Київський національний університет
імені Т.Шевченка

Звіт

до лабораторної роботи 5
з предмету Нейронні мережі та нейрообчислення
«Задача Комівояжера»

Студента четвертого курсу
Групи ТК-41
Факультету комп'ютерних наук
та кібернетики
Некряча Владислава

Київ
2023

Постановка задачі

Задача комівояжера для n міст. Відомі відстані d_{XY} між кожною парою міст X, Y; комівояжер, виходячи з одного міста, повинен відвідати $n-1$ інших міст, заходячи по одному разу в кожен, і повернутися в початковий. Потрібно визначити порядок обходу міст, при якому загальна пройдена відстань буде мінімальна.

Розв’язати задачу комівояжера мережею Хопфілда та за допомогою машини Больцмана. Порівняти отримані результати.

Алгоритм навчання мережі Хопфілда для розв’язку TSP

В нашому формулюванні, розв’язком вважається матриця переходів. X_{ij} = чи переходить комівояжер з міста i в місто j. Для шляху [1,0,2,1] матриця буде виглядати наступним чином:

V =
[0,0,1]
[1,0,0]
[0,1,0]

Енергія системи (перші три вкладені суми):

$$E = A/2 \sum_X \sum_i \sum_{j \neq i} V_{Xi} V_{Xj} + B/2 \sum_i \sum_X \sum_{X \neq Y} V_{Xi} V_{Yi} + C/2 \left(\sum_X \sum_i V_{Xi} - n \right)^2,$$

Ці суми відповідають за обмеження коректності на розв’язок задачі комівояжера (кожна рядок/кожен стовпець має зберігати рівно одну одиницю).

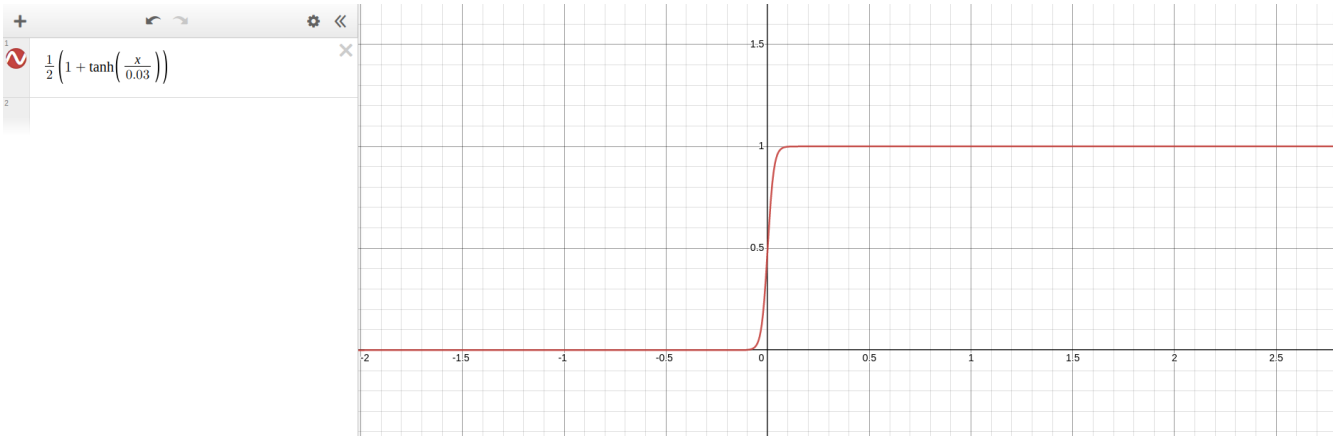
$$1/2 D \sum_X \sum_{Y \neq X} \sum_i d_{XY} V_{Xi} (V_{Y,i+1} + V_{Y,i-1}). \qquad (9)$$

Дане обмеження є 4-им доданком в енергії системи. Дане обмеження провокує покращення розв’язку задачі комівояжера.
Взагалом, енергію системи треба мінімізувати.

Матриця вагів формується наступним чином (базуючись на функції енергії):

$$T_{Xi,Yj} = -A\delta_{XY}(1-\delta_{ij}) \text{ “inhibitory connections within each row”}$$
$$-B\delta_{ij}(1-\delta_{XY}) \text{ “inhibitory connections within each column”}$$
$$-C \text{ “global inhibition”}$$
$$-D\,d_{XY}(\delta_{j,i+1}+\delta_{j,i-1}) \text{ “data term”}$$
$$[\delta_{ij}=1 \quad \text{if} \quad i=j \text{ and is 0 otherwise}]. \qquad (10)$$

Під час оновлення матриці шляху використовується наступна функція:



Алгоритм розв’язку задачі:

1. Ініціалізувати матрицю шляху V (описану вище) з деяким шумом, близьким до нуля, щоб додати схильність до конкретного рішення (незашумлений початковий стан може не зійтись до рішення).
2. Згенерувати матрицю вагів між нейронами (10)
3. В циклі від 1 до 1000:
    4. Оновити матрицю шляху.
    5. Вирахувати енергію системи.
    6. Якщо енергія не міняється, вийти з циклу.

Реалізація

Використовуємо наступні координати міст і по них вираховуємо матрицю дистанцій:

```
city[0] = (0.25, 0.16)
city[1] = (0.85, 0.35)
city[2] = (0.65, 0.24)
city[3] = (0.70, 0.50)
city[4] = (0.15, 0.22)
city[5] = (0.25, 0.78)
city[6] = (0.40, 0.45)
city[7] = (0.90, 0.65)
city[8] = (0.55, 0.90)
city[9] = (0.60, 0.25)

def calc_d(cities):
   n = cities.shape[0]
   d = np.zeros([n, n])
   for i in range(n):
      for j in range(n):
         d[i][j] = np.sqrt(np.square(cities[i][0] - cities[j][0]) + np.square(cities[i][1] -
cities[j][1]))
   return d
```

Min Path Dist: 2.674805
Max Path Dist: 6.277897

Власне алгоритм:

```
def predict(self, A, B, C, D, max_iterations):
   self.train(A, B, C, D)

   u = np.zeros([self.neurons, 1])
   for i in range(self.cities):
      for j in range(self.cities):
         u[i][0] = uniform(0, 0.03)
```

```
    prev_error = self.calc_error(u, A, B, C, D)
    repeated = 0
    max_repeat = 15
    for iteration in range(max_iterations):
        u = self.update(u, C)
        error = self.calc_error(u, A, B, C, D)
        if error == prev_error:
            repeated += 1
        else:
            repeated = 0

        if repeated > max_repeat:
            break
        prev_error = error
    ret = np.zeros([self.cities, self.cities])
    for i in range(self.cities):
        for j in range(self.cities):
            ret[i][j] = u[self.x_to_index[(i, j)]][0]

    return ret
```

Вирахування енергії системи (див. Теорію):

```
def calc_error(self, u, A, B, C, D):
    tmpA = 0
    n = self.cities
    for i in range(n):
        for a in range(n):
            for b in range(n):
                if a != b:
                    tmpA += u[self.x_to_index[(i, a)]][0] * u[self.x_to_index[(i, b)]][0]
    tmpA *= (A/2.0)

    tmpB = 0
    for i in range(n):
        for j in range(n):
```

```
        for a in range(n):
            if i != j:
                tmpB += u[self.x_to_index[(i, a)]][0] * u[self.x_to_index[(j, a)]][0]
    tmpB *= (B/2.0)

    tmpC = 0
    for i in range(n):
        for a in range(n):
            tmpC += u[self.x_to_index[(i, a)]][0]
    tmpC = (tmpC - n)**2
    tmpC *= (C/2.0)

    tmpD = 0
    for i in range(n):
        for j in range(n):
            for a in range(n):
                if 0 < a < n-1:
                    tmpD += self.d[i][j]*u[self.x_to_index[(i, a)]][0]*(u[self.x_to_index[(j,
a+1)]][0] + u[self.x_to_index[(j, a-1)]][0])
                elif a > 0:
                    tmpD += self.d[i][j]*u[self.x_to_index[(i, a)]][0]*(u[self.x_to_index[(j,
a-1)]][0] + u[self.x_to_index[(j, 0)]][0])
                elif a < n-1:
                    tmpD += self.d[i][j]*u[self.x_to_index[(i, a)]][0]*(u[self.x_to_index[(j,
a+1)]][0] + u[self.x_to_index[(j, n-1)]][0])
    tmpD *= (D/2.0)

    return tmpA + tmpB + tmpC + tmpD
```

Оновлення матриці V:

```
def update(self, u, C):
    n = self.cities
    for iteration in range(5*n**2):
        i = randint(0, n-1)
        x = randint(0, n-1)
        u[self.x_to_index[(i, x)]][0] = self.f(np.dot(u.transpose(), self.w[:,
```

```
self.x_to_index[(i, x)]]) + C*(n+1))
    return u

def f(self, x):
    return 0.5*(1+np.tanh(self.alpha*x))
```

Результати:

Мережа Хопфілда запускалася 3 рази.
Мінімальна дистанція, максимальна дистанція та середня дистанція (оптимальна дистанція = 2.674805, найдовша дистанція = 6.277897).

```
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]]
From City 6 To City 9
From City 9 To City 8
From City 8 To City 4
From City 4 To City 2
From City 2 To City 3
From City 3 To City 10
From City 10 To City 7
From City 7 To City 5
From City 5 To City 1
From City 1 To City 6
Distance: 2.853770130883146

Min: 2.853770130883146
Max: 3.316856530830952
Average: 3.1124260927330134
```

Машина Больцманна:

2) Boltzmann Machine (BM): It is a well-known stochastic neural network, and it can be viewed as a generalized Hopfield network. It is a fully connected network comprising two states. The neurons are activated simultaneously based on the current state of their neighbors and the corresponding edges. The probability of accepting neural transition is defined as follows [28]:

$$\begin{cases} P(x_i(t)=1) = \frac{1}{1+e^{-\frac{u(t)}{T}}}, \\ P(x_i(t)=0) = 1 - P(x_i(t)=1), \end{cases} \quad (4)$$

Алгоритм роботи машини Больцмана:

1. До тих пір, поки температура вище порогу
2. Повторити N^2 разів:
[
3. Випадковим чином вибрати місто та шлях
4. Вирахувати параметри біноміального розподілу
5. Підкинути “монету” і змінити стан нейрону якщо в цьому є потреба
]
6. Зменшити температуру

```
# keep going until we've cooled enough...
while self.temperature > lowest_temp:

    # within each cooling epoch, keep exploring states
    for _ in range(self.numStates**2):

        # select a random city and tour
        city = np.random.random_integers(0, self.numCities-1, 1)[0]
        tour = np.random.random_integers(0, self.tourSteps-1, 1)[0]

        # delta consensus is only returned for reporting purposes
```

```
        stateProbability, deltaConsensus = self._stateProbability(city, tour,
self.temperature)

        # randomly flip the state based on the state probability (simulates a
        # coinflip with a biased coin)
        if np.random.binomial(1, stateProbability) == 0:
            changes += 1    # just used for printing status...

            # flip the state
            self.states[city, tour] = 1 - self.states[city, tour]


            if tour == 0:
                self.states[city, self.tourSteps-1] = self.states[city, tour]
            elif tour == self.tourSteps-1:
                self.states[city, 0] = self.states[city, tour]

            if utils.isPathValid(self.states):
                lastValidState = self.states.copy()

                statesExplored[str(lastValidState)] += 1
                validHits += 1

    # cooling...
    self.temperature *= 0.975
```

Вирахування ймовірності (див. Теорію, deltaConsensus = u(t)) :

```
def _stateProbability(self, city, tour, temperature):
    #### DELTA Consensus
##################################################

    # we are copying the state before modifying it since this is a "what if"
    # branch.... so don't affect the "real" state
    states = self.states.copy()
```

```
    # this is the weights and current state for the given city/tour step
    state = self.states[city, tour]
    weights = self.weights[city, tour]

    # "what if" we flipped the state at the given city/tour...
    states[city, tour] = (1 - state)

    # calculate the activity value with this "what if" scenario...
    weightEffect = np.sum(weights * states)
    biasEffect = weights[city, tour]
    activityValue = weightEffect + biasEffect

    # check the consensus as if we had flipped this spot in the tour...
    deltaConsensus = ((1 - state) - state) * activityValue

    #### Probability of flipping state
##########################################

    # sigmoid activation function
    exponential = np.exp(-1 * deltaConsensus / temperature)
    probability = 1 / (1 + exponential)

    return probability, deltaConsensus
```

Резултати:

Машина Больцмана запускалася 3 рази.

Running...
Iteration: 0

Temp:35096.56 PercentComplete:0.72 % ETA:0:02:29 Flips:18287
BestDist:4.652843104097155 DeltaConsensus:-35.89956709107695 ValidStates:0/0

...

Temp:0.10 PercentComplete:99.11 % ETA:0:00:01 Flips:4
BestDist:3.4113100687909195 DeltaConsensus:3.7977977144774187
ValidStates:709/1295

Boltzmann paths (Distance=3.411310):
[5, 8, 4, 0, 6, 3, 2, 9, 1, 7, 5]

Score (out of 1, higher is better): 0.80

Iteration: 1
Temp:35096.56 PercentComplete:0.72 % ETA:0:02:28 Flips:18189
BestDist:4.652843104097155 DeltaConsensus:-37.07262081064182 ValidStates:0/0

....

Temp:0.10 PercentComplete:99.11 % ETA:0:00:01 Flips:2
BestDist:3.050732387964145 DeltaConsensus:3.8194781794400106
ValidStates:735/1245

Boltzmann paths (Distance=3.050732):
[4, 6, 5, 8, 1, 7, 3, 9, 2, 0, 4]

Score (out of 1, higher is better): 0.90

Iteration: 2
Temp:35096.56 PercentComplete:0.72 % ETA:0:02:27 Flips:18091
BestDist:4.652843104097155 DeltaConsensus:-37.11827137961423 ValidStates:0/0
....

Temp:0.10 PercentComplete:99.09 % ETA:0:00:01 Flips:0
BestDist:4.009352871772176 DeltaConsensus:4.393410831338059
ValidStates:735/1354
Boltzmann paths (Distance=4.009353):
[0, 3, 2, 1, 7, 8, 6, 5, 4, 9, 0]
Score (out of 1, higher is better): 0.63

Min: *3.050732*
Max: *4.009353*
Avg: 3.490465

Виходячи з результатів, бачимо, що мережа Хопфілда краще розв’язує задачу комівояжера.