Y_p):
90          return ((Y_t - Y_p)**2).mean()**0.5
91
92      def gen_values(self, num, inpt = None):
93          if inpt is not None:
94              self.last_predict = inpt
95          if self.last_predict is None:
96              raise ValueError("Last predict doesn‘t know. " +
97                               "Please set last predicted value or make prediction.")
98          for i in range(num):
99              yield self.predict(np.array([self.last_predict]))[0]
```

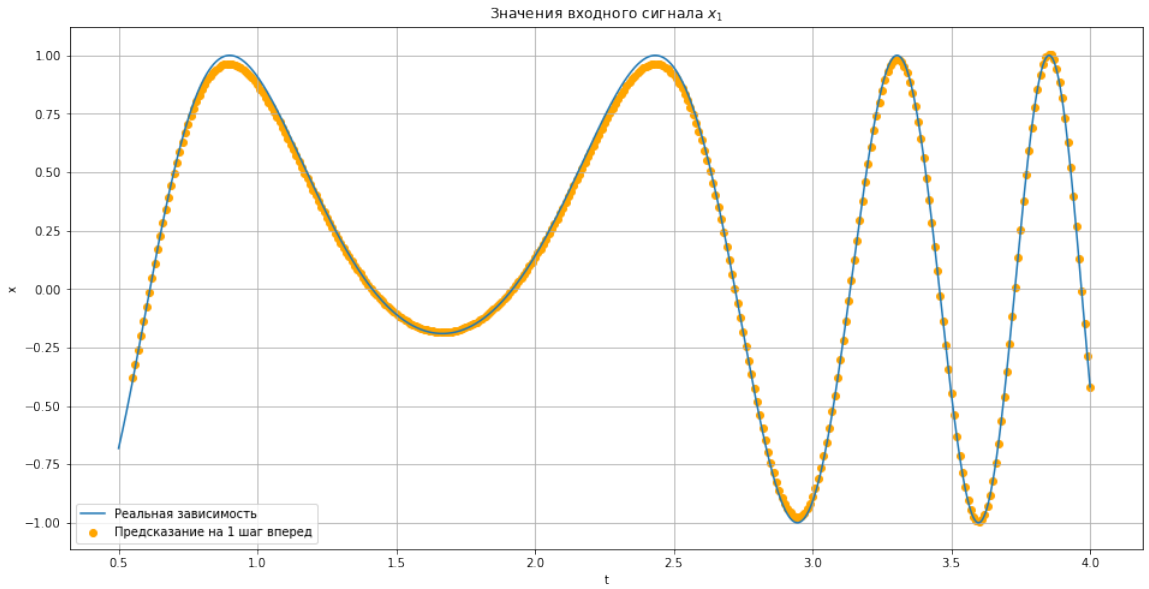
2 Ход работы

Я сгенерировал обучающее множество на основе заднного шага и интервала значений t и передал его для обучению модели фильтратора, который был инициализирован значениями, требуемыми по заданию:

```
T = np.append(np.arange(*t_lim1, h1), t_lim1[1])
X = x1_t(T)

D = 5
steps = 50
learn_rate = 0.01

model = Filtrator(D, False, steps, learn_rate).fit(X)
print(model.display())

>>> TDL(5) --> Input(n,5) --> Linear_Layer(1) --> Output(n, 1) -->
```

Далее я инициализировал модель первыми $D = 5$ значениями и сделал одношаговый прогноз на обучающих данных, после чего сравнил полученный результат с эталонным:

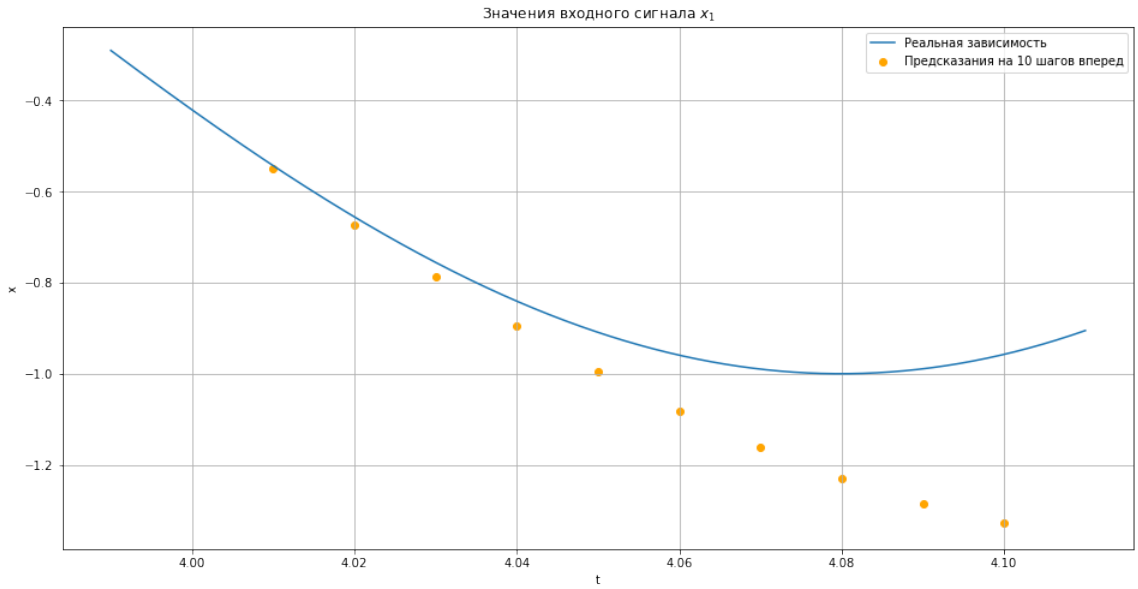
```
X_init = X[:D - 1]
X_test = X[D - 1:-1]
X_ans = X[D:]

model.tdl_init(X_init)
X_pred = model.predict(X_test)

t = np.arange(*t_lim1, 0.0001)
x = x1_t(t)
plt.figure(figsize=(16, 8))
plt.plot(t, x, label=" ")
#plt.scatter(T, X, label=" ")
plt.scatter(T[D:], X_pred, color="orange", label="  1  ")
plt.title("   $x_1$")
plt.xlabel("t")
plt.ylabel("x")
plt.grid()
plt.legend()

print("RMSE =", model.score_value(X_ans, X_pred))

>>> RMSE = 0.01860670996085224
```



Далее для второго задания я практически полностью повторил описанные выше шаги (только с другими парпаметрами модели, требуемыми по заданию), после чего попробовал сделать многошаговый прогноз за пределы заданного интервала на $K = 10$ шагов вперед:

```
K = 10
X_pred = np.array(list(model.gen_values(K)))

T_pred = []
for i in range(1, K + 1):
    T_pred.append(t_lim1[1] + i*h1)

T_pred = np.array(T_pred)
X_ans = x1_t(T_pred)

t = np.arange(t_lim1[1] - h1, t_lim1[1] + h1*11, 0.0001)
x = x1_t(t)
plt.figure(figsize=(16, 8))
plt.plot(t, x, label=" ")
#plt.scatter(T, X, label=" ")
plt.scatter(T_pred, X_pred, color="orange", label="  10  ")
plt.title("   $x_1$")
plt.xlabel("t")
plt.ylabel("x")
plt.grid()
plt.legend()

print("RMSE =", model.score_value(X_ans, X_pred))

```

```
>>> RMSE = 0.18256148130858663
```



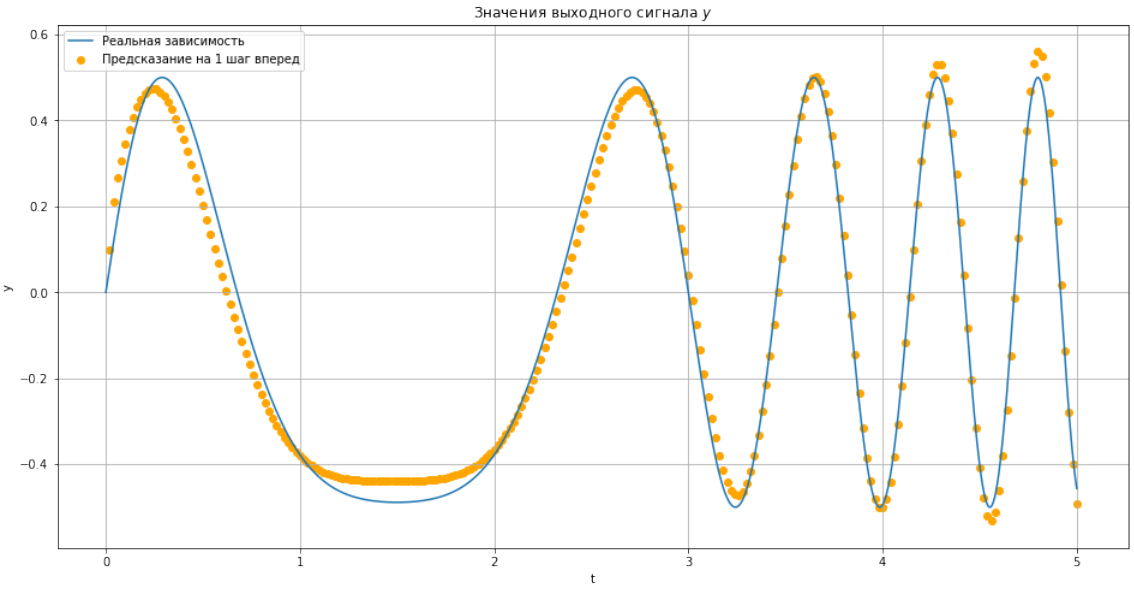
После этого я перешел к последнему заданию, в котором в качестве входных данных для обучения я использовал входное множество №2, дополненное нулями для генерации в качестве входов для линейного слоя не задержки, а погружение временного ряда. В качестве целевой переменной выступает не значение следующего по порядку входного сигнала, а значениие шумового выходного сигнала. В остальном же обучение фильтра мало чем отличается от обучения в первых двух заданиях:

```
T = np.append(np.arange(*t_lim2, h2), t_lim2[1])
X = x2_t(T)
Y = y_t(T)

D = 4
steps = 500
learn_rate = 0.0025
stop_val = 10e-6

model = Filtrator(D, True, steps, learn_rate, stop_val).fit(X, Y)
```

Для предсказания шумового значения я инициализировал модель нулевыми значениями, после чего сделал предсказание модели и оценил его:

```
X_test = X[:-1]
X_ans = X[1:]
Y_ans = Y[1:]

model.tdl_init_zeros()

```

```
Y_pred = model.predict(X_test)

t = np.arange(*t_lim2, 0.0001)
x = y_t(t)
plt.figure(figsize=(16, 8))
plt.plot(t, x, label=" ")
#plt.scatter(T, X, label=" ")
plt.scatter(T[1:], Y_pred, color="orange", label="  1  ")
plt.title("   $y$")
plt.xlabel("t")
plt.ylabel("y")
plt.grid()
plt.legend()

print("RMSE =", model.score_value(Y_ans, Y_pred))

>>> RMSE = 0.041820303477404716
```



3 Выводы

Выполнив вторую лабораторную работу по курсу «Нейроинформатика», я узнал о линейных слоях, правилах их обучения, а также узнал где и как они применяются на примере создания адаптивного фильтратора.

Полученные результаты сообщают, что рассматренная нами модель хорошо справляется с одношаговыми прогнозами, но начинает терять динамику изменения функции при небольшом увеличении шагов предсказания. Это связанно во многом с тем, что обученная по четырёхточечному входу модель научиась определять первую производную прогнозируемой функции (что видно по совпадающему направлению прогнозируемых точек) и слабо определять характер второй (это видно по изменению направления последних точек предсказания), но не может по 4 входным точкам определять зависмость в целом. Недаром для аппроксимации первых производных используют трехточечные схемы.

Линейный слой наиболее поход на современные слои нейронных сетей, в том числе алгоритмом Уидроу-Хоффа обучения, который очень напоминает стохастический градиентный спуск.

Эта работа была интересной, но оказалась не такой простой, как первая лабораторная, но я с ней справился, и в будущем я ожидаю более трудных и интересных задач.