Nevronska mreža

Tomaž Tomažič (63100281)

17. maj 2015

1 Uvod

Cilj naloge je bil cim bolj pravilno napovedati eno izmed devetih skupin, kateri pripada nek izdelek.

2 Podatki

Podatki so imeli 93 značilk katere so bile anonime. Vsi podatki so zasegali diskretne vrednosti in so bili nenegativni. Na voljo je bilo 50000 učnih primerov, v katerih sta močno prevladala drugi in šesti razred.

3 Metode

Najprej sem napisal kodo za nevronsko mrežo po članku Šparse autoencoder;̧ ki je delovala samo za 2 skriti plasti. Ko sem se prepričal, da ta koda deluje in da z numeričnim preverjanjem odvod skoraj enake rezultate kot analitični, sem napisal verzijo, ki deluje za poljubno stevilo skritih plasti.

Na zadnjem nivoju sem zatem zamenjal aktivacijsko funkcijo za softmax funkcijo, saj želim imeti na koncu verjetnosti za določen razred, ki se seštejejo v 1. Poleg nove aktivacije na zadnjem nivoju sem moral zamenjati cenilno funkcijo za cenilko softmax regresije in posodobitev napake na zadnjem nivoju v odvodu.

Implementiral sem tudi dropout tehniko, da sem zmanjšal overfit nevronske mreze. Nevrone izlocim iz mreze po s predlagano verjetnostjo po bernoullijevi distribuciji. Na vhodnem nivoju odstrnim nevrone z vrjetnostjo 0.2 na ostalih nivojih pa to verjetnost postopoma povecujem. Nevroni se nakljucnjo izberejo ob vsakem klicu backprop funkcije. Za racunanju cenilne funkcije se pa uporabijo vsi nevroni.

Za vsako tehniko izboljsav sem preveril pravilnost gradienta z numeričnim odvajanjem (funkcija test_grad()) in preveril model s precnim preverjanjem. To sem naredil taok da sem podatke razdelil na 2 dela. En del je ostal vedno neviden za koncno evalvacijo modela. Za izbiro najboljse lambde sem uporabil funkcijo iz paketa sklearn - cross_val_score() v kateri se je testiralo podatke v treh delih/foldih.

Ker se je pri softmax domaci nalogi podatke splačalo normalizirati, sem to tudi tu poiskusil, vendar sem vedno dobil veliko slabše rezultate in sem zato to tehniko opustil.

4 Rezultati

Ker je v nevronskih mrežah ogromno parametrov ketere je potrebno izbrati, je optimalna izbira praktično časovno nemogoča. Zato sem se odločil da vse modele za primerjavo testiram pri enaki arhitekturi (93, 20, 20, 9) z dvema skritima nivojema in poiščem čim boljši regularizacijski parameter lambda_. Rezultati modelov po prečnem preverjanju so podani v tabeli 1. Zraven so, za enake modele, rezultati iz spletnega strežnika.

Tabela 1: Rezultati uporabljenih tehnik v neronski mreži.

tehnika	prečno preverjanje	oddaja	lambda
sparse autoencoder	0.615234227501	0.61665	0.000001
softmax	0.574657559294	0.58094	0.00027
dropout	0.560926567503	0.56929	0.0005

Na strežniku sem poizkušal nekaj oddaj tudi z drugačnimi arhitekturami, vendar rezultati niso bili boljši.

5 Izjava o izdelavi domače naloge

Domačo nalogo in pripadajoče programe sem izdelal sam.

Priloge

A Programska koda

programska koda je v 4 datotekah zato bom v komentar napisal na zacetek ime datoteke

```
#nn.py
from NeuralNet import NeuralNet
import IO
import CV
import Orange
import numpy as np

# layers = (4, 10, 10, 3)
# iris = Orange.data.Table("iris")
# CV.simple_prediction_test(iris.X, iris.Y, layers)
# CV.eval_model(iris.X, iris.Y, layers)

layers = (93, 20, 20, 9)
# X, Y = IO.readFile()
evalX = IO.readTestFile()
```

```
X, Y = IO.readFile('train.csv')

# CV.eval_model(X, Y, layers)
CV.predict(X, Y, evalX, layers, "result-softmax-20-20")


#CV.py
from NeuralNet import NeuralNet
import IO
import numpy as np
import Orange
from sklearn import preprocessing, metrics, grid_search
from sklearn.cross_validation import cross_val_score, ShuffleSplit,
    train_test_split

def find_lambda(X, Y, nnl):
    '''get best lambda'''

    lambdas = list(map(float, np.linspace(0.0000001, .001, 5))) # +\
              # list(map(float, np.linspace(.1, 2, 5))) +\
              # list(map(float, np.linspace(2, 100, 10)))


    cv_split = ShuffleSplit(Y.shape[0], n_iter=3, test_size=0.3, random_state
    =42)
    cv_score = lambda l: -np.mean(cross_val_score(nnl, X, Y, \
        cv=cv_split, fit_params={'lambda_': l}, scoring = 'log_loss', n_jobs=3)
    )

    best_lambda = min([(cv_score(l), l) for l in lambdas])
    print("best labmda", best_lambda[1], " got mean score ", best_lambda[0], "
    for 3 folds")
    return best_lambda[1]

def eval_model(X, Y, layers, lambda_=None):
    '''evaluate model with best lambda on unseen data'''

    X_train, X_test, Y_train, Y_test = train_test_split(
        X, Y, test_size=0.2, random_state=42
    )

    nnl = NeuralNet(layers)
    lambda_ =  find_lambda(X_train, Y_train, nnl) if lambda_ is None else
    lambda_
    nnl.fit(X_train, Y_train, lambda_=lambda_)
    y = nnl.predict(X_test)
    result = metrics.log_loss(Y_test, y)
    print("this model got score ", result, " with lambda ", lambda_, " and arch
     ", layers)
    return lambda_

def predict(X_train, Y_train, X, layers, filename="result"):
    nnl = NeuralNet(layers)
    lambda_ = eval_model(X_train, Y_train, layers)
    nnl.fit(X_train, Y_train, lambda_=lambda_)
    prediction = nnl.predict(X)
    IO.savePrediction(prediction, filename)
    print("prediction finished")

def simple_prediction_test(X, Y, layers, lambda_=0.001):
```

```
    nnl = NeuralNet(layers)
    X_train, X_test, Y_train, Y_test = train_test_split(
        X, Y, test_size=0.2, random_state=42
    )
    nnl.fit(X_train, Y_train, lambda_=lambda_)
    prediction = nnl.predict(X_test)
    print(prediction)
    print(metrics.log_loss(Y_test, prediction))

# IO.py
import csv
import numpy as np
from sklearn import preprocessing

def readFile(path="data4_reduced.csv"):
    with open(path, newline='') as csvfile:
        data = csv.reader(csvfile, delimiter=',')
        data = [i[1:-1] + [i[-1][-1]] for i in data]

    data.pop(0)
    data = np.array(data).astype(int)
    dataX = data[:, :-1]
    dataY = data[:, -1]-1
    return dataX, dataY

def readTestFile(path="test.csv"):
    with open(path, newline='') as csvfile:
        data = csv.reader(csvfile, delimiter=',')
        data = [i[1:] for i in data]

    data.pop(0)
    data = np.array(data).astype(int)
    return data

def savePrediction(p, name="result"):
    f = open(name + ".csv", 'w')
    f.write("id,Class_1,Class_2,Class_3,Class_4,Class_5,Class_6,Class_7,Class_8
    ,Class_9\n")
    np.set_printoptions(suppress=True)
    np.set_printoptions(precision=7)
    for i, j in enumerate(p):
        f.write(str(i+1) + "," + ",".join(j.astype(str)) + "\n")
    f.close()



# NeuralNet.py

import numpy as np
import Orange
from scipy import optimize, stats

# epsilon = 0.00000001
np.random.seed(42)

class NeuralNet:
    def __init__(self, arch):
        self.arch = arch
        self.theta_shape = np.array([(arch[i]+1, arch[i+1])
                                     for i in range(len(arch)-1)])
```

```
        ind = np.array([s1*s2 for s1, s2 in self.theta_shape])
        self.theta_ind = np.cumsum(ind[:-1])
        self.theta_len = sum(ind)

    def init_thetas(self, epsilon=1):
        return np.random.rand(self.theta_len) * 2 * epsilon - epsilon

    def shape_thetas(self, thetas):
        t = np.split(thetas, self.theta_ind)
        return [t[i].reshape(shape) for i, shape in enumerate(self.theta_shape)
    ]

    def h(self, a, thetas):
        """feed forward, prediction"""
        thetas = self.shape_thetas(thetas)
        for theta in thetas[:-1]:
            a = self.g(self.add_ones(a).dot(theta))

        # without softmax
        # return self.g(self.add_ones(a).dot(thetas[-1]))

        # softmax last layer
        s = self.softmax(self.add_ones(a), thetas[-1])
        return s

    def J(self, thetas):
        Y = self.Y
        theta = self.shape_thetas(thetas)
        h0 = np.maximum(np.minimum( self.h(self.X, thetas), 1 - 1e-15), 1e-15)

        # logistic output layer
        # J = .5 * np.sum(np.power(h0 - Y, 2)) /self.m
        # reg = np.sum([np.sum(t[1:].dot(t[1:].T)) for t in theta])
        # return J + self.lambda_ /2.0 * reg

        #cost for softmax
        J = -np.sum(Y*np.log(h0))
        # remove bias in theta for regularization
        reg = np.sum([np.sum(np.power(t[1:], 2)) for t in theta])
        return (J + self.lambda_ /2.0 * reg) / self.m

    def grad_approx(self, thetas, e=1e-2):
        return np.array([(self.J(thetas+eps) - self.J(thetas-eps))/(2*e)
                         for eps in np.identity(len(thetas)) * e])

    def softmax(self, activation, theta):
        z_last = activation.dot(theta)
        s = np.exp(z_last - np.max(z_last, axis=1)[:,None])
        s /= np.sum(s, axis=1)[:, None]
        return s

    def backprop(self, thetas):
        X = self.X
        Y = self.Y
        theta = self.shape_thetas(thetas)
        dropout = self.dropout_init()

        ######################################################
        # feed forward with history
        ######################################################
```

```
        # act = [self.add_ones(X)]                  #activation on first layer (a1)
        # for t in theta[:-1]:                      #activation on middle layers
            # act.append( self.add_ones(self.g(act[-1].dot(t))) )

        #with dropout
        act = [self.add_ones(X) * dropout[0]] #activation on first layer (a1)
        # act = [self.add_ones(X)] #activation on first layer (a1)
        for i, t in enumerate(theta[:-1]):          #activation on middle layers
            act.append( self.add_ones(self.g(act[-1].dot(t))) * dropout[i+1] )
            # act.append( self.add_ones(self.g(act[-1].dot(t))) )

        # logistic output  if u have no softmax
        # act.append( self.g(act[-1].dot(theta[-1])) )  #activation on last layer

        #softmax for last layer
        s = self.softmax(act[-1], theta[-1])
        act.append(s)

        #reverse for backprop
        act.reverse(), theta.reverse(), dropout.reverse()

        ######################################################
        # backpropagation
        ######################################################
        d = [(act[0]-Y)]                                     #error for softmax
        # d = [-(Y-act[0]) * act[0] * (1-act[0])]            #error on normal last layer
        for a, t in zip(act[1:-1], theta[:-1]):              #errors on middle layers
            d.append((t.dot(d[-1].T).T * (a * (1-a)))[:, 1:])

        #without regularization
        # new theta for every layer reversed and flatten back to vector
        # D = [ a.T.dot(e).ravel() for a, e in zip(reversed(act[1:]), reversed(d))]
        # return np.hstack(D) / self.m

        D = [(a * drop).T.dot(e) / self.m for a, e, drop in zip(act[1:], d, dropout)]
        # D = [a.T.dot(e) / self.m for a, e in zip(act[1:], d)]

        # with regularization
        _l = self.lambda_
        for i in range(len(theta)): theta[i][0] = 0 # remove bias
        D = [ (d + _l * t).ravel() for d, t in zip(reversed(D), reversed(theta))]
        return np.hstack(D)

        ######################################################
        # step by step code for 2 hidden layers only.
        # usefull to debug upper code
        # here is without regularization, softmax and dropout
        ######################################################
        # a1 = self.add_ones(X)

        # z2 = a1.dot(theta[0])
        # a2 = self.add_ones(self.g(z2))

```

```
        # z3 = a2.dot(theta[1])
        # a3 = self.add_ones(self.g(z3))

        # z4 = a3.dot(theta[2])
        # a4 = self.g(z4)

        # d4 = (a4-Y)
        # d3 = (theta[2].dot(d4.T).T * (a3 * (1-a3)))[:, 1:]
        # d2 = (theta[1].dot(d3.T).T * (a2 * (1-a2)))[:, 1:]

        # T3 = a3.T.dot(d4) / self.m
        # T2 = a2.T.dot(d3) / self.m
        # T1 = a1.T.dot(d2) / self.m

        # return np.hstack((T1.flat, T2.flat, T3.flat))

    def fit(self, X, y, **kwargs):
        # init variables
        self.lambda_ = kwargs['lambda_'] if 'lambda_' in kwargs else 0
        self.X, self.y = X, y
        self.Y = np.eye(self.arch[-1])[self.y.astype(int)]
        self.m = self.X.shape[0]
        thetas = self.init_thetas()

        # self.test_grad(thetas)

        #get new thetas
        thetas, fmin, d = optimize.fmin_l_bfgs_b(func=self.J, x0=thetas, fprime
=self.backprop)
        self.thetas = thetas
        if d['warnflag'] != 0: print("ERROR!!, LBFGS is not working")
        return self

    def test_grad(self, thetas):
        # print(self.init_thetas)
        approx = np.absolute(self.grad_approx(thetas))
        real = np.absolute(self.backprop(thetas))
        print( np.absolute(sum((approx - real))))
        print( np.sum((approx - real)**2))
        # print( (np.absolute(approx - real) < 1e-4).all())

    def dropout_init(self):
        dropout = []
        m = self.m
        passtrough = .8
        for i, a in enumerate(self.arch[:-1]):
            dropout.append(stats.bernoulli.rvs(passtrough, size=(self.m, a+1)))
            passtrough -= 0.05
            dropout[-1][:,0] = 1
            #disable dropout
            # dropout[-1] = 1
        return dropout

    def g(self, z):
        return 1/(1+np.exp(-z))

    def add_ones(self, X):
        return np.column_stack((np.ones(len(X)), X))

    def get_params(self, deep = False):
```

```
        '''used for CV'''
        return {'arch': self.arch}

    def predict_proba(self, X):
        '''used for CV'''
        return self.h(X, self.thetas)

    def predict(self, X):
        '''used for CV'''
        return self.predict_proba(X)
```