**Московский авиационный институт**

**(Национальный исследовательский университет)**

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

**Лабораторная работа № 3**

по курсу «Нейроинформатика».

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

Студент: Якимович А.И.

Группа: 80-408Б

Вариант: 20

Оценка:

Москва, 2018

Постановка задачи.

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

Задание 1: использовать многослойную нейронную сеть для классификации точек в случае, когда классы не являются линейно разделимыми.

*Вариант 20:*

*Эллипс: a = 0.4, b = 0.4, α = 0, x0 = −0.1, y0 = 0.15*

*Эллипс: a = 0.7, b = 0.7, α = 0, x0 = 0, y0 = 0*

*Парабола: p = −1, α = 0, x0 = 0.8, y0 = 0*

Задание 2: использовать многослойную нейронную сеть для аппроксимации функции. Произвести обучение с помощью одного из методов первого порядка.

Задание 3: использовать многослойную нейронную сеть для аппроксимации функции. Произвести обучение с помощью одного из методов второго порядка.

*Вариант 20: x = cos(cos(t)\*t^2 + 5t) , t ∈ [0, 3.5], h = 0.01*

Описание алгоритма.

В качестве метода поиска экстремума функции многих переменных первого порядка был выбран метод градиентного спуска, второго – Метод Левенберга-Марквардта.

Алгоритм наискорейшего спуска.

*– скорость обучения*

Слоистая сеть представляет собой композицию отображений или, что то же самое, сложную функцию, например, если y = f (u), где u = g(x), такая функция имеет вид y = f (g(x)). Производная сложной функции равна произведению ее производной по промежуточному аргументу на производную этого аргумента по независимой переменной (цепное правило дифференцирования).

Применение этого правила играет ключевую роль в методе обратного распространения ошибки и его разновидностях. Используя цепное правило дифференцирования, можно осуществить вычисление градиента функции ошибки. Следовательно, требуется дифференцируемость активационных функций в сети, а также желательность существования для этих функций производных в виде явных формул.

Алгоритм обратного распространения ошибки.

1. Весам сети W присваиваются небольшие начальные значения.
2. Выбирается очередной обучающий пример из обучающего набора

; вектор подается на вход сети.

1. Вычисляется выход сети Y(; W), отвечающий входному вектору и принятым значениям весов W.
2. На основе желаемого выхода для примера и реально полученного (вычисленного на предыдущем шаге) выхода Y( ;W) вычисляется ошибка сети для данного примера: = -
3. Шаги 2, 3 и 4 повторяются для всех обучающих примеров ; при одном и том же значении W; эти шагов составляют одну эпоху в процессе обучения рассматриваемой сети.
4. Вычисляется суммарная ошибка сети для данной r–й эпохи: =.
5. Веса сети W корректируются так, чтобы уменьшить ошибку .
6. Шаги со 2 по 7 повторяются до тех пор, пока не будет выполнено условие < , где >0 — наперед заданное малое число; процесс обучения может быть остановлен также по достижении предельно допустимого числа эпох.

Рассмотрим пункт 7.

Ошибка сети непосредственно связана лишь с выходами сети и весами связей выходного слоя и предшествующего ему скрытого слоя. Надо уметь определять ошибку на выходах нейронов скрытых слоев по ошибке в последующем слое: ошибку для последнего скрытого слоя по ошибке выходного слоя, ошибку для предпоследнего скрытого слоя — по ошибке последнего скрытого слоя и т. д. Ошибка всей сети «распространяется назад», от выходного слоя сети к ее входному слою, отсюда название соответствующего алгоритма. Переход «назад» от слоя к слою в алгоритме BP сопровождается не только вычислением ошибки на соответствующем слое сети, но и пересчетом весов данного слоя, уменьшающим его ошибку. Веса на всех слоях изменяются одновременно.

Общая информация.

Программа написана на языке Python с применением библиотек numpy, neupy, sklearn, pandas и matplotlib (для графиков).

Данная программа может решать задачи связанные с аппроксимацией функций многих переменных, классификацией объектов, в том числе и линейно-неразделимых.

Запуск программы.

Чтобы воспользоваться программой, необходимо запустить программу на любом интерпретаторе Python версии 3.x.x.

Входные данные и результаты.

Вариант 20.

|  |  |
| --- | --- |
| **Вход** | **Выход** |
| C:\Users\now20\AppData\Local\Microsoft\Windows\INetCache\Content.Word\lab3task1start.png | C:\Users\now20\AppData\Local\Microsoft\Windows\INetCache\Content.Word\lab3_task1_train.png  C:\Users\now20\AppData\Local\Microsoft\Windows\INetCache\Content.Word\test.pngC:\Users\now20\AppData\Local\Microsoft\Windows\INetCache\Content.Word\valid.png |
| *x = cos(cos(t)\*t^2 + 5t) , t ∈ [0, 3.5], h = 0.01*  *Метод градиентного спуска* | C:\Users\now20\AppData\Local\Microsoft\Windows\INetCache\Content.Word\lab3_task2.png  **Квадратичная ошибка на train = 0.02084**  **Квадратичная ошибка на test = 0.23274** |
| *x = cos(cos(t)\*t^2 + 5t) , t ∈ [0, 3.5], h = 0.01*  *Метод Левенберга-Марквардта* | **Квадратичная ошибка на train = 0.00049**  **Квадратичная ошибка на test = 0.03559** |

Выводы:

Метод обратного распространения ошибки является мощным инструментом обучения многослойных сетей прямого распространения, это подтверждают результаты лабораторной работы. Однако несмотря на многочисленные успешные применения обратного распространения, оно не является универсальным решением. Больше всего неприятностей приносит неопределённо долгий процесс обучения. В сложных задачах для обучения сети могут потребоваться дни или даже недели, она может и вообще не обучиться.

Для обучения используют различные методы нахождения градиента функции многих переменных. Их можно разделить на две группы: первого и второго порядка. Методы первого порядка обычно проще для понимания и программирования, к тому же в каждой эпохе применение этих методов менее затратно с точки зрения производительности, но к сожалению, эти методы имеют ряд существенных недостатков, в частности — метод наискорейшего спуска может очень долго сходиться в конце оптимизации. Поэтому применяют методы второго порядка, в частности метод Левенберга — Марквардта. Он имеет такие преимущества как: высокая устойчивость, большие шансы найти глобальный экстремум, очень высокая скорость сходимости(адаптивная). Но есть и недостатки: он налагает ограничения на вид целевой функции, сложен в реализации.

Сеть показывает плохие результаты на том отрезке, где не было обучающих данных. Это подтверждает утверждение от том, что нельзя давать сети на вход данные из области, не покрытые обучающей выборкой.

Исходный код.

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 = 0.4

b1 = 0.4

x0\_1 = 0.1

y0\_1 = -0.15

a2 = 0.7

b2 = 0.7

x0\_2 = 0

y0\_2 = 0

p3 = -1

x0\_3 = 0.8

y0\_3 = -0.8

step = 0.025

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

x = np.linspace(-1, 1, int(2/step), endpoint=True)

def f(x0, a, t):

return x0 + a\*np.cos(t)

def g(y0, b, t):

return y0 + b\*np.sin(t)

x1 = f(x0\_1, a1, t)

y1 = g(y0\_1, b1, t)

x2= f(x0\_2, a2, t)

y2= g(y0\_2, b2, t)

#x3= f(a3, t)

#y3= g(b3, t)

x3 = x + x0\_3

y3 = p3 \* x \* x + y0\_3

plt.plot(x1, y1, 'navy')

plt.plot(x2, y2, 'purple')

plt.plot(x3, y3, 'yellow')

plt.grid(True)

plt.show()

df1 = pd.DataFrame({'x' : x1, 'y' : y1, 'target' : 0})

df2 = pd.DataFrame({'x' : x2, 'y' : y2, 'target' : 1})

df3 = pd.DataFrame({'x' : x3, 'y' : y3, 'target' : 2})

def split\_df(df):

x\_train, x\_test = train\_test\_split(df, test\_size=0.3, shuffle=True, random\_state=21)

x\_valid, x\_test = train\_test\_split(x\_test, test\_size=0.3, shuffle=True, random\_state=14)

return x\_train, x\_valid, x\_test

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'])

train = []

valid = []

test = []

for df in (df1, df2 ,df3):

tr, v, te = split\_df(df)

train.append(tr)

valid.append(v)

test.append(te)

train = pd.concat(train)

valid = pd.concat(valid)

test = pd.concat(test)

y = pd.get\_dummies(train['target'])

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

p = []

p.append(model.predict\_classes(train.iloc[:, :-1]))

accuracy\_score(train['target'], p[-1])

p.append(model.predict\_classes(test.iloc[:, :-1]))

accuracy\_score(test['target'], p[-1])

p.append(model.predict\_classes(valid.iloc[:, :-1]))

accuracy\_score(valid['target'], p[-1])

titles = ['train', 'test', 'valid']

for idx, df in enumerate((train, test, valid)):

plt.scatter(df.x, df.y, c=p[idx], cmap=plt.cm.plasma)

plt.grid(True)

plt.title(titles[idx])

plt.show()

h = 0.025

grid\_pred = [model.predict(np.array([[i, j]])).round(1) for i in np.arange(-1.2, 1.2+h, h)

for j in np.arange(-1.2, 1.2+h, h)]

x\_vals = np.arange(-1.2, 1.2+h, h)

y\_vals = np.arange(-1.2, 1.2+h, h)

xx, yy = np.meshgrid(x\_vals, y\_vals)

rows = len(grid\_pred)

colors = np.array(grid\_pred).reshape((rows, 3))

colors.shape

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

plt.show()

………………………………………………………………………………………………………………………………………………………………………………………………………………………………………

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

h = 0.01

t = np.linspace(0, 3.5, int(3.5/0.02), endpoint=True)

x = np.cos(np.cos(t) \* t\*\*2 + 5\*t)

plt.plot(t, x);

train\_size = int(t.shape[0] \* 0.9)

X\_train = t[:train\_size]

y\_train = x[:train\_size]

X\_test = t[train\_size:]

y\_test = x[train\_size:]

scaler\_x = StandardScaler()

scaler\_y = StandardScaler()

tmp\_train\_scaled\_x = scaler\_x.fit\_transform(X\_train[:, np.newaxis])

tmp\_test\_scaled\_x = scaler\_x.transform(X\_test[:, np.newaxis])

tmp\_train\_scaled\_y = scaler\_y.fit\_transform(y\_train[:, np.newaxis])

model = Sequential()

model.add(Dense(20, input\_shape=(1,), activation='tanh'))

model.add(Dense(1))

model.compile(loss='mean\_squared\_error', optimizer='adam')

history = model.fit(tmp\_train\_scaled\_x, tmp\_train\_scaled\_y , epochs=12000, verbose=1)

pred\_x = model.predict(tmp\_train\_scaled\_x)

pred\_x = scaler\_y.inverse\_transform(pred\_x)

mse = mean\_squared\_error(y\_train, pred\_x.flatten())

print(f'RMSE = {np.sqrt(mse)}')

plt.plot(X\_train, y\_train, label='train')

plt.plot(X\_train, pred\_x, label='predict')

plt.legend();

pred\_x = model.predict(tmp\_test\_scaled\_x[:, 0])

pred\_x = scaler\_y.inverse\_transform(pred\_x)

mse = mean\_squared\_error(y\_test, pred\_x.flatten())

print(f'RMSE = {np.sqrt(mse)}')

plt.plot(X\_test, y\_test, label='test')

plt.plot(X\_test, pred\_x, label='predict')

plt.legend();

plt.show()

……………………………………………………………………………………………………………………………………………………………………………………………………………………………………………

import numpy as np

import sklearn.metrics

import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler

from neupy import algorithms

from neupy.layers import Input, Tanh, Linear

h = 0.01

t = np.linspace(0, 3.5, int(3.5/0.02), endpoint=True)

x = np.cos(np.cos(t) \* t\*\*2 + 5\*t)

train\_size = int(t.shape[0] \* 0.9)

train\_size

X\_train = t[:train\_size]

y\_train = x[:train\_size]

X\_test = t[train\_size:]

y\_test = x[train\_size:]

scaler\_x = StandardScaler()

scaler\_y = StandardScaler()

tmp\_train\_scaled\_x = scaler\_x.fit\_transform(X\_train[:, np.newaxis])

tmp\_test\_scaled\_x = scaler\_x.transform(X\_test[:, np.newaxis])

tmp\_train\_scaled\_y = scaler\_y.fit\_transform(y\_train[:, np.newaxis])

lmnet = algorithms.LevenbergMarquardt((Input(1), Tanh(60), Linear(1)), verbose=True)

lmnet.train(X\_train, y\_train, epochs=100)

pred\_x = lmnet.predict(X\_train)

mse = sklearn.metrics.mean\_squared\_error(y\_train, pred\_x.flatten())

print(f'RMSE = {np.sqrt(mse)}')

plt.plot(X\_train, y\_train, label='train')

plt.plot(X\_train, pred\_x, label='predict')

plt.legend();

pred\_x = lmnet.predict(X\_test)

mse = sklearn.metrics.mean\_squared\_error(y\_test, pred\_x.flatten())

print(f'RMSE = {np.sqrt(mse)}')

plt.plot(X\_test, y\_test, label='test')

plt.plot(X\_test, pred\_x, label='predict')

plt.legend();

plt.show()