Одеський національний університет імені І. І. Мечникова
Інститут математики, економіки і механіки
Кафедра методів математичної фізики

# Дипломна робота

бакалавра

на тему: **«Нечіткі штучні нейронні мережі»**

«Fuzzy artificial neural networks»

Виконав: студент денної форми навчання
напряму підготовки
6.040301 Прикладна математика
Бондаренко Станіслав Сергійович

Керівник: к. ф.-м. н., доц. Шумихин С. А.
Рецензент: к. ф.-м. н., доц. Мойсєєнок О. П.

Рекомендовано до захисту:
Протокол засідання кафедри
№ ___ від «_____» _____ р.
Завідувач кафедри
_____ _____

Захищено на засіданні ЕК № _____
Протокол № ____ від «____» ____ р.
Оцінка _____ / _____ / _____
Голова ЕК
_____ _____

Одеса — 2017 р.

# ЗМІСТ

# ВСТУП

Штучні нейронні мережі застосовують в різноманітних сферах науки: починаючи з систем розпізнавання мовлення до розпізнавання вторинної структури білка, класифікації різних видів раку та у генній інженерії. Коли мова йде про задачі, відмінні від обробки великих масивів даних, людський мозок має багато переваг у порівнянні з комп'ютером. Людина може розпізнавати обличчя, навіть якщо у приміщенні буде багато сторонніх об'єктів та погане освітлення. Людина легко розпізнає звуки навіть коли знаходиться в шумному приміщенні. Але найбільш цікавою особливістю мозку є те, що він здатен навчатися.

Будь-яка нейронна мережа використовується в якості самостійної системи уявлення знань, яка на практиці, як правило, є компонентом системи керування чи прийняття рішень, який передає результат на інші елементи, які не пов'язані с мережею. Функції нейронних мереж можна розділити на декілька основних груп: апроксимація й інтерполяція; розпізнавання і класифікація образів; стиснення даних; прогнозування; керування; асоціації. У кожній наведеній групі нейронна мережа виконує роль універсального апроксиматора функції багатьох змінних[1].

Існує дуже багато систем для яких математичні моделі дужне складні або ще не існують і для них неможливо застосувати традиційні методи аналізу. В таких випадках, альтернативою може бути підхід, який ґрунтується на м'яких розрахунках. Концепція м'яких розрахунків (soft computation) бере свій початок в роботах Лотфі Заде, який працював над нечіткою логікою та мягким аналізом (soft analysis). Це стало новим поштовхом у розвитку інтелектуальних систем. Проте, створення справжніх інтелектуальних систем стало можливим відносно недавно (у 2009 році рекурентні та глибинні нейронні мережі показали найкращі результати в задачах розпізнавання образів та машинного навчання), що пов'язана зі стрімким ростом розрахункових потужностей. На відміну від жорстких обчислень (hard computing), м'які обчислення враховують похибки, які дуже часто зустрічаються у реальному житті. М'які обчислення не такі чутливі до похибок та невизначеності, що дозволяє отримати більш надійні та приближенні до реальності рішення.

Замість одного методу, м'які обчислення можуть використовувати комбінацію різних теорій та методів, як наприклад, нейронні мережі, нечітка логіка та генетичні алгоритми.

Метою дипломної роботи є дослідження нечітких нейронних мереж, їх ефективність в різних задачах аналізу даних та машинного навчання. Дослідження відомих алгоритмів навчання нечітких мереж та їх модифікацій.

# ВИСНОВКИ

У роботі розглянуто нечітку штучну нейронну мережу типу ANFIS (Adaptive Neuro Fuzzy Inference System) на основі нечіткого виводу. Розроблена програма на мові програмування Python, яка моделює саму мережу та реалізую два методи її навчання: гібридний алгоритм та алгоритм PSO (Particle Swatm Optimization). Для порівняння результатів проведено тести на чотирьох класичних наборах даних для задач класифікації.

Метод PSO є відносно новим підходом до вирішення задачі навчання нейронних мереж в цілому. У роботі було розглянуто класичний алгоритм PSO, проте він також показав задовільні результати у вирішенні задачі класифікації за допомогою ANFIS. Точність роботи мережі значною мірою залежить навчальної вибірки та структури самої мережі: кількість правил, види функцій приналежності, тощо. Звичайно, нелегко знайти найпростішу структура, яка даватиме найкращу точність і ще важче знайти оптимальну мережу, яка даватиме найкращий результат на будь-яких наборах даних. Проте, запропонований метод PSO має значний потенціал за рахунок модифікацій. Серед них вже відомі паралельні та асинхронні реалізації, а також модифікація на основі генетичних операцій.

# СПИСОК ЛІТЕРАТУРИ


1. С. Осовский "Нейронные сети для обработки информации"/ Пер. с польского И.Д. Рудинского. - М.: Финансы и статистика, 2002.
2. Мак-Каллок У. С., Питтс В. Логическое исчисление идей, относящихся к нервной активности // Автоматы / Под ред. К. Э. Шеннона и Дж. Маккарти. — М.: Изд-во иностр. лит., 1956. — С. 363—384. (Перевод английской статьи 1943 г.)
3. Cybenko, G.V. (1989). Approximation by Superpositions of a Sigmoidal function, Mathematics of Control, Signals and Systems, vol. 2 no. 4 pp. 303-314.
4. Галушкин А. И. Синтез многослойных систем распознавания образов. — М.: «Энергия», 1974.
5. Rumelhart D.E., Hinton G.E., Williams R.J., Learning Internal Representations by Error Propagation. In: Parallel Distributed Processing, vol. 1, pp. 318—362. Cambridge, MA, MIT Press. 1986.
6. Wasserman P. D. Experiments in translating Chinese characters using backpropagation. Proceedings of the Thirty-Third IEEE Computer Society International Conference.. — Washington: D. C.: Computer Society Press of the IEEE, 1988.
7. Штовба, С.Д. Введение в теорию нечетких множеств и нечеткую логику [Текст]: Монография / С.Д. Штоба. – Винница: Континент-Прим, 2003. –198 с.
8. Kennedy J, Eberhart R (1995) Particle swarm optimization. In: Neural networks, 1995. Proceedings., IEEE international conference on IEEE, vol 4. pp 1942–1948
9. Deng W, Wang G, Yang S, Hu F (2011) A new method for inconsistent multicriteria classification. Rough sets and knowledge technology, pp 600–609

Додаток А

## Код програми

```
# -*- coding: utf-8 -*-
import itertools
import numpy as np
from membership import mfDerivs, membershipfunction
import copy
import random
import sys


class ANFIS:

    def __init__(self, X, Y, memFunction):
        self.X = np.array(copy.copy(X))
        self.Y = np.array(copy.copy(Y))
        self.XLen = len(self.X)
        self.memClass = copy.deepcopy(memFunction)
        self.memFuncs = self.memClass.MFList
        self.memFuncsByVariable = [[x for x in range(len(
            self.memFuncs[z]))] for z in range(len(self.
            memFuncs))]
        self.rules = np.array(list(itertools.product(*
            self.memFuncsByVariable)))
        self.consequents = np.empty(self.Y.ndim * len(
            self.rules) * (self.X.shape[1] + 1))
        self.consequents.fill(0)
        self.errors = np.empty(0)
        self.memFuncsHomo = all(len(i) == len(self.
            memFuncsByVariable[0]) for i in self.
            memFuncsByVariable)
        self.trainingType = 'Not trained yet'
```

```
def LSE(self, A, B, initialGamma=1000.):
    coeffMat = A
    rhsMat = B
    S = np.eye(coeffMat.shape[1]) * initialGamma
    x = np.zeros((coeffMat.shape[1], 1))  # need to
       correct for multi-dim B
    for i in range(len(coeffMat[:, 0])):
        a = coeffMat[i, :]
        b = np.array(rhsMat[i])
        S = S - (np.array(np.dot(np.dot(np.dot(S, np.
           matrix(a).transpose()), np.matrix(a)), S))
           ) / (
             1 + (np.dot(np.dot(S, a), a)))
        x = x + (np.dot(S, np.dot(np.matrix(a).
           transpose(), (np.matrix(b) - np.dot(np.
           matrix(a), x)))))
    return x

def trainHybridJangOffLine(self, epochs=5, tolerance
   =1e-5, initialGamma=1000, k=0.01):

    self.trainingType = 'trainHybridJangOffLine'
    convergence = False
    epoch = 1

    while (epoch < epochs) and (convergence is not
       True):

        # layer four: forward pass
        [layerFour, wSum, w] = forwardHalfPass(self,
           self.X)

        # layer five: least squares estimate
        layerFive = np.array(self.LSE(layerFour, self
```

```
    .Y, initialGamma))
self.consequents = layerFive
layerFive = np.dot(layerFour, layerFive)

# error
error = np.sum((self.Y - layerFive.T) ** 2)
print('current error: ', error)
average_error = np.average(np.absolute(self.Y
    - layerFive.T))
self.errors = np.append(self.errors, error)

if len(self.errors) != 0:
    if self.errors[len(self.errors) - 1] <
        tolerance:
          convergence = True

# back propagation
if convergence is not True:
    cols = range(len(self.X[0, :]))
    dE_dAlpha = list(backprop(self, colX,
        cols, wSum, w, layerFive) for colX in
        range(self.X.shape[1]))

if len(self.errors) >= 4:
    if self.errors[-4] > self.errors[-3] >
        self.errors[-2] > self.errors[-1]:
          k = k * 1.1

if len(self.errors) >= 5:
    if (self.errors[-1] < self.errors[-2])
        and (self.errors[-3] < self.errors
        [-2]) and (
                self.errors[-3] < self.errors
                    [-4]) and (self.errors[-5]
```

```
                    > self.errors[-4]):
            k = k * 0.9

    # handling of variables with a different
       number of MFs
    t = []
    for x in range(len(dE_dAlpha)):
        for y in range(len(dE_dAlpha[x])):
            for z in range(len(dE_dAlpha[x][y])):
                t.append(dE_dAlpha[x][y][z])

    eta = k / np.abs(np.sum(t))

    if np.isinf(eta):
        eta = k

    # handling of variables with a different
       number of MFs
    dAlpha = copy.deepcopy(dE_dAlpha)
    if not self.memFuncsHomo:
        for x in range(len(dE_dAlpha)):
            for y in range(len(dE_dAlpha[x])):
                for z in range(len(dE_dAlpha[x][y
                   ])):
                    dAlpha[x][y][z] = -eta *
                       dE_dAlpha[x][y][z]
    else:
        dAlpha = -eta * np.array(dE_dAlpha)

    for varsWithMemFuncs in range(len(self.
       memFuncs)):
        for MFs in range(len(self.
           memFuncsByVariable[varsWithMemFuncs]))
           :
```

```
                    paramList = sorted(self.memFuncs[
                        varsWithMemFuncs][MFs][1])
                    for param in range(len(paramList)):
                        self.memFuncs[varsWithMemFuncs][
                            MFs][1][paramList[param]] = \
                                self.memFuncs[
                                    varsWithMemFuncs][MFs][1][
                                    paramList[param]] + dAlpha
                                    [varsWithMemFuncs][MFs][
                                        param]
            epoch = epoch + 1

        self.fittedValues = predict(self, self.X)
        self.residuals = self.Y - self.fittedValues[:, 0]

        return self.fittedValues

    def PSOTrain(self, max_epochs=10, swarm_size=50,
        tolerance=1e-5):
        mfParams_dim = 0
        # layer four: forward pass
        [layerFour, wSum, w] = forwardHalfPass(self, self
            .X)
        # layer five: least squares estimate
        layerFive = np.array(self.LSE(layerFour, self.Y))
        self.consequents = layerFive
        for i in range(len(self.memFuncs)):
            for j in range(len(self.memFuncs[i])):
                for key in self.memFuncs[i][j][1]:
                    mfParams_dim += 1
        newMfParams = PSO(self, max_epochs, swarm_size,
            len(self.consequents), -50, 50, params_type='
            consequent')
        self.consequents = copy.deepcopy(newMfParams)
```

```
        newMfParams = PSO(self, max_epochs, swarm_size,
            mfParams_dim, -50, 50, params_type='antecedent
            ')
        self.memFuncs = copy.deepcopy(set_params(self,
            newMfParams))

    def plotErrors(self):
        if self.trainingType == 'Not trained yet':
            print(self.trainingType)
        else:
            import matplotlib.pyplot as plt
            plt.plot(range(len(self.errors)), self.errors
                , 'ro', label='errors')
            plt.ylabel('error')
            plt.xlabel('epoch')
            plt.show()

    def plotMF(self, x, inputVar):
        import matplotlib.pyplot as plt
        from skfuzzy import gaussmf, gbellmf, sigmf

        for mf in range(len(self.memFuncs[inputVar])):
            if self.memFuncs[inputVar][mf][0] == 'gaussmf
                ':
                y = gaussmf(x, **self.memClass.MFList[
                    inputVar][mf][1])
            elif self.memFuncs[inputVar][mf][0] == '
                gbellmf':
                y = gbellmf(x, **self.memClass.MFList[
                    inputVar][mf][1])
            elif self.memFuncs[inputVar][mf][0] == 'sigmf
                ':
                y = sigmf(x, **self.memClass.MFList[
                    inputVar][mf][1])
```

```
                plt.plot(x, y)

            plt.show()

        def plotResults(self):
            if self.trainingType == 'Not trained yet':
                print(self.trainingType)
            else:
                import matplotlib.pyplot as plt
                plt.plot(range(len(self.fittedValues)), self.
                    fittedValues, 'r', label='trained')
                plt.plot(range(len(self.Y)), self.Y, 'b',
                    label='original')
                plt.legend(loc='upper left')
                plt.show()


def forwardHalfPass(ANFISObj, Xs):
    layerFour = np.empty(0, )
    wSum = []

    for pattern in range(len(Xs[:, 0])):
        # layer one
        layerOne = ANFISObj.memClass.evaluateMF(Xs[
            pattern, :])

        # layer two
        miAlloc = [[layerOne[x][ANFISObj.rules[row][x]]
            for x in range(len(ANFISObj.rules[0]))] for
            row in
                    range(len(ANFISObj.rules))]
        layerTwo = np.array([np.product(x) for x in
            miAlloc]).T
```

```
            if pattern == 0:
                w = layerTwo
            else:
                w = np.vstack((w, layerTwo))

            # layer three
            wSum.append(np.sum(layerTwo))
            if pattern == 0:
                wNormalized = layerTwo / wSum[pattern]
            else:
                wNormalized = np.vstack((wNormalized,
                    layerTwo / wSum[pattern]))

            # prep for layer four (bit of a hack)
            layerThree = layerTwo / wSum[pattern]
            rowHolder = np.concatenate([x * np.append(Xs[
                pattern, :], 1) for x in layerThree])
            layerFour = np.append(layerFour, rowHolder)

        w = w.T
        wNormalized = wNormalized.T

        layerFour = np.array(np.array_split(layerFour,
            pattern + 1))

        return layerFour, wSum, w


def backprop(ANFISObj, columnX, columns, theWSum, theW,
    theLayerFive):
        paramGrp = [0] * len(ANFISObj.memFuncs[columnX])
        for MF in range(len(ANFISObj.memFuncs[columnX])):
            parameters = np.empty(len(ANFISObj.memFuncs[
                columnX][MF][1]))
```

```
timesThru = 0
for alpha in sorted(ANFISObj.memFuncs[columnX][MF
    ][1].keys()):
        bucket3 = np.empty(len(ANFISObj.X))
        for rowX in range(len(ANFISObj.X)):
            varToTest = ANFISObj.X[rowX, columnX]
            tmpRow = np.empty(len(ANFISObj.memFuncs))
            tmpRow.fill(varToTest)
            bucket2 = np.empty(ANFISObj.Y.ndim)
            for colY in range(ANFISObj.Y.ndim):
                rulesWithAlpha = np.array(np.where(
                    ANFISObj.rules[:, columnX] == MF))
                    [0]
                adjCols = np.delete(columns, columnX)

                senSit = mfDerivs.partial_dMF(
                    ANFISObj.X[rowX, columnX],
                    ANFISObj.memFuncs[columnX][MF],
                    alpha)
                # produces d_ruleOutput/
                    d_parameterWithinMF
                dW_dAplha = senSit * np.array(
                    [np.prod([ANFISObj.memClass.
                        evaluateMF(tmpRow)[c][ANFISObj
                        .rules[r][c]] for c in adjCols
                        ]) for r
                     in rulesWithAlpha])

                bucket1 = np.empty(len(ANFISObj.rules
                    [:, 0]))
                for consequent in range(len(ANFISObj.
                    rules[:, 0])):
                    fConsequent = np.dot(np.append(
                        ANFISObj.X[rowX, :], 1.),
```

```
                    ANFISObj.consequents[(
                      (ANFISObj.X.shape[1] + 1) *
                        consequent):(
                      ((ANFISObj.X.shape[1] + 1) *
                        consequent) + (ANFISObj.X.
                        shape[1] + 1)), colY])
                acum = 0
                if consequent in rulesWithAlpha:
                    acum = dW_dAplha[np.where(
                        rulesWithAlpha ==
                        consequent)] * theWSum[
                        rowX]

                acum = acum - theW[consequent,
                    rowX] * np.sum(dW_dAplha)
                acum = acum / theWSum[rowX] ** 2
                bucket1[consequent] = fConsequent
                    * acum

            sum1 = np.sum(bucket1)

            if ANFISObj.Y.ndim == 1:
                bucket2[colY] = sum1 * (ANFISObj.
                    Y[rowX] - theLayerFive[rowX,
                    colY]) * (-2)
            else:
                bucket2[colY] = sum1 * (ANFISObj.
                    Y[rowX, colY] - theLayerFive[
                    rowX, colY]) * (-2)

        sum2 = np.sum(bucket2)
        bucket3[rowX] = sum2

    sum3 = np.sum(bucket3)
```

```
                parameters[timesThru] = sum3
                timesThru = timesThru + 1

            paramGrp[MF] = parameters

        return paramGrp


def predict(ANFISObj, varsToTest):
    [layerFour, wSum, w] = forwardHalfPass(ANFISObj,
        varsToTest)

    # layer five
    layerFive = np.dot(layerFour, ANFISObj.consequents)

    return layerFive


# ------------------------------------------

def show_vector(vector):
    for i in range(len(vector)):
        if i % 8 == 0:  # 8 columns
            print("\n", end="")
        if vector[i] >= 0.0:
            print(' ', end="")
        print("%.4f" % vector[i], end="")  # 4 decimals
        print(" ", end="")
    print("\n")


def set_params(ANFISobj, params):
    item = 0
    for i in range(len(ANFISobj.memFuncs)):
```

```
            for j in range(len(ANFISobj.memFuncs[i])):
                for key in ANFISobj.memFuncs[i][j][1]:
                    ANFISobj.memFuncs[i][j][1][key] = params[
                        item]
                    item += 1
    return ANFISobj.memFuncs


def MSE(ANFISobj, position, param_type):
    if param_type == 'antecedent':
        anf = ANFIS(ANFISobj.X, ANFISobj.Y,
            membershipfunction.MemFuncs(set_params(
            ANFISobj, position)))
        anf.consequents = copy.copy(ANFISobj.consequents)
        err = np.sum((anf.Y - predict(anf, anf.X)) ** 2)
    elif param_type == 'consequent':
        ANFISobj.consequents = copy.copy(position)
        err = np.sum((ANFISobj.Y - predict(ANFISobj,
            ANFISobj.X)) ** 2)
    return err


# ----------------------------------------------------------

class Particle:
    def __init__(self, dim, minx, maxx, seed):
        self.rnd = random.Random(seed)
        self.position = [0.0 for i in range(dim)]
        self.velocity = [0.0 for i in range(dim)]
        self.best_part_pos = [0.0 for i in range(dim)]

        for i in range(dim):
            self.position[i] = ((maxx - minx) *
                                self.rnd.random() + minx)
```

```
            self.velocity[i] = ((maxx - minx) *
                                self.rnd.random() + minx)

        self.error = 100.  # curr error
        self.best_part_pos = copy.copy(self.position)
        self.best_part_err = self.error  # best error


def PSO(ANFISobj, max_epochs, swarm_size, dim, minx, maxx
    , params_type, inertia=0.75, c1=2, c2=2):
    rnd = random.Random(0)

    # create random particles
    swarm = [Particle(dim, minx, maxx, i) for i in range(
        swarm_size)]

    for i in range(swarm_size):
        swarm[i].error = MSE(ANFISobj, swarm[i].position,
            params_type)
        swarm[i].best_part_err = swarm[i].error

    best_swarm_pos = [0.0 for i in range(dim)]  # not
        necess.
    best_swarm_err = sys.float_info.max  # swarm best
    for i in range(swarm_size):  # check each particle
        if swarm[i].error < best_swarm_err:
            best_swarm_err = swarm[i].error
            best_swarm_pos = copy.copy(swarm[i].position)

    epoch = 0
    w = inertia  # inertia
    c1 = c1  # cognitive (particle)
    c2 = c2  # social (swarm)
```

```
while epoch < max_epochs:
    if epoch % 10 == 0 and epoch > 1:
        print("Epoch = " + str(epoch) +
              " best error = %.3f" % best_swarm_err)
    for i in range(swarm_size):  # process each
        particle
        # compute new velocity of curr particle
        for k in range(dim):
            r1 = rnd.random()  # randomizations
            r2 = rnd.random()

            swarm[i].velocity[k] = ((w * swarm[i].
                velocity[k]) +(c1 * r1 * (swarm[i].
                best_part_pos[k] -swarm[i].position[k
                ])) +(c2 * r2 * (best_swarm_pos[k] -
                swarm[i].position[k])))

            if swarm[i].velocity[k] < minx:
                swarm[i].velocity[k] = minx
            elif swarm[i].velocity[k] > maxx:
                swarm[i].velocity[k] = maxx

        # compute new position using new velocity
        for k in range(dim):
            swarm[i].position[k] += swarm[i].velocity
                [k]

        # compute error of new position
        # is new position a new best for the particle
            ?
        if swarm[i].error < swarm[i].best_part_err:
            swarm[i].best_part_err = swarm[i].error
            swarm[i].best_part_pos = copy.copy(swarm[
                i].position)
```

```
            # is new position a new best overall?
            if swarm[i].error < best_swarm_err:
                best_swarm_err = swarm[i].error
                best_swarm_pos = copy.copy(swarm[i].
                    position)

    epoch += 1
return best_swarm_pos
```