Московский авиационный институт
(Национальный исследовательский университет)
Институт №8 «Информационные технологии и прикладная математика»

Кафедра вычислительной математики и программирования

Лабораторная работа № 3

по курсу «Нейроинформатика».

Тема: «Многослойные сети. Алгоритм обратного распространения ошибки».

Студент: Вельтман Л.Я.
Группа: 80-407Б
Преподаватели: Тюменцев Ю.В.
Аносова Н.П.

Вариант: 7

Оценка:

Москва, 2020

Цель работы.

Целью работы является исследование свойств многослойной нейронной сети прямого распространения и алгоритмов ее обучения, применение сети в задачах классификации и аппроксимации функции.

Основные этапы работы.

1. Использовать многослойную нейронную сеть для классификации точек в случае, когда классы не являются линейно разделимыми.
2. Использовать многослойную нейронную сеть для аппроксимации функции. Произвести обучение с помощью одного из методов первого порядка.
3. Использовать многослойную нейронную сеть для аппроксимации функции. Произвести обучение с помощью одного из методов второго порядка.

Оборудование.

Операционная система: macOS Catalina version 10.15.5

Процессор: 2,3 GHz 2-ядерный процессор Intel Core i5

Оперативная память: 8 ГБ 2133 MHz LPDDR3

Программное обеспечение.

Работа выполнена на Python3 с применением библиотек numpy (для вычислений) и matplotlib (для графиков) при помощи командной оболочки Jupyter Notebook.

Сценарий выполнения работы.

Функция traincgp обучает нейронную сеть, используя метод сопряженного градиента с обратным распространением ошибки в модификации Полака – Рибейры.

Алгоритм:
Функция traincgp выполняет процедуру обучения, если функции взвешивания, накопления и активации имеют производные. Для вычисления производных критерия качества обучения по переменным веса и смещения используется метод обратного распространения ошибки. В соответствии с алгоритмом метода сопряженных градиентов вектор настраиваемых переменных получает следующие приращения:
X = X + a*dX,
где dX– направление поиска. Параметр *a* выбирается так, чтобы минимизировать критерий качества обучения в направлении поиска. Функция одномерного поиска searchFcn используется для вычисления минимума. Начальное направление поиска задается вектором, противоположным градиенту критерия качества. При успешных итерациях направление поиска определяется на основе нового значения градиента с учетом прежнего направления поиска согласно формуле
dX = –gX + dX_old*Z,
где gX– вектор градиента; параметр Z может быть вычислен отдельными различными способами. Для метода сопряженного градиента в модификации Полака – Рибейры он рассчитывается согласно формуле
Z = ((gX – gX_old)’*gX)/norm_sqr,
где gX_old – вектор градиента на предыдущей итерации; norm_sqr – квадрат нормы вектора градиента.
Обучение прекращается, когда выполнено одно из следующих условий:

- значение функции качества стало меньше предельного;
- градиент критерия качества стал меньше значения min_grad;
- достигнуто предельное число циклов обучения;
- превышено максимальное время, отпущенное на обучение;
- ошибка контрольного подмножества превысила ошибку обучающего более чем в max_fail раз.

Квазиньютоновские методы — методы оптимизации, основанные на накоплении информации о кривизне целевой функции по наблюдениям за изменением градиента.

Алгоритм Бройдена — Флетчера — Гольдфарба — Шанно (BFGS) (англ. Broyden — Fletcher — Goldfarb — Shanno algorithm) — итерационный метод численной оптимизации, предназначенный для нахождения локального максимума/минимума нелинейного функционала без ограничений.

BFGS — один из наиболее широко применяемых квазиньютоновских методов. В квазиньютоновских методах не вычисляется напрямую гессиан функции. Вместо этого гессиан оценивается приближенно, исходя из сделанных до этого шагов.

$$B_{k+1} = B_k - \frac{1}{\vec{s}_k^T B_k \vec{s}_k} B_k \vec{s}_k \vec{s}_k^T B_k + \frac{1}{\vec{y}_k^T \vec{s}_k} \vec{y}_k \vec{y}_k^T$$

Алгоритм

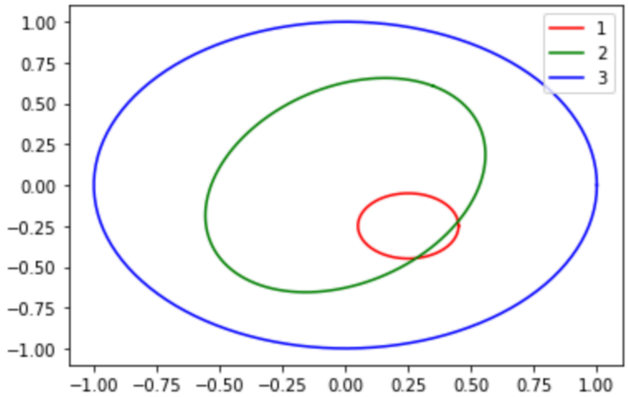
дано $\varepsilon,\ x_0$
инициализировать C_0
$k = 0$
while $||\nabla f_k|| > \varepsilon$
  найти направление $p_k = -C_k \nabla f_k$
  вычислить $x_{k+1} = x_k + \alpha_k p_k$, α_k удовлетворяет условиям Вольфе
  обозначить $s_k = x_{k+1} - x_k$ и $y_k = \nabla f_{k+1} - \nabla f_k$
  вычислить C_{k+1}
  $k = k + 1$
end

Входные данные и результаты

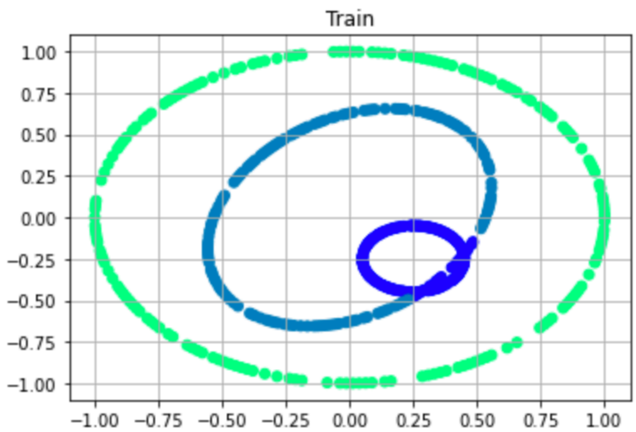
Задание 1

In [6]:
```
plt.plot(x1, y1, 'r', label='1')
plt.plot(x2, y2, 'g', label='2')
plt.plot(x3, y3, 'b', label='3')
plt.legend()
```

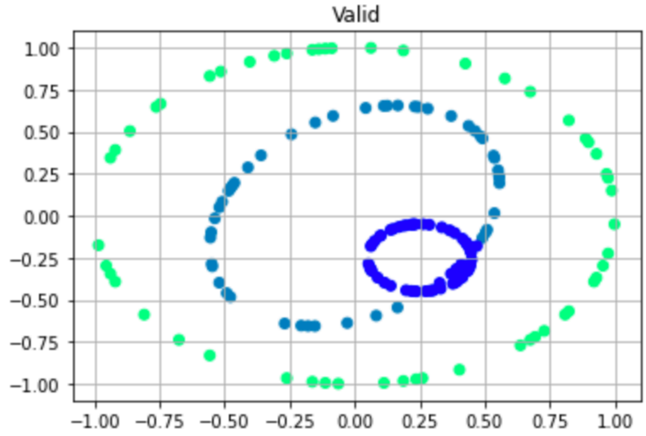
Out[6]: <matplotlib.legend.Legend at 0x7faa9dc3a828>



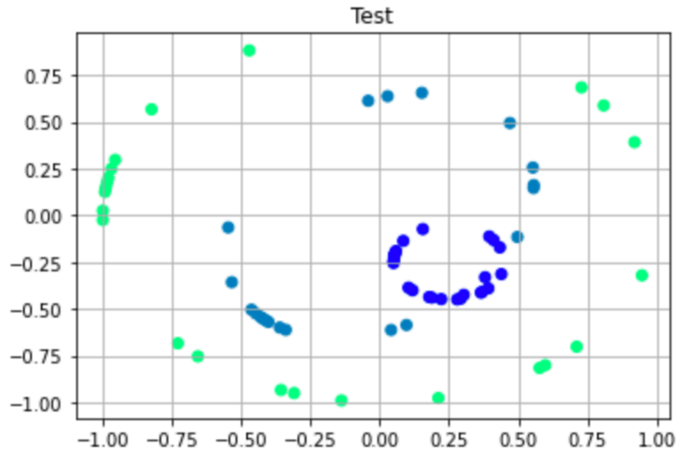
In [15]:
```
plt.scatter(train['x'], train['y'], c=p[0], cmap=plt.cm.winter)
plt.grid(True)
plt.title('Train')
plt.show()
```



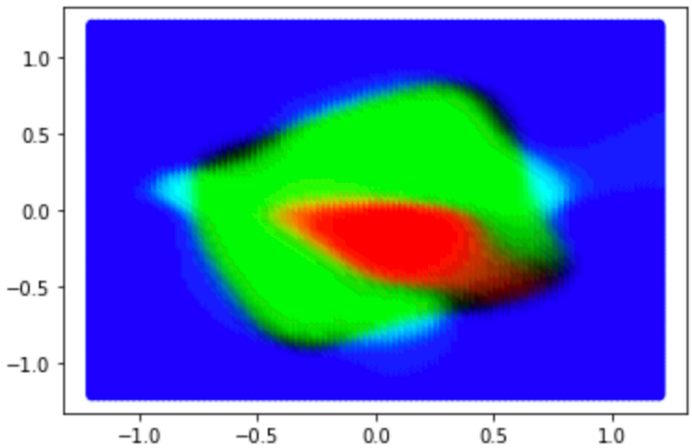
In [17]:
```
plt.scatter(valid['x'], valid['y'], c=p[2], cmap=plt.cm.winter)
plt.grid(True)
plt.title('Valid')
plt.show()
```



In [16]:
```
plt.scatter(test['x'], test['y'], c=p[1], cmap=plt.cm.winter)
plt.grid(True)
plt.title('Test')
plt.show()
```



In [24]:
```
plt.scatter(yy, xx, c=colors, cmap=plt.cm.winter);
plt.show()
```



Задание 2

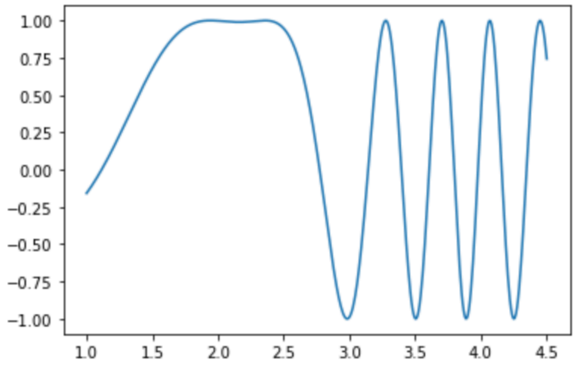
Функция для обучения в соответствии с вариантом задания

In [26]:
```
def f(t):
    return np.sin(np.sin(t) * t**2 - t)

t = np.linspace(1, 4.5, int(4.5 / 0.01), endpoint=True)
x = f(t)
```

In [27]:
```
plt.plot(t, x)
```

Out[27]: [<matplotlib.lines.Line2D at 0x7faa9f0cb668>]



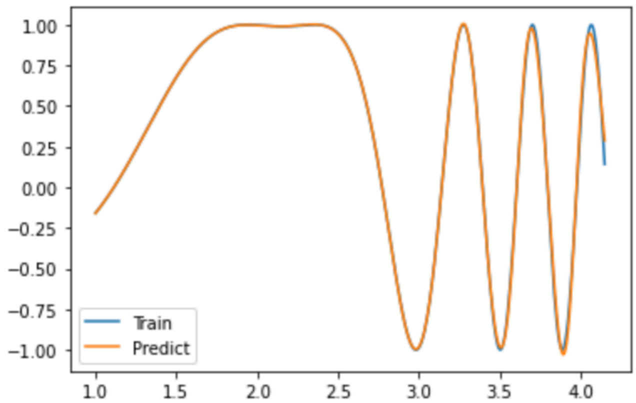
Ошибка предсказания на тренировочной выборке

In [33]:
```
mse = mean_squared_error(yTrain, xPredicted.flatten())
print(f'MSE = {mse}')
print(f'RMSE = {np.sqrt(mse)}')
```

```
MSE = 0.00047683684225462847
RMSE = 0.021836594108391272
```

In [34]:
```
plt.plot(xTrain, yTrain, label='Train')
plt.plot(xTrain, xPredicted, label='Predict')
plt.legend()
```

Out[34]: <matplotlib.legend.Legend at 0x7faaa1427d30>

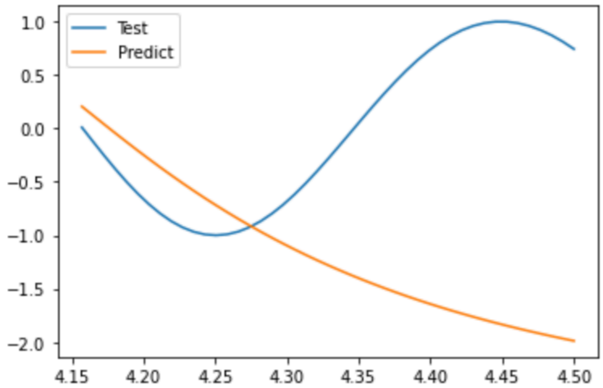


Ошибка предсказания на тестовой выборке

In [36]:
```
mse = mean_squared_error(yTest, xPredicted.flatten())
print(f'MSE = {mse}')
print(f'RMSE = {np.sqrt(mse)}')
```

```
MSE = 2.954872256659715
RMSE = 1.7189741873162945
```

In [37]:
```
plt.plot(xTest, yTest, label='Test')
plt.plot(xTest, xPredicted, label='Predict')
plt.legend()
plt.show()
```



Задание 3

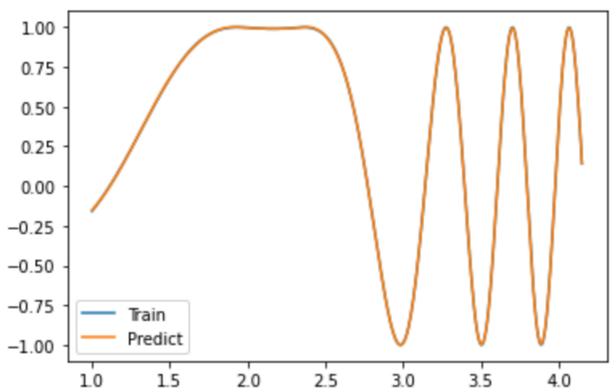
Ошибка предсказания на тренировочной выборке

In [41]:
```
mse = mean_squared_error(yTrain, xPredicted.flatten())
print(f'MSE = {mse}')
print(f'RMSE = {np.sqrt(mse)}')
```

```
MSE = 8.665136320779325e-06
RMSE = 0.0029436603609756555
```

In [42]:
```
plt.plot(xTrain, yTrain, label='Train')
plt.plot(xTrain, xPredicted, label='Predict')
plt.legend()
```

Out[42]: <matplotlib.legend.Legend at 0x7faa838a8080>

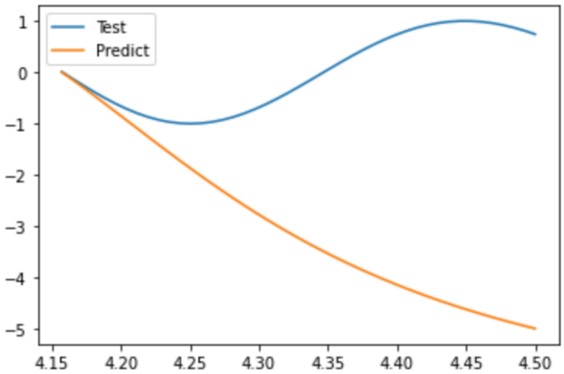


Ошибка предсказания на тестовой выборке

In [44]:
```
mse = mean_squared_error(yTest, xPredicted.flatten())
print(f'MSE = {mse}')
print(f'RMSE = {np.sqrt(mse)}')
```

```
MSE = 13.175384495324069
RMSE = 3.629791246796993
```

In [45]:
```
plt.plot(xTest, yTest, label='Test')
plt.plot(xTest, xPredicted, label='Predict')
plt.legend()
plt.show()
```



Код программы.

```
#!/usr/bin/env python
# coding: utf-8

# ## Задание 1


import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam, Adagrad

from sklearn.metrics import accuracy_score, mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler


a1 = 0.2
b1 = 0.2
x0_1 = 0.25 # координаты параллельного переноса
y0_1 = -0.25
alpha1 = 0 #угол поворота

a2 = 0.7
b2 = 0.5
x0_2 = 0
y0_2 = 0
alpha2 = -np.pi/3

a3 = 1
b3 = 1
x0_3 = 0
y0_3 = 0
alpha3 = 0


t = np.linspace(0, 2 * np.pi, int(2 * np.pi / 0.025), endpoint=True)


def f(alpha, x0, a, t, y0, b):
    return (x0 + a * np.cos(t)) * np.cos(alpha) + (y0 + b * np.sin(t)) * np.sin(alpha)

def g(alpha, x0, a, t, y0, b):
    return -(x0 + a * np.cos(t)) * np.sin(alpha) + (y0 + b * np.sin(t)) * np.cos(alpha)


x1 = f(alpha1, x0_1, a1, t, y0_1, b1)
y1 = g(alpha1, x0_1, a1, t, y0_1, b1)

x2 = f(alpha2, x0_2, a2, t, y0_2, b2)
y2 = g(alpha2, x0_2, a2, t, y0_2, b2)

x3 = f(alpha3, x0_3, a3, t, y0_3, b3)
y3 = g(alpha3, x0_3, a3, t, y0_3, b3)


plt.plot(x1, y1, 'r', label='1')
plt.plot(x2, y2, 'g', label='2')
plt.plot(x3, y3, 'b', label='3')
plt.legend()


# Разделим выборку на тренировочную, тестовую и валидационную в соотношении 70% 10% 20%
соответственно


df1 = pd.DataFrame({'x' : x1, 'y' : y1, 'class' : 0})
df2 = pd.DataFrame({'x' : x2, 'y' : y2, 'class' : 1})
df3 = pd.DataFrame({'x' : x3, 'y' : y3, 'class' : 2})


def Splitter(data):
    xTrain, xTest = train_test_split(data, test_size=0.3, shuffle=True, random_state=42)
    xValid, xTest = train_test_split(xTest, test_size=0.3, shuffle=True, random_state=42)
    return xTrain, xTest, xValid


train, test, valid = [], [], []

tmpTrain, tmpTest, tmpValid = Splitter(df1)
train.append(tmpTrain)
test.append(tmpTest)
valid.append(tmpValid)

tmpTrain, tmpTest, tmpValid = Splitter(df2)
train.append(tmpTrain)
test.append(tmpTest)
valid.append(tmpValid)
```

```
tmpTrain, tmpTest, tmpValid = Splitter(df3)
train.append(tmpTrain)
test.append(tmpTest)
valid.append(tmpValid)

train = pd.concat(train)
valid = pd.concat(valid)
test = pd.concat(test)


# Архитектура сети



model = Sequential()
model.add(Dense(20, input_shape=(2,), activation='tanh'))
model.add(Dense(3, activation='sigmoid'))


model.compile(Adam(lr=0.01), 'binary_crossentropy', metrics=['accuracy'])


# Обучаем нейросеть на 300 эпохах.


y = pd.get_dummies(train['class'])
history = model.fit(train.iloc[:, :-1], y, epochs=300, shuffle=True)


# Метрики обучения

p = []

p.append(model.predict_classes(train.iloc[:, :-1]))
accTrain = accuracy_score(train['class'], p[-1])
mseTrain = mean_squared_error(train['class'], p[-1])

p.append(model.predict_classes(test.iloc[:, :-1]))
accTest = accuracy_score(test['class'], p[-1])
mseTest = mean_squared_error(test['class'], p[-1])

p.append(model.predict_classes(valid.iloc[:, :-1]))
accValid = accuracy_score(valid['class'], p[-1])
mseValid = mean_squared_error(valid['class'], p[-1])


print('Train accuracy = {}'.format(accTrain))
print(f'Train MSE = {mseTrain}')
print(f'Train RMSE = {np.sqrt(mseTrain)}\n')

print('Test accuracy = {}'.format(accTest))
print(f'Test MSE = {mseTest}')
print(f'Test RMSE = {np.sqrt(mseTest)}\n')

print('Valid accuracy = {}'.format(accValid))
print(f'Valid MSE = {mseValid}')
print(f'Valid RMSE = {np.sqrt(mseValid)}')


plt.scatter(train['x'], train['y'], c=p[0], cmap=plt.cm.winter)
plt.grid(True)
plt.title('Train')
plt.show()



plt.scatter(test['x'], test['y'], c=p[1], cmap=plt.cm.winter)
plt.grid(True)
plt.title('Test')
plt.show()


plt.scatter(valid['x'], valid['y'], c=p[2], cmap=plt.cm.winter)
plt.grid(True)
plt.title('Valid')
plt.show()


# Зададим область точек [-1.2, 1.2] x [-1.2, 1.2]. Получим сетку для указанной области с шагом h =
0.025.



h = 0.025
x = np.arange(-1.2, 1.2 + h, h)
y = np.arange(-1.2, 1.2 + h, h)


# По уже обученной модели предскажем класс для каждой точки сетки.


h = 0.025
predictions = [model.predict(np.array([[i, j]])).round(1) for i in x for j in y]


# Получим матрицу координат из координатных векторов.


xx, yy = np.meshgrid(x, y)
```

```
# Формируем таблицу цветов


colors = np.array(predictions).reshape((len(predictions), 3))
colors.shape


plt.scatter(yy, xx, c=colors, cmap=plt.cm.winter);
plt.show()


# ## Задание 2

# #### Метод 1-го порядка:
# ### Метод сопряженных градиентов:
# #### traincgp - метод полака-рибейры
# количество нейронов в скрытом слое 10

# In[25]:


from neupy import algorithms
from neupy.layers import Input, Tanh, Linear, Sigmoid


# Функция для обучения в соответствии с вариантом задания


def f(t):
    return np.sin(np.sin(t) * t**2 - t)

t = np.linspace(1, 4.5, int(4.5 / 0.01), endpoint=True)
x = f(t)


plt.plot(t, x)


# Делим на тестовую и тренировочную выборку

percentTrain = 0.9

trainSize = int(len(t) * percentTrain)

xTrain = t[:trainSize]
yTrain = x[:trainSize]
xTest = t[trainSize:]
yTest = x[trainSize:]


# Нормализуем тестовые и тренировочные данные.



scaler_x = StandardScaler()
scaler_y = StandardScaler()

scaledTrainX = scaler_x.fit_transform(xTrain[:, np.newaxis])
scaledTestX = scaler_x.transform(xTest[:, np.newaxis])
scaledTrainY = scaler_y.fit_transform(yTrain[:, np.newaxis])


# Архитектура нейросети


traincgp = algorithms.ConjugateGradient(network=[Input(1),
                                                 Tanh(15),
                                                 Linear(1),],
                                        update_function='polak_ribiere', verbose=True)


traincgp.train(scaledTrainX, scaledTrainY, epochs=4000)


# Предсказание по тренировочной выборке


xPredicted = traincgp.predict(scaledTrainX)
xPredicted = scaler_y.inverse_transform(xPredicted)


# Ошибка предсказания на тренировочной выборке


mse = mean_squared_error(yTrain, xPredicted.flatten())
print(f'MSE = {mse}')
print(f'RMSE = {np.sqrt(mse)}')


plt.plot(xTrain, yTrain, label='Train')
plt.plot(xTrain, xPredicted, label='Predict')
plt.legend()


# Предсказание по тестовой выборке
```

```
xPredicted = traincgp.predict(scaledTestX)
xPredicted = scaler_y.inverse_transform(xPredicted)


# Ошибка предсказания на тестовой выборке


mse = mean_squared_error(yTest, xPredicted.flatten())
print(f'MSE = {mse}')
print(f'RMSE = {np.sqrt(mse)}')


plt.plot(xTest, yTest, label='Test')
plt.plot(xTest, xPredicted, label='Predict')
plt.legend()
plt.show()


# ## Задание 3

# #### Метод второго порядка:
# ### Квазиньютоновский метод, предложенный Бройденом, Флетчером, Гольдфарбом, Шанно.



trainbfg = algorithms.QuasiNewton(network=[Input(1),
                                           Tanh(128),
                                           Linear(1),],
                                  update_function='bfgs', verbose=True)


# Обучаем нейросеть на 300 эпохах.


trainbfg.train(scaledTrainX, scaledTrainY, epochs=300)


# Предсказание по тренировочной выборке

xPredicted = trainbfg.predict(scaledTrainX)
xPredicted = scaler_y.inverse_transform(xPredicted)


# Ошибка предсказания на тренировочной выборке

mse = mean_squared_error(yTrain, xPredicted.flatten())
print(f'MSE = {mse}')
print(f'RMSE = {np.sqrt(mse)}')

plt.plot(xTrain, yTrain, label='Train')
plt.plot(xTrain, xPredicted, label='Predict')
plt.legend()


# Предсказание по тестовой выборке


xPredicted = trainbfg.predict(scaledTestX)
xPredicted = scaler_y.inverse_transform(xPredicted)


# Ошибка предсказания на тестовой выборке


mse = mean_squared_error(yTest, xPredicted.flatten())
print(f'MSE = {mse}')
print(f'RMSE = {np.sqrt(mse)}')



plt.plot(xTest, yTest, label='Test')
plt.plot(xTest, xPredicted, label='Predict')
plt.legend()
plt.show()
```

Выводы:

Применение алгоритма **обратного распространения ошибки** — один из известных методов, используемых для глубокого обучения нейронных сетей прямого распространения.
К плюсам можно отнести простоту в реализации и устойчивость к выбросам и аномалиям в данных, и это основные преимущества. Но есть и минусы:

- неопределенно долгий процесс обучения;

- вероятность «паралича сети» (при больших значениях рабочая точка функции активации попадает в область насыщения сигмоиды, а производная величина приближается к 0, в результате чего коррекции весов почти не происходят, а процесс обучения «замирает»;
- алгоритм уязвим к попаданию в локальные минимумы функции ошибки.