ЛАБОРАТОРНА РОБОТА № 6

ДОСЛІДЖЕННЯ РЕКУРЕНТНИХ НЕЙРОННИХ МЕРЕЖ

Мета роботи: *використовуючи спеціалізовані бібліотеки та мову програмування Python навчитися дослідити деякі типи нейронних мереж.*

1. ТЕОРЕТИЧНІ ВІДОМОСТІ

Основні теоретичні відомості подані на лекціях. Також доцільно вивчити матеріал поданий в літературі:

Джоши Пратик. Искусственный интеллект с примерами на Python. : Пер. с англ. - СПб. : ООО "Диалектика", 2019. - 448 с. - Парал. тит. англ. ISBN 978-5-907114-41-8 (рус.)

https://python-scripts.com/recurrent-neural-network

Додатково деякі теоретичні відомості подано у кожному завданні окремо.

Можна використовувати Google Colab або Jupiter Notebook.

2. ЗАВДАННЯ НА ЛАБОРАТОРНУ РОБОТУ ТА МЕТОДИЧНІ РЕКОМЕНДАЦІЇ ДО ЙОГО ВИКОНАННЯ

Завдання 2.1. Ознайомлення з Рекурентними нейронними мережами

Прочитайте та виконайте дії згідно рекомендацій до виконання.

РЕКОМЕНДАЦІЇ ДО ВИКОНАННЯ

Рекурентні нейронні мережі (RNN) - це тип нейронних мереж, що спеціалізуються на обробці послідовностей. Найчастіше їх використовують у таких завданнях, як обробка природної мови (Natural Language Processing) через їхню ефективність в аналізі тексту.

Один із нюансів роботи з рекурентними нейронними мережами полягає в тому, що вони працюють із попередньо заданими параметрами. Вони приймають вхідні дані з фіксованими розмірами та виводять результат, який також є фіксованим. Плюс рекурентних нейронних мереж, або RNN, у тому, що вони забезпечують послідовності з варіативними довжинами як для входу, так і для виходу. На рис. 6.1 подано кілька прикладів того, як може виглядати рекурентна нейронна мережа. Вхідні дані позначені червоним, нейронна мережа RNN – зеленим, а виводи – синім.

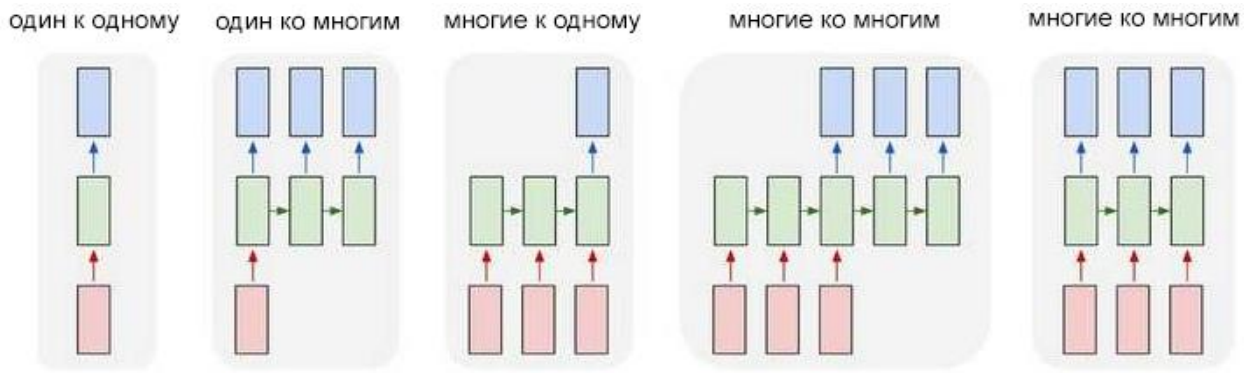


Рис.6.1. Рекурентні нейронні мережі

Здатність обробляти послідовності робить рекурентні нейронні мережі RNN дуже корисними. Області використання:

- Машинний переклад (приклад Google Translate) виконується за допомогою нейронних мереж за принципом «багато до багатьох». Оригінальна послідовність тексту подається в рекурентну нейронну мережу, яка потім створює перекладений текст як результат виведення.
- Аналіз настроїв часто виконується за допомогою рекурентних нейронних мереж з принципом «багато до одного». Цей відгук позитивний чи негативний? Така постановка є одним із прикладів аналізу настроїв. Аналізований текст подається в нейронну мережу, яка потім створює єдину класифікацію виведення. Наприклад - Цей відгук позитивний.

Припустимо, що у нас є нейронна мережа, яка працює за принципом «багато до багатьох». Вхідні дані – x_0, x_1, … x_n, а результати виведення – y_0, y_1, … y_n. Дані x_i та y_i є векторами і можуть бути довільних розмірів.

Рекурентні нейронні мережі RNN працюють шляхом ітерованого оновлення прихованого стану h, яке є вектором, що може мати довільний розмір. Варто враховувати, що на будь-якому заданому етапі t (рис. 6.2):

- Наступний прихований стан h_t підраховується за допомогою попереднього h_{t-1} та наступним вводом x_t;
- Наступний вивід y_t підраховується за допомогою h_t.

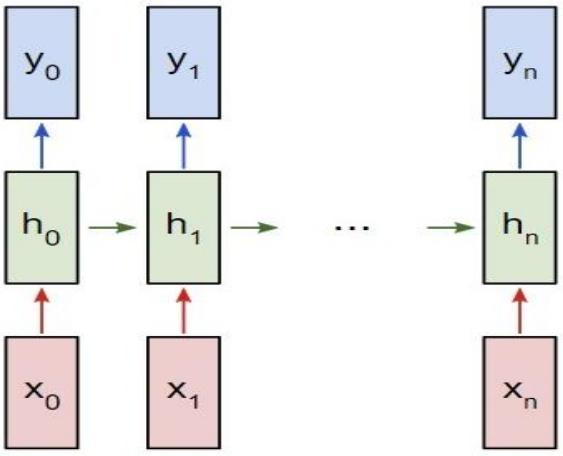


Рис. 6.2. Рекурентна нейронна мережа RNN багато до багатьох

Ось що робить нейронну мережу рекурентною: на кожному кроці вона використовує ту саму вагу. Говорячи точніше, типова класична рекурентна нейронна мережа використовує лише три набори параметрів ваги для виконання необхідних підрахунків:

- W_{xh} використовується для всіх зв'язок $x_t \rightarrow h_t$
- W_{hh} використовується для всіх зв'язок $h_{t-1} \rightarrow h_t$
- W_{hy} використовується для всіх зв'язок $h_t \rightarrow y_t$

Для рекурентної нейронної мережі ми також використовуємо два зміщення:

- b_h додається при підрахунку h_t
- b_y додається при підрахунку y_t.

Вага буде представлена як матриця, а зміщення як вектор. В даному випадку рекурентна нейронна мережа складається з трьох параметрів ваги та двох зсувів.

Наступні рівняння є компактним уявленням всього вищесказаного:

$$h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$
$$y_t = W_{hy}h_t + b_y$$

Створення рекурентної нейронної мережі

Розбір рівнянь краще не пропускати. Зупиніться на хвилинку та вивчіть їх уважно. Пам'ятайте, що вага це матриця, а інші змінні є векторами.

Говорячи про вагу, ми використовуємо матричне множення, після чого вектори вносяться до кінцевого результату. Потім застосовується гіперболічна функція як функція активації першого рівняння. Варто мати на увазі, що інші методи активації, наприклад сигмоїду, також можна використовувати.

Формулювання задачі для рекурентної нейронної мережі

На даний момент ми описали рекурентну нейронну мережу RNN. Вона має виконати простий аналіз настрою. У подальшому прикладі ми попросимо мережу визначити, чи буде заданий рядок нести позитивний чи негативний характер.

Використайте набір даних, що міститься у файлі data.py

Він являє собою набір типу питання-відповідь англійською мовою, як показано в табл.6.1.

Таблиця 6.1

Текст	**Позитивний?**
Я гарний	Так
Я поганий	Ні
Це дуже добре	Так
Це непогано	Так
Я поганий, а не хороший	Ні
Я нещасний	Ні
Це було добре	Так
Я почуваюся непогано, мені не сумно.	Так

Складання плану для нейронної мережі

Тут буде використано класифікацію рекурентної мережі «багато до одного». Принцип її використання нагадує роботу схеми «багато до багатьох», що була описана раніше. Однак цього разу буде задіяно лише прихований стан для одного пункту виведення y:

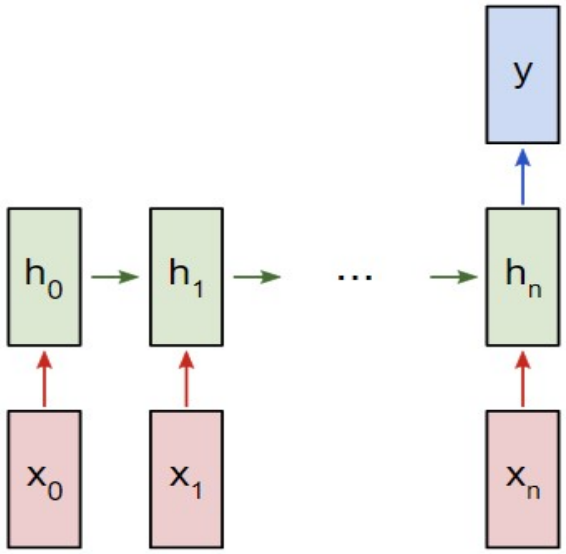


Рис.6.3. RNN багато до одного

Кожен x_i буде вектором, який представляє певне слово з тексту. Вивід y буде вектором, що містить два числа. Одне репрезентує позитивний настрій, а друге — негативний. Ми використовуємо функцію Softmax, щоб перетворити ці значення на ймовірність, і в кінцевому рахунку виберемо між позитивним і негативним.

Softmax це узагальнення логістичної функції для багатовимірного випадку. Функція перетворює вектор z розмірності K у вектор σ тієї ж розмірності, де кожна координата σ_i отриманого вектора представлена дійсним числом в інтервалі [0,1] та сума координат дорівнює 1. Координати σ_i отриманого вектора при цьому трактуються як ймовірність того, що об'єкт належить до класу **i**. Функція Softmax застосовується в машинному навчанні для класифікаційних завдань, коли кількість можливих класів більше двох (для двох класів використовується логістична функція).

Попередня обробка даних рекурентної нейронної мережі RNN

Згаданий набір даних data.py складається з двох словників Python:

```
train_data = {
  'good': True,
  'bad': False,
  # ... більше даних
}
```

```
test_data = {
  'this is happy': True,
  'i am good': True,
  # ... більше даних
}
```

True = Позитивний, False = Негативний

Для отримання даних у зручному форматі потрібно зробити певну попередню обробку. Для початку необхідно створити словник Python з усіх слів, які вживаються в наборі даних:

```
from data import train_data, test_data

# Створити словник
vocab = list(set([w for text in train_data.keys() for w in text.split(' ')]))
vocab_size = len(vocab)

print('%d unique words found' % vocab_size) # знайдено 18 унікальних слів
```

vocab тепер містить перелік всіх слів, які вживаються щонайменше в одному навчальному тексті. Далі надамо кожному слову з vocab індекс типу integer (ціле число).

```
# Призначити індекс кожному слову
word_to_idx = { w: i for i, w in enumerate(vocab) }
idx_to_word = { i: w for i, w in enumerate(vocab) }

print(word_to_idx['good']) # 16 (це може змінитися)
print(idx_to_word[0]) # сумно (це може змінитися)
```

Тепер можна відобразити будь-яке слово за допомогою індексу цілого числа. Це дуже важливий пункт, оскільки:

Рекурентна нейронна мережа не розрізняє слів – лише числа.

Насамкінець нагадаємо, що кожне введення x_i для аналізованої рекурентної нейронної мережі є вектором. Ми використовуватимемо вектори, які представлені у вигляді унітарного коду. Одиниця у кожному векторі перебуватиме у відповідному цілочисельному індексі слова.

Унітарний код (в англомовній літературі unitary code, one-hot) - двійковий код фіксованої довжини, що містить лише одну 1 - прямий унітарний код, або лише один 0 - зворотний (інверсний) унітарний код. Довжина коду визначається кількістю об'єктів, що кодуються, тобто кожному

об'єкту відповідає окремий розряд коду, а значення коду положенням 1 або 0 в кодовому слові.

Оскільки у словнику 18 унікальних слів, кожен x_i буде 18-мірним унітарним вектором.

```
import numpy as np


def createInputs(text):
  '''
Повертає масив унітарних векторів
  які представляють слова у введеному рядку тексту
  - текст є рядком string
  - Унітарний вектор має форму (vocab_size, 1)
  '''

  inputs = []
  for w in text.split(' '):
    v = np.zeros((vocab_size, 1))
    v[word_to_idx[w]] = 1
    inputs.append(v)

  return inputs
```

Ми використовуємо createInputs() пізніше для створення вхідних даних у вигляді векторів та подальшої їх передачі в рекурентну нейронну мережу RNN.

Фаза прямого поширення нейронної мережі

Настав час для створення рекурентної нейронної мережі. Почнемо ініціалізацію з трьома параметрами ваги та двома зсувами.

```
import numpy as np
from numpy.random import randn


class RNN:
  # Класична рекурентна нейронна мережа

  def __init__(self, input_size, output_size, hidden_size=64):
    # Вес
    self.Whh = randn(hidden_size, hidden_size) / 1000
    self.Wxh = randn(hidden_size, input_size) / 1000
```

```
        self.Why = randn(output_size, hidden_size) / 1000

        # Зміщення
        self.bh = np.zeros((hidden_size, 1))
        self.by = np.zeros((output_size, 1))
```

Зверніть увагу: щоб прибрати внутрішню варіативність ваг, ми ділимо на 1000. Це не найкращий спосіб ініціалізації ваг, але він досить простий, підійде для новачків і непогано працює для даного прикладу.

Для ініціалізації ваги стандартного нормального розподілу ми використовуємо np.random.randn().

Тепер ми реалізуємо пряму передачу нейронної мережі, що розглядається. Згадайте перші два рівняння, що розглядаються раніше.

$$h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$
$$y_t = W_{hy}h_t + b_y$$

Ці ж рівняння, реалізовані в коді:

```
class RNN:
  # ...

  def forward(self, inputs):
    '''
    Виконання передачі нейронної мережі за допомогою вхідних даних
    Повернення результатів виведення та прихованого стану
    Вивід – це масив одного унітарного вектора з формою (input_size, 1)
    '''
    h = np.zeros((self.Whh.shape[0], 1))

    # Виконання кожного кроку в нейронній мережі RNN
    for i, x in enumerate(inputs):
      h = np.tanh(self.Wxh @ x + self.Whh @ h + self.bh)

    # Compute the output
    y = self.Why @ h + self.by

    return y, h
```

Зверніть увагу, що ми ініціалізували h для нульового вектора в першому кроці, так як у нас немає попереднього h, який тепер можна використовувати.

Зробіть наступне:

```
# ...

def softmax(xs):
  # Застосування функції Softmax для вхідного масиву
  return np.exp(xs) / sum(np.exp(xs))


# Ініціалізація нашої рекурентної нейронної мережі RNN
rnn = RNN(vocab_size, 2)


inputs = createInputs('i am very good')
out, h = rnn.forward(inputs)
probs = softmax(out)
print(probs) # [[0.50000095], [0.49999905]]
```

Наша рекурентна нейронна мережа працює, проте її важко назвати корисною. Давайте виправимо цей недолік.

Фаза зворотного розповсюдження нейронної мережі

Для тренування рекурентної нейронної мережі буде використано функцію втрати. Тут буде використано втрату перехресної ентропії, яка у більшості випадків сумісна з функцією Softmax. Формула для підрахунку:

$$L = -\ln(p_c)$$

Тут P_C є передбачуваною ймовірністю рекурентної нейронної мережі для класу correct (позитивний або негативний). Наприклад, якщо позитивний текст передбачається рекурентною нейронною мережею як позитивний текст на 90%, то втрата складе:

$$L = -\ln(0.90) = 0.105$$

За наявності параметрів втрати можна натренувати нейронну мережу таким чином, щоб вона використовувала **градієнтний спуск для мінімізації втрат**. Отже, тут знадобляться градієнти.

Зверніть увагу: наступний розділ має на увазі наявність у читача базових знань про багатоваріантне обчислення. Ви можете пропустити кілька абзаців, проте рекомендується все ж таки пробігтися очима. У міру отримання нових даних код доповнюватиметься, і пояснення стануть зрозумілішими.

Параметри нейронної мережі, що розглядається

Параметри даних, які будуть використані надалі:

y - необроблені вхідні дані нейронної мережі;

p - Кінцева ймовірність: p = softmax (y);

c - справжня мітка певного зразка тексту, так званий "правильний" клас;

L - Втрата перехресної ентропії: $L = -\ln(p_c)$;

W_{xh}, W_{hh} і W_{hy} — три матриці ваги в нейронній мережі, що розглядається;

b_h і b_y — два вектори зміщення в аналізованій рекурентній нейронній мережі RNN.

Наступним кроком буде налаштування фази прямого розповсюдження. Це необхідно для кешування окремих даних, які будуть використовуватися у фазі зворотного розповсюдження нейронної мережі. Паралельно з цим можна буде встановити основний скелет фази зворотного поширення. Це буде виглядати так:

```
class RNN:
  # ...

  def forward(self, inputs):
    '''
    Виконання фази прямого поширення нейронної мережі з
    використанням введених даних.
    Повернення підсумкової видачі та прихованого стану.
    - Вхідні дані у масиві однозначного вектора з формою (input_size, 1).
    '''
    h = np.zeros((self.Whh.shape[0], 1))

    self.last_inputs = inputs
    self.last_hs = { 0: h }

    # Виконання кожного кроку нейронної мережі RNN
    for i, x in enumerate(inputs):
      h = np.tanh(self.Wxh @ x + self.Whh @ h + self.bh)
      self.last_hs[i + 1] = h

    # Підрахунок значення виводу
    y = self.Why @ h + self.by

    return y, h

  def backprop(self, d_y, learn_rate=2e-2):
    '''
```

```
    Виконання фази зворотного розповсюдження мережі RNN.
    - d_y (dL/dy) має форму (output_size, 1).
    - learn_rate є дійсним числом float.
    '''
    pass
```

Градієнти

Почнемо з обчислення RNN $\frac{\partial L}{\partial y}$. Що нам відомо:

$$L = -\ln(p_c) = -\ln(\text{softmax}(y_c))$$

Тут використовується фактичне значення $\frac{\partial L}{\partial y}$, а також застосовується диференціювання складної функції. Результат наступний:

$$\frac{\partial L}{\partial y_i} = \begin{cases} p_i & \text{if } i \neq c \\ p_i - 1 & \text{if } i = c \end{cases}$$

Наприклад, якщо p = [0.2, 0.2, 0.6], а коректним класом є c = 0, кінцевим результатом буде значення $\frac{\partial L}{\partial y}$ = [-0.8, 0.2, 0.6]. Даний вислів можна перевести в код

```
# Цикл для кожного прикладу тренування
for x, y in train_data.items():
  inputs = createInputs(x)
  target = int(y)

  # Пряме разповсюдження
  out, _ = rnn.forward(inputs)
  probs = softmax(out)

  # Створення dL/dy
  d_L_d_y = probs
  d_L_d_y[target] -= 1

  # Зворотнє разповсюдження
  rnn.backprop(d_L_d_y)
```

Тепер розберемося з градієнтами для W_{hy} і b_y, які використовуються тільки для переходу кінцевого прихованого стану в результат виведення нейронної мережі, що розглядається, RNN. Використовуємо такі дані:

$$\frac{\partial L}{\partial W_{hy}} = \frac{\partial L}{\partial y} * \frac{\partial y}{\partial W_{hy}}$$

$$y = W_{hy} h_n + b_y$$

Тут h_n є кінцевим прихованим станом. Таким чином:

$$\frac{\partial y}{\partial W_{hy}} = h_n$$

$$\frac{\partial L}{\partial W_{hy}} = \boxed{\frac{\partial L}{\partial y} h_n}$$

Аналогічним способом обчислюємо:

$$\frac{\partial y}{\partial b_y} = 1$$

$$\frac{\partial L}{\partial b_y} = \boxed{\frac{\partial L}{\partial y}}$$

Тепер можна приступити до реалізації backprop().

```
class RNN:
  # ...

  def backprop(self, d_y, learn_rate=2e-2):
    '''
    Виконує фазу зворотного поширення нейронної мережі RNN.
    - d_y (dL/dy) має форму (output_size, 1).
    - learn_rate є дійсним числом float.
    '''
    n = len(self.last_inputs)

    # Подсчет dL/dWhy и dL/dby.
    d_Why = d_y @ self.last_hs[n].T
```

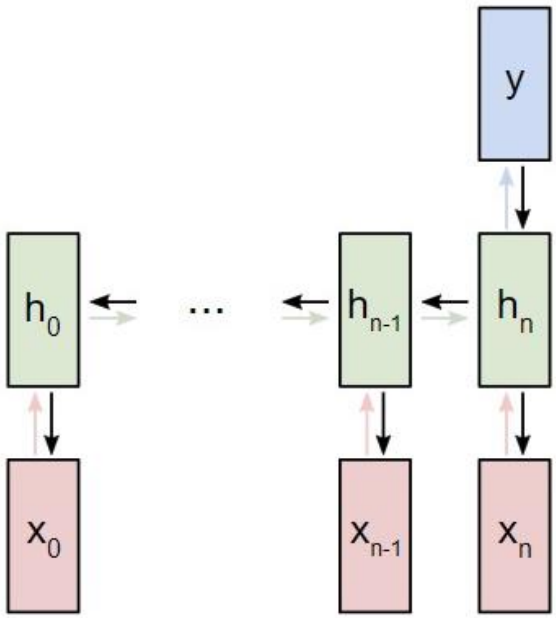
```
d_by = d_y
```

Нагадування: ми створили self.last_hs forward() у попередніх прикладах.

Нарешті, нам знадобляться градієнти для W_{hh}, W_{xh} і b_h, які використовувалися в кожному кроці нейронної мережі. У нас є:

$$\frac{\partial L}{\partial W_{xh}} = \frac{\partial L}{\partial y} \sum_t \frac{\partial y}{\partial h_t} * \frac{\partial h_t}{\partial W_{xh}}$$

Зміна W_{xh} впливає як на кожен h_t, а й у все , що, своєю чергою, призводить до змін у L. Щоб повністю підрахувати градієнт W_{xh}, необхідно провести зворотне поширення через всі часові кроки. Його також називають Зворотним поширенням у часі, або **Backpropagation Through Time (BPTT):**



Зворотне поширення у часі

W_{xh} використовується для всіх прямих посилань $x_t \rightarrow h_t$, тому нам потрібно провести зворотне розповсюдження назад до кожного з цих посилань.

Наблизившись до заданого кроку t, потрібно підрахувати $\frac{\partial h_t}{\partial W_{xh}}$:

$$h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

Похідна гіперболічна функція tanh нам вже відома:

$$\frac{d\tanh(x)}{dx} = 1 - \tanh^2(x)$$

Використовуємо диференціювання складної функції, або ланцюгове правило:

$$\frac{\partial h_t}{\partial W_{xh}} = \boxed{(1 - h_t^2)x_t}$$

Аналогічним способом обчислюємо:

$$\frac{\partial h_t}{\partial W_{hh}} = \boxed{(1 - h_t^2)h_{t-1}}$$

$$\frac{\partial h_t}{\partial b_h} = \boxed{(1 - h_t^2)}$$

Останнє потрібне значення — $\frac{\partial y}{\partial h_t}$. Його можна підрахувати рекурсивно:

$$\frac{\partial y}{\partial h_t} = \frac{\partial y}{\partial h_{t+1}} * \frac{\partial h_{t+1}}{\partial h_t}$$
$$= \frac{\partial y}{\partial h_{t+1}}(1 - h_t^2)W_{hh}$$

Реалізуємо зворотне поширення у часі, або BPTT, відштовхуючись від прихованого стану як початкову точку. Далі працюватимемо у зворотному порядку. Тому на момент підрахунку $\frac{\partial y}{\partial h_t}$ значення $\frac{\partial y}{\partial h_{t+1}}$ буде відома. Винятком стане лише останній прихований стан h_n:

Прихований стан нейронної мережі

Тепер у нас є все необхідне, щоб нарешті реалізувати зворотне поширення в часі **BPTT** і закінчити backprop():

```
class RNN:
  # ...

  def backprop(self, d_y, learn_rate=2e-2):
    '''
```

```
Виконання фази зворотного розповсюдження RNN.
- d_y (dL/dy) має форму (output_size, 1).
- learn_rate є дійсним числом float.
'''
n = len(self.last_inputs)

# ОБчислення dL/dWhy і dL/dby.
d_Why = d_y @ self.last_hs[n].T
d_by = d_y

# Ініціалізація dL/dWhh, dL/dWxh, і dL/dbh до нуля.
d_Whh = np.zeros(self.Whh.shape)
d_Wxh = np.zeros(self.Wxh.shape)
d_bh = np.zeros(self.bh.shape)

# Обчислення dL/dh для останнього h.
d_h = self.Why.T @ d_y

# Зворотне розповсюдження по часу.
for t in reversed(range(n)):
   # Среднее значение: 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

   # dL/dWhh = dL/dh * (1 - h^2) * h_{t-1}
   d_Whh += temp @ self.last_hs[t].T

   # dL/dWxh = dL/dh * (1 - h^2) * x
   d_Wxh += temp @ self.last_inputs[t].T

   # Далее dL/dh = dL/dh * (1 - h^2) * Whh
   d_h = self.Whh @ temp

# Відсікаємо, щоб попередити розрив градієнтів.
for d in [d_Wxh, d_Whh, d_Why, d_bh, d_by]:
   np.clip(d, -1, 1, out=d)

# Обновляємо ваги і зміщення з використанням 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
```

Моменти, на які варто звернути увагу:

- Ми об'єднали $\frac{\partial L}{\partial y} * \frac{\partial y}{\partial h}$ в $\frac{\partial L}{\partial h}$ для зручності.
- Ми постійно оновлюємо змінну d_h, яка тримає останню версію $\frac{\partial y}{\partial h_{t+1}}$, що потрібно для підрахунку $\frac{\partial L}{\partial h_t}$.
- Закінчивши зі зворотним поширенням у часі BPTT, ми використовуємо np.clip() на значеннях градієнта нижче -1 або вище 1. Це допоможе позбавитися проблеми з вибуховими градієнтами. Таке трапляється, коли градієнти стають занадто великими через велику кількість помножених параметрів. Вибух, а також зникнення градієнтів не є рідкістю для класичних рекурентних нейронних мереж. Складніші рекурентні нейронні мережі, наприклад LSTM, краще підійдуть для їх обробки.
- Коли всі градієнти підраховані, ми оновлюємо параметри ваги та зміщення за допомогою градієнтного спуску.

Наша рекурентна нейронна мережа готова.

Тестування рекурентної нейронної мережі

Протестуємо готову рекурентну нейронну мережу.

Для початку, напишемо допоміжну функцію для обробки даних рекурентної нейронної мережі, що розглядається:

```
import random

def processData(data, backprop=True):
    '''
    Повернення втрат RNN і точності для даних
    - дані подані як словник, що відображує текст як True або False.
    - backprop визначає, чи потрібно використовувати звороднє розподілення
    '''
    items = list(data.items())
    random.shuffle(items)

    loss = 0
    num_correct = 0
```

```
    for x, y in items:
        inputs = createInputs(x)
        target = int(y)

        # Пряме розподілення
        out, _ = rnn.forward(inputs)
        probs = softmax(out)

        # Обчислення втрат / точності
        loss -= np.log(probs[target])
        num_correct += int(np.argmax(probs) == target)

        if backprop:
            # Создание dL/dy
            d_L_d_y = probs
            d_L_d_y[target] -= 1

            # Зворотне розподілення
            rnn.backprop(d_L_d_y)

    return loss / len(data), num_correct / len(data)
```

Тепер можна написати цикл для тренування мережі:

```
# Цикл тренування
for epoch in range(1000):
    train_loss, train_acc = processData(train_data)

    if epoch % 100 == 99:
        print('--- Epoch %d' % (epoch + 1))
        print('Train:\tLoss %.3f | Accuracy: %.3f' % (train_loss, train_acc))

        test_loss, test_acc = processData(test_data, backprop=False)
        print('Test:\tLoss %.3f | Accuracy: %.3f' % (test_loss, test_acc))
```

Збережіть код робочої програми під назвою LR_6_task_1.py
Код програми, результати виведення main.py занесіть у звіт.
Зробіть висновок

Завдання 2.2. Дослідження рекурентної нейронної мережі Елмана (Elman Recurrent network (newelm))

Рекурентні нейронні мережі Елмана часто застосовують для детектування амплітуди сигналу (апроксимації, представлення, наближення сигналів). Тобто нейронна мережа за попередніми відліками сигналу повинна передбачити наступні значення і якомога ближче до реальних.

`neurolab.net.`**`newelm`**(*minmax*, *size*, *transf=None*)
Create a Elman recurrent network

Parameters:	minmax: list of list, the outer list is the number of input neurons, inner lists must contain 2 elements: min and max Range of input value size: the length of list equal to the number of layers except input layer, the element of the list is the neuron number for corresponding layer Contains the number of neurons for each layer
Returns:	net: Net

Проведіть дослідження рекурентної нейронної мережі Елмана відповідно до рекомендацій наведених нижче.

РЕКОМЕНДАЦІЇ ДО ВИКОНАННЯ

Створіть новий файл Python та імпортуйте такі пакети.

```
import neurolab as nl
import numpy as np
```

Створіть модель сигналу для навчання мережі

```
# Створення моелей сигналу для навчання
i1 = np.sin(np.arange(0, 20))
i2 = np.sin(np.arange(0, 20)) * 2

t1 = np.ones([1, 20])
t2 = np.ones([1, 20]) * 2


input = np.array([i1, i2, i1, i2]).reshape(20 * 4, 1)
target = np.array([t1, t2, t1, t2]).reshape(20 * 4, 1)
```

Створіть мережу з 2 прошарками

```
# Створення мережі з 2 прошарками
net = nl.net.newelm([[-2, 2]], [10, 1], [nl.trans.TanSig(),
nl.trans.PureLin()])
```

```
# Ініціалізуйте початкові функції ваґів
net.layers[0].initf = nl.init.InitRand([-0.1, 0.1], 'wb')
net.layers[1].initf = nl.init.InitRand([-0.1, 0.1], 'wb')
net.init()
```

Натренуйте та запустіть мережу

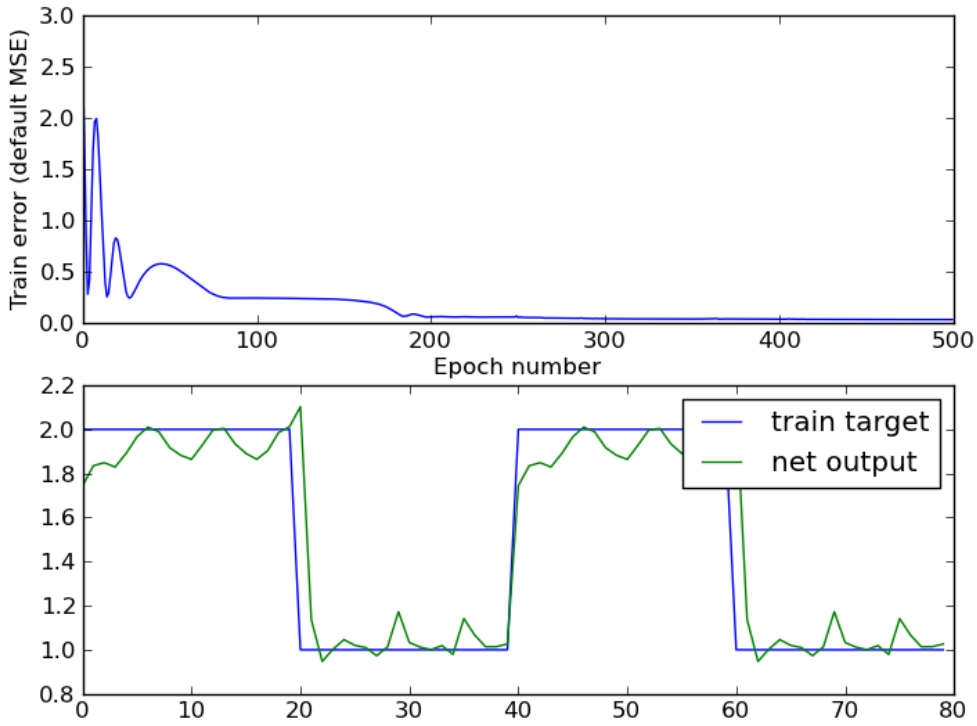
```
# Тренування мережі
error = net.train(input, target, epochs=500, show=100, goal=0.01)
# Запустіть мережу
output = net.sim(input)
```

Побудуйте графіки результатів

```
# Побудова графіків
import pylab as pl
pl.subplot(211)
pl.plot(error)
pl.xlabel('Epoch number')
pl.ylabel('Train error (default MSE)')

pl.subplot(212)
pl.plot(target.reshape(80))
pl.plot(output.reshape(80))
pl.legend(['train target', 'net output'])
pl.show()
```

У результаті отримаєте графіки подібні до поданих на рисунку.



Графік помилки та апроксимації сигналу занесіть у звіт.

У вікні терміналу також відобразиться інформація. ***Скрін з інформацією виріжте та занесіть у звіт!***

Зверніть увагу, що коли ви будете запускати виконання коду програми ще раз, то отримаєте дещо інший результат апроксимації мережею. Це пов’язано з випадковістю формування ваг у процесі навчання.

Збережіть код робочої програми з обов'язковими коментарям під назвою LR_6_task_2.py

Зробіть висновок

Завдання 2.3. Дослідження нейронної мережі Хемінга (Hemming Recurrent network)

Проведіть дослідження нейронної мережі Хемінга (Hemming Recurrent network). Для цього виконайте наступні рекомендації.

РЕКОМЕНДАЦІЇ ДО ВИКОНАННЯ

Створіть новий файл Python та імпортуйте такі пакети.

```
import numpy as np
import neurolab as nl
```

Введіть вхідні дані для навчання мережі

```
target = [[-1, 1, -1, -1, 1, -1, -1, 1, -1],
          [1, 1, 1, 1, -1, 1, 1, -1, 1],
          [1, -1, 1, 1, 1, 1, 1, -1, 1],
          [1, 1, 1, 1, -1, -1, 1, -1, -1],
          [-1, -1, -1, -1, 1, -1, -1, -1, -1]]

input = [[-1, -1, 1, 1, 1, 1, 1, -1, 1],
         [-1, -1, 1, -1, 1, -1, -1, -1, -1],
         [-1, -1, -1, -1, 1, -1, -1, 1, -1]]
```

Створіть та натренуйте нейромережу

```
# Створення та тренування нейромережі
net = nl.net.newhem(target)

output = net.sim(target)
print("Test on train samples (must be [0, 1, 2, 3, 4])")
print(np.argmax(output, axis=0))

output = net.sim([input[0]])
print("Outputs on recurent cycle:")
print(np.array(net.layers[1].outs))

output = net.sim(input)
print("Outputs on test sample:")
print(output)
```

Код програми та результати занесіть у звіт.

Програмний код збережіть під назвою LR_6_task_3.py

Завдання 2.4. Дослідження рекурентної нейронної мережі Хопфілда Hopfield Recurrent network (newhop)

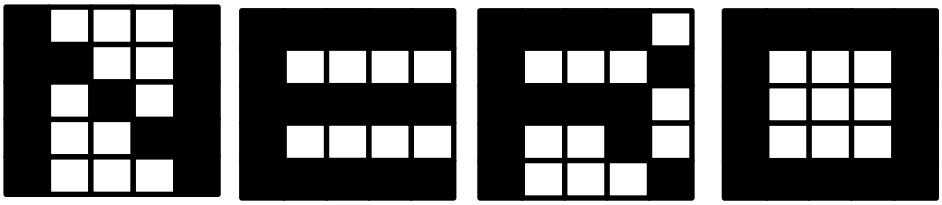
Необхідно навчити рекурентну нейронну мережу Хопфілда розпізнавати букви слова.

Формат звернення до відповідної функції для neurolab подано у таблиці.

`neurolab.net.`**`newhop`**(*target*, *transf=None*, *max_init=10*, *delta=0*)	
Create a Hopfield recurrent network	
Parameters:	target: array like (l x net.co) train target patterns transf: func (default HardLims) Activation function max_init: int (default 10) Maximum of recurrent iterations delta: float (default 0) Minimum difference between 2 outputs for stop recurrent cycle
Returns:	net: Net
Example:	`>>> net = newhem([[-1, -1, -1], [1, -1, 1]])` `>>> output = net.sim([[-1, 1, -1], [1, -1, 1]])`

Нехай необхідно навчити мережу розпізнавати букви у слові NERO.

Будемо вважати, що шрифт подання кожної букви у цифровому вигляді складає матрицю пікселів 5×5. Зображення букви чорно-біле (бінарне), як це показано на рисунку.



Тоді таке зображення кожної букви можна закодувати так: якщо піксель чорний то - 1, якщо білий , то - 0 (як показано на наступному рисунку).

1	0	0	0	1
1	1	0	0	1
1	0	1	0	1
1	0	0	1	1
1	0	0	0	1

1	1	1	1	1
1	0	0	0	0
1	1	1	1	1
1	0	0	0	0
1	1	1	1	1

1	1	1	1	0
1	0	0	0	1
1	1	1	1	0
1	0	0	1	0
1	0	0	0	1

1	1	1	1	1
1	0	0	0	1
1	0	0	0	1
1	0	0	0	1
1	1	1	1	1

Тоді кожен елемент матриці і буде вхідним сигналом для нейронної мережі.

РЕКОМЕНДАЦІЇ ДО ВИКОНАННЯ

Створіть новий файл Python та імпортуйте такі пакети.

```
import numpy as np
import neurolab as nl
```

Подайте вхідні дані у вигляді складного списку та приведіть до форми, що сприймається функцією з бібліотеки.

```
# N E R O
target =  [[1,0,0,0,1,
            1,1,0,0,1,
            1,0,1,0,1,
            1,0,0,1,1,
            1,0,0,0,1],
           [1,1,1,1,1,
            1,0,0,0,0,
            1,1,1,1,1,
            1,0,0,0,0,
            1,1,1,1,1],
           [1,1,1,1,0,
            1,0,0,0,1,
            1,1,1,1,0,
            1,0,0,1,0,
            1,0,0,0,1],
           [0,1,1,1,0,
            1,0,0,0,1,
            1,0,0,0,1,
            1,0,0,0,1,
            0,1,1,1,0]]

chars = ['N', 'E', 'R', 'O']
target = np.asfarray(target)
target[target == 0] = -1
```

Створіть та навчіть нейронну мережу Хопфілда.

```
# Create and train network
net = nl.net.newhop(target)

output = net.sim(target)
print("Test on train samples:")
for i in range(len(target)):
    print(chars[i], (output[i] == target[i]).all())
```

У вікні терміналу появиться інформація про результати навчання.
Інформацію з вікна терміналу занесіть у звіт.

Протестуйте навчену нейронну мережу Хопфілда. Для цього будемо вважати, що при відображенні букви N були помилки (деякі білі пікселі стали чорними і навпаки).

```
print("\nTest on defaced N:")
test =np.asfarray([0,0,0,0,0,
                   1,1,0,0,1,
                   1,1,0,0,1,
                   1,0,1,1,1,
```

```
                    0,0,0,1,1])
test[test==0] = -1
out = net.sim([test])
print ((out[0] == target[0]).all(), 'Sim. steps',len(net.layers[0].outs))
```

У вікні терміналу появиться інформація про результати.

Інформацію з вікна терміналу занесіть у звіт.

Якщо навчання пройшло правильно то мережа при невеликій кількості помилок буде вгадувати букву правильно.

Спробуйте протестувати інші букви з помилками. Код тесту та результати з вікна терміналу занесіть у звіт.

Збережіть код робочої програми з обов'язковими коментарям під назвою LR_6_task_4.py

Код програми та рисунок занесіть у звіт.

Зробіть висновок. Висновок занесіть у звіт.

Завдання 2.5. Дослідження рекурентної нейронної мережі Хопфілда для ваших персональних даних

По аналогії з попереднім завданням візьміть перші букви **Ваших Прізвища, Імя та По батькові (кирилицею)**. Закодуйте їх матрицею пікселів та кодом одиниць і нулів. Навчіть мережу розпізнавати Ваші букви. Протестуйте мережу на можливість розпізнавання кожної букви шляхом внесення помилок в тест. ***Код тесту та результати тестування занесіть у звіт.***

РЕКОМЕНДАЦІЇ ДО ВИКОНАННЯ

Завдання виконується шляхом модифікації коду попереднього завдання.

У коді слід врахувати, що букв буде 3.

При тестуванні не робіть багато помилок у букві (1-2 піксела), бо мала вибірка для навчання.

Код програми та результати занесіть у звіт повністю аналогічно як у попередньому занятті.

Програмний код збережіть під назвою LR_6_task_5.py

Коди комітити на GitHab. У кожному звіті повинно бути посилання на GitHab.

Назвіть бланк звіту СШІ-ЛР-6-NNN-XXXXX.doc
де NNN – позначення групи
XXXXX – позначення прізвища студента.

Переконвертуйте файл звіту в СШІ-ЛР-6-NNN-XXXXX.pdf
Надішліть чи представте звіт викладачу.