Міністерство освіти і науки України

Національний технічний університет "ХПІ"

кафедра "Інформатика та інтелектуальна власність"

**ЗВІТ**

до лабораторної роботи № 7

Тема : «**Рекурентні нейронні мережі** (RNN)»

Варіант номер 18

з дисципліни "Основи штучного інтелекту"

Виконав:

студент групи КН-321В

Хома Д.М.

Перевірив:

Паржин Ю. В.

Харків 2023

**1 Формулювання завдання класифікації**

Я обрав завдання на класифікацію коктейлів за їх складом. Тобто на алкогольні та безалкогольні. Таким чином, що в коктелях у яких є водка, сидр, текіла, джин, ром, виски, мартіні – вони алкогольні, навіть якщо в них додають безалкогольні компоненти. Безалкогольними коклелями будемо вважати різні соки, молочний коктель, лимонад, содову, чай. Якщо до цих компонентів додати алкогольний компенент напій вже я рахую алкогольним.

**2 Створені набори даних**

Створені набори даних наведені на рис. 2.1. train\_data – навчальний набір,

test\_data – тестовий набір даних, в якому немає прикладів з навчального. Самі набори – це словники в яких ключ це – рецепт коктелю, а значення True – якщо напій алкогольний, False – якщо ні.

train\_data = {  
 'vodka': True, # Алкогольний  
 'vodka tequila': True, # Алкогольний  
 'gin tequila tonic': True, # Алкогольний  
 'soda': False, # Безалкогольний  
 'non-alcoholic fruit punch': False, # Безалкогольний  
 'tequila': True, # Алкогольний  
 'tequila with sour': True, # Алкогольний  
 'lemonade with lime': False, # Безалкогольний  
 'lemonade with strawberry': False, # Безалкогольний  
 'gin and tonic': True, # Алкогольний  
 'fruit pineapple juice smoothie': False, # Безалкогольний  
 'strawberry pineapple juice smoothie': False, # Безалкогольний  
 'rum and coke': True, # Алкогольний  
 'non-alcoholic mojito with lemon': False, # Безалкогольний  
 'whiskey with sour': True, # Алкогольний  
 'iced tea with lime': False, # Безалкогольний  
 'martini with olives': True, # Алкогольний  
 'non-alcoholic mocktail with lime': False, # Безалкогольний  
 'milkshake with lemon': False, # Безалкогольний  
 'cranberry vodka': True, # Алкогольний  
 'martini vodka whiskey': True, # Алкогольний  
 'vodka whiskey': True, # Алкогольний  
 'iced whiskey with olives': True, # Алкогольний  
 'milkshake with strawberry': False, # Безалкогольний  
 'soda with strawberry': False, # Безалкогольний  
 'vodka rum with olives': True, # Алкогольний  
}  
  
# Тестові дані  
test\_data = {  
 'gin vodka': True, # Алкогольний  
 'strawberry lemonade with lime': False, # Безалкогольний  
 'whiskey': True, # Алкогольний  
 'pineapple juice': False, # Безалкогольний  
 'pineapple punch': False, # Безалкогольний  
 'martini rum': True, # Алкогольний  
 'iced tequila': True, # Алкогольний  
 'non-alcoholic soda with lime': False, # Безалкогольний  
 'vodka martini sour': True, # Алкогольний  
 'cranberry vodka rum': True, # Алкогольний  
 'iced tea with lemon fruit': False, # Безалкогольний  
 'iced whiskey with pineapple juice': True, # Безалкогольний  
 'martini tea with lemon fruit': False, # Безалкогольний  
 'milkshake with lime': False, # Безалкогольний  
 'iced juice strawberry smoothie': False, # Безалкогольний  
}

Рисунок 2.1 – набори даних

**3 Архітектура мережі**

Це реалізація мережі - багато-до-одного рекурентного нейронного шару (RNN), яка використовується для моделювання послідовних даних, таких як мовний текст

Архітектура мережі:

Ініціалізація: Клас RNN має ваги та зміщення, що представлені матрицями: Whh (ваги для попередніх станів), Wxh (ваги для вхідних даних), Why (ваги для прогнозованого виходу), bh (зміщення для прихованого шару), by (зміщення для вихідного шару).

Вхідні дані inputs обробляються поетапно. Кожен крок використовує tanh для обчислення нового стану h, який використовується для наступного кроку. Вихід y обчислюється на основі останнього стану h.

Зворотнє поширення використовується для корекції ваг та зміщень. Градієнти обмежуються, щоб уникнути вибуху градієнтів. Потім ваги та зміщення оновлюються за допомогою градієнтного спуску.

Гіперпараметри:

hidden\_size: обрав 32 для збільшення швидкості навчання нейромережі.

learn\_rate: Швидкість навчання для оновлення ваг обрав 0.07, так як точність в 100% досягається вже на приблизно 30 епосі.

Активаційну функцію залишив softmax, також спробував використовувати сигмоїдну та ReLU, та LeakyReLU, але найкраще себе показала саме softmax, вона за найменшу кількість епох та з найменшими втратами класифікує коктелі.

**4 Код програми**

Додав обмеження градієнту на рис.4.1.

clip\_value = 0.5  
for d in [d\_Wxh, d\_Whh, d\_Why, d\_bh, d\_by]:  
 np.clip(d, -clip\_value, clip\_value, out=d)

Рисунок 4.1 – Обмеження градієнту

Це може бути корисним для запобігання великих значень градієнтів, які можуть призвести до проблеми вибуху або затухання градієнту під час навчання нейромережі. Зазвичай це використовується для стабілізації навчання і уникнення нестабільності ваг моделі.

Код програми main.py наведено на рис.4.2.

import numpy as np  
import random  
  
from rnn import RNN  
from data import train\_data, test\_data  
  
# Створення словника слів з навчальних даних  
vocab = list(set([w for text in train\_data.keys() for w in text.split(' ')]))  
vocab\_size = len(vocab)  
print('%d унікальних слів знайдено' % vocab\_size)  
  
# Створення словників для відповідності індексу та слову  
word\_to\_idx = {w: i for i, w in enumerate(vocab)}  
idx\_to\_word = {i: w for i, w in enumerate(vocab)}  
  
# Функція для створення вектора входів з тексту  
def createInputs(text):  
 inputs = []  
 for w in text.split(' '):  
 v = np.zeros((vocab\_size, 1))  
 v[word\_to\_idx[w]] = 1  
 inputs.append(v)  
 return inputs  
  
# Функція для визначення функції softmax  
def softmax(xs):  
 return np.exp(xs) / sum(np.exp(xs))  
  
# Ініціалізація RNN з розміром словника та кількістю класів (у даному випадку 2)  
rnn = RNN(vocab\_size, 2)  
  
# Функція для обробки навчальних даних та тестування  
def processData(data, backprop=True):  
 items = list(data.items())  
 random.shuffle(items)  
  
 loss = 0  
 num\_correct = 0  
  
 for x, y in items:  
 inputs = createInputs(x)  
 target = int(y)  
  
 out, \_ = rnn.forward(inputs)  
 probs = softmax(out)  
  
 loss -= np.log(probs[target])  
 num\_correct += int(np.argmax(probs) == target)  
  
 if backprop:  
 # Обчислення градієнтів та навчання мережі  
 d\_L\_d\_y = probs  
 d\_L\_d\_y[target] -= 1  
 rnn.backprop(d\_L\_d\_y)  
  
 return loss / len(data), num\_correct / len(data)  
  
# Навчання та тестування моделі протягом 1000 епох  
for epoch in range(1000):  
 # Навчання на навчальних даних  
 train\_loss, train\_acc = processData(train\_data)  
 print('--- Епоха %d' % (epoch + 1))  
 print('Навчання:\tВтрата %.4f | Точність: %.4f' % (train\_loss, train\_acc))  
  
 # Тестування на валідаційних даних  
 test\_loss, test\_acc = processData(test\_data, backprop=False)  
 print('Тест:\t\tВтрата %.4f | Точність: %.4f' % (test\_loss, test\_acc))

Рисунок 4.2 – Файл main.py

Код програми rnn.py наведено на рис.4.3.

import numpy as np  
from numpy.random import randn  
  
class RNN:  
 def \_\_init\_\_(self, input\_size, output\_size, hidden\_size=32):  
 # Ваги  
 self.Whh = randn(hidden\_size, hidden\_size) / 1000  
 self.Wxh = randn(hidden\_size, input\_size) / 1000  
 self.Why = randn(output\_size, hidden\_size) / 1000  
  
 # Зсуви (байаси)  
 self.bh = np.zeros((hidden\_size, 1))  
 self.by = np.zeros((output\_size, 1))  
  
 def forward(self, inputs):  
 h = np.zeros((self.Whh.shape[0], 1))  
 self.last\_inputs = inputs  
 self.last\_hs = { 0: h }  
  
 # Прохід вперед по часу  
 for i, x in enumerate(inputs):  
 h = np.tanh(self.Wxh @ x + self.Whh @ h + self.bh)  
 self.last\_hs[i + 1] = h  
  
 y = self.Why @ h + self.by  
  
 return y, h  
  
 def backprop(self, d\_y, learn\_rate=0.07):  
 n = len(self.last\_inputs)  
  
 # Обчислення градієнтів dL/dWhy та dL/dby.  
 d\_Why = d\_y @ self.last\_hs[n].T  
 d\_by = d\_y  
  
 # Ініціалізація градієнтів dL/dWhh, dL/dWxh, dL/dbh нулями.  
 d\_Whh, d\_Wxh, d\_bh = np.zeros\_like(self.Whh), np.zeros\_like(self.Wxh), np.zeros\_like(self.bh)  
  
 # Обчислення градієнтів dL/dh для останнього h.  
 # dL/dh = dL/dy \* dy/dh  
 d\_h = self.Why.T @ d\_y  
  
 # Зворотнє поширення (backpropagation) у часі.  
 for t in reversed(range(n)):  
 # Проміжне значення: dL/dh \* (1 - h^2)  
 temp = ((1 - self.last\_hs[t + 1] \*\* 2) \* d\_h)  
  
 # dL/db = dL/dh \* (1 - h^2)  
 d\_bh += temp  
  
 # Обчислення градієнтів для Whh, Wxh  
 d\_Whh += temp @ self.last\_hs[t].T  
 d\_Wxh += temp @ self.last\_inputs[t].T  
  
 # Оновлення dL/dh = dL/dh \* (1 - h^2) \* Whh  
 d\_h = self.Whh.T @ temp  
  
 # Обмеження градієнтів, щоб уникнути їхнього вибуху.  
 clip\_value = 0.5  
 for d in [d\_Wxh, d\_Whh, d\_Why, d\_bh, d\_by]:  
 np.clip(d, -clip\_value, clip\_value, out=d)  
  
 # Оновлення ваг та зсувів за допомогою градієнтного спуску.  
 self.Whh -= learn\_rate \* d\_Whh  
 self.Wxh -= learn\_rate \* d\_Wxh  
 self.Why -= learn\_rate \* d\_Why  
 self.bh -= learn\_rate \* d\_bh  
 self.by -= learn\_rate \* d\_by

Рисунок 4.3 – Файл rnn.py

**5 Результати класифікації**

Результати класифікації наведено на рис. 5.1.

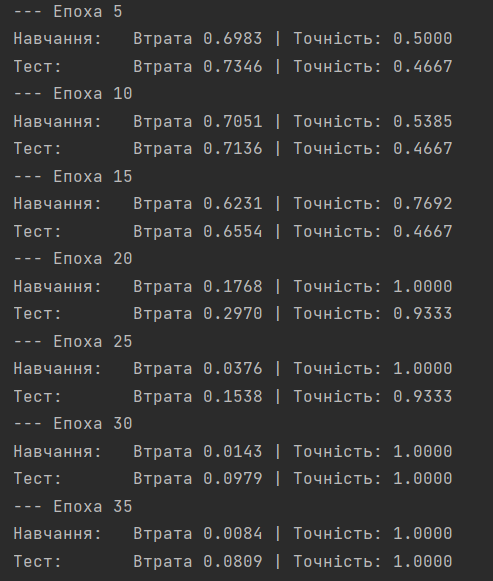


Рисунок 5.1 – Результати класифікації

Мінімальних втрат нейромережа досягає на 800 епосі на рис.5.2.

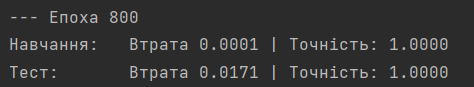


Рисунок 5.2 – Мінімальні втрати 800 епоха

**6. Висновки**

Було розглянуто реалізацію RNN, код представляє простий приклад рекурентної нейронної мережі (RNN), яка навчається класифікувати тексти (коктейлі) на два класи: алкогольні та безалкогольні напої., створено набори даних для класифікації коктелів.

Процес навчання мережі відбувається протягом 1000 епох. На кожній епосі вона проходить через навчальні дані, обчислює втрати та точність на них, а потім виконує тестування на окремих тестових даних, щоб оцінити її загальні можливості.

RNN використовує три основні шари: вхідний шар, прихований шар та вихідний шар. Вхідний шар кодує слова у вектори за допомогою one-hot енкодингу. Потім ці вектори проходять через прихований шар, де використовується гіперболічний тангенс для створення нового стану шару на кожному кроці часу. На останньому шарі отримуємо вихід, який проходить через функцію softmax для отримання ймовірностей належності до класів.

Функція втрати, яка використовується, є кросс-ентропією, яка обчислюється відповідно до прогнозів мережі та фактичних значень. У разі помилкових прогнозів виконується зворотнє поширення помилки, щоб оновити ваги мережі.