МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

**І. В. ФЕДОРІН**

# МЕТОДИ ТА ТЕХНОЛОГІЇ ОБЧИСЛЮВАЛЬНОГО ІНТЕЛЕКТУ: ПРАКТИКУМ

**Навчальний посібник**

*Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського як навчальний посібник для здобувачів другого (магістерського) рівня вищої освіти за освітньою програмою «Комп'ютерні технології в біології та медицині» спеціальності 122 «Комп'ютерні науки»*

Київ
КПІ ім. Ігоря Сікорського
2022

Рецензент *Олійник А.С.,* д.фіз.-мат.н., доцент, професор кафедри алгебри і комп'ютерної математики Київського національного університету імені Тараса Шевченка

*Сахненко Н.К,* д.фіз.-мат.н., доцент, доцент кафедри математичного моделювання та аналізу даних, Навчально-науковий фізико-технічний інститут, Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського»

Відповідальний редактор *Носовець О. К.*, к.т.н., доцент кафедри біомедичної кібернетики, Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського»

*Гриф надано Методичною радою КПІ ім. Ігоря Сікорського*
*(протокол № 2 від 30.09.2022р.)*
*за поданням Вченої ради факультету біомедичної інженері*
*(протокол № 1 від 01.09.2022 р.)*

Електронне мережне навчальне видання

*Федорін Ілля Валерійович*, канд. фіз.-мат. наук

# МЕТОДИ ТА ТЕХНОЛОГІЇ ОБЧИСЛЮВАЛЬНОГО ІНТЕЛЕКТУ: ПРАКТИКУМ

## НАВЧАЛЬНИЙ ПОСІБНИК

Методи та технології обчислювального інтелекту: Практикум [Електронний ресурс] : навч. посіб. для студ. спеціальності 122 «Комп’ютерні науки» / І. В. Федорін; КПІ ім. Ігоря Сікорського. – Електронні текстові дані (1 файл: 4,2 Мбайт). – Київ: КПІ ім. Ігоря Сікорського, 2022. – 317 с.

Навчальний посібник призначено для закріплення на практичних заняттях теоретичних знань, методів та алгоритмів обчислювального інтелекту. Надані загальні відомості з теорії машинного навчання, штучного інтелекту та сучасних архітектур штучних нейронних мереж. У навчальному посібнику врахований сучасний стан розвитку алгоритмі та мереж глибокого навчання.

# ЗМІСТ

# ВСТУП

Основою метою вивчення навчальної дисципліни «Методи та технології обчислювального інтелекту» є формування у студентів розуміння принципів, вмінь та практичних навичок, які дозволять спроектувати медичну систему штучного інтелекту, яка виконує наступні функції:

1) автоматично обробляє великий обсяг персоналізованої інформації з метою подальшого аналізу

2) за результатами автоматичної обробки даних надає персоналізовані рекомендації

3) за результатами автоматичної обробки з певною вірогідністю надає скринінгові результат дослідження

4) використовує нові дані, для автоматичного донавчання системи

5) являє собою зв’язковою ланкою системи користувач-медичний працівник

Згідно з вимогами програми навчальної дисципліни студенти після засвоєння кредитного модуля мають продемонструвати такі результати навчання:

- використовувати методи штучного та обчислювального інтелекту до проектування, розробки та тестування програмних продуктів біомедичного спрямування
- забезпечити пошук, обробку та зберігання біомедичної інформації та баз даних у бажаному форматі, використовуючи мережеві системи
- ідентифікувати об’єкт дослідження, порядок зберігання та обробки даних методами штучного інтелекту для вирішення задач біомедичного спрямування
- моделювати біомедичні системи методами обчислювального інтелекту та машинного навчання

− визначати рівень достовірності та ефективність роботи розробленого програмного продукту на основі методів обчислювального інтелекту та машинного навчання

− визначати критерії якості та основні характеристики об’єкту дослідження та програмного продукту біомедичного спрямування розробленого з використанням методів обчислювального та штучного інтелекту.

− ідентифікувати алгоритми, моделі та методи штучного інтелекту які підходять для вирішення тієї чи іншої задачі біомедичного спрямування

− забезпечити декомпозицію задачі біомедичного спрямування на складові елементи з метою застосування відомих методів і технологій для її вирішення

− оптимізувати роботу обраних алгоритмів та методів штучного інтелекту

В методичних вказівках до всіх практичних робіт спочатку наводиться теоретичний виклад теми роботи. Це спрощує розуміння та виконання прикладної (практичної) частини роботи.

Порядок виконання практичної роботи полягає в наступному:

1. Перед початком виконання роботи студенти вивчають опис практичної роботи.

2. Після ознайомлення з теоретичним матеріалом студенти виконують завдання практичної роботи.

3. Практична робота вважається виконаною, якщо вона захищена. При захисті практичної роботи викладач може спитати не лише про суть виконання роботи та її результатах, але і теоретичний матеріал того розділу, до якого відноситься дана робота.

# 1: ОСНОВИ РОБОТИ В PYTHON

**Мета роботи** – засвоїти базовий синтаксис мови програмування Python.

## Теоретичні відомості

Python – це високорівнева динамічна типізована мова програмування із багатьма парадигмами. Код Python дуже схожий із псевдокодом, оскільки він дозволяє висловлювати дуже потужні ідеї в дуже мало рядків коду, при цьому він є дуже читабельним. Як приклад, наведемо реалізацію класичного алгоритму quicksort в Python:

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)
print(quicksort([3,6,8,10,1,2,1]))
# Prints "[1, 1, 2, 3, 6, 8, 10]"
```

**Версії Python та необхідні бібліотеки.**

Наразі підтримуються та використовуються дві версії Python 2.7 та 3.9. Python 3.0 ввів у мову багато змін, тому код, написаний для версії 2.7, може не працювати під 3.9 і навпаки. Ми будемо використовувати Python версії 3.5 та старше. Ви можете перевірити версію Python, що встановлена на вашому комп'ютері запустивши із командної строки *python --version*. Також для наступної роботи буде доречно встановити програму PyCharm

community (безкоштовна програма) разом із бібліотекою Anaconda (використовуйте офіційні сайти програм для установки, вибравши версію, що відповідає вашій операційній системі та характеристикам комп'ютера *https://www.jetbrains.com/ru-ru/pycharm/download/#section=windows*, *https://www.anaconda.com/distribution/*). Крім того необхідно установити бібліотеку *pytorch*, це можна зробити із командної строки Windows командою *pip install torch*.

**Основні типи даних в Python.**

Як і більшість мов, Python має ряд основних типів, включаючи цілі числа, числа з плаваючою комою, булевий тип і рядки. Ці типи даних ведуть себе так, як це знайоме з інших мов програмування.

*Числа*: Цілі числа чи числа з плаваючою комою працюють так, як ви очікували від інших мов:

```
x = 3
print(type(x)) # Prints "<class 'int'>"
print(x)       # Prints "3"
print(x + 1)   # Addition; prints "4"
print(x - 1)   # Subtraction; prints "2"
print(x * 2)   # Multiplication; prints "6"
print(x ** 2)  # Exponentiation; prints "9"
x += 1
print(x)  # Prints "4"
x *= 2
print(x)  # Prints "8"
y = 2.5
print(type(y)) # Prints "<class 'float'>"
print(y, y + 1, y * 2, y ** 2) # Prints "2.5 3.5 5.0 6.25"
```

Зауважте, що на відміну від багатьох мов, Python не має операторів унарного збільшення (x++) або зменшення (x--).

Python також має вбудовані типи для комплексних чисел.

*Booleans*: Python реалізує всі звичні для булевої логіки оператори, але використовує англійські слова, а не символи (&&, || тощо):

```
t = True
f = False
print(type(t)) # Prints "<class 'bool'>"
print(t and f) # Logical AND; prints "False"
print(t or f)  # Logical OR; prints "True"
print(not t)   # Logical NOT; prints "False"
print(t != f)  # Logical XOR; prints "True"
Рядки: Python має велику підтримку для рядків:
hello = 'hello'    # String literals can use single quotes
world = "world"    # or double quotes; it does not matter.
print(hello)       # Prints "hello"
print(len(hello))  # String length; prints "5"
hw = hello + ' ' + world  # String concatenation
print(hw)  # prints "hello world"
hw12 = '%s %s %d' % (hello, world, 12)  # sprintf style string formatting
print(hw12)  # prints "hello world 12"
Рядки мають купу корисних методів; наприклад:
s = "hello"
print(s.capitalize())  # Capitalize a string; prints "Hello"
print(s.upper())       # Convert a string to uppercase; prints "HELLO"
print(s.rjust(7))      # Right-justify a string, padding with spaces; prints "
hello"
print(s.center(7))     # Center a string, padding with spaces; prints "hello"
```

*print(s.replace('l', '(ell)'))  # Replace all instances of one substring with another; prints "he(ell)(ell)o"*

*print('  world '.strip())  # Strip leading and trailing whitespace; prints "world"*

**Контейнери Python.**

Python включає кілька вбудованих типів контейнерів: списки, словники, набори та кортежі.

1) **Списки**

Список є еквівалентом масиву Python, але його можна змінити за розміром і він може містити елементи різних типів:

```
xs = [3, 1, 2]    # Create a list
print(xs, xs[2])  # Prints "[3, 1, 2] 2"
print(xs[-1])     # Negative indices count from the end of the list; prints "2"
xs[2] = 'foo'     # Lists can contain elements of different types
print(xs)         # Prints "[3, 1, 'foo']"
xs.append('bar')  # Add a new element to the end of the list
print(xs)         # Prints "[3, 1, 'foo', 'bar']"
x = xs.pop()      # Remove and return the last element of the list
print(x, xs)      # Prints "bar [3, 1, 'foo']"
```

**Зрізи**: крім доступу до елементів списку один за одним, Python надає стислий синтаксис для доступу до списків; це відомо як зрізи:

```
nums = list(range(5))     # range is a built-in function that creates a list of integers
print(nums)               # Prints "[0, 1, 2, 3, 4]"
print(nums[2:4])          # Get a slice from index 2 to 4 (exclusive); prints "[2, 3]"
```

```
print(nums[2:])          # Get a slice from index 2 to the end; prints "[2, 3, 4]"
print(nums[:2])          # Get a slice from the start to index 2 (exclusive); prints "[0, 1]"
print(nums[:])           # Get a slice of the whole list; prints "[0, 1, 2, 3, 4]"
print(nums[:-1])         # Slice indices can be negative; prints "[0, 1, 2, 3]"
nums[2:4] = [8, 9]       # Assign a new sublist to a slice
print(nums)              # Prints "[0, 1, 8, 9, 4]"
```

Ми знову далі побачимо зрізи у контексті нумерованих масивів (numpy arrays).

**Цикли**: Ви можете переходити до елементів такого списку:

```
animals = ['cat', 'dog', 'monkey']
for animal in animals:
    print(animal)
# Prints "cat", "dog", "monkey", each on its own line.
```

Якщо ви хочете отримати доступ до індексу кожного елемента в тілі циклу, використовуйте вбудовану функцію перерахування (enumerate):

```
animals = ['cat', 'dog', 'monkey']
for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
# Prints "#1: cat", "#2: dog", "#3: monkey", each on its own line
```

**List comprehensions**: часто програмуючи, ми хочемо перетворити один тип даних в інший. Як простий приклад, розглянемо наступний код, який обчислює квадрати чисел:

```
nums = [0, 1, 2, 3, 4]
squares = []
for x in nums:
    squares.append(x ** 2)
print(squares)   # Prints [0, 1, 4, 9, 16]
```

Ви можете зробити цей код простішим використавши list comprehension:

```
nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]
print(squares)   # Prints [0, 1, 4, 9, 16]
List comprehensions може також мати умови:
nums = [0, 1, 2, 3, 4]
even_squares = [x ** 2 for x in nums if x % 2 == 0]
print(even_squares)  # Prints "[0, 4, 16]"
```

2) **Словники**

Словник зберігає пари (ключ, значення), схожі з Map на Java або об'єктом у Javascript. Ви можете використовувати його так:

```
d = {'cat': 'cute', 'dog': 'furry'}  # Create a new dictionary with some data
print(d['cat'])       # Get an entry from a dictionary; prints "cute"
print('cat' in d)     # Check if a dictionary has a given key; prints "True"
d['fish'] = 'wet'     # Set an entry in a dictionary
print(d['fish'])      # Prints "wet"
# print(d['monkey'])  # KeyError: 'monkey' not a key of d
print(d.get('monkey', 'N/A'))  # Get an element with a default; prints "N/A"
print(d.get('fish', 'N/A'))    # Get an element with a default; prints "wet"
del d['fish']         # Remove an element from a dictionary
```

*print(d.get('fish', 'N/A')) # "fish" is no longer a key; prints "N/A"*

**Цикли**: легко перебирати ключі в словнику:

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal in d:
    legs = d[animal]
    print('A %s has %d legs' % (animal, legs))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

Якщо ви хочете отримати доступ до ключів та відповідних їм значень, використовуйте метод елементів:

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal, legs in d.items():
    print('A %s has %d legs' % (animal, legs))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

**Dictionary comprehensions:** вони подібні до List comprehensions, але дозволяють легко будувати словники. Наприклад:

```
nums = [0, 1, 2, 3, 4]
even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}
print(even_num_to_square)  # Prints "{0: 0, 2: 4, 4: 16}"
```

3) **Набори**

Набір – це не упорядкована колекція різних елементів. Як простий приклад розглянемо наступне:

```
animals = {'cat', 'dog'}
```

```
print('cat' in animals)   # Check if an element is in a set; prints "True"
print('fish' in animals)  # prints "False"
animals.add('fish')       # Add an element to a set
print('fish' in animals)  # Prints "True"
print(len(animals))       # Number of elements in a set; prints "3"
animals.add('cat')        # Adding an element that is already in the set does nothing
print(len(animals))       # Prints "3"
animals.remove('cat')     # Remove an element from a set
print(len(animals))       # Prints "2"
```

**Цикли**: Ітерація над набором має той самий синтаксис, що ітерація над списком; однак набори не упорядковані, ви не можете робити припущення про порядок відвідування елементів набору:

```
animals = {'cat', 'dog', 'fish'}
for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
# Prints "#1: fish", "#2: dog", "#3: cat"
```

*Set comprehensions:* подібно до списків та словників, ви можете легко побудувати набор використовуючи set comprehensions:

```
from math import sqrt
nums = {int(sqrt(x)) for x in range(30)}
print(nums)  # Prints "{0, 1, 2, 3, 4, 5}"
```

4) **Кортеж**

Кортеж – це (незмінний) упорядкований список значень. Кортеж багато в чому схожий на список; одна з найважливіших відмінностей

полягає в тому, що кортежі можна використовувати як ключі в словниках і як елементи наборів, тоді як списки не можна. Ось тривіальний приклад:

```
d = {(x, x + 1): x for x in range(10)}  # Create a dictionary with tuple keys
t = (5, 6)        # Create a tuple
print(type(t))    # Prints "<class 'tuple'>"
print(d[t])       # Prints "5"
print(d[(1, 2)])  # Prints "1"
```

**Функції Python**

Функції Python визначаються за допомогою ключового слова *def*. Наприклад:

```
def sign(x):
    if x > 0:
        return 'positive'
    elif x < 0:
        return 'negative'
    else:
        return 'zero'
for x in [-1, 0, 1]:
    print(sign(x))
# Prints "negative", "zero", "positive"
```

Ми часто визначаємо функції, які можуть приймати необов'язкові (keyword) аргументи:

```
def hello(name, loud=False):
    if loud:
        print('HELLO, %s!' % name.upper())
```

```
    else:
        print('Hello, %s' % name)
hello('Bob') # Prints "Hello, Bob"
hello('Fred', loud=True)  # Prints "HELLO, FRED!"
```

**Класи Python**

Синтаксис для визначення класів у Python є простим:

```
class Greeter(object):
    # Constructor
    def __init__(self, name):
        self.name = name  # Create an instance variable
    # Instance method
    def greet(self, loud=False):
        if loud:
            print('HELLO, %s!' % self.name.upper())
        else:
            print('Hello, %s' % self.name)
g = Greeter('Fred')  # Construct an instance of the Greeter class
g.greet()           # Call an instance method; prints "Hello, Fred"
g.greet(loud=True)   # Call an instance method; prints "HELLO, FRED!"
```

**Практична частина**

1. Написати програму на Python, яка:
   a) Сортує словник за значеннями
   b) Додає ключ до словника
      Було : {0: 10, 1: 20}
      Результат : {0: 10, 1: 20, 2: 30}
   c) Поєднає наступні словники та створює новий
      dic1={1:10, 2:20}

dic2={3:30, 4:40}

dic3={5:50,6:60}

Результат : {1: 10, 2: 20, 3: 30, 4: 40, 5: 50, 6: 60}

d) Перевіряє чи входить даний ключ до словника
e) Ітерується по словнику використовуючи for loops.
f) Генерує та виводить словник, який складається с чисел від 1 до n в формі (x, x*x).

   Ввід (n = 5). Результат : {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
g) Генерує словник у якому ключі це числа від 1 до 15, а значення квадрати ключів.

   Приклад {1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100, 11: 121, 12: 144, 13: 169, 14: 196, 15: 225}
h) Яка створює будь який set.
i) Яка ітерується крізь sets.
j) Додає значення до set.
k) Видаляє елемент з set
l) Видаляє елемент якщо він є в set.
m) Створює об’єднання sets (intersection).
n) Яка знаходить суму всіх елементів списку.
o) Яка помножує всі елементи списку
p) Яка виводить найбільше та найменше значення у списку.
q) яка отримує список, відсортований у порядку зростання за останнім елементом у кожному кортежі із заданого списку не порожніх кортежів.

   Приклад : [(2, 5), (1, 2), (4, 4), (2, 3), (2, 1)]

   Результат : [(2, 1), (1, 2), (2, 3), (4, 4), (2, 5)]
r) Яка видаляє елементи які повторюються
s) Яка друкує лист після видалення 0го, 4го та 5го елементів.

   Приклад : ['Red', 'Green', 'White', 'Black', 'Pink', 'Yellow']

   Результат : ['Green', 'White', 'Black']

t) Яка помножує всі значення в списку.

Приклад: (8, 2, 3, -1, 7). Результат: -336

u) Розраховує факторіал числа.

v) Приймає строку та підраховує кількість великих та маленьких букв.

Ввід : 'The quick Brow Fox'

Результат : No. of Upper case characters : 3 No. of Lower case Characters : 12

w) Друкує наступний патерн, використовуючи цикли

```
*
 *
 * *
 * * *
 * * * *
 * * *
 * *
 *
*
```

2. Розписати коротко під кожен пункт, що може бути використано для заданого варіанту.

## Висновки

У висновках обґрунтувати переваги та недоліки мови програмування Python.

## Контрольні запитання

1. Що таке словник?
2. Що таке sets?
3. Що таке кортеж?

# 2: ОСНОВИ РОБОТИ З БІБЛІОТЕКОЮ NUMPY

**Мета роботи:** ознайомитися з базовими методами бібліотеки numpy.

## Теоретична частина

Numpy – це основна бібліотека наукових обчислень на Python. Вона забезпечує високоефективний багатовимірний об'єкт масиву та інструменти для роботи з цими масивами.

**Масиви**

Масив numpy – це сітка значень, однакових типів, що індексується кортежем невід'ємних цілих чисел. Кількість розмірів – це ранг масиву; форма масиву – це набір цілих чисел, що задають розмір масиву вздовж кожного виміру.

Ми можемо створити numpy масиви зі вкладених списків Python та отримати доступ до елементів за допомогою квадратних дужок:

```
import numpy as np
a = np.array([1, 2, 3])   # Create a rank 1 array
print(type(a))            # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5                  # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
print(b.shape)                     # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
```

Numpy також пропонує безліч функцій для створення масивів:

```
import numpy as np
```

```
a = np.zeros((2,2))   # Create an array of all zeros
print(a)              # Prints "[[ 0.  0.]
                      #         [ 0.  0.]]"

b = np.ones((1,2))    # Create an array of all ones
print(b)              # Prints "[[ 1.  1.]]"

c = np.full((2,2), 7)  # Create a constant array
print(c)               # Prints "[[ 7.  7.]
                       #         [ 7.  7.]]"

d = np.eye(2)        # Create a 2x2 identity matrix
print(d)             # Prints "[[ 1.  0.]
                     #         [ 0.  1.]]"

e = np.random.random((2,2))  # Create an array filled with random values
print(e)                     # Might print "[[ 0.91940167  0.08143941]
                             #              [ 0.68744134  0.87236687]]"
```

**Індексація в масивах**

Numpy пропонує декілька способів індексувати масиви.

Зрізи: Подібно до списків Python, у numpy масивах можна брати зрізи. Оскільки масиви можуть бути багатовимірними, необхідно вказати фрагмент (діапазон) для кожного виміру масиву:

```
import numpy as np
# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
```

```
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]

# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0, 1])   # Prints "2"
b[0, 0] = 77     # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])   # Prints "77"
```

Також можна змішати цілочисельну індексацію з індексуванням фрагментів. Однак це дозволить отримати масив нижчого рангу, ніж початковий масив.

```
import numpy as np
# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Two ways of accessing the data in the middle row of the array.
# Mixing integer indexing with slices yields an array of lower rank,
# while using only slices yields an array of the same rank as the
# original array:
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
```

```
print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"
# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)  # Prints "[ 2  6 10] (3,)"
print(col_r2, col_r2.shape)  # Prints "[[ 2]
                             #         [ 6]
                             #         [10]] (3, 1)"
```

Індексація масиву цілими числами: Коли numpy масиви індексуються за допомогою зрізів, результат подання масиву завжди буде підмасивом вихідного масиву. Навпаки, індексація масиву цілими числами дозволяє будувати довільні масиви, використовуючи дані з іншого масиву, наприклад:

```
import numpy as np
a = np.array([[1,2], [3, 4], [5, 6]])
# An example of integer array indexing.
# The returned array will have shape (3,) and
print(a[[0, 1, 2], [0, 1, 0]])  # Prints "[1 4 5]"
# The above example of integer array indexing is equivalent to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))  # Prints "[1 4 5]"
# When using integer array indexing, you can reuse the same
# element from the source array:
print(a[[0, 0], [1, 1]])  # Prints "[2 2]"
# Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]]))  # Prints "[2 2]"
```

Один з корисних трюків з індексуванням масиву цілими числами – це вибір або зміна одного елемента з кожного рядка матриці:

```
import numpy as np
# Create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
print(a)  # prints "array([[ 1,  2,  3],
          #             [ 4,  5,  6],
          #             [ 7,  8,  9],
          #             [10, 11, 12]])"

# Create an array of indices
b = np.array([0, 2, 0, 1])

# Select one element from each row of a using the indices in b
print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"

# Mutate one element from each row of a using the indices in b
a[np.arange(4), b] += 10

print(a)  # prints "array([[11,  2,  3],
          #             [ 4,  5, 16],
          #             [17,  8,  9],
          #             [10, 21, 12]])
```

Індексація масиву булевим значенням: Індексування масиву булевим значенням дозволяє вибирати довільні елементи масиву. Часто цей тип індексації використовується для вибору елементів масиву, які відповідають певній умові, наприклад:

```
import numpy as np
a = np.array([[1,2], [3, 4], [5, 6]])
```

```
bool_idx = (a > 2)   # Find the elements of a that are bigger than 2;
                     # this returns a numpy array of Booleans of the same
                     # shape as a, where each slot of bool_idx tells
                     # whether that element of a is > 2.

print(bool_idx)      # Prints "[[False False]
                     #         [ True  True]
                     #         [ True  True]]"

# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(a[bool_idx])  # Prints "[3 4 5 6]"

# We can do all of the above in a single concise statement:
print(a[a > 2])     # Prints "[3 4 5 6]"
```

**Типи даних**

Кожен масив numpy – це сітка елементів одного типу. Numpy пропонує великий набір числових типів даних, які можна використовувати для побудови масивів. Numpy намагається відгадати тип даних під час створення масиву, але функції, що будують масиви, зазвичай також включають необов'язковий аргумент, щоб чітко вказати тип даних, наприклад:

```
import numpy as np
x = np.array([1, 2])   # Let numpy choose the datatype
print(x.dtype)         # Prints "int64"
```

```
x = np.array([1.0, 2.0])   # Let numpy choose the datatype
print(x.dtype)             # Prints "float64"

x = np.array([1, 2], dtype=np.int64)   # Force a particular datatype
print(x.dtype)                         # Prints "int64"
```

**Математичні операції над масивами**

Основні математичні функції функціонують елементарно на масивах і доступні як перевантаження оператора, так і як функції модуля numpy:

```
import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))

# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
```

```
print(x * y)
print(np.multiply(x, y))

# Elementwise division; both produce the array
# [[ 0.2        0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))

# Elementwise square root; produces the array
# [[ 1.         1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

Зауважте, що на відміну, наприклад, від MATLAB, * – це поелементне множення, а не матричне множення. Замість цього використовуємо *dot* функцію для обчислення внутрішніх добутків векторів, множення вектора на матрицю та множення матриць. *dot* доступна і як функція в модулі numpy, і як метод об'єкту масив:

```
import numpy as np

x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print(v.dot(w))
```

```
print(np.dot(v, w))

# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))

# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

Numpy забезпечує безліч корисних функцій для виконання обчислень на масивах; однією з найкорисніших є sum:

```
import numpy as np

x = np.array([[1,2],[3,4]])
print(np.sum(x))  # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

Крім обчислення математичних функцій за допомогою масивів, часто потрібно переробляти або іншим чином маніпулювати даними в масивах. Найпростіший приклад цього типу операцій – транспонування матриці; щоб транспонувати матрицю, просто використовується атрибут T об’єкта масиву:

```
import numpy as np

x = np.array([[1,2], [3,4]])
print(x)    # Prints "[[1 2]
            #         [3 4]]"
print(x.T)  # Prints "[[1 3]
            #         [2 4]]"

# Note that taking the transpose of a rank 1 array does nothing:
v = np.array([1,2,3])
print(v)    # Prints "[1 2 3]"
print(v.T)  # Prints "[1 2 3]"
```

**Broadcasting**

Broadcasting це потужний механізм, який дозволяє numpy працювати з масивами різної форми при виконанні арифметичних операцій. Часто у нас є менший масив і більший масив, і ми хочемо використовувати менший масив кілька разів, щоб виконати деяку операцію над більшим масивом.

Наприклад, припустимо, що ми хочемо додати постійний вектор до кожного рядка матриці. Ми могли б зробити це так:

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)   # Create an empty matrix with the same shape as x

# Add the vector v to each row of the matrix x with an explicit loop
```

```
for i in range(4):
    y[i, :] = x[i, :] + v
# Now y is the following
# [[ 2  2  4]
#  [ 5  5  7]
#  [ 8  8 10]
#  [11 11 13]]
print(y)
```

Це працює, однак, коли матриця *x* дуже велика, обчислення явного циклу в Python може бути повільним. Зауважимо, що додавання вектора *v* до кожного рядка матриці *x* еквівалентно формуванню матриці *vv* шляхом складання декількох копій *v* вертикально, а потім виконання поелементного сумування *x* і *vv*. Ми могли б реалізувати такий підхід так:

```
import numpy as np
# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
vv = np.tile(v, (4, 1))   # Stack 4 copies of v on top of each other
print(vv)                 # Prints "[[1 0 1]
                          #         [1 0 1]
                          #         [1 0 1]
                          #         [1 0 1]]"
y = x + vv  # Add x and vv elementwise
print(y)  # Prints "[[ 2  2  4
          #         [ 5  5  7]
          #         [ 8  8 10]
          #         [11 11 13]]"
```

Numpy broadcasting дозволяє нам виконувати ці обчислення, фактично не створюючи декількох копій *v*. Розглянемо цю версію, використовуючи broadcasting:

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # Add v to each row of x using broadcasting
print(y)  # Prints "[[ 2  2  4]
          #         [ 5  5  7]
          #         [ 8  8 10]
          #         [11 11 13]]"
```

Рядок *y = x + v* виконується навіть незважаючи на те, що *x* має форму (4, 3) а *v* має форму (3,) завдяки broadcasting; цей рядок працює так, ніби *v* фактично мав форму (4, 3), де кожен рядок був копією *v*, а сума виконувалася поелементно.

Broadcasting двох масивів відповідає наступним правилам:

1. Якщо масиви не мають однакового рангу, додає форму масиву нижнього рангу на 1, поки обидві форми не мають однакової довжини.

2. Кажуть, що два масиви сумісні у вимірі, якщо вони мають однаковий розмір у цьому вимірі, або якщо один з масивів має розмір 1 у цьому вимірі.

3. Масиви можна транслювати (broadcast) разом, якщо вони сумісні в усіх вимірах.

4. Після broadcasting кожен масив веде себе так, ніби має форму, рівну максимуму елементів двох форм вхідних масивів.

5. У будь-якому вимірі, де один масив мав розмір 1, а інший масив мав розмір більше 1, перший масив веде себе так, ніби він був скопійований уздовж цього виміру.

Функції, що підтримують broadcasting, відомі як універсальні функції.

Наведемо кілька прикладів broadcasting:

```
import numpy as np

# Compute outer product of vectors
v = np.array([1,2,3])  # v has shape (3,)
w = np.array([4,5])    # w has shape (2,)
# To compute an outer product, we first reshape v to be a column
# vector of shape (3, 1); we can then broadcast it against w to yield
# an output of shape (3, 2), which is the outer product of v and w:
# [[ 4  5]
#  [ 8 10]
#  [12 15]]
print(np.reshape(v, (3, 1)) * w)

# Add a vector to each row of a matrix
x = np.array([[1,2,3], [4,5,6]])
# x has shape (2, 3) and v has shape (3,) so they broadcast to (2, 3),
# giving the following matrix:
# [[2 4 6]
#  [5 7 9]]
print(x + v)

# Add a vector to each column of a matrix
# x has shape (2, 3) and w has shape (2,).
# If we transpose x then it has shape (3, 2) and can be broadcast
```

```
# against w to yield a result of shape (3, 2); transposing this result
# yields the final result of shape (2, 3) which is the matrix x with
# the vector w added to each column. Gives the following matrix:
# [[ 5  6  7]
#  [ 9 10 11]]
print((x.T + w).T)
# Another solution is to reshape w to be a column vector of shape (2, 1);
# we can then broadcast it directly against x to produce the same
# output.
print(x + np.reshape(w, (2, 1)))

# Multiply a matrix by a constant:
# x has shape (2, 3). Numpy treats scalars as arrays of shape ();
# these can be broadcast together to shape (2, 3), producing the
# following array:
# [[ 2  4  6]
#  [ 8 10 12]]
print(x * 2)
```

Broadcasting зазвичай робить код більш стислим і швидшим, тому ви повинні прагнути використовувати його там, де це можливо.

**SciPy**

Numpy забезпечує високоефективний багатовимірний масив та основні інструменти для обчислення та маніпулювання цими масивами. SciPy ґрунтується на цьому і надає велику кількість функцій, які працюють на numpy масивах і корисні для різних типів наукових та інженерних застосувань.

**Операції із зображенням**

SciPy надає деякі основні функції для роботи із зображеннями. Наприклад, у нього є функції для зчитування зображень з диска в нумеровані масиви, для запису масивів на диск як зображення та для зміни розмірів зображень. Наведемо простий приклад, який демонструє ці функції:

```
from scipy.misc import imread, imsave, imresize

# Read an JPEG image into a numpy array
img = imread('assets/cat.jpg')
print(img.dtype, img.shape)  # Prints "uint8 (400, 248, 3)"

# We can tint the image by scaling each of the color channels
# by a different scalar constant. The image has shape (400, 248, 3);
# we multiply it by the array [1, 0.95, 0.9] of shape (3,);
# numpy broadcasting means that this leaves the red channel unchanged,
# and multiplies the green and blue channels by 0.95 and 0.9
# respectively.
img_tinted = img * [1, 0.95, 0.9]

# Resize the tinted image to be 300 by 300 pixels.
img_tinted = imresize(img_tinted, (300, 300))

# Write the tinted image back to disk
imsave('assets/cat_tinted.jpg', img_tinted)
```

Рисунок 2.1 – Зліва: оригінальне зображення. Праворуч: тоноване і змінене зображення.

**Файли MATLAB**

Функції *scipy.io.loadmat* та *scipy.io.savemat* дозволяють зчитувати та записувати файли MATLAB.

**Відстань між точками**

SciPy визначає деякі корисні функції для обчислення відстаней між множинами точок.

Функція *scipy.spatial.distance.pdist* обчислює відстань між усіма парами точок у заданому наборі:

```
import numpy as np
from scipy.spatial.distance import pdist, squareform

# Create the following array where each row is a point in 2D space:
```

```
# [[0 1]
#  [1 0]
#  [2 0]]
x = np.array([[0, 1], [1, 0], [2, 0]])
print(x)

# Compute the Euclidean distance between all rows of x.
# d[i, j] is the Euclidean distance between x[i, :] and x[j, :],
# and d is the following array:
# [[ 0.         1.41421356  2.23606798]
#  [ 1.41421356  0.         1.        ]
#  [ 2.23606798  1.         0.        ]]
d = squareform(pdist(x, 'euclidean'))
print(d)
```

Аналогічна функція (*scipy.spatial.distance.cdist*) обчислює відстань між усіма парами в двох наборах точок

**Matplotlib**

Matplotlib – це бібліотека графіків. Найважливішою функцією в matplotlib є *plot*, який дозволяє зобразити 2D дані, наприклад:

```
import numpy as np
import matplotlib.pyplot as plt
# Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

# Plot the points using matplotlib
plt.plot(x, y)
plt.show()  # You must call plt.show() to make graphics appear.
```

Запуск цього коду створює рисунок 2.2.

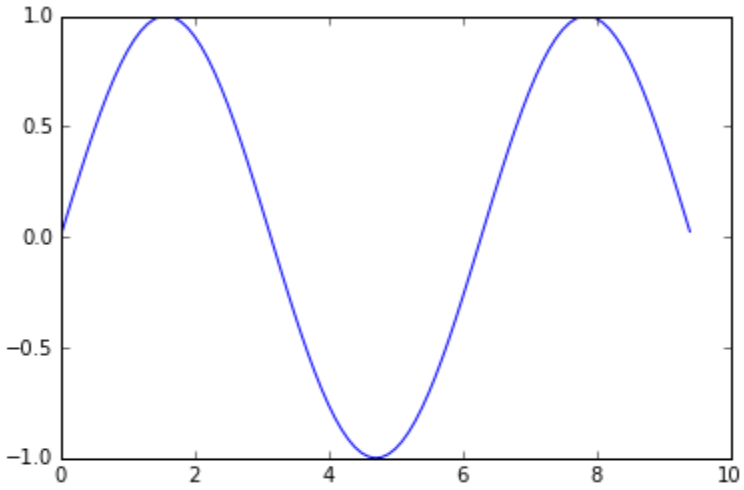



Рисунок 2.2 – Приклад використання функції *plot*

Зробивши трохи додаткових робіт, ми можемо легко побудувати відразу кілька графіків та додати назву, легенду та мітки осі (рисунок 2.3)

```
import numpy as np
import matplotlib.pyplot as plt
# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Plot the points using matplotlib
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
plt.show()
```

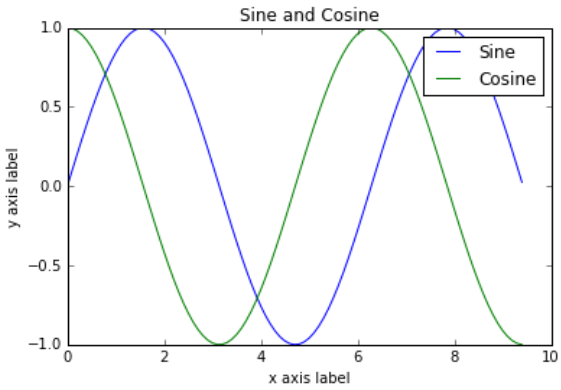



Рисунок 2.3 – Приклад побудови декількох графіків одразу

**Subplots**

Також можна побудувати на одному рисунку різні речі за допомогою функції *subplot*, наприклад (рисунок 2.4):

```
import numpy as np
import matplotlib.pyplot as plt

# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Set up a subplot grid that has height 2 and width 1,
# and set the first such subplot as active.
plt.subplot(2, 1, 1)

# Make the first plot
plt.plot(x, y_sin)
plt.title('Sine')

# Set the second subplot as active, and make the second plot.
```

```
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')

# Show the figure.
plt.show()
```

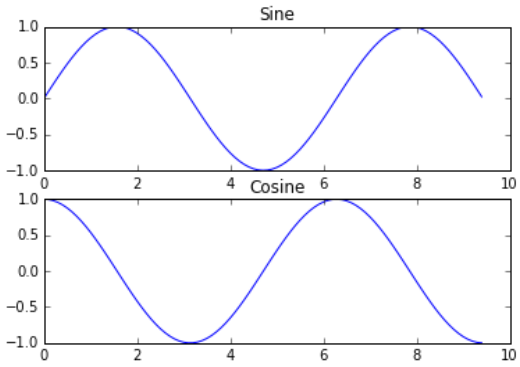



Рисунок 2.4 – Приклад використання функції *subplot*

**Images**

Також можна використовувати функцію *imshow* для показу картинок, наприклад (рисунок 2.5):

```
import numpy as np
from scipy.misc import imread, imresize
import matplotlib.pyplot as plt

img = imread('assets/cat.jpg')
img_tinted = img * [1, 0.95, 0.9]

# Show the original image
plt.subplot(1, 2, 1)
plt.imshow(img)
```

```
# Show the tinted image
plt.subplot(1, 2, 2)

# A slight gotcha with imshow is that it might give strange results
# if presented with data that is not uint8. To work around this, we
# explicitly cast the image to uint8 before displaying it.
plt.imshow(np.uint8(img_tinted))
plt.show()
```


Рисунок 2.5 – Приклад використання функції imshow

**Практична частина**

1. Написати програму мовою Python, яка вирішує наступні завдання:

   1) Створіть 1D масив чисел від 0 до 9

   #> array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

   2) Створіть 3×3 numpy масив, що складається с усіх True’s

   3) Отримайте всі непарні числа з масиву

   Input: arr = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

   output: #> array([1, 3, 5, 7, 9])

   4) Треба замінити всі непарні числа на -1

   Input: arr = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

   Output: #>  array([ 0, -1,  2, -1,  4, -1,  6, -1,  8, -1])

5) Треба замінити всі непарні числа на -1 не змінюючі масив (where)

Input: arr = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

Output:

Out #>  array([ 0, -1,  2, -1,  4, -1,  6, -1,  8, -1])

Arr #>  array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

6) Перетворіть  1D масив у 2D масив з 2 рядками (reshape)

Input: np.arange(10)

#> array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

Output: #> array([[0, 1, 2, 3, 4], #>  [5, 6, 7, 8, 9]])

7) Складіть вертикально масиви a та b (функції vstack, hstack, concatenate)

Input: a=np.arange(10).reshape(2,-1), b=np.repeat(1, 10).reshape(2,-1)

Output:

#> array([[0, 1, 2, 3, 4],

#>  [5, 6, 7, 8, 9],

#>  [1, 1, 1, 1, 1],

#>  [1, 1, 1, 1, 1]])

8) Складіть горизонтально два масиви

Input:

a = np.arange(10).reshape(2,-1)

b = np.repeat(1, 10).reshape(2,-1)

Output:

#> array([[0, 1, 2, 3, 4, 1, 1, 1, 1, 1],

#>  [5, 6, 7, 8, 9, 1, 1, 1, 1, 1]])

9) Зробіть наступний патерн не використовуючи жорстке прописування коду. Використовуйте тільки функції numpy (використовуйте функцію tile)

Input:

a = np.array([1,2,3])

Output:

#> array([1, 1, 1, 2, 2, 2, 3, 3, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3])

10) Отримайте однакові значення між масивами a та b (можете використати функцію intersect1d)

Input:

a = np.array([1,2,3,2,3,4,3,4,5,6])

b = np.array([7,2,10,2,7,4,9,4,9,8])

Output:

array([2, 4])

11) З масиву a видалити значення, які є у масиві b (функція setdiff1d)

Input:

a = np.array([1,2,3,4,5])

b = np.array([5,6,7,8,9])

Output:

array([1,2,3,4])

12) Отримайте індекси елементів які співпадають у двох масивах (where)

Input:

a = np.array([1,2,3,2,3,4,3,4,5,6])

b = np.array([7,2,10,2,7,4,9,4,9,8])

Output:

#> (array([1, 3, 5, 7]),)

13) Отримайте всі значення які більші за 5 та менші або рівні 10 (where, logical_and).

Input:

a = np.array([2, 6, 1, 9, 10, 3, 27])

Output:

(array([6, 9, 10]),)

14) Перетворіть функцію maxx яка працює з двома скалярними значеннями так, щоб вона могла працювати з масивами (vectorize(maxx, otypes=[float]))

Input:

```
def maxx(x, y):
    """Get the maximum of two items"""
    if x >= y:
        return x
    else:
        return y
maxx(1, 5)
#> 5
```

Output:

```
a = np.array([5, 7, 9, 8, 6, 4, 5])
b = np.array([6, 3, 4, 8, 9, 7, 1])
pair_max(a, b)
#> array([ 6.,  7.,  9.,  8.,  9.,  7.,  5.])
```

15) Поміняйте місцями колонку 1 та 2

```
arr = np.arange(9).reshape(3,3)
arr
```

16) Створіть 2D масив розміром 5x3 який складається з довільних десяткових чисел у діапазоні між 5 та 10.

17) Як виводити тільки 3 знаки після коми в масиві numpy (set_printoptions(precision=))?

Input:

```
rand_arr = np.random.random((5,3))
```

18) Як друкувати масив змінивши наукову нотацію на інженерну (1e-01 –> 0.1) (set_printoptions(suppress = , precision=))

Input:

```
# Create the random array
```

```
np.random.seed(100)
rand_arr = np.random.random([3,3])/1e3
rand_arr
#> array([[  5.434049e-04,   2.783694e-04,   4.245176e-04],
#>        [  8.447761e-04,   4.718856e-06,   1.215691e-04],
#>        [  6.707491e-04,   8.258528e-04,   1.367066e-04]])
```

Desired Output:

```
#> array([[ 0.000543,  0.000278,  0.000425],
#>        [ 0.000845,  0.000005,  0.000122],
#>        [ 0.000671,  0.000826,  0.000137]])
```

19) Як надрукувати весь масив numpy без обрізання значень (set_printoptions(threshold =))

Input:

```
np.set_printoptions(threshold=6)
a = np.arange(15)
a
#> array([ 0,  1,  2, ..., 12, 13, 14])
```

Output:

```
a
#> array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

20) Імпортуйте датасет ***iris*** зберігаючи текст неушкодженим

```
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data'
iris = np.genfromtxt(url, delimiter=',', dtype='object')
names = ('sepallength', 'sepalwidth', 'petallength', 'petalwidth', 'species')
```

21) Отримайте текст з колонки ***species*** ([row[4] for row in iris_1d])

Input:

```
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data'
```

iris_1d = np.genfromtxt(url, delimiter=',', dtype=None)

22) Змініть 1D ***iris*** на 2D масив ***iris_2d*** опустивши текстове поле ***species*** ([row.tolist()[:4] for row in iris_1d]).

Input:

url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data'

iris_1d = np.genfromtxt(url, delimiter=',', dtype=None)

23) Знайдіть mean, median, standard deviation першої колонки sepallength (genfromtxt(usecols=))

url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data'

iris = np.genfromtxt(url, delimiter=',', dtype='object')

24) Створіть нормалізовану форму з *iris's sepallength* із значеннями між 0 та 1 так що мінімальне значення буде 0 а максимальне 1 (numpy.ptp).

Input:

url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data'

sepallength = np.genfromtxt(url, delimiter=',', dtype='float', usecols=[0])

25) Розрахуйте 5ий та 95ий перцентилі *iris's sepallength*

url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data'

sepallength = np.genfromtxt(url, delimiter=',', dtype='float', usecols=[0])

26) Знайдіть позиції пропущених значень в першій колонці (where, isnan) *iris_2d's sepallength*

# Input

url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data'

iris_2d = np.genfromtxt(url, delimiter=',', dtype='float')

iris_2d[np.random.randint(150, size=20), np.random.randint(4, size=20)] = np.nan

27) Виберіть рядки *iris_2d* датасету які не мають *nan* значень ([~np.any(np.isnan(row) for row in iris_2d)]).

# Input

url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data'

iris_2d = np.genfromtxt(url, delimiter=',', dtype='float', usecols=[0,1,2,3])

28) Знайдіть кореляцію між *SepalLength* (перша колонка) та *PetalLength* (третя колонка) в *iris_2d*

# Input

url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data'

iris_2d = np.genfromtxt(url, delimiter=',', dtype='float', usecols=[0,1,2,3])

29) Розрахуйте евклідові відстані між двома масивами (linalg.norm).

Input:

a = np.array([1,2,3,4,5])

b = np.array([4,5,6,7,8])

30) Знайдіть всі піки в 1D numpy масиві *a*. Піки це точки з двох боків які мають менших сусідів (diff, where, sign).

Input:

a = np.array([1, 3, 7, 1, 2, 6, 0, 1])

Output:

#> array([2, 5])

Де 2 та 7 це позиції піків 7 та 6

2. Розписати коротко під кожен пункт, що може бути використано для заданого варіанту.

## Висновки

У висновках обґрунтувати переваги та недоліки бібліотеки numpy.

## Контрольні запитання

1. Які основні операції із матрицями реалізовано у бібліотеці numpy?
2. Яка бібліотека використовується для побудови рисунків у Python?

# 3: ОСНОВИ РОБОТИ З БІБЛІОТЕКОЮ PANDAS

**Мета роботи:**

1. Засвоїти базові методи та принципи обробки даних за допомогою бібліотеки Pandas.

2. Опанувати практичні прийоми роботи із бібліотекою Pandas.

## Теоретичні відомості

Pandas – це пакет Python, що забезпечує швидку, гнучку та виразну структуру даних, розроблену для того, щоб зробити роботу з "реляційними" або "розміченими" даними легкою та інтуїтивно зрозумілою. Наявність цього пакету є основоположним складовим фактором на високому рівні для практичного аналізу реальних даних у Python.

Pandas добре підходить для різних типів даних:

• Табличні дані з гетерогенно набраними стовпцями, як у таблиці SQL або таблиці Excel

• Впорядковані та не упорядковані дані (не обов'язково з фіксованою частотою) часових рядів.

• Довільні матричні дані з мітками рядків та стовпців

• Будь-яка інша форма спостережних / статистичних наборів даних.

**Основні команди Pandas.**

Спершу треба імпортувати наступні модулі для роботи

```
import pandas as pd
import numpy as np
```

Перевірка версії Pandas:

```
import pandas as pd
```

*print(pd.__version__)*

Основні об’єкти Pandas:

df – pandas DataFrame object; s – pandas Series object

Створення Dataframe:

*import pandas as pd*

*df=pd.DataFrame({'X':[78,85,96,80,86],'Y':[84,94,89,83,86],'Z':[86,97,96,72,83]});*

*print(df)*

Результат:

```
   X   Y   Z
0  78  84  86
1  85  94  97
2  96  89  96
3  80  83  72
4  86  86  83
```

Створення DataSeries:

*import pandas as pd*

*s = pd.Series([2, 4, 6, 8, 10])*

*print(s)*

Результат:

```
0    2
```

*1 4*

*2 6*

*3 8*

*4 10*

*dtype: int64*

Створення тестових об’єктів

| | |
|---|---|
| pd.DataFrame(np.random.rand(20,5)) | 5 стовпців і 20 рядів випадкових чисел з плаваючою комою |
| pd.Series(my_list) | Створює серію з листа my_list |
| df.index=pd.date_range('1900/1/30', periods=df.shape[0]) | Додає індекс дати |

Перегляд даних

| | |
|---|---|
| df.head(n) | Перші n рядків даних |
| df.tail(n) | Останні n рядків даних |
| df.shape | Кількість рядків та стовпців |
| df.info() | Індекс, тип даних та інформація про пам'ять |
| df.describe() | Підсумкова статистика для чисельних стовпців |
| s.value_counts(dropna=False) | Переглянути унікальні значення та кількість |

| | |
|---|---|
| df.apply(pd.Series.value_counts) | Унікальні значення та кількість для всіх стовпців |

Вибір даних

| | |
|---|---|
| df[col] | Повертає стовпчик із міткою col як Series |
| df[[col1, col2]] | Повертає стовпці як новий DataFrame |
| s.iloc[0] | Вибір за позицією |
| s.loc['index_one'] | Вибір за індексом |
| df.iloc[0,:] | Перший ряд |
| df.iloc[0,0] | Перший елемент першого стовпця |

Чистка даних

| | |
|---|---|
| df.columns = ['a','b','c'] | Перейменовує стовпці |
| pd.isnull() | Перевіряє нульові значення, повертає булевий масив |
| pd.notnull() | Працює навпаки до pd.isnull() |
| df.dropna() | Скинути всі рядки, що містять нульові значення |
| df.dropna(axis=1) | Відкинути всі стовпці, що містять нульові значення |
| df.dropna(axis=1,thresh=n) | Скинути всі рядки, що мають менше n не нульових значень |

| | |
|---|---|
| df.fillna(x) | Замінити усі нульові значення на x |
| s.fillna(s.mean()) | Замінити усі нульові значення на середнє значення |
| s.astype(float) | Перетворити тип даних серії в Float |
| s.replace(1,'one') | Замінити усі значення, рівні 1 на "one" |
| s.replace([2,3],['two', 'three']) | Замінити усі 2 на "two" та на 3 "three" |
| df.rename(columns=lambda x: x + 1) | Масове перейменування стовпців |
| df.rename(columns={'old_name': 'new_name'}) | Вибіркове перейменування |
| df.set_index('column_one') | Змінити індекс |
| df.rename(index=lambda x: x + 1) | Масове перейменування індексу |

Фільтрація, сортування, групування

| | |
|---|---|
| df[df[col] > 0.6] | Рядки там, де колонка col > 0.6 |
| df[(df[col] > 0.6) & (df[col] < 0.8)] | Рядки, де 0.8> col> 0.6 |
| df.sort_values(col1) | Сортувати значення за col1 |
| df.sort_values(col2, ascending=False) | Сортувати значення col2 у порядку зменшення. |

| | |
|---|---|
| df.sort_values([col1,col2], ascending=[True,False]) | Сортувати значення за допомогою col1 у порядку зростання, а потім col2 у порядку зменшення |
| df.groupby(col) | Повертає об'єкт GroupBy для значень з одного стовпця |
| df.groupby([col1,col2]) | Повертає об'єкт GroupBy для значень із декількох стовпців |
| df.groupby(col1)[col2] | Повертає середнє значення значень у col2, згрупованих значеннями в col1 |
| df.pivot_table(index=col1, values=[col2,col3],aggfunc=mean) | Створює поворотну таблицю, яка згруповую за col1 і обчислює середнє значення col2 та col3 |
| df.groupby(col1).agg(np.mean) | Знаходить середнє значення для всіх стовпців для кожної унікальної групи col1 |
| df.apply(np.mean) | Застосовує функцію np.mean() по кожному стовпцю |
| nf.apply(np.max,axis=1) | Застосує функцію np.max() по всьому рядку |

Об’єднання

| | |
|---|---|
| df1.append(df2) | Додає рядки в DF1 до кінця DF2 (стовпці повинні бути однаковими) |

| | |
|---|---|
| pd.concat([df1, df2],axis=1) | Додає стовпці в DF1 до кінця DF2 (рядки повинні бути однаковими) |
| df1.join(df2,on=col1, how='inner') | У стилі SQL приєднує до стовпців DF1 стовпці DF2, де рядки для COL мають однакові значення. "How" може бути 'left', 'right', 'outer' або 'inner' |

Статистика

| | |
|---|---|
| df.describe() | Підсумкова статистика для чисельних стовпців |
| df.mean() | Повертає середнє значення всіх стовпців |
| df.corr() | Повертає кореляцію між стовпцями у даних |
| df.count() | Повертає кількість значень у кожному стовпці даних |
| df.max() | Повертає найвищі значення в кожному стовпці |
| df.min() | Повертає найнижче значення у кожному стовпці |
| df.median() | Повертає медіану кожного стовпця |
| df.std() | Повертає стандартне відхилення кожного стовпця |

Імпортування даних

| | |
|---|---|
| pd.read_csv(filename) | З файлу CSV |
| pd.read_table(filename) | З розмежованого текстового файлу (наприклад, TSV) |
| pd.read_excel(filename) | З файлу Excel |

| | |
|---|---|
| pd.read_sql(query, connection_object) | З таблиці/бази даних SQL |
| pd.read_json(json_string) | З відформатованого рядка JSON, URL - адреси або файлу. |
| pd.read_html(url) | Розписує HTML URL, рядок або файл та витягує таблиці до списку даних |
| pd.read_clipboard() | Приймає вміст вашого буфера обміну і передає його для read_table () |
| pd.DataFrame(dict) | З словників, ключі для імен стовпців, значення для даних як списки |

Експортування даних

| | |
|---|---|
| df.to_csv(filename) | Запис у файл CSV |
| df.to_excel(filename) | Запис у файл Excel |
| df.to_sql(table_name, connection_object) | Запис на таблицю SQL |
| df.to_json(filename) | Запис у файл у форматі JSON |

## Практична частина

Використовуючи бібліотеки Pandas, NumPy и SciPy виконати наступні завдання

1. **Аналіз набору даних пасажирів Titanic**: https://www.kaggle.com/c/titanic/data

Для того, щоб почати працювати з даними, необхідно спочатку завантажити їх з файлу. У цьому завдані ми будемо працювати з даними у форматі CSV, призначений для зберігання табличних даних.

Завантажте датасет в Pandas:

```
import pandas
data = pandas.read_csv('titanic.csv', index_col='PassengerId')
```

Дані будуть завантажені у вигляді DataFrame, з підтримкою якого зручно працювати з ними. В даному випадку параметр: *index_col='PassengerId'*, означає, що колонка PassengerId задає нумерацію строк даного набору даних.

Для того, щоб побачити, що представляють з себе дані, можна використовувати декілька способів: *data[:10]*; або методом датафрейма: *data.head()*.

Один із способів доступу до стовбців даних фрейма – використовувати квадратні скобки та назву стовпчика: *Print(data['Pclass'])*.

Для підрахунку деяких статистик (кількості, середнє значення, максимум, мінімум) можна також використовувати методи датафрейма: *Print(data['Pclass'].value_counts())*

Інструкція по виконанню.

Завантажте датасет titanic.csv (можна скачати тут https://public.opendatasoft.com/explore/dataset/titanic-passengers/export/ або тут https://www.kaggle.com/c/titanic/data) і, використовуючи описані вище способи роботи з даними, знайдіть відповіді на наступні запитання:

a) Яка кількість чоловіків і жінок їхала на кораблі? Приведіть два числа через пробіл.
b) Якій частині пасажирів вдалося вижити? Порахуйте частку пасажирів, що вижили. Відповідь приведіть у відсотках (число в

інтервалі від 0 до 100, знак відсотка не потрібний), округливши до двох знаків.

c) Яку частку пасажири першого класу складали серед всіх пасажирів? Відповідь приведіть у відсотках (число в інтервалі від 0 до 100, знак відсотка не потрібний), округливши до двох знаків.
d) Якого віку були пасажири? Порахуйте середнє і медіану віку пасажирів. Як відповідь приведіть два числа через пробіл.
e) Чи корелює число братів / сестер / подружжя з числом батьків / дітей? Порахуйте кореляцію Пірсона між ознаками SibSp і Parch.
f) Яке найпопулярніше жіноче ім'я на кораблі? Виделіть з повного імені пасажира (колонка Name) його особисте ім'я (First Name). Спробуйте вручну розібрати кілька значень стовпця Name і виробити правило для вилучення імен, а також поділу їх на жіночі і чоловічі.

2. **Аналіз даних світового рейтингу університетів**. З усіх університетів світу, які є кращими? (https://www.kaggle.com/mylesoneill/world-university-rankings)

Ранжування університетів – складна, політична і спірна практика. Існують сотні різних національних і міжнародних університетських рейтингових систем, багато з яких не згодні один з одним. Цей набір даних містить три глобальні рейтинги університетів з різних місць.

Дані про ранжирування університетів.

a. Рейтинг Times університетів світу є одним з найвпливовіших і широко спостережуваних університетських рейтингів. Заснований в Сполученому Королівстві в 2010 році, він піддається критиці за комерціалізацію і підривання авторитету неангломовних навчальних закладів.
b. Академічний рейтинг світових університетів, також відомий як Шанхайський рейтинг, є настільки ж впливовим. Він був

заснований в Китаї в 2003 році і піддався критиці з точки зору людських ресурсів і освіти.

c. Рейтинг центру світових рейтингів університетів, був заснований в 2012 році.

Таблиця 1. Опис файлу wurData.csv (https://www.kaggle.com/mylesoneill/world-university-rankings)

| **Ім'я поля** | **Опис** | **Тип** |
|---|---|---|
| world_rank | світовий ранг університету | Numeric |
| institution | назва університету | String |
| country | Країна університету | String |
| national_rank | ранг університету в межах його країни | Numeric |
| quality_of_education | рейтинг якості освіти | Numeric |
| alumni_employment | ранг для роботи випускників | Numeric |
| quality_of_faculty | ранг за якістю факультету | Numeric |
| publications | числовий | Numeric |
| influence | Ранг публікацій | Numeric |
| citations | рейтинг впливу | Numeric |
| broad_impact | рейтинг цитування | String |

| | | |
|---|---|---|
| patents | ранг широкого впливу (доступно тільки для 2014 і 2015 років) | Numeric |
| score | Ранг патентів | Numeric |
| Year | загальний бал, який використовується для визначення світового рангу | Numeric |

Завдання:

a. У рейтингу залиште тільки дані за 2015 рік.
b. Обчисліть максимальний, мінімальний і середній загальний бал світового рейтингу.
c. Для всіх згаданих країн виведіть кількість університетів, які увійшли до світового рейтингу. Висновок організуйте за спаданням кількості університетів.
d. Обчисліть для кожної країни величину що є сумою загальних балів його університетів. Висновок організуйте по спадаючій цієї величини.
e. Обчисліть кращий університет для кожної країни, висновок організуйте по цій величині.
f. Зробіть висновок про те, чи є кореляція у двох останніх величин.
g. Обчисліть п'ять університетів з найбільшим рейтингом, виведіть їх назви, загальний бал, країни, рейтинги в своїй країні.
h. Випускники яких 5 університетів володіють найбільшим робочим рейтингом?
i. Як оцінити вплив окремих рейтингів на загальний. Який із зазначених рейтингів дає найбільший внесок у загальний рейтинг?

## Висновки

У висновках навести основні методи бібліотеки Pandas, які використовуються при обробці баз даних. Надати відповіді на питання із практичної частини

## Контрольні запитання

1. Для чого використовується бібліотека Pandas?
2. Що являє собою Pandas DataFrame?

# 4: ОСНОВИ РОБОТИ У PYTORCH

**Мета роботи –** засвоїти принципи роботи бібліотеки pytorch.

## Теоретичні відомості

PyTorch – бібліотека на основі Python, яка сприяє побудові моделей глибокого навчання та їх використанню в різних додатках. Але це не просто ще одна бібліотека Deep Learning – це науковий пакет обчислень (як зазначено в офіційних документах PyTorch).

Це науковий обчислювальний пакет на основі Python, орієнтований на два пункти:

1. Заміна NumPy, яка надає можливість для використання потужності графічних процесорів (GPUs).

2. Глибока дослідницька платформа, яка забезпечує максимальну гнучкість та швидкість.

**Поглиблене навчання з PyTorch.**

PyTorch використовує Tensor в якості своєї основної структури даних, яка схожа на масив Numpy. Якщо Ви здивовані таким специфічним вибором структури даних, відповідь полягає в тому, що за наявності відповідного програмного забезпечення та апаратних засобів, тензори забезпечують прискорення різних математичних операцій. Ці операції, що проводяться у великій кількості в Deep Learning, роблять величезну різницю у швидкості.

PyTorch, подібно до Python, фокусується на простоті використання та дає можливість навіть користувачам з дуже базовими знаннями з програмування використовувати Deep Learning у своїх проектах. Це також робить його ідеальною "першою бібліотекою для глибокого навчання".

Чому треба вивчати PyTorch?

Існує багато інших бібліотек Deep Learning: Keras, Tensorflow, Caffe, Theano (RIP) та багато інших. Але чим PyTorch відрізняється?

Ідеальна бібліотека глибокого навчання повинна бути простою для вивчення та використання, достатньо гнучкою для використання в різних програмах, ефективною, щоб ми могли мати справу з величезними наборами даних і достатньо точною, щоб забезпечити правильні результати навіть за наявності невизначеності вхідних даних.

PyTorch дуже добре задовольняє всім вказаним вимогам. «Python подібний» стиль кодування спрощує навчання та використання. Прискорення графічного процесора (GPU), підтримка розподілених обчислень та автоматичний розрахунок градієнта допомагає автоматично виконувати зворотне поширення помилки, починаючи з прямого виразу.

Звичайно, через Python він стикається з ризиком уповільнення виконання, але високоефективний API C++(libtorch) мінімізує ці витрати. Це робить перехід від НДДКР до виробництва дуже плавним.

**Огляд бібліотеки PyTorch**

На рисунку 4.1 описаний типовий робочий процес разом з важливими модулями, пов'язаними з кожним кроком.

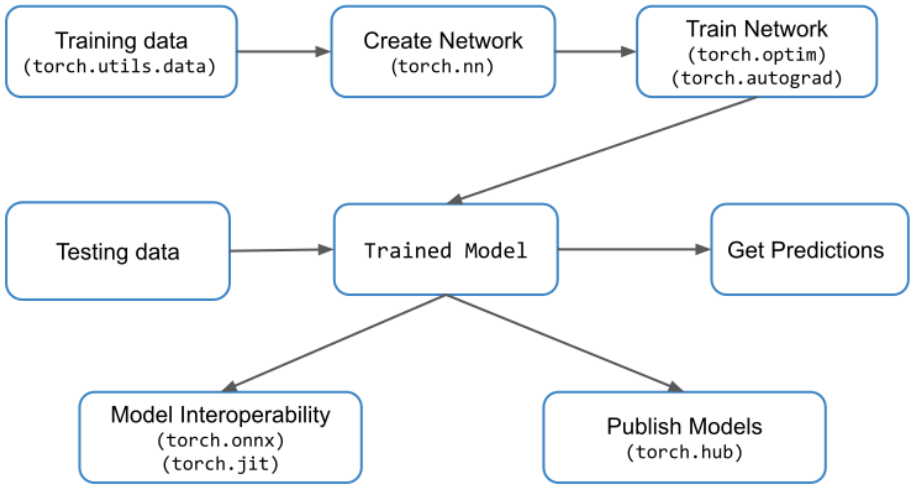



Рисунок 4.1 – Основний робочий процес PyTorch

Важливими модулями PyTorch є: torch.nn, torch.optim, torch.utils та torch.autograd.

1. Завантаження даних та обробка даних.

Перший крок у будь-якому проекті глибокого навчання стосується завантаження та обробки даних. PyTorch надає утиліти для цього у torch.utils.data.

Два важливих класи в цьому модулі – це Dataset та DataLoader.

• Dataset побудований поверх типу даних Tensor і використовується головним чином для спеціальних наборів даних.

• DataLoader використовується, коли у вас є великий набір даних і ви хочете завантажити дані з набору даних у фоновому режимі, щоб він був готовий і чекав навчального циклу.

Також можемо використовувати torch.nn.DataParallel і torch.distributed, якщо у нас є доступ до декількох машин або GPU.

2. Побудова нейронної мережі

Модуль torch.nn використовується для створення нейронних мереж. Він забезпечує всі загальні шари нейронної мережі, як повністю пов'язані шари (fully connected layers), згорткові шари (convolutional layers), функції активації та втрати (activation and loss functions) тощо.

Після того, як архітектура мережі буде створена і дані будуть готові для подачі в мережу, нам потрібні методи для оновлення ваг і зміщення (biases), щоб мережа почала вчитися. Ці утиліти надаються в модулі torch.optim. Аналогічно, для автоматичного диференціювання, яке потрібне під час проходу назад, ми використовуємо модуль torch.autograd.

3. Висновок моделі та сумісність

Після того, як модель буде навчена, її можна використовувати для прогнозування результатів для тестових випадків або навіть нових наборів даних. Цей процес називають висновком моделі.

PyTorch також пропонує TorchScript, який можна використовувати для запуску моделей незалежно від часу виконання Python. Це можна вважати віртуальною машиною з інструкціями, головним чином характерними для тензорів.

Ви також можете конвертувати модель, підготовлену за допомогою PyTorch, у формати типу ONNX, які дозволяють використовувати ці моделі в інших структурах DL, таких як MXNet, CNTK, Caffe2. Ви також можете конвертувати моделі ONNX в Tensorflow.

**Вступ до тензорів**

Тензор – це просто ім'я, яке дається матрицям. Якщо ви знайомі з масивами NumPy, зрозуміти та використовувати тензори PyTorch буде дуже просто. Скалярне значення представлене 0-мірним тензором. Аналогічно матриця стовпців / рядків представлена з використанням 1-D тензора, тощо. Деякі приклади тензорів з різними розмірами наведені нижче на рисунку 4.2, щоб отримати кращу картину.

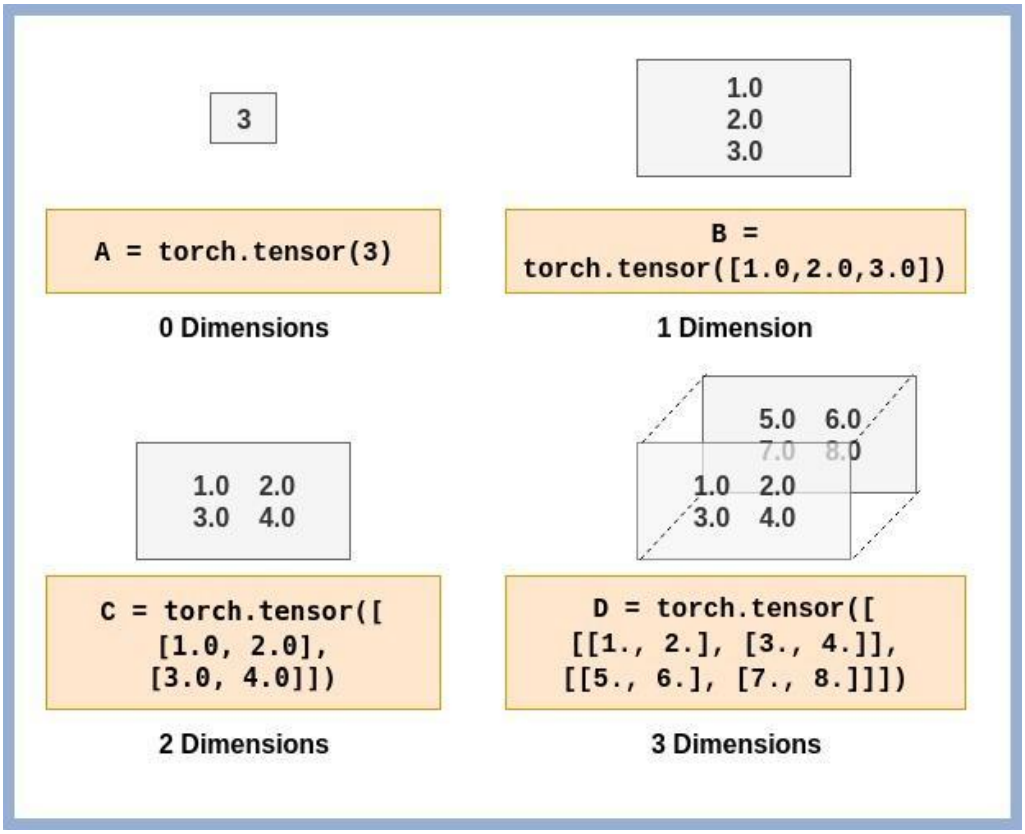



Рисунок 4.2 – Розміри тензорів в PyTorch

Перш ніж розпочати вступ в тензори та у PyTorch, необхідно встановити PyTorch (надалі код, який наданий розроблявся для версії PyTorch 1.1.0, для більш нових версій доведеться його трохи адаптувати, але принципово він не зміниться): *conda install -c pytorch pytorch-cpu (or pip*

*install pytorch).* Краще скористатися Google Colab для пришвидшення розрахунків (оберіть тип виконання GPU у меню).

Побудуємо свій перший тензор.

Подивимося, як ми можемо створити тензор PyTorch:

```
import torch
# Create a Tensor with just ones in a column
a = torch.ones(5)

# Print the tensor we created
print(a)
# tensor([1., 1., 1., 1., 1.])

# Create a Tensor with just zeros in a column
b = torch.zeros(5)
print(b)
# tensor([0., 0., 0., 0., 0.])
```

Ми можемо аналогічно створити Tensor за допомогою спеціальних значень, як показано нижче.

```
c = torch.tensor([1.0, 2.0, 3.0, 4.0, 5.0])
print(c)
# tensor([1., 2., 3., 4., 5.])
```

У всіх перерахованих вище випадках ми створили вектори або тензори розміру 1. Тепер давайте створимо кілька тензорів більшої розмірності.

```
d = torch.zeros(3,2)
print(d)
```

```
# tensor([[0., 0.],
#         [0., 0.],
#         [0., 0.]])

e = torch.ones(3,2)
print(e)
# tensor([[1., 1.],
#         [1., 1.],
#         [1., 1.]])

f = torch.tensor([[1.0, 2.0],[3.0, 4.0]])
print(f)
# tensor([[1., 2.],
#         [3., 4.]])

# 3D Tensor
g = torch.tensor([[[1., 2.], [3., 4.]], [[5., 6.], [7., 8.]]])
print(g)

# tensor([[[1., 2.],
#          [3., 4.]],
#         [[5., 6.],
#          [7., 8.]]])
```

Ми також можемо дізнатися форму тензора за допомогою методу shape:

```
print(f.shape)
# torch.Size([2, 2])
```

*print(e.shape)*
*# torch.Size([3, 2])*

*print(g.shape)*
*# torch.Size([2, 2, 2])*

**Доступ до елемента в Tensor**

Тепер, коли ми створили декілька тензорів, давайте подивимось, як ми можемо отримати доступ до елемента в тензорі. Спочатку давайте подивимося, як це зробити для вектора 1D Tensor як для вектору

*# Get element at index 2*
*print(c[2])*
*# tensor(3.)*

А що робити з 2D або 3D-тензором? Для доступу до одного конкретного елемента в тензорі нам потрібно буде вказати індекси, що дорівнюють розміру тензора. Ось чому для тензора ***c*** нам потрібно було вказати лише один індекс.

*# All indices starting from 0*
*# Get element at row 1, column 0*
*print(f[1,0])*
*# We can also use the following*
*print(f[1][0])*
*# tensor(3.)*
*# Similarly for 3D Tensor*
*print(g[1,0,0])*
*print(g[1][0][0])*
*# tensor(5.)*

Але що робити, якщо ви хотіли отримати доступ до одного цілого ряду у двовимірному тензорі? Ми можемо використовувати той самий синтаксис, як і в NumPy-масивах.

```
# All elements
print(f[:])

# All elements from index 1 to 2 (inclusive)
print(c[1:3])

# All elements till index 4 (exclusive)
print(c[:4])

# First row
print(f[0,:])

# Second column
print(f[:,1])
```

**Тип даних елементів**

Щоразу, коли ми створюємо тензор, PyTorch вирішує який тип даних елементів тензора таким чином, що тип даних може охоплювати всі елементи тензора. Ми можемо це змінити, вказавши тип даних під час створення тензора.

```
int_tensor = torch.tensor([[1,2,3],[4,5,6]])
print(int_tensor.dtype)
# torch.int64

# What if we changed any one element to floating point number?
```

```
int_tensor = torch.tensor([[1,2,3],[4.,5,6]])
print(int_tensor.dtype)
# torch.float32

print(int_tensor)
# tensor([[1., 2., 3.],
#         [4., 5., 6.]])

# This can be overridden as follows
int_tensor = torch.tensor([[1,2,3],[4.,5,6]], dtype=torch.int32)
print(int_tensor.dtype)
# torch.int32

print(int_tensor)
# tensor([[1, 2, 3],
#         [4, 5, 6]], dtype=torch.int32)
```

Список можливих значень аргументу dtype наведено на рисунку 4.3.

| Numpy type | dtype | Torch type | Description |
|---|---|---|---|
| int64 | torch.int64 torch.float | LongTensor | 64 bit integer |
| int32 | torch.int32 torch.int | IntegerTensor | 32 bit signed integer |
| uint8 | torch.uint8 | ByteTensor | 8 bit unsigned integer |
| float64 double | torch.float64 torch.double | DoubleTensor | 64 bit floating point |
| float32 | torch.float32 torch.float | FloatTensor | 32 bit floating point |
| | torch.int16 torch.short | ShortTensor | 16 bit signed integer |
| | torch.int8 | CharTensor | 6 bit signed integer |

Рисунок 4.3 – Список можливих значень аргументу dtype

Тензор до / з NumPy масиву

Тензори PyTorch та масиви NumPy досить схожі. З’являється питання, чи можна перетворити одну структуру даних в іншу? Подивимося, як ми можемо це зробити.

```
# Import NumPy
import numpy as np

# Tensor to Array
f_numpy = f.numpy()
print(f_numpy)

# array([[1., 2.],
#        [3., 4.]], dtype=float32)

# Array to Tensor
h = np.array([[8,7,6,5],[4,3,2,1]])
h_tensor = torch.from_numpy(h)
print(h_tensor)

# tensor([[8, 7, 6, 5],
#         [4, 3, 2, 1]])
```

Арифметичні операції із тензорами

Подивимося, як ми можемо виконувати арифметичні операції на тензорах PyTorch:

```
# Create tensor
tensor1 = torch.tensor([[1,2,3],[4,5,6]])
tensor2 = torch.tensor([[-1,2,-3],[4,-5,6]])

# Addition
print(tensor1+tensor2)
# We can also use
print(torch.add(tensor1,tensor2))
```

```
# tensor([[ 0,  4,  0],
#         [ 8,  0, 12]])

# Subtraction
print(tensor1-tensor2)
# We can also use
print(torch.sub(tensor1,tensor2))

# tensor([[ 2,  0,  6],
#         [ 0, 10,  0]])

# Multiplication
# Tensor with Scalar
print(tensor1 * 2)
# tensor([[ 2,  4,  6],
#         [ 8, 10, 12]])

# Tensor with another tensor
# Elementwise Multiplication
print(tensor1 * tensor2)
# tensor([[ -1,   4,  -9],
#         [ 16, -25,  36]])

# Matrix multiplication
tensor3 = torch.tensor([[1,2],[3,4],[5,6]])
print(torch.mm(tensor1,tensor3))
# tensor([[22, 28],
#         [49, 64]])

# Division
```

```
# Tensor with scalar
print(tensor1/2)
# tensor([[0, 1, 1],
#         [2, 2, 3]])

# Tensor with another tensor
# Elementwise division
print(tensor1/tensor2)
# tensor([[-1,  1, -1],
#         [ 1, -1,  1]])
```

**CPU v/s GPU Tensor**

PyTorch має різну реалізацію Tensor для процесора (CPU) та GPU. Кожен тензор може бути переміщений на GPU, щоб виконати масово паралельні, швидкі обчислення. Усі операції, які будуть виконуватись на тензорі, будуть здійснюватися за допомогою специфічних для GPU підпрограм, що постачаються з PyTorch (якщо у вас немає доступу до GPU, ви можете виконати ці приклади в Google Colab).

Давайте спочатку подивимося, як створити тензор для GPU:

```
# Create a tensor for CPU
# This will occupy CPU RAM
tensor_cpu = torch.tensor([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]], device='cpu')

# Create a tensor for GPU
# This will occupy GPU RAM
tensor_gpu = torch.tensor([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]], device='cuda')
```

Якщо ви використовуєте Google Colab, зосередьтеся на лічильнику споживання оперативної пам’яті у верхньому правому куті, і ви побачите збільшення споживання оперативної пам’яті GPU, як тільки створите tensor_gpu.

Подібно до створення тензорів, операції, проведені для тензорів на процесорі та GPU, також відрізняються і споживають оперативну пам’ять, відповідну вказаному пристрою.

```
# This uses CPU RAM
tensor_cpu = tensor_cpu * 5

# This uses GPU RAM
# Focus on GPU RAM Consumption
tensor_gpu = tensor_gpu * 5
```

Ключовим моментом, який слід зазначити тут, є те, що немає потоків інформації до процесора CPU в тензорних операціях GPU (за винятком випадків, коли ми друкуємо або отримуємо доступ до тензора).

Ми можемо перемістити тензор GPU до процесора і навпаки, як показано нижче:

```
# Move GPU tensor to CPU
tensor_gpu_cpu = tensor_gpu.to(device='cpu')

# Move CPU tensor to GPU
tensor_cpu_gpu = tensor_cpu.to(device='cuda')
```

**Storage (сховище)**

Storage – це одновимірний масив числових даних, такий як суміжний блок пам'яті, що містить числа заданого типу: float або int32.

Pytorch Tensor – це перегляд такої пам’яті, яка здатна індексувати в цьому сховищі, використовуючи зміщення (offset) та кроки (per-dimension strides). Кілька тензорів можуть індексувати одне сховище, навіть якщо вони індексують дані по різному. Приклад можна побачити на рисунку 4.4.

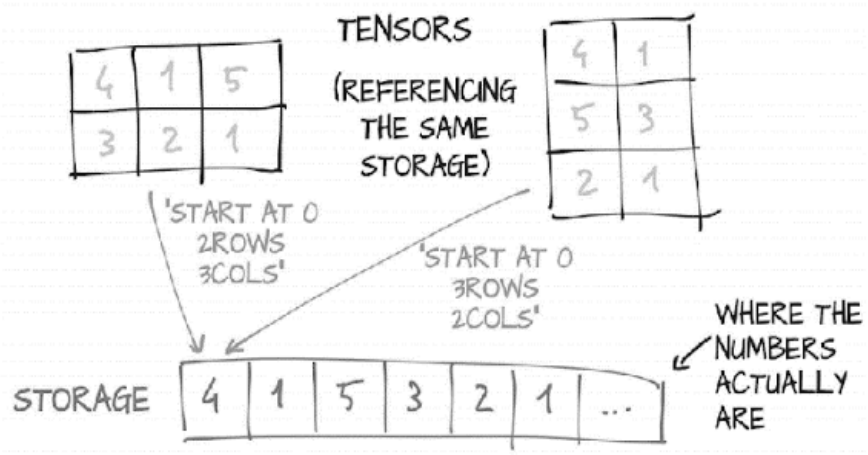



Рисунок 4.4 – Приклад індексування у сховищі PyTorch

Однак основна пам'ять виділяється лише один раз, тому створення альтернативних тензорних копій на даних може бути виконано швидко, незалежно від розміру даних, якими керує екземпляр Storage.

Далі ви побачите, як індексація у сховищі працює на практиці з 2D даними. Ви можете отримати доступ до сховища для певного тензора, використовуючи властивість .storage:

```
points = torch.tensor([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])
print(points.storage())
# 1.0 4.0 2.0 1.0 3.0 5.0 [torch.FloatStorage of size 6]
```

Незважаючи на те, що тензор повідомляє про те, що він має три ряди та два стовпчики, сховище є суміжним масивом розміру 6. У цьому сенсі тензор знає, як перевести пару індексів у місце зберігання в storage.

Ви також можете проіндексувати у storage вручну:

```
points_storage = points.storage()
print(points_storage[0])
# 1.0
print(points.storage()[1])
# 4.0
```

На даний момент не повинно викликати здивування те, що зміна значення сховища змінює вміст його тензору:

```
points = torch.tensor([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])
points_storage = points.storage()
points_storage[0] = 2.0
print(points )
# tensor([[2., 4.], [2., 1.], [3., 5.]])
```

Ви рідко, якщо взагалі, використовуватимете екземпляри пам’яті безпосередньо, але розуміння взаємозв'язку між тензором та базовим сховищем корисно для розуміння вартості певних операцій пізніше. Цю ментальну модель добре враховувати, коли ви хочете написати ефективний код у PyTorch.

Size, storage offset, та strides (розмір, зміщення, крок).

Щоб індексувати сховище, тензори покладаються на кілька фрагментів інформації, які разом із їх сховищем однозначно визначають їх: розмір, зміщення сховища та крок (рисунок 4.5).

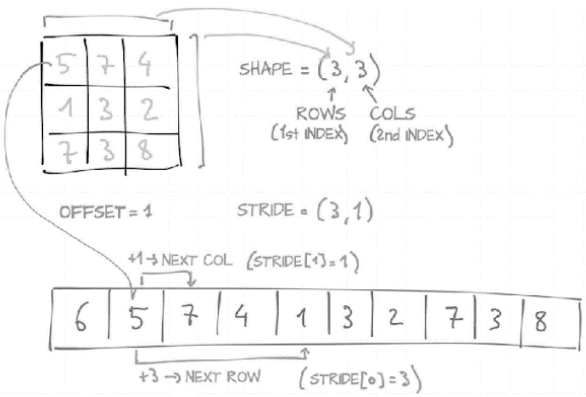



Рисунок 4.5 – Приклад змінних: розмір, зміщення сховища та крок

Розмір (або форма, на мові NumPy) є кортежем, що вказує, скільки елементів у кожному вимірі представляє тензор.

Зміщення – це індекс у сховищі, який відповідає першому елементу в тензорі.

Крок – це кількість елементів у сховищі, які потрібно пропустити, щоб отримати наступний елемент уздовж кожного виміру.

Ви можете отримати другу точку в тензорі, вказавши відповідний індекс:

```
points = torch.tensor([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])
second_point = points[1]
print (second_point.storage_offset())
# 2
print(second_point.size())
# torch.Size([2])
```

Отриманий тензор має зміщення 2 у сховищі (оскільки нам потрібно пропустити першу точку, яка має два елементи), а розмір – це екземпляр класу Size, що містить один елемент, оскільки тензор одновимірний. *Важлива примітка*: ця інформація – це та сама інформація, що міститься у властивості форми об'єктів тензора:

```
print(second_point.shape)
# torch.Size([2])
```

Нарешті, крок – це кортеж із зазначенням кількості елементів у сховищі, які потрібно пропустити, коли індекс збільшується на 1 у кожному вимірі. Наприклад, у тензора точок є крок:

```
print(points.stride())
# (2, 1)
```

Доступ до елемента (i, j) у двовимірному тензорі призводить до доступу до елемента storage_offset + stride [0] * i + stride [1] * j у сховищі. Зсув, як правило, дорівнює нулю; якщо цей тензор представляє собою сховище, створене для розміщення більшого тензора, зміщення може бути позитивним значенням.

Таке опосередкування між Tensor і Storage призводить до того, що деякі операції, такі як переміщення тензора або вилучення субтензора, недорогі, оскільки вони не призводять до перерозподілу пам'яті; натомість вони складаються з виділення нового тензорного об’єкта з різним значенням за розміром, зміщенням пам’яті або кроком.

Ви бачили, як витягнути субтензор, коли проіндексували певну точку і помітили, що зсув пам’яті збільшується. Тепер подивіться, що відбувається з розміром і кроком:

```
points = torch.tensor([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])
second_point = points[1]
print(second_point.size())
# torch.Size([2])
print(second_point.storage_offset())
# 2
```

```
print(second_point.stride())
# (1,)
```

Нижня лінія, субтензор має один менший вимір (як ви очікували), але при цьому індексує той самий обсяг пам’яті, що і оригінальний тензор points. Зміна субтензора також побічно впливає на початковий тензор:

```
points = torch.tensor([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])
second_point = points[1]
second_point[0] = 10.0
print(points)
# tensor([[ 1., 4.],
 [10., 1.],
 [ 3., 5.]])
```

Цей ефект може бути не завжди бажаним, тому ви можете згодом клонувати субтензор у новий тензор:

```
# points = torch.tensor([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])
second_point = points[1].clone()
second_point[0] = 10.0
print(points)
# tensor([[1., 4.],
 [2., 1.],
 [3., 5.]])
```

Спробуємо transposing. Візьмемо тензор точок, який має окремі точки у рядках та координати x та y у стовпцях, та обернемо його так, щоб окремі точки знаходились уздовж стовпців:

```
points = torch.tensor([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])
print(points)
# tensor([[1., 4.],
 [2., 1.],
 [3., 5.]])
points_t = points.t()
print(points_t)
# tensor([[1., 2., 3.],
 [4., 1., 5.]])
```

Ви можете легко переконатися, що два тензори поділяють сховище

```
print(id(points.storage()) == id(points_t.storage()))
# True
```

і що вони відрізняються лише формою та кроком:

```
print(points.stride())
# (2, 1)
print(points_t.stride())
# (1, 2)
```

Цей результат говорить про те, що збільшення першого індексу на 1 у points – тобто перехід від точок points[0,0] до points[1,0] – пропускається вздовж сховища на два елементи, а також збільшення другого індексу з points[0,0] до points[0,1] пропускається вздовж сховища на одиницю. Іншими словами, сховище містить елементи в тензорі послідовно рядок за рядком.

Приклад transpose points в points_t наведено на рисунку 4.6. Ви змінюєте порядок елементів у кроці. Після цього збільшення рядка (перший

показник тензора) пропускає сховище на 1 крок, як коли ви рухалися по стовпцях у точках. Це визначення transposing. Нова пам'ять не виділяється: переміщення отримується лише шляхом створення нового екземпляра Tensor з різним упорядкуванням кроку з оригіналу.

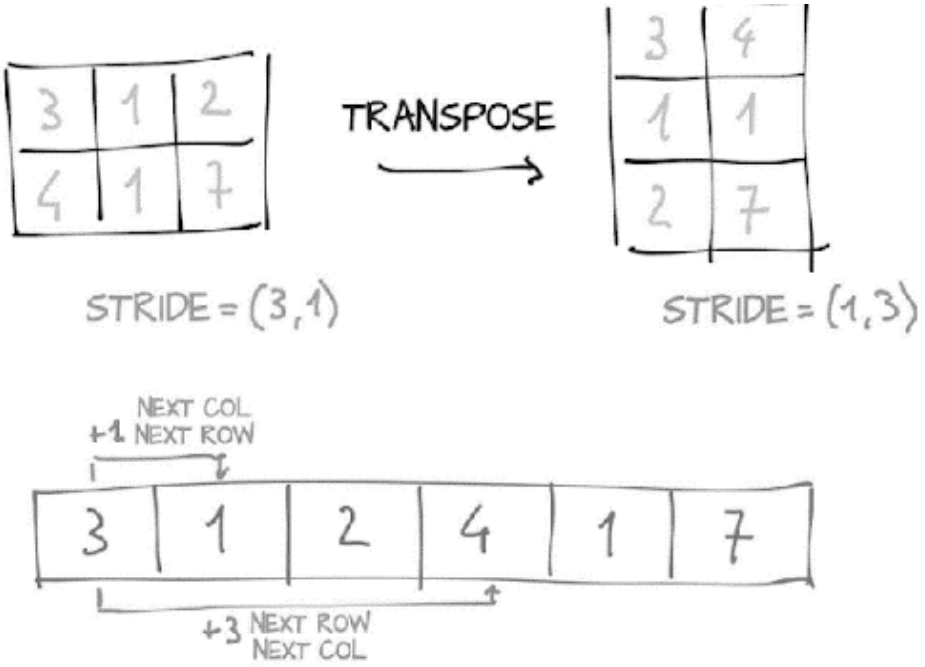



Рисунок 4.6 – Приклад операції transpose

Транспонування в PyTorch не обмежується матрицями. Ви можете транспонувати багатовимірний масив, вказавши два виміри, уздовж яких повинне відбуватися транспонування (наприклад, гортання форми та кроку):

```
some_tensor = torch.ones(3, 4, 5)
some_tensor_t = some_tensor.transpose(0, 2)
print(some_tensor.shape)
# torch.Size([3, 4, 5])
print(some_tensor_t.shape)
# torch.Size([5, 4, 3])
print(some_tensor.stride())
# (20, 5, 1)
```

*print(some_tensor_t.stride())*

*# (1, 5, 20)*

Тензор, значення якого викладені в сховищі, починаючи від самого правого розміру вперед (наприклад, рухаючись по рядках для двовимірного тензора), визначається як суміжний. Суміжні тензори зручні тим, що ви можете відвідувати їх ефективно та не стрибаючи в сховищі. (Покращення локальності даних покращує продуктивність завдяки тому, як працює доступ до пам'яті в сучасних процесорах.)

У цьому випадку points є суміжними, але його transpose не є:

*print(points.is_contiguous())*

*#True*

*print(points_t.is_contiguous())*

*# False*

Ви можете отримати новий суміжний тензор з неспорідненого, використовуючи суміжний метод (contiguous). Вміст тензора залишається однаковим, але крок змінюється, як і сховище:

*points = torch.tensor([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])*

*points_t = points.t()*

*print(points_t)*

*# tensor([[1., 2., 3.],*

*[4., 1., 5.]])*

*print(points_t.storage())*

*# 1.0 4.0 2.0 1.0 3.0 5.0 [torch.FloatStorage of size 6]*

*print(points_t.stride())*

*# (1, 2)*

*points_t_cont = points_t.contiguous()*

```
print(points_t_cont)
# tensor([[1., 2., 3.],
 [4., 1., 5.]])
print(points_t_cont.stride())
# (3, 1)
print(points_t_cont.storage())
# 1.0  2.0  3.0 4.0 1.0 5.0 [torch.FloatStorage of size 6]
```

Зауважте, що сховище було перестановлено для елементів, які будуть викладені рядок за рядком у новому сховищі. Крок змінено, щоб відобразити новий макет.

Методи об’єкта тензор.

Невелика кількість операцій існує лише як методи об’єкта тензора. Їх можна розпізнати за підкресленням у своєму імені, наприклад, zero_, що вказує на те, що метод працює in-place, змінюючи вхідне значення замість створення нового тензора та повернення його. Наприклад, метод zero_ повертає нулі всіх елементів вхідного тензора. Будь-який метод без підкреслення залишає вихідний тензор незмінним і повертає новий тензор:

```
a = torch.ones(3, 2)
a.zero_()
print(a)
# tensor([[0., 0.], [0., 0.], [0., 0.]])
```

## Практична частина

Виконати наступні завдання використовуючи бібліотеку PyTorch.

1) Створіть із списку list(range(9)) тензор. Передбачте та перевірте, який розмір, зсув та кроки він має.

2) Створіть тензор b = a.view(3, 3). Яке значення має b[1,1]?

3) Створіть тензор c = b[1:,1:]. Передбачте та перевірте, який розмір, зсув та кроки він має.

4) Виберіть математичну операцію, таку як косинус або квадратний корінь. Чи можете ви знайти відповідну функцію в бібліотеці torch?

5) Чи існує версія вашої функції, яка працює in-place?

## Висновки

У висновках навести базових принцип зберігання даних у Pytorch та надати відповіді на завдання.

## Контрольні запитання

1. Що являє із себе бібліотека PyTorch?
2. Які основні операції з масивами реалізовані у бібліотеці PyTorch?

# 5: ГРАДІЄНТНИЙ СПУСК

**Мета роботи –** ознайомитись з механізмом градієнтного спуску та навчитися його програмній реалізації у PyTorch.

## Теоретичні відомості

Напишемо код градієнтного спуску. Зробити це дуже просто, нам вистачить всього декількох рядків. Припустимо, що у нас є тензор "x", це torch.tensor. Потрібно вказати, що з цього тензора ми потім, згодом, хочемо порахувати похідні. Це робиться за допомогою аргументу "requires_grad = True". Тепер, якщо ми будемо складати з цього тензора деяку формулу, то ця формула буде пам'ятати, що це насправді не якась константа, а насправді це змінна, по якій можна потім порахувати похідні. Далі ми цей тензор можемо, якщо хочемо, перекласти на GPU. Якщо не хочемо – якщо ми хочемо його залишити на CPU – то він там так і залишиться. Якщо у нас є тільки CPU, то тензор перейде на CPU (нічого не робиться, по суті), якщо у нас є деякий GPU, то переводимо на нульову відео карту.

```
device = torch.device('cuda:0'
                      if torch.cuda.is_available()
                      else 'cpu')
x = x.to(device)
```

І далі ми складаємо певну функцію, яка залежить від тензора "x", який є вже змінною. І ця функція точно така ж, як в нашому прикладі: це 10 помножити на суму квадратів.

```
function = 10 * (x ** 2).sum()
```

Отже, як ми пам'ятаємо, в PyTorch все – це тензори, операції з ними – по-компонентні, якщо це не обумовлено якось інакше, якщо це не якась спеціальна операція. Якщо "x" звести в квадрат, то кожна компонента тензора "x" зведеться в квадрат. Якщо у нас стоїть ".sum ()", то значить, що весь тензор підсумовується, виходить скаляр. У цій функції (function) ми повинні порахувати похідні. Робиться це за допомогою методу "backward".

*function.backward()*

Чому метод називається "backward"? Справа в тому, що якщо ви обчислюєте функцію послідовно, тобто у вас береться деякий аргумент у функції, зводиться в квадрат, після цього проводиться сумування за всіма компонентами цього аргументу, після цього ви помножуєте результат на 10, це – "forward pass " – це правильний порядок.

Якщо ви будете обчислювати похідні, то вам потрібно йти з кінця: взяти останню операцію, потім взяти передостанню, потім перед-передостанню, і так далі, поки ви не дійдете до самого аргументу "x". Тому це називається "backward". І, що найголовніше, як ви пам'ятаєте, результат обчислення похідної по тензору був розміру, як наш тензор початковий. І, мабуть тому, творці PyTorch вирішили зберігати градієнт, тобто результат обчислення похідної за всіма компонентами, в самій змінній "x", в самому тензорі "x". Тобто тут треба розуміти, що беремо ми похідну від функції, а результат цієї похідної, результат операції "backward" у нас з'являється в атрибуті ".grad" тензора.

Тоді, весь початковий кусок коду має наступний вигляд:

```
import torch
x = torch.tensor(
   [[1., 2., 3., 4.],
    [5., 6., 7., 8.],
```

```
    [9., 10., 11., 12.]], requires_grad=True)

device = torch.device('cuda:0'
                      if torch.cuda.is_available()
                      else 'cpu')
x = x.to(device)

function = 10 * (x ** 2).sum()
function.backward()
print(x.grad, '<- gradient')
tensor([[ 20.,  40.,  60.,  80.],
        [100., 120., 140., 160.],
        [180., 200., 220., 240.]]) <- gradient
```

Може виникнути питання – а як наша функція розуміє, як обчислити по собі похідну? Справа в тому, що функція розуміє, в якому порядку ми робили операції і знає аналітичні формули: як за цими операціями робити обчислення похідної. Ми можемо побачити порядок обчислення, точніше порядок тих операцій, які ми робили, і навіть вивести його на екран. Якщо ми візьмемо у функції атрибут ".grad_fn", це буде остання функція, яка привела до кінцевого результату.

```
print(function.grad_fn)
```

Передостанню можна подивитися таким чином,

```
print(function.grad_fn.next_functions[0][0])
```

перед-перед-передостанню

*print(function.grad_fn.next_functions[0][0].next_functions[0][0])*

і, нарешті, кінцеву:

*print(function.grad_fn.next_functions[0][0].next_functions[0][0].next_functions[0][0])*

Ми бачимо, що в кінці ми робили "Multiply Backward", це означає що ми робили множення. Передостаннє – це було підсумовування. Перед-перед-останнє – це було обчислення квадрата. Коли всі функції закінчилися, остання функція – це "AccumulateGrad", тобто "порахувати весь градієнт": ми закінчили розгортати формулу, давайте рахувати.

Весь кусок коду тепер має наступний вигляд:

*print(function.grad_fn)*
*print(function.grad_fn.next_functions[0][0])*
*print(function.grad_fn.next_functions[0][0].next_functions[0][0])*
*print(function.grad_fn.next_functions[0][0].next_functions[0][0].next_functions[0][0])*
*<MulBackward0 object at 0x10ce83c88>*
*<SumBackward0 object at 0x10ce83e10>*
*<PowBackward0 object at 0x10ce83c88>*
*<AccumulateGrad object at 0x10ce83e10>*

Для того, щоб закінчити написання градієнтного спуску, нам не вистачає одного елементу: це оновлення тензора "x". Воно відбувається наступним чином:

*x.data -= 0.001 * x.grad*

Тут є деяка тонкість. Ми взяли градієнт, який у нас зберігається в тензорі "x". Помножили його на градієнтний крок, і хочемо оновити тензор "x". Чому тут написано "x.data", а не просто "x"? Справа в тому, що якщо ви хочете оновити тензор, у якого можна обчислити градієнт, то не дуже зрозуміло, як це використовувати в подальших обчисленнях. Тобто – як повинна повести себе функція від цього оновленого тензора, чи повинна вона включити це обчислення свого градієнта, або це деяка передобробка тензора "x". Відповідно, PyTorch не дозволяє таку операцію, але ми можемо це обійти: ми можемо обновити не тензор, за яким можна обчислити градієнт, а самі дані, які лежать в цьому тензор. Тобто це буде точно такий же тензор, тільки у нього флаг "requires_grad" буде стояти в False. І для цього можна взяти ".data", це буде той самий тензор, тільки з "requires_grad = False". І це ніяк не вплине на обчислення наступних функцій.

Другий тонкий момент – це те, що якщо ми будемо робити ці операції багаторазово (у нас градієнтний спуск – це ітеративна операція), в PyTorch для зручності градієнти не оновлюється після градієнтних кроків – вони весь час накопичуються. Тобто, підсумовуються. Якщо ми зробимо один градієнтний крок, потім зробимо другий крок, то у нас результатом "x.data" буде сума попереднього градієнта і поточного. І для того щоб таке не відбувалося, потрібно обнуляти градієнт – тут ми це зробимо самостійно:

*x.grad.zero_()*

Метод "zero_" з нижнім підкреслюванням дозволяє обнулити весь градієнт. Взагалі, в PyTorch дуже часто використовуються функції з нижнім підкреслюванням на кінці, вони означають, що результат цієї операції буде проведений на тому ж об'єкті, до якого застосовується цей метод (тобто, це "in-place" операція). Треба сказати, що той метод який ми тут реалізували – це найпростіша модифікація градієнтного спуску. Якщо ми хочемо щось

більш просунуте, то нам потрібно використовувати під-модуль "PyTorch.optim". Отже, фінальний код такий:

```
x.data -= 0.001 * x.grad
x.grad.zero_()
```

На минулому кроці ми реалізували градієнтний спуск у вигляді декількох рядків, які потрібно послідовно виконувати, щоб робити градієнтні кроки. Непогано було б з цього зробити деякі функції, і тоді ми зможемо їх викликати скільки завгодно раз і робити градієнтний спуск будь-якої довжини. Реалізуємо функцію make_gradient_step, яка буде робити градієнтний крок. На вхід вона буде отримувати функцію, яку ми хочемо оптимізувати, можемо назвати це функцією втрат або loss-функцією, а другим аргументом вона буде отримувати змінну: попередній стан, в якій вона перебувала – ту змінну, по якій ми хочемо робити градієнтні кроки. Усередині вона буде робити наступне: вона буде підраховувати значення в попередньому положенні, вона буде обчислювати похідну, робити "backward", далі ми будемо робити градієнтний крок, тобто помножувати розмір градієнта на learning rate, віднімати значення, що вийшло з попереднього значення змінної і обнуляти градієнт, тому що в PyTorch за замовчуванням градієнти накопичуються.

```
def make_gradient_step(function, variable):
    function_result = function(variable)
    function_result.backward()
    variable.data -= 0.001 * variable.grad
    variable.grad.zero_()
```

Тепер ми можемо ставити будь-яку функцію втрат, яку ми хочемо оптимізувати. У нашому випадку це параболічна функція, помножена на 10.

```
def function_parabola(variable):
    return 10 * (variable ** 2).sum()
```

Давайте скористаємося цією функцією і подивимося що вийде: передаємо туди нашу параболічну функцію, назвемо її "function parabola". Візьмемо тензор "x" (тепер всього з двох координат, тому що так зручніше для візуалізації), початкова точка наближення – це координати (8, 8), градієнтний крок – одна тисячна, зробимо 500 градієнтних кроків.

```
for i in range(500):
    var_history.append(x.data.cpu().numpy().copy())
    fn_history.append(function_parabola(x).data.cpu().numpy().copy())
    make_gradient_step(function_parabola, x)
```

Що нам потрібно змінити в цьому коді, який вже працює, щоб застосовувати будь-які інші градієнтні спуски, які реалізовані в PyTorch? Насправді, досить змінити всього три рядки.

По-перше нам знадобиться деякий "optimizer", тобто це буде деякий об'єкт, який знає, як робити градієнтні кроки. Раніше ми просто помножували на learning rate, і зменшували значення "x". Можливо, більш сильні методи градієнтного спуску будуть робити щось інакше. Відповідно, давайте створимо такий об'єкт "optimizer". У найпростішому випадку це буде SGD, тобто стохастичний градієнтний спуск з пакета torch.optim. Всередину об'єкта torch.optim.SGD можемо передати дуже багато різних параметрів для його створення, але два необхідних – це та змінна, всередині якої будуть рахуватися градієнти, і learning rate, розмір градієнтного кроку, тут він, як у нас раніше, одна тисячна. Є багато інших параметрів. Наприклад, сюди ж можна передати "Нестеров моментум", використовувати його чи ні, можна передати L2-регуляризацію на змінну

"x", але якщо ми залишимо все як є, то це буде рівно той же градієнтний спуск, який ми реалізовували раніше. І тепер, зауважте, у нас optimizer – це деяка обгортка змінної "x". Відповідно, якщо ми скажемо "optimizer.step" після того, як ми зробили "backward" (після того, як ми зробили обчислення похідної), то optimizer сам зробить градієнтний крок і значення "x" зміниться. Крім того, потрібно обнулити градієнт, тому що, ще раз наголосимо, що градієнти в PyTorch накопичуються. Все інше залишається, як раніше.

Весь код даного пункту наступний:

1) Без optimizer

```
import torch

x = torch.tensor(
   [8., 8.], requires_grad=True)
var_history = []
fn_history = []

def function_parabola(variable):
   return 10 * (variable ** 2).sum()

def make_gradient_step(function, variable):
   function_result = function(variable)
   function_result.backward()
   variable.data -= 0.001 * variable.grad
   variable.grad.zero_()

for i in range(500):
   var_history.append(x.data.cpu().numpy().copy())
```

```
    fn_history.append(function_parabola(x).data.cpu().numpy().copy())
    make_gradient_step(function_parabola, x)
```

2) Із optimizer

```
import torch

x = torch.tensor(
    [8., 8.], requires_grad=True)
var_history = []
fn_history = []

optimizer = torch.optim.SGD([x], lr=0.001)
def function_parabola(variable):
    return 10 * (variable ** 2).sum()

def make_gradient_step(function, variable):
    function_result = function(variable)
    function_result.backward()
    optimizer.step()
    optimizer.zero_grad()

for i in range(500):
    var_history.append(x.data.numpy().copy())
    fn_history.append(function_parabola(x).data.cpu().numpy().copy())
    make_gradient_step(function_parabola, x)
```

Чим цей код хороший? Ми можемо тепер замість torch.optim.SGD поставити будь-який інший оптимізатор – наприклад, Adam або RProp, або RMSProp, або що завгодно інше з будь-якими параметрами і подивитися, як

зміниться наша збіжність. Якщо ми хочемо візуалізувати те, що у нас виходить, внаслідок градієнтного спуску, ми можемо спробувати намалювати графік ліній рівня функції, яку ми оптимізуємо (в нашому випадку, це будуть кола (для параболи)). Це лінії, на яких функція має одне і те ж значення, і червоними крапками відзначимо деяку траєкторію руху "x", тобто траєкторію нашої змінної "x", яку вона проходить в процесі градієнтного спуску (рисунок 5.1). Ми починали з початкового наближення в точці (8, 8), ми бачимо що у нас градієнтний спуск спускається, дійсно, в мінімум функції, в точку (0, 0) – там, де функція дорівнює нулю.

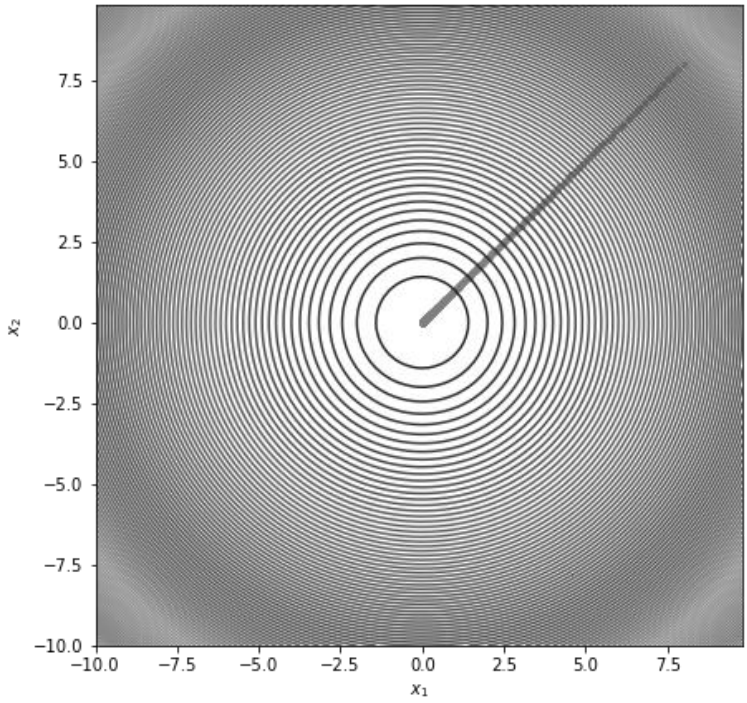



Рисунок 5.1 – Приклад візуалізації роботи градієнтного спуску

Код для візуалізації:

```
import torch
import numpy as np
import matplotlib.pyplot as plt

def show_contours(objective,
```

```
                x_lims=[-10.0, 10.0],
                y_lims=[-10.0, 10.0],
                x_ticks=100,
                y_ticks=100):
        x_step = (x_lims[1] - x_lims[0]) / x_ticks
        y_step = (y_lims[1] - y_lims[0]) / y_ticks
        X, Y = np.mgrid[x_lims[0]:x_lims[1]:x_step, y_lims[0]:y_lims[1]:y_step]
        res = []
        for x_index in range(X.shape[0]):
            res.append([])
            for y_index in range(X.shape[1]):
                x_val = X[x_index, y_index]
                y_val = Y[x_index, y_index]
                res[-1].append(objective(np.array([[x_val, y_val]]).T))
        res = np.array(res)
        plt.figure(figsize=(7,7))
        plt.contour(X, Y, res, 100)
        plt.xlabel('$x_1$')
        plt.ylabel('$x_2$')
    show_contours(function_parabola)
    plt.scatter(np.array(var_history)[:,0], np.array(var_history)[:,1], s=10, c='r');
```

Подивимося, як швидко відбувається зменшення функції втрат – тієї функції, яку ми оптимізуємо. Виявляється що зменшення відбувається експоненціально (рисунок 5.2).

Код для візуалізації:

```
    plt.figure(figsize=(7,7))
```

*plt.plot(fn_history);*

*plt.xlabel('step')*

*plt.ylabel('function value');*

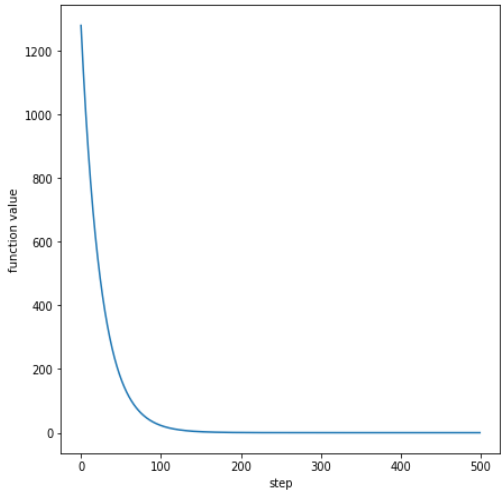



Рисунок 5.2 – Приклад зменшення значення функції втрат

На рисунку 5.2 зазначені значення функції на різних етапах градієнтного спуску від нульового кроку до 500 – отримали вигляд експоненти. Ми можемо в цьому переконатися якщо ми візьмемо шкалу "y" за логарифмом. Тобто якщо ми візьмемо від "y" логарифм, вийде, що наш графік перетворюється в пряму (рисунок 5.3). Це означає що спочатку там було щось у вигляді "$e^x$". І це цікава властивість градієнтного спуску на квадратичних функціях – що він досить швидко сходиться за експоненціальним законом.

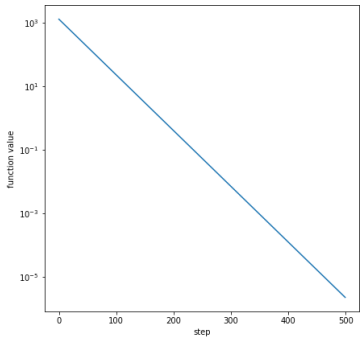



Рисунок 5.3 – Приклад візуалізації логарифму функції втрат

Код візуалізації:

```
plt.figure(figsize=(7,7))
plt.semilogy(fn_history);
plt.xlabel('step')
plt.ylabel('function value');
```

Тепер візьмемо більш складну функцію, лінії рівня якої представлені на рисунку 5.4 (@ – це матричне множення).

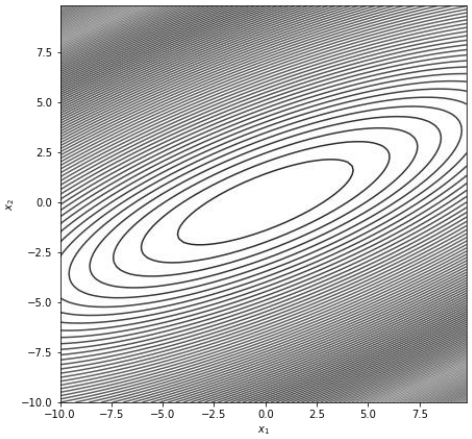



Рисунок 5.4 – Приклад лінії рівня складної функції

Код для візуалізації:

```
def function_skewed(variable):
    gramma = torch.tensor([[1., -1.], [1., 1.]]) @ torch.tensor([[1.0, 0.0], [0.0, 4.0]])
    res = 10 * (variable.unsqueeze(0) @ (gramma @ variable.unsqueeze(1))).sum()
    return res

def function_skewed_np(variable):
```

*gramma = np.array([[1, -1], [1, 1]]) @ np.array([[1.0, 0.0], [0.0, 4.0]])*

*res = 10 * (variable.transpose(1, 0) @ (gramma @ variable)).sum()*

*return res*

*show_contours(function_skewed_np)*

Для того щоб її реалізувати, довелося це зробити і в PyTorch, і в NumPy. NumPy-реалізація потрібна була для візуалізації ліній рівня, а PyTorch потрібна була власне, щоб передавати цю функцію всередину градієнтного спуску. Якщо ми застосуємо наш градієнтний спуск на цій функції, то побачимо трошки інший характер оптимізації (рисунок 5.5).

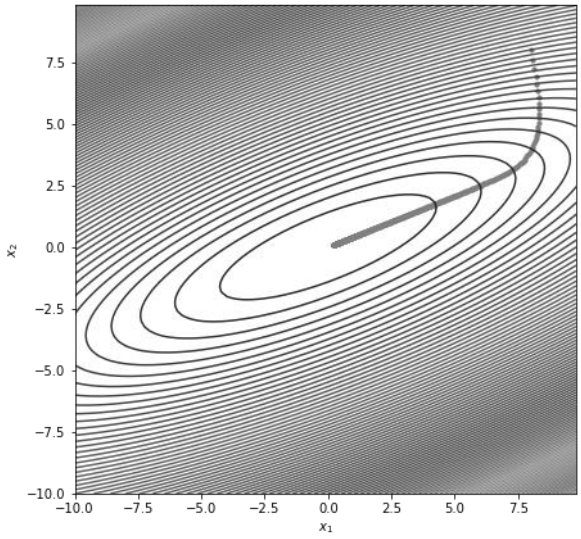



Рисунок 5.5 – Характер оптимізації складної функції

Код візуалізації та градієнтного спуску:

*x = torch.tensor(*

*[8., 8.], requires_grad=True)*

*var_history = []*

*fn_history = []*

*optimizer = torch.optim.SGD([x], lr=0.001)*

*for i in range(500):*

*var_history.append(x.data.cpu().numpy().copy())*

*fn_history.append(function_skewed(x).data.cpu().numpy().copy())*

*make_gradient_step(function_skewed, x)*

*show_contours(function_skewed_np)*

*plt.scatter(np.array(var_history)[:,0], np.array(var_history)[:,1], s=10, c='r');*

**Приклад лінійної регресії.**

Розглянемо тепер приклад реальної прикладної задачі, а саме розробимо алгоритм лінійної регресії.

Створимо модель, яка передбачає врожайність яблук та апельсинів (цільові змінні), по середній температурі, кількості опадів та вологості (вхідні змінні чи особливості) у регіоні. Дані для навчання наведені на рисунку 5.6.

| Region | Temp. (F) | Rainfall (mm) | Humidity (%) | Apples (ton) | Oranges (ton) |
|---|---|---|---|---|---|
| Kanto | 73 | 67 | 43 | 56 | 70 |
| Johto | 91 | 88 | 64 | 81 | 101 |
| Hoenn | 87 | 134 | 58 | 119 | 133 |
| Sinnoh | 102 | 43 | 37 | 22 | 37 |
| Unova | 69 | 96 | 70 | 103 | 119 |

Рисунок 5.6 – Дані для навчання лінійної регресії

У лінійній регресійній моделі кожна цільова змінна оцінюється як зважена сума вхідних змінних, зміщена деякою постійною, відомою як зміщення:

*yield_apple = w11 * temp + w12 * rainfall + w13 * humidity + b1*

*yield_orange = w21 * temp + w22 * rainfall + w23 * humidity + b2*

Дані для тренування можуть бути представлені за допомогою 2 матриць: вхід та ціль, кожна з яких має один рядок на спостереження та один стовпець на змінну:

```
# Input (temp, rainfall, humidity)
inputs = np.array([[73, 67, 43],
                   [91, 88, 64],
                   [87, 134, 58],
                   [102, 43, 37],
                   [69, 96, 70]], dtype='float32')

# Targets (apples, oranges)
targets = np.array([[56, 70],
                    [81, 101],
                    [119, 133],
                    [22, 37],
                    [103, 119]], dtype='float32')
```

Ми розділили вхідні та цільові змінні, оскільки будемо працювати над ними окремо. Виконаємо деяку обробку, а потім перетворимо їх у тензори PyTorch наступним чином:

```
# Convert inputs and targets to tensors
inputs = torch.from_numpy(inputs)
targets = torch.from_numpy(targets)
```

Лінійна модель регресії з нуля.

Ваги та зміщення (w11, w12, ... w23, b1 & b2) також можна представити у вигляді матриць, ініціалізованих як випадкові величини. Перший ряд w і перший елемент b використовуються для прогнозування

першої цільової змінної, тобто врожайності яблук, і аналогічно другий для апельсинів:

```
w = torch.randn(2, 3, requires_grad=True)
b = torch.randn(2, requires_grad=True)
```

torch.randn створює тензор із заданою формою, з елементами, вибраними випадковим чином із звичайного розподілу із середнім значенням 0 та стандартним відхиленням 1.

Наша модель – це просто функція, яка виконує матричне множення входів inputs та ваг w (транспонована матриця) і додає зміщення b (повторюється для кожного спостереження).

Ми можемо визначити модель наступним чином:

```
def model(x):
   return x @ w.t() + b
```

@ представляє матричне множення в PyTorch, а метод .t() повертає транспонований тензор. На рисунку 5.7 наведено математичне подання штучного нейрона.

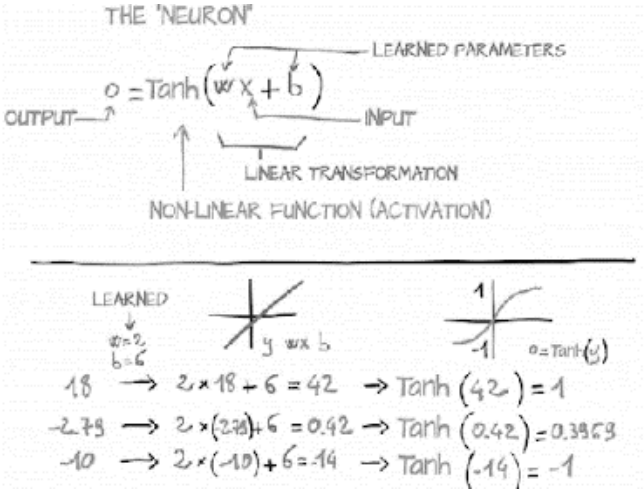



Рисунок 5.7 – Математична модель нейрона

Матриця, отримана при передачі вхідних даних у модель, є набором прогнозів для цільових змінних.

```
preds = model(inputs)
print(preds)
```

Порівняємо прогнози нашої моделі з фактичними цілями.

```
print(targets)
```

Ви можете бачити, що існує велика різниця між прогнозами нашої моделі та фактичними значеннями цільових змінних. Очевидно, це тому, що ми ініціалізували нашу модель випадковими вагами та зміщеннями, і не можемо очікувати, що вона просто спрацює.

Функція втрати

Перш ніж вдосконалити модель, нам потрібен спосіб оцінити ефективність нашої моделі. Ми можемо порівняти прогнози моделі з фактичними цілями, розрахувавши середню помилка в квадраті (MSE).

```
# MSE loss
def mse(t1, t2):
    diff = t1 - t2
    return torch.sum(diff * diff) / diff.numel()
```

torch.sum повертає суму всіх елементів у тензорі, а метод .numel повертає кількість елементів у тензорі. Давайте обчислимо середню квадратичну помилку для поточних прогнозів нашої моделі.

```
# Compute loss
loss = mse(preds, targets)
```

*print(loss)*

Результат називається втратою, оскільки він вказує, наскільки погана модель при прогнозуванні цільових змінних. Чим нижче втрати, тим краще модель.

За допомогою PyTorch ми можемо автоматично обчислити градієнт або похідну втрат до ваг і зміщень, оскільки вони мають встановленим значення підрахувати градієнт:

*# Compute gradients*
*loss.backward()*

Градієнти зберігаються у властивості .grad відповідних тензорів.

*# Gradients for weights*
*print(w)*
*print(w.grad)*

Функція втрат – це квадратична функція наших ваг і зміщень, і наша мета – знайти набір ваг, де втрати найменші. Якщо ми побудуємо графік втрат з будь-яким окремим елементом ваги або зміщення, він буде мати вигляд, як показано на рисунку 5.8. Основне розуміння з розрахунку полягає в тому, що градієнт вказує на швидкість зміни втрати або нахил функції втрат.

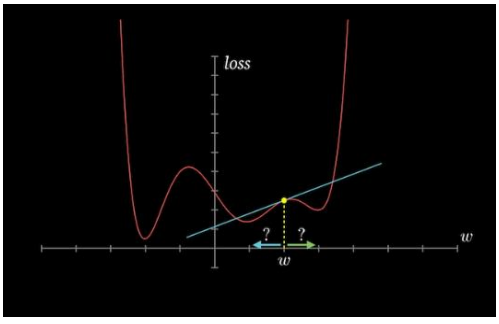



Рисунок 5.8 – Розрахунок градієнта функції (градієнт позитивний)

Якщо елемент градієнта позитивний: незначне збільшення значення елемента збільшить втрати; зменшення значення елемента трохи зменшить втрати (рисунок 5.8).

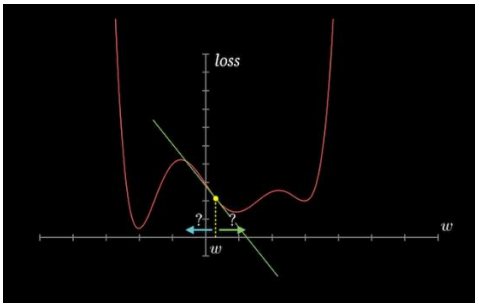



Рисунок 5.9 – Розрахунок градієнта функції (градієнт негативний)

Якщо елемент градієнта негативний: збільшення значення елемента трохи зменшить втрати; незначне зменшення значення елемента призведе до збільшення втрат (рисунок 5.9).

Збільшення або зменшення втрат при зміні вагового елемента пропорційно величині градієнта втрат – є основою для алгоритму оптимізації, який ми будемо використовувати для вдосконалення нашої моделі.

Перш ніж продовжувати, ми скидаємо градієнти до нуля, викликаючи метод .zero_ (). Нам потрібно це зробити, оскільки PyTorch накопичує, градієнти, тобто наступного разу, коли ми викличемо .backward, нові значення градієнта будуть додані до існуючих значень градієнта, що може призвести до несподіваних результатів.

*w.grad.zero_()*
*b.grad.zero_()*

Відрегулюємо ваги та зміщення за допомогою градієнтного спуску. Зменшимо втрати та вдосконалимо нашу модель, використовуючи алгоритм оптимізації градієнтного спуску, який має такі кроки:

1) Створює прогнози

2) Розраховує втрати

3) Обчислює градієнти з вагами та зміщеннями

4) Регулює ваги, віднімаючи невелику кількість, пропорційну градієнту

5) Скидає градієнти до нуля

```
# Generate predictions
preds = model(inputs)
print(preds)

# Calculate the loss
loss = mse(preds, targets)
print(loss)

# Compute gradients
loss.backward()
print(w.grad)
print(b.grad)

# Adjust weights & reset gradients
with torch.no_grad():
    w -= w.grad * 1e-5
    b -= b.grad * 1e-5
    w.grad.zero_()
    b.grad.zero_()
```

Ми використовуємо torch.no_grad, щоб вказати PyTorch, що не слід відстежувати, обчислювати чи змінювати градієнти під час оновлення ваг та зміщень.

Ми множимо градієнти на дійсно невелике число ($10^{-5}$ в цьому випадку), щоб переконатися, що ми не змінюємо ваги на дійсно велику суму, оскільки ми хочемо лише зробити невеликий крок у напрямку спуску градієнту. Це число називається швидкістю навчання алгоритму.

Після оновлення ваг ми скидаємо градієнти назад до нуля, щоб уникнути будь-яких майбутніх обчислень.

З новими вагами та зміщеннями модель повинна мати менші втрати:

```
# Calculate loss
preds = model(inputs)
loss = mse(preds, targets)
print(loss)
```

Ми вже досягли значного зменшення втрат, просто злегка скоригувавши ваги та зміщення за допомогою градієнтного спуску.

## Практична частина

1. Натренуйте модель лінійної регресії надану в теоретичному описі (для аналізу врожайності яблук та апельсинів) деяку кількість епох. Подивіться як змінюються ваги та похибка (втрати). Знайдіть оптимальну кількість епох для тренування. Порахуйте MSE та зробіть висновки щодо оптимальної кількості епох та отриманих результатів.

## Висновки

У висновках пояснити базовий принцип роботи градієнтного спуску. Навести результати аналізу роботи моделі лінійної регресії.

## Контрольні запитання

1. Що являє собою градієнтний спуск?
2. Як змінюються ваги при роботі градієнтного спуску?

# 6: НЕЙРОННА МЕРЕЖА ДЛЯ ЗАДАЧІ РЕГРЕСІЇ

**Мета роботи** – засвоїти принцип роботи та способи побудови нейронної мережі для задачі регресії.

## Теоретичні відомості

У цьому практикумі необхідно розробити дуже просту нейронну мережу. Це буде нейронна мережа, яка вирішує завдання регресії. Завдання регресії – це передбачення деякого дійсного числа. У якості навчального завдання необхідно передбачити поведінку функції sin(y).

### Імпортування бібліотек

Отже спочатку імпортуємо необхідні бібліотеки. Нам вистачить імпортувати бібліотеку torch та бібліотеку для рисування графіків matplotlib (окремо імпортуємо функцію pyplot яку далі будемо називати plt, та встановимо розмір рисунку):

```
import torch
import matplotlib.pyplot as plt
import matplotlib
matplotlib.rcParams['figure.figsize'] = (13.0, 5.0)
```

### Підготовка даних

Перше, що нам необхідно зробити – це скласти тренувальний датасет ("train dataset"). Для цього візьмемо 100 точок з рівномірного розподілу від нуля до одиниці, кожну точку помножимо на 20, віднімемо від неї 10, щоб графік був приблизно по центру – це будуть наші "X" – вхідні значення для тренування, які ми будемо подавати до функції синус. А "y" (наше цільове значення) – це будуть синуси від даних точок. Побудуємо нашу функцію. Вона досить проста – це, як і очікувалось, графік синуса. Ось так це можна зробити (x_train.numpy() – переводить torch тензор до numpy масиву):

```
x_train = torch.rand(100)
x_train = x_train * 20.0 - 10.0
y_train = torch.sin(x_train)
plt.plot(x_train.numpy(), y_train.numpy(), 'o')
plt.title('$y = sin(x)$')
```

Далі ускладнимо завдання, додаємо трохи шуму. Шум буде з нормального (Гаусового) розподілу (розподіл ймовірностей випадкової величини, що характеризується густиною ймовірності де $\mu$ – це математичне сподівання, σ – дисперсія випадкової величини. Параметр σ також відомий, як стандартне відхилення. Розподіл із μ=0 та $\sigma^2$=1 називають стандартним нормальним розподілом). Цей шум додаємо до кожної точки попереднього графіка (кожну точку з шуму додаємо до кожної точки з попереднього графіка). Це можна зробити, використавши функцію rand() модуля torch наступним чином:

```
noise = torch.randn(y_train.shape) / 5.
print(x_train.numpy().shape)
plt.plot(x_train.numpy(), noise.numpy(), 'o')
plt.axis([-10, 10, -1, 1])
plt.title('Gaussian noise')
```

Це для побудови самого шуму (noise). Тепер додаємо його до нашого попереднього графіку:

```
y_train = y_train + noise
plt.plot(x_train.numpy(), y_train.numpy(), 'o')
plt.title('noisy sin(x)')
plt.xlabel('x_train')
```

*plt.ylabel('y_train')*

Узагальнимо трохи наші дані. Зараз у нас наш об`єкт це одне число (координата Х по якій ми хочемо передбачити координату У), але, в загальному випадку, це може бути багато чисел (багато ознак, тобто Х може бути багатовимірним вектором, а не рядками). Тому, давайте узагальнимо розмірності наших об’єктів. Нам потрібно наш вектор Х (який зараз є строчкою, тобто має розмір (100, ) – 1 строка із 100 значеннями), перетворити у стовпець, у якого в кожному рядку буде одне число Х (тобто буде мати розмір (100, 1) – 1 стовпець і 100 строчок).

Це робить метод unsqueeze_() – повертає новий тензор із розмірністю розміру один, вставлений у вказане положення. Якщо ви в PyTorch бачите нижнє підкреслення в назві методу, це означає, що цей метод трансформує той об'єкт до якого він застосовується, тобто після його виконання зміняться самі об'єкти.

*x_train.unsqueeze_(1)*
*y_train.unsqueeze_(1)*
*print(x_train.shape)*
*print(y_train.shape)*

Після цього у нас X_train і Y_train змінилися, і тепер це стовпці розміром (100,1) – тобто двовимірний вектор. Ця навчальна вибірка вже більш складна і потребую більше зусиль для того, щоб виділити тут шум та складову синуса і наша нейронна мережа повинна буде зробити це. Таким чином ми зробили датасет для тренування (train dataset). Давайте зробимо датасет для перевірки роботи нашої нейронної мережі (validation dataset).

Зазвичай у задачах машинного навчання, ми ділимо наш dataset на тренувальні дані, і ті  дані, на яких ми будемо тестувати або валідувати наш

алгоритм, і мережа навчається на тренувальних даних і, відповідно, валідується (перевіряється) на тих даних які вона ніколи не бачила. А тут ми знаємо, що той закон природи, який згенерував наші дані – це функція синуса. Тому ми просто візьмемо в якості даних для перевірки просто функцію синуса, та не будемо до нього додавати ніякого шуму. Звичайно, це не дуже життєво, тому що у вас ніколи такого не буде – що ваші дані будуть не зашумлені, там завжди буде певний шум. Але в нашому тренувальному прикладі ми візьмемо для перевірки звичайний синус. Тобто створимо дві змінні "x_validation" і "y_validation". Подивимося, як виглядає графік цієї функції. Це звичайний синус на точках, які розподілені просто рівномірно, від мінус десяти до десяти (torch.linspace(-10, 10, 100)). Ми будемо їх передавати в сітку, відповідно, вони повинні бути правильної розмірності – це повинен бути двовимірний тензор, де кожен рядок відповідає одному елементу, одній точці. Робимо x_validation unsqueeze (1) і y_validation unsqueeze (1):

```
x_validation = torch.linspace(-10, 10, 100)
y_validation = torch.sin(x_validation.data)
plt.plot(x_validation.numpy(), y_validation.numpy(), 'o')
plt.title('sin(x)')
plt.xlabel('x_validation')
plt.ylabel('y_validation')

x_validation.unsqueeze_(1)
y_validation.unsqueeze_(1)
```

**Модель**

Тепер потрібно створити нейронну мережу.

Щоб створити нейронну мережу, нам потрібно створити клас, назвемо його "SineNet", припускаючи, що це буде нейронна мережа, яка вирішує завдання відновлення синуса.

Її ми повинні успадковувати від класу torch.nn.Module. Бібліотека torch.nn складається із функцій, модулів та класів, які використовуються при побудові нейронних мереж. torch.nn.Module – це базовий клас для всіх модулів нейронної мережі. Ваші моделі повинні бути підкласом цього класу (успадкованими). Модулі також можуть містити інші модулі, що дозволяє вкладати їх у структуру. Ви можете призначити підмодулі як звичайні атрибути. Ось таке спадкування внесе в наш об'єкт додаткові функції, які ми зараз же будемо використовувати:

*class SineNet(torch.nn.Module):*

...

...

Крім того, нам потрібно проініціалізувати ті шари, які будуть використовуватися в мережі. Тобто, конструктор класу (функція "__init__"), на вхід може приймати будь-що, будь-які параметри, які нам буде цікаво передати в цю мережу в момент конструювання. Нам, наприклад, цікаво передати кількість прихованих нейронів, які будуть зберігатися в кожному шарі, тобто ми припускаємо, що всі шари будуть однакового розміру із n_hidden_neurons прихованих нейронів. Відповідно ініціюємо батьківський об'єкт:

*(super(SineNet, self).__init__())*

**Довідка по методу super у Python.**

Досить часта ситуація в програмуванні, коли ми хочемо перевизначити якийсь метод батьківського класу. І все б добре, але іноді

потрібно зберегти поведінку батьків. Ось тут виникає проблема: "Копіювати код з батьківського класу або ж виносити в іншу функцію?" Припустимо, що у нас є наступний клас:

```
class ParentClass:
    def __init__(self, arg1, arg2):
        self.arg1 = arg1
        self.arg2 = arg2
```

Припустимо ми хочемо зробити його нащадка, у якого перші 2 аргументу конструктора такі ж, але ще є третій.

```
class ChildClass(ParentClass):
    def __init__(self, arg1, arg2, arg3):
        self.arg3 = arg3
```

Ми опрацювали новий аргумент arg3. Але що робити з arg1 і arg2? Можна просто скопіювати код з ParentClass, але що якщо ми там щось потім змінимо і ці аргументи треба буде обробляти по-іншому?

У Python є функція super, яка покликана вирішити цю проблему. У загальному вигляді, вона використовується так: super (subclass, obj), де obj – це об'єкт, а subclass – його клас. Повертає ця функція проміжний об'єкт, який є тим же об'єктом, але в якому все – властивості і методи, замінені на "батьківські". З тим пріоритетом, який вказаний при оголошенні класу (вище у того класу, який вказаний раніше).

Таким чином, ми можемо в перевизначені методу скористатися super, зберігаючи батьківське поводження, а потім дописати те, що додається. ChildClass з прикладу вище із застосуванням функції super буде мати вигляд:

```
class ChildClass(ParentClass):
    def __init__(self, arg1, arg2, arg3):
        super(ChildClass, self).__init__(arg1, arg2)
        self.arg3 = arg3
```

Таким чином, ми на нашому об'єкті ChildClass явно викликаємо батьківський конструктор, зберігаючи поведінку ParentClass, а потім доповнюємо тим функціоналом, який потрібно.

**Шари мережі**

1) перший шар, який буде називатися fc1, це "fully connected" шар – повністю пов'язаний шар. У PyTorch "fully connected" шар називається "linear(input_n, output_n)". Ми передаємо на вхід функції кількість вхідних нейронів (input_n) і кількість вихідних нейронів (output_n). Вхідних нейронів у нас буде один. Тобто це, насправді, не нейрон буде, а сам вхід в нейрон. Це одне число "x", координата нашої точки, за якою ми будемо щось прогнозувати. Якби у нас координата точки була якась багатовимірна (точка у багатовимірному просторі), то тут була би розмірність того простору, яке задає точку. Вихідних нейронів, у нас буде "n_hidden_neurons".

```
self.fc1 = torch.nn.Linear(1, n_hidden_neurons)
```

2) Після цього нам потрібна функція активації. Функцію активації візьмемо Tanh, або будь-яку іншу.

```
self.act1 = torch.nn.Tanh()
```

3) Крім того, додаємо ще один "fully connected" шар, але у нього буде всього один нейрон на виході та "n_hidden_neurons" з попереднього шару на вході. Цей нейрон буде відповіддю на питання. Так як у завданні регресії

нас цікавить відповідь – число, то на виході нашої мережі повинен бути один нейрон:

```
self.fc2 = torch.nn.Linear(n_hidden_neurons, 1)
```

У підсумку, наша нейрона мережа буде виглядати, як два шари, в одному з них буде кілька нейронів а в другому буде один. Тепер потрібно написати функцію forward, тобто те, як наші шари послідовно застосовуються. Спочатку ми застосовуємо шар "fc1", на "x". Те що вийшло ми передаємо в функцію активації, то що вийшло з функцій активації ми передаємо в "fc2", і ось це ж ми повертаємо. Тобто, в принципі, функція forward повторює нашу ініціалізацію. Давайте створимо таку мережу. Кількість прихованих нейронів – 50. У нас тепер є "SineNet" – об'єкт, який можна навчати:

```
class SineNet(torch.nn.Module):
    def __init__(self, n_hidden_neurons):
        super(SineNet, self).__init__()
        self.fc1 = torch.nn.Linear(1, n_hidden_neurons)
        self.act1 = torch.nn.Tanh()
        self.fc2 = torch.nn.Linear(n_hidden_neurons, 1)

    def forward(self, x):
        x = self.fc1(x)
        x = self.act1(x)
        x = self.fc2(x)
        return x

sine_net = SineNet(50)
print(sine_net)
```

**Передбачення**

Напишемо функцію "predict", всередині вона буде дуже проста – там буде виклик методу "forward". І якщо ви передаєте туди деяку змінну X, на виході у вас буде деякий prediction (y_pred = net.forward(x) – це те, що ми хочемо: одне число). І далі є певний код, який рисує цей prediction:

```
def predict(net, x, y):
    y_pred = net.forward(x)
    plt.plot(x.numpy(), y.numpy(), 'o', label='Groud truth')
    plt.plot(x.numpy(), y_pred.data.numpy(), 'o', c='r', label='Prediction')
    plt.legend(loc='upper left')
    plt.xlabel('$x$')
    plt.ylabel('$y$')
predict(sine_net, x_validation, y_validation)
```

На рисунку 6.1 приведений варіант, який може бути на виході розробленої мережі.

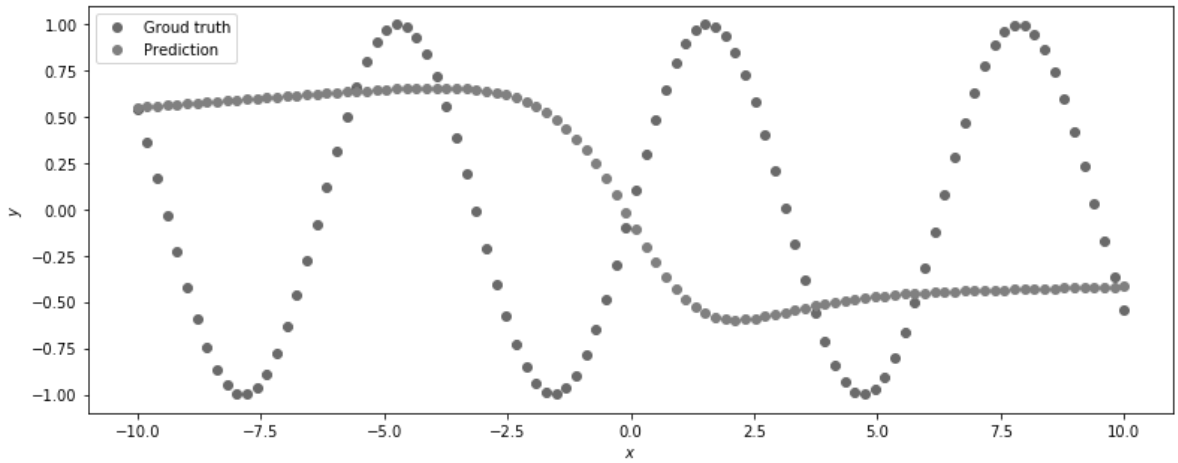



Рисунок 6.1 – Приклад передбачень нейронної мережі

На рисунку 6.1 зображено два графіка, один позначений groud truth, (те, що ми б хотіли побачити на валідації), X – це те, що ми передаємо в мережу а Y – це те що ми б хотіли щоб мережа повернула, а червоними крапками

позначено те, що мережа нам передбачила. Неважко здогадатися, що, так як у нас мережа була ініційована випадковими числами, то на виході у нас вийшла деяка випадкова крива (вона може бути різна в залежності від запуску).

Тепер навчимо цю нейромережу. Щоб навчити нейромережу, нам потрібно кілька речей, додатково.

**Оптимізатор**

По-перше нам потрібен деякий оптимізатор – деякий об'єкт, який буде здійснювати для нас кроки градієнтного спуску. Візьмемо для прикладу та простоти torch.optim.adam (torch.optim – це пакет, що реалізує різні алгоритми оптимізації. Найчастіше використовувані методи вже підтримуються та реалізовані у torch, а інтерфейс є загальним, щоб в майбутньому можна було легко інтегрувати і більш складні методи):

*optimizer = torch.optim.Adam(sine_net.parameters(), lr=0.01)*

На вхід Adam передаються ті параметри, які ми хочемо модифікувати – ті параметри, які ми хочемо навчати в нейронної мережі. Ми можемо вплинути на ваги нейронної мережі – ті ваги, які зберігаються в нейронах (ці ваги знаходяться в sine_net.parameters()). Це одна з тих причин, чому нейромережу ми наслідували, а не створювали як клас, з нуля. Відповідно, якщо ми передаємо в Adam такий об'єкт, то Adam зрозуміє, що тут лежать всі ті змінні, які він може модифікувати внаслідок градієнтного спуску. І ще необхідно передати "learning rate" – крок градієнтного спуску.

**Функція втрат**

Нам потрібна ще функція втрат (Loss Function, або просто Loss) – це та функція, яка говорить, на скільки неправильно ми передбачили, на скільки ми помилилися. Це та функція, по якій ми будемо рахувати градієнт і яка буде брати участь у градієнтному спуску. Припустимо, що функція втрат у нас MSE (середня квадратична помилка)

*squares = (pred - target) ** 2*

Ми беремо передбачення нейронної мережі pred, target – тобто реальне значення яке відповідає даній точці, зводимо це в квадрат та рахуємо середнє значення суми квадратів.

```
def loss(pred, target):
    squares = (pred - target) ** 2
    return squares.mean()
```

**Процедура навчання**

Одна епоха навчання це:

1) Якщо ми візьмемо весь наш набір даних, пропустимо його через нейромережу то отримаємо деякі передбачення.
2) Після цього, на цих передбаченнях рахуємо функцію втрат, яку ми тільки що задали.
3) Після цього у цій функції втрат підрахуємо похідну та зробимо шаг градієнтного спуску.

Тобто за 1 епоху ми подивилися на всі наші дані та зробили градієнтний шаг. Ми можемо зробити градієнтні кроки кілька разів, переглянути всі дані. Всередині однієї ітерації, всередині однієї епохи, ми спочатку обнуляємо градієнти.

*optimizer.zero_grad()*

Кожна епоха починається з того, що в нас в optimizer обнуляються градієнти. Після цього ми рахуємо forward, тобто ми беремо наш весь X_train і передаємо його в функцію forward, розраховуємо prediction (розраховуємо передбачення нашої нейромережі), після цього ми розраховуємо функцію втрат, отримуємо деяке число: це скаляр, за яким ми

можемо зробити backward. Робимо з цього скаляру backward, тобто це певний тензор, який залежить від параметрів мережі, тобто від ваг нейромережі, який обгорнутий в optimizer, і, відповідно, коли ми робимо loss_val.backward, optimizer розуміє, що там зважили градієнти, і значить optimizer може зробити крок.

```
for epoch_index in range(2000):
    optimizer.zero_grad()

    y_pred = sine_net.forward(x_train)
    loss_val = loss(y_pred, y_train)

    loss_val.backward()

    optimizer.step()

predict(sine_net, x_validation, y_validation)
```

**Практична частина**

1. **Проекспериментуйте із нейронною мережею наведеною у теоретичній частині, а саме:**
   a) Спробуйте змінити кількість нейронів (1,2,3,4,5,6,7,8,9,10 – проаналізуйте як змінюються передбачення)
   b) Спробуйте змінити кількість епох навчання
   c) Спробуйте змінити функцію активації
   d) Спробуйте змінити (зменшити або додати додаткові шари)
   e) Спробуйте змінити функцію втрат
   f) Запишіть значення функції втрат після кожної ітерації та виведіть графік залежності функції втрат від номера епохи.
   g) Поясніть отримані результати.

2. **Багатовимірна регресія**

Спробуйте вирішити задачу регресії на прикладі набору даних Енергоефективність з великої бібліотеки UCI (http://archive.ics.uci.edu/ml/datasets/Energy+efficiency). Це набір даних у якому описані наступні атрибути:

| Поле | Опис | Тип |
|---|---|---|
| **X1** | Відносна компактність | FLOAT |
| **X2** | Площа | FLOAT |
| **X3** | Площа стіни | FLOAT |
| **X4** | Площа стелі | FLOAT |
| **X5** | Загальна висота | FLOAT |
| **X6** | Орієнтація | INT |
| **X7** | Площа скління | FLOAT |
| **X8** | Розподілена площа скління | INT |
| **y1** | Навантаження при обігріві | FLOAT |
| **y2** | Навантаження при охолодженні | FLOAT |

Маємо x1…x8 – характеристики приміщення на підставі яких буде проводитися аналіз, а y1 та y2 – значення навантаження, які треба спрогнозувати.

При вирішенні завдання спробуйте вибрати чи всі ознаки Вам потрібні (якщо будете використовувати pandas – метод .corr()), архітектуру, кількість

епох, спосіб ділення даних (зазвичай 60 (тренування) / 20(валідація) / 20(тест)), тощо.

3. **Проаналізуйте аналогічно до другого завдання набір даних «Діабет»** (https://www4.stat.ncsu.edu/~boos/var.select/diabetes.html), який можна завантажити із бібліотеки sklearn:

```
from sklearn import datasets
# Load the diabetes dataset
diabetes_X, diabetes_y = datasets.load_diabetes(return_X_y=True)
```

Набір даних про діабет складається з 10 фізіологічних змінних (вік, стать, вага, артеріальний тиск, тощо) виміряних на 442 пацієнтах та вказує на прогресування захворювання через рік (колонка таргету). Дані вже нормовані (мають середнє значення 0 та евклідову норму 1). Проаналізуйте дані, які поля та ознаки в нас є, кореляція між ознаками та таргетом, тощо. Натренуйте модель та отримайте передбачення.

Можна використовувати ненормовані оригінальні дані: https://www4.stat.ncsu.edu/~boos/var.select/diabetes.tab.txt - але тоді реалізуйте нормалізацію самостійно.

## Висновки

У висновках поясніть отримані результати експериментів із нейронною мережею із теоретичної частини. Зробіть висновок щодо оптимальних мереж, які ви отримали для вирішення другого завдання та третього завдання.

## Контрольні запитання

1. Які основні компоненти повинна містити програма для побудови нейронної мережі написана у PyTorch?
2. Які доступні вбудовані оптимізатори реалізовані у PyTorch?

# 7: НЕЙРОНА МЕРЕЖА ДЛЯ ЗАДАЧІ КЛАСИФІКАЦІЇ

**Мета роботи** – засвоїти принцип роботи та способи побудови нейронної мережі для задачі класифікації.

## Теоретичні відомості

У даній практичній роботі реалізуємо нейромережу для завдання класифікації.

Імпортуємо необхідні бібліотеки: torch та random і numpy

```
import torch
import random
import numpy as np
```

Тому, перш ніж приступати до реалізації, давайте поговоримо про випадкові числа, які виникають в результаті роботи нейронної мережі, і як вони відтворюються.

Якщо ми візьмемо один і той же код і запустимо його кілька разів, то ми отримаємо різні результати. Це відбувається тому, що кожен раз у нас ініціалізуються заново ваги, відбувається по-різному навчання нейронної мережі, і це все залежить від того, як відпрацював генератор випадкових чисел в нашому процесорі. Можливо, ми хочемо щоб експерименти були відтворені: щоб, взявши один і той же python файл і виконавши його, ми отримали б той же самий результат, як і раніше. Наприклад, це потрібно для того щоб розуміти: а чи правда ті зміни, які ми робимо з нейромережею, покращують наші результати, або це результат певної випадковості. Давайте зафіксуємо ось цю випадковість, яка з експерименту в експеримент нас переслідує.

Є таке поняття як random seed, це можна інтерпретувати, як номер послідовності випадкових чисел, яку видає нам випадковий генератор, якщо його попросити видати нам послідовність. Випадкових генераторів є кілька: є випадковий генератор модуля random в Python, є випадковий генератор модуля numpy.random (вже бібліотеки numpy), є також випадкові генератори бібліотеки PyTorch, і інші. Давайте зафіксуємо їх всі. Для цього візьмемо random seed і поставимо його в конкретне значення.

*random.seed(0)*
*np.random.seed(0)*
*torch.manual_seed(0)*
*torch.cuda.manual_seed(0)*
*torch.backends.cudnn.deterministic = True*

Тобто, ми завжди будемо використовувати нульову послідовність при виклику випадкового генератора бібліотеки random (це бібліотека мови python). Також нам потрібно зафіксувати сиди в numpy і в PyTorch, випадкові сиди, які відповідають за обрахування і CPU і на GPU – вони різні, відповідно потрібно ще зафіксувати випадковий seed підмодуля CUDA. І якщо ви розраховуєте на відеокарті, ви використовуєте бібліотеку cudnn, і вона може виконуватися в детерміністичному режимі, а може в недетерміністичному. Недетерміністичний режим набагато швидший, але якщо ми хочемо встановити детерміністичний режим, щоб була відтворюваність, то нам потрібно виставити цей параметр в "True". Виставивши всі ці параметри, ми можемо практично гарантувати, що експерименти будуть відтворені. Якщо це не так то швидше за все потрібно шукати помилку в коді.

**Набір даних**

Для прикладу будемо використовувати датасет класифікації вин. Він є і в бібліотеці sklearn Ми можемо його завантажити через

*sklearn.datasets.load_wine()*

Цей датасет містить 178 різних пляшок вин, у кожної пляшки виміряно 13 параметрів, це дійсні числа, і є три класи, на які можна класифікувати конкретну пляшку.

Нам потрібно цей датасет розбити на дві частини: частина для тренування (train part), на якій ми будемо навчатися, і на тестову (test part, можливо додати ще validation part але кількість даних в нас замала), на якій ми будемо рахувати метрики. Скористаємося функцією train_test_split з бібліотеки scikit_learn. Якщо ми передаємо в функцію train_test_split першим параметром, власне, dataset (тут ми використовуємо тільки перші дві колонки (wine.data[:, :2]), загалом колонок 13, стільки ж, скільки у нас параметрів вина, ми використовуємо лише дві для зручності подальшої візуалізації), другим параметром ми передаємо таргети (wine.target), то є ті класи, які нам потрібно передбачити – це буде номер класу, і ми скажемо що ми заберемо 30 відсотків в тест (test_size=0.3), і перед тим, як цей датасет ділити на дві частини, нам потрібно його перемішати, щоб упевнитися, що якщо він був впорядкований, наприклад за номером класу, то тепер це сортування не працює (shuffle=True).

```
import sklearn.datasets
wine = sklearn.datasets.load_wine()
wine.data.shape
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    wine.data[:, :2],
    wine.target,
    test_size=0.3,
    shuffle=True)
```

Після цього ми всі ці "фолди": X_train, X_test, Y_train і Y_test переведемо до torch тензорів (якщо число з плаваючою комою – обернемо у float тензор, якщо ні (наш таргет) – обернемо у long тензор).

```
X_train = torch.FloatTensor(X_train)
X_test = torch.FloatTensor(X_test)
y_train = torch.LongTensor(y_train)
y_test = torch.LongTensor(y_test)
```

**Модель**

Реалізуємо клас WineNet, це буде наша нейромережа для класифікації. Спадкуємося від torch.nn.Module, у функціях __init__ (в конструкторі) буде: кількість прихованих нейронів, n_hidden_neurons. У цей раз спробуємо два прихованих шари, тобто наша нейромережа буде складатися з трьох шарів і два з них будуть прихованими.

1) Перший шар – це fully connected шар, з двох входів (у нас дві колонки для кожної пляшки вина), на виході N прихованих нейронів;
2) Далі активація: сигмоїда, можна поставити будь-яку іншу, якщо хочете.
3) Після цього – прихований шар, який складається з N нейронів, перетворює їх теж в N нейронів;
4) Знову сигмоїдна активація.
5) Після цього знову fully connected шар, який видає нам три нейрона, кожен нейрон буде відповідати за свій клас. Тобто на виході цих трьох нейронів будуть деякі числа, які після цього ми передаємо в софтмакс, і отримаємо ймовірності класів.
6) Софтмакс далі ініціалізується.

Функція "forward" – вона буде реалізовувати граф нашої нейронної мережі. Ми передаємо двомірний тензор з двома колонами в перший fully connected шар, після цього в першу активацію, в другій fully connected шар, в другу активацію, в третій fully connected шар, у якого три виходи. І зауважте, ми тут не використовували softmax.

Чому ми так зробили? Чому ми не прогнали вихід з третього шару через softmax? Справа в тому, що після того, як ми порахуємо виходи нейронної мережі, ми хочемо прогнати їх через softmax і порахувати функцію втрат крос-ентропію. Але якщо ви пам'ятаєте формулу крос-ентропії, там є логарифм, тобто виходи нейронної мережі пропускаються через логарифм. А у softmax теж беруть участь експоненти. Так ось, ці експоненти і логарифми взаємно знищуються, і виходить що нам не потрібно обчислювати експоненти, щоб порахувати крос-ентропію. Ми можемо її порахувати навіть беручи до уваги softmax. Відповідно, якщо ми хочемо просто розрахувати функцію втрат, нам softmax не потрібен. Якщо ми хочемо порахувати ймовірності, то нам доведеться використовувати softmax.

$$\text{loss}(x, class) = -\log\left(\frac{\exp(x[class])}{\sum_j \exp(x[j])}\right) = -x[class] + \log\left(\sum_j \exp(x[j])\right)$$

Softmax – взагалі, функція досить довгого обчислення, тому ми намагаємося її уникати. Для того, щоб розрахувати ймовірності, напишемо також окрему функцію inference, яка буде викликати функцію "forward", і пропускати її через softmax. Ініціалізуємо нашу нейромережу з кількістю прихованих нейронів = 5. Нехай вона буде мати назву wine_net.

```
class WineNet(torch.nn.Module):
    def __init__(self, n_hidden_neurons):
        super(WineNet, self).__init__()
```

```
        self.fc1 = torch.nn.Linear(2, n_hidden_neurons)
        self.activ1 = torch.nn.Sigmoid()
        self.fc2 = torch.nn.Linear(n_hidden_neurons, n_hidden_neurons)
        self.activ2 = torch.nn.Sigmoid()
        self.fc3 = torch.nn.Linear(n_hidden_neurons, 3)
        self.sm = torch.nn.Softmax(dim=1)

    def forward(self, x):
        x = self.fc1(x)
        x = self.activ1(x)
        x = self.fc2(x)
        x = self.activ2(x)
        x = self.fc3(x)
        return x

    def inference(self, x):
        x = self.forward(x)
        x = self.sm(x)
        return x

wine_net = WineNet(5)
```

Нам залишилося тільки задати функцію втрат – бінарну крос-ентропію – torch.nn.CrossEntropyLoss, яка використовує не виходи після софтмаксу, а виходи нейронної мережі, що не пропущені ще через софтмакс. Нам потрібен optimizer – той метод, який буде використовуватися для обчислення градієнтних кроків. У optimizer ми передаємо всі параметри нейронної мережі. Параметри нейронної мережі – це ваги. Це ті приховані значення, які перебувають в наших нейронах, які ми хочемо підбирати:

*loss = torch.nn.CrossEntropyLoss()*

*optimizer = torch.optim.Adam(wine_net.parameters(), lr=1.0e-3)*

**Батч (пакет)**

Минулого разу ми брали весь наш датасет, рахували loss-функцію, далі робили градієнтний крок, і повторювали цей процес багаторазово. Але в реальному житті навряд чи поміститься в пам'яті весь датасет. Тобто, навчання в реальному житті відбувається по частинах даних – вони називаються Батч (batch, або пакет). Ми повинні відрізати деякий шматочок наших даних, порахувати по ньому втрати, порахувати по ньому градієнтний крок, зробити градієнтний крок, взяти наступний шматочок, і так далі. Відповідно, одна епоха, тобто ітерація перегляду всього датасету, б'ється на багато маленьких частин.

Нам знадобиться функція numpy.random.permutation – вона дає випадково перемішані значення. Якщо ми сюди підставимо розмір нашого тренувального датасета, то отримаємо деякі індекси, у випадковому порядку. Якщо ми від нашого датасета візьмемо ці індекси, то отримаємо "перемішаний" (shuffle – перемішувати) датасет. Кожну епоху ми будемо перемішувати датасет, і потім різати його на частини. Скажімо, що ці частини будуть розміром десять елементів (batch_size). Можна взяти будь-яке інше значення. Отже, кожну епоху ми будемо перемішувати наш датасет, у нас є змінна "order", яка визначається кожну епоху, яка визначає порядок індексів, який потрібно застосувати до датасета. З нього ми будемо вирізати ділянки довжиною batch_size. Тобто, кожну епоху ми будемо робити перемішування нашого датасета, визначати змінну order, яка відповідає за порядок елементів. А після цього ми з цього порядку (batch_indexes = order[start_index:start_index+batch_size]) будемо обчислювати деяку підмножину, починаючи з start_index, який буде 0, 10, 20 і так далі, до кінця Батч. Тобто, якщо start index "0", то кінець буде 0 плюс 10, значить 10-й індекс в кінець не включається. Виходить, що batch_indexes

– це деякі індекси, які відповідають поточному Батчу. Таким чином ми кожну епоху гарантовано проходимо всі значення в датасеті, при цьому кожна ітерація навчання відбувається по десяти елементах. Ці batch_indexes, ми візьмемо з X_train і y_train, щоб отримати і дані, і відповіді. Зробимо forward, тобто пропустимо весь Батч через нейромережу, отримаємо деякі prediction. Це без софтмаксу, тобто це виходи на останніх трьох нейронах. Після цього порахуємо loss на виходах нейронної мережі і реальних значеннях, і порахуємо backward, тобто у результату виконання loss функції, ми порахуємо похідну. А результат цієї похідної, тобто градієнти, які вийшли, підуть у optimizer, тому що він обертає всі ваги нейронної мережі. Далі optimizer може зробити step, тобто крок градієнтного спуску.

```
batch_size = 10

for epoch in range(5000):
    order = np.random.permutation(len(X_train))
    for start_index in range(0, len(X_train), batch_size):
        optimizer.zero_grad()
        batch_indexes = order[start_index:start_index+batch_size]
        x_batch = X_train[batch_indexes]
        y_batch = y_train[batch_indexes]
        preds = wine_net.forward(x_batch)
        loss_value = loss(preds, y_batch)
        loss_value.backward()
        optimizer.step()

    if epoch % 100 == 0:
        test_preds = wine_net.forward(X_test)
        test_preds = test_preds.argmax(dim=1)
        print((test_preds == y_test).float().mean())
```

Крім того, кожні 100 епох ми будемо обчислювати метрики на тестовому датасеті щоб подивитися, навчається у нас нейромережа, чи ні. А саме, кожні 100 епох ми робимо forward за тестовими даними, отримуємо тестові prediction, і обчислюємо, який вихід був максимальний. Знову ж щоб зрозуміти, який клас передбачає нейромережа, не обов'язково обчислювати софтмакс, не обов'язково обчислювати ймовірності. Нам досить подивитися: а який вихід був найбільший, і він же і буде згодом виходом з максимальною ймовірністю. Отже нам потрібно порахувати argmax виходів нейронної мережі, це буде номер нейрона, і порівняти його з тим номером класу, який знаходиться в Y_test. Після цього ми хочемо порахувати: а яка частка цього збігу, коли у нас нейрон з максимальним виходом збігся з реально правильним класом. Нам потрібно порахувати середнє значення, але середнє значення ми не можемо порахувати у цілочисельному тензорі, який виходить в результаті цього порівняння, тому ми спочатку його перетворимо до тензора з плаваючою комою і викличемо у нього метод mean().

Код для візуалізації результату може мати наступний вигляд:

```
import matplotlib.pyplot as plt
%matplotlib inline

plt.rcParams['figure.figsize'] = (10, 8)

n_classes = 3
plot_colors = ['g', 'orange', 'black']
plot_step = 0.02

x_min, x_max = X_train[:, 0].min() - 1, X_train[:, 0].max() + 1
y_min, y_max = X_train[:, 1].min() - 1, X_train[:, 1].max() + 1
```

```
xx, yy =  torch.meshgrid(torch.arange(x_min, x_max, plot_step),
                         torch.arange(y_min, y_max, plot_step))

preds = wine_net.inference(
    torch.cat([xx.reshape(-1, 1), yy.reshape(-1, 1)], dim=1))

preds_class = preds.data.numpy().argmax(axis=1)
preds_class = preds_class.reshape(xx.shape)
plt.contourf(xx, yy, preds_class, cmap='Accent')

for i, color in zip(range(n_classes), plot_colors):
    indexes = np.where(y_train == i)
    plt.scatter(X_train[indexes, 0],
                X_train[indexes, 1],
                c=color,
                label=wine.target_names[i],
                cmap='Accent')
    plt.xlabel(wine.feature_names[0])
    plt.ylabel(wine.feature_names[1])
    plt.legend()
```

На рисунку 7.1 наведена візуалізація рішення використовуючи наведений вище код.

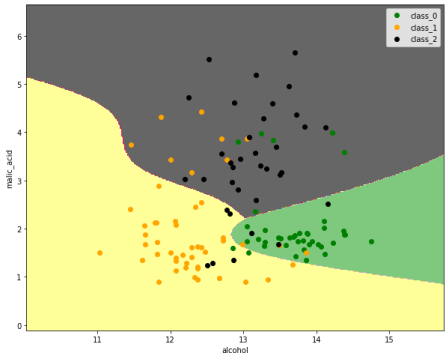



Рисунок 7.1 – Приклад візуалізації рішення

На рисунку 7.1 точками позначено тренувальний датасет, тобто ті точки, на яких навчалася нейронна мережа, а заповненими областями – то, як би нейромережа класифікувала точки у відповідних значеннях. Ми бачимо, що нейромережа досить непогано справляється з тим, щоб відокремлювати точки різних класів.

## Практична частина

1. **Спробувати поліпшити результат нейронної мережі, що надана в теоретичній частині, а саме**
   a) Оптимізуйте архітектуру нейронної мережі.
   b) Спробуйте змінити кількість прихованих шарів
   c) Змінити оптимізатор.
   d) Змінити Learning rate.
   e) Змінити функцію активації.
   f) Спробуйте використати весь набор ознак (зараз ми використовували лише 2 ознаки для простої візуалізації)
   g) Спробуйте інші поліпшення/зміни.

2. **Проаналізуйте тренувальний датасет Breast Cancer Wisconsin** (https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)) та розробіть модель класифікації (класифікація пухлини – колонка таргету – Diagnosis (M = malignant(злоякісна), B = benign (доброякісна))). Цей датасет також можна знайти в бібліотеці sklearn:

   *from sklearn.datasets import load_breast_cancer*
   *data = load_breast_cancer()*
   *X, y = data.data, data.target*

## Висновки

У висновках обґрунтувати обрану найкращу конфігурацію нейронної мережі. Проаналізуйте дані, подивіться кореляції, видаліть сильно корелюючи між собою ознаки (із коефіцієнтом кореляції Пірсона більше ніж 0.9), натренуйте модель, зробіть відповідні висновки.

## Контрольні запитання

1. У чому суть задачі класифікації?
2. Чим принципово відрізняється нейронна мережа для задачі регресії та класифікації?

# 8: КЛАСИФІКАЦІЯ РУКОПИСНИХ ЧИСЕЛ ПОВНОЗВ'ЯЗНОЮ МЕРЕЖЕЮ. МЕТОДИ ОПТИМІЗАЦІЇ

**Мета роботи** – засвоїти принцип побудови повнозв'язної нейронної мережі для класифікації зображень.

## Теоретичні відомості

У цій практичній роботі необхідно навчитися класифікувати зображення. Ми зробимо це за допомогою вже відомої нам повнозв'язної нейронної мережі. І крім того ми навчимося прискорювати обчислення нейронної мережі, перекладаючи їх на GPU (за наявності). Почнемо з того, що як і минулого разу, зробимо ініціалізацію random seed та завантажимо необхідні бібліотеки.

```
import torch
import random
import numpy as np

random.seed(0)
np.random.seed(0)
torch.manual_seed(0)
torch.cuda.manual_seed(0)
torch.backends.cudnn.deterministic = True
```

### Набір даних

Крім того, нам знадобиться датасет. Найпростіший (класичний та найпоширеніший з навчальної точки зору) датасет для класифікації зображень – це набір даних MNIST, він містить в собі рукописні цифри від 0 до 9, і його можна завантажити за допомогою бібліотеки

torchvision.datasets (він вже окремо має train і test), і отримаємо відповідно mnist_train, mnist_test:

```
import torchvision.datasets
MNIST_train = torchvision.datasets.MNIST('./', download=True, train=True)
MNIST_test = torchvision.datasets.MNIST('./', download=True, train=False)
X_train = MNIST_train.train_data
y_train = MNIST_train.train_labels
X_test = MNIST_test.test_data
y_test = MNIST_test.test_labels
```

Синтаксис завантаження даних може трохи відрізнятися у різних версіях PyTorch: зауважте, наприклад, що у новій версії train_data та test_data – це просто data; train_labels та test_labels – це targets.

Подивимося, що ми маємо за дані і «лейбли» – спочатку подивимося які типи даних ми маємо:

```
print(X_train.dtype, y_train.dtype)
```

Бачимо що X_data, тобто самі картинки, мають тип dtype "unsigned int8", а ось лейбли мають тип "int64".

Для більшої точності під час розрахунків, краще, щоб дані були в числах з плаваючою комою. Відповідно, ми відразу перетворимо X_train і X_test у float.

```
X_train = X_train.float()
X_test = X_test.float()
```

Подивимося на розмірності датасетів, які ми скачали:

```
print(X_train.shape, X_test.shape)
print(y_train.shape, y_test.shape)
```

Бачимо, що X_train і X_test мають розмірності 60 тисяч зображень і 10 тисяч відповідно, і самі зображення розміром 28 на 28 – тобто, це дуже маленькі картинки, тому ми і можемо застосовувати повнозв'язну нейронну мережу для такого завдання. А лейбли y_train і y_test мають відповідний розмір і у них немає додаткової розмірності. Це одновимірний тензор, і тому ми позначили X_train з великої літери а y_train – з маленької, припускаючи, що X_train – це багатовимірний тензор, y_train – це одновимірний. Крім того, цікаво – як же ці картинки виглядають: можна імпортувати matplotlib – бібліотеку для малювання графіків, там є чудова функція imshow і вона може намалювати картинку:

```
import matplotlib.pyplot as plt
plt.imshow(X_train[0, :, :])
plt.show()
print(y_train[0])
```

Коли ми працювали з датасетом про вина, у нас кожна пляшка вина описувалася 13-ма ознаками, і це був деякий одновимірний тензор, який відповідав пляшці вина, і, відповідно, Батч був двовимірним тензором. А тут кожна картинка описується двовимірним тензором. Щоб це вирішити можна розтягнути цю картинку в один довгий вектор, і тоді кожен піксель знайде своє місце в цьому довгому векторі. Звичайно, загубиться деяка інформація про те, які пікселі були поруч які перебували далеко, але в принципі нам цього вистачить. Розтягнути наші картинки допоможе функція reshape:

```
X_train = X_train.reshape([-1, 28 * 28])
X_test = X_test.reshape([-1, 28 * 28])
```

Отже, у нас був тензор X_train, це тривимірний тензор, а ми хочемо двомірний тензор, де перша розмірність буде незмінною – 60 000 зображень (-1), а друга розмірність розтягнеться – було 28x28, а вийде одна розмірність: 784 пікселя. Якщо ми застосуємо цю функцію і до train, і до test, у нас вийдуть вже двомірні тензори X_train і X_test, і їх уже можна передавати в нашу нейронну мережу.

**Модель**

Давайте створимо таку нейронну мережу: вона буде дуже схожою на ту яку ми створювали для вин, вона також буде складатися з декількох повнозв'язних шарів. Тут буде їх два: fc1 – це (fully connected) шар, на вхід якого приходить 28 на 28 = 784 пікселя. Далі вони передаються в N hidden neurons, який ми можемо як завгодно ставити в залежності від того, скільки нам потрібно інформації в прихованому шарі. Після цього у нас буде сигмоїдна активація, щоб додати нелінійність, і після активації, результат буде передаватися в ще один повнозв'язний шар. На вході у нього N hidden neurons, а на виході 10 (тому, що у нас 10 класів): це цифри від 0 до 9. Тобто відбувається класифікація на 10 класів. Крім того потрібно написати функцію forward, яка пропускає тензор X через всі ці шари послідовно: fully connected, активація, fully connected другий, і видає результуючий тензор, який являє собою виходи з другого повнозв'язного шару розміром 10. Давайте створимо таку нейронну мережу, назвемо її MNISTnet, скажімо що у неї всередині 100 прихованих нейронів:

```
class MNISTNet(torch.nn.Module):
    def __init__(self, n_hidden_neurons):
        super(MNISTNet, self).__init__()
        self.fc1 = torch.nn.Linear(28 * 28, n_hidden_neurons)
```

```
        self.ac1 = torch.nn.Sigmoid()
        self.fc2 = torch.nn.Linear(n_hidden_neurons, 10)

    def forward(self, x):
        x = self.fc1(x)
        x = self.ac1(x)
        x = self.fc2(x)
        return x

mnist_net = MNISTNet(100)
```

Крім того нам знадобиться loss, як і минулого разу буде крос-ентропія, тому що це loss, який використовується в класифікації. Потрібно підкреслити, що функція CrossEntropyLoss на вхід приймає не вірогідність, а ті виходи, які були ще до софтмаксу, тобто функція forward, яку ми написали, не містила софтмакс, тому що ми хочемо трошки прискорити наші обчислення, уникнувши софтмаксу. Для прискорення обчислень можна софтмакс і крос-ентропію з'єднати однією функцією, і тоді будуть трошки швидше і стабільніше виконуватися обчислення. Ще нам потрібен оптимізатор, тобто деякий метод градієнтного спуску, нехай це буде – Adam, learning rate = 1e-3. На вхід оптимізатору передаються, як і в минулий раз, всі параметри нейронної мережі. Тобто, оптимізуються параметри нейронної мережі – це її ваги, тобто ті значення, які зберігаються в нейронах. Тому ми передаємо саме їх всередину оптимізатора. Якщо у Вас є відеокарта та ви хочете використовувати її під час розрахунків (для пришвидшення) необхідно буде додати декілька строчок коду. В коді нижче, спитаємо тільки чи є у Вас відеокарта print(torch.cuda.is_available()) – якщо на виході буде True то відеокарта є і доступна для використання. Якщо ви використовуєте Jupyter Notebook, то Ви можете перевірити

інформацію про доступну відеокарту і доступні ресурси на ній, процеси на ній, тощо, виконавши консольну команду !nvidia-smi:

```
print(torch.cuda.is_available())
#!nvidia-smi

loss = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(mnist_net.parameters(), lr=1.0e-3)
```

Якщо відеокарта у Вас є, та ви хочете її використовувати, то нам потрібно перекласти обчислення на відео карту. Якщо карти нема, можна перейти до пункту «навчання моделі».

**Що саме перекласти на відеокарту?**

По-перше це ваги нейронної мережі – це те, що буде оптимізуватися, це те, що бере участь в обчисленнях, це обов'язково має бути на відео карті.

По-друге – це входи – тобто ті картинки, які ми передаємо в нейромережу.

Давайте почнемо з нейронної мережі. По-перше нам потрібно створити змінну "device", яка буде або рядком "cuda: 0", що відповідає нульовій відео-картці (якщо у вас їх раптом багато), або "cpu", якщо відео-карти немає. Тобто, зараз ми створюємо крос-платформний код, який буде працювати і на CPU, і на GPU, в залежності від того, що видає цей рядок: *torch.cuda_is_available*. Якщо буде True, то, відповідно, device перетвориться в "cuda: 0", а якщо у вас "cuda_is_available" – це False, значить буде CPU, як зазвичай. Тепер нам потрібно на цей device перекласти нейронну мережу. Це робиться дуже просто – ми можемо сказати "mnist_net.to (device)", як зі звичайним тензором.

Як переконатися, чи переклали ми нашу нейронну мережу на GPU? Це можна зробити, якщо ми візьмемо її параметри, які ми зазвичай

передаємо в оптимізатор, і просто виведемо їх на екран print(list(mnist_net.parameters())).

Якщо ми це зробимо, ми побачимо деякий список в якому елементи – це параметри нейронної мережі. І якщо ми бачимо, що скрізь прописаний *device cuda: 0* – це означає, що всі ці параметри тепер лежать на відео-карті.

Крім того, нам потрібно перекласти на відео карту вхідні дані. Ми можемо взяти, і весь наш тестовий датасет X_test відразу перекласти на device, тому що ми тестуємо по всьому датасету відразу і вважаємо, що він поміщається в відео-пам'яті.

```
#X_test = X_test.to(device)
# y_test = y_test.to(device)
```

Ми можемо X_test та y_test сказати, щоб вони перейшов на device. А ось Y_train і X_train ми не будемо відразу класти на відео-карту, ми будемо це робити по Батчу, тобто ми візьмемо з датасета деякий Батч, і його будемо перекладати на відео карту.

Зробити це можна в місці, де ми створюємо Батч, тобто перекладати кожен конкретний Батч на device. Навчання, яке відбувається на відеокартці, як правило, в 10-50 разів буде швидше, ніж навчання, яке відбувається на CPU.

```
# device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
# mnist_net = mnist_net.to(device)
# list(mnist_net.parameters())
```

**Навчання моделі.**

Як і минулого разу, ми будемо навчатися батчевим або стохастичним градієнтним спуском. Тобто, ми будемо ділити наш набір даних на маленькі частини (так звані Батчі), передавати ці Батчі в нейронну мережу за

допомогою функції forward. Після цього ми будемо викликати loss-функцію, яка скаже нам розмір помилки. Після цього ми зможемо порахувати градієнт за допомогою функції backward, і далі ми зможемо зробити градієнтний крок, викликавши optimizer.step. Нехай у нас буде batch_size розміром 100 (це число, яке вибрано навмання), поставимо 10 000 епох. Щоразу ми будемо перемішувати набір даних, на кожній епосі і виділяти звідти послідовні ділянки, таким чином, щоб усередині однієї епохи кожен елемент, кожна картинка була показана всього один раз.

Крім того нам би хотілося побачити як росте наша якість після кожної епохи навчання, а також зростання цієї якості на тестовому наборі даних – на тих даних, які нейромережа не бачила. Тому будемо викликати mnistnet.forward ще й на тестовому датасеті, на X_test. Ми будемо робити forward по всьому тестовому набору даних, припускаючи, що він не дуже великий, а батчевий градієнтний спуск нам потрібен всього з двох причин. По-перше, тому що таким чином сам процес оптимізації відбувається швидше – тобто, краще зробити 10 градієнтних кроків на епоху, ніж зробити один. А по-друге, тому що у нас можуть не поміститися Батчі в GPU (тобто на відео картку), якщо ми будемо робити їх досить великими. Тому ми обмежили batch_size 100 на тренуванні, а в тесті ми припустили, що все буде нормально і тому ми можемо зробити forward по всьому набору даних. Крім того, нам потрібно порахувати частку правильних відповідей – accuracy і, відповідно, нам потрібно зрозуміти, а який же клас передбачила нейронна мережа. Нейронна мережа передбачає клас, для якого вона видає максимальне значення. Тобто, після цього це значення відправляється в софтмакс, якщо потрібно дізнатися його ймовірність, але вже перед софтмаксом ми можемо зрозуміти – а який найбільш ймовірний клас передбачила нейронна мережа. Це буде той нейрон, у якого максимальний вихід. Відповідно, нам потрібно зробити argmax (argmax віддає номер нейрона, одного з десяти, у якого максимальний вихід), і порівняти це з y_test. Це accuracy ми будемо кожен раз виводити на екран.

```
batch_size = 100

# test_accuracy_history = []
# test_loss_history = []

# X_test = X_test.to(device)
# y_test = y_test.to(device)

for epoch in range(10000):
    order = np.random.permutation(len(X_train))

    for start_index in range(0, len(X_train), batch_size):
        optimizer.zero_grad()

        batch_indexes = order[start_index:start_index+batch_size]

        X_batch = X_train[batch_indexes] #.to(device)
        y_batch = y_train[batch_indexes] #.to(device)

        preds = mnist_net.forward(X_batch)

        loss_value = loss(preds, y_batch)
        loss_value.backward()

        optimizer.step()

    test_preds = mnist_net.forward(X_test)
#     test_loss_history.append(loss(test_preds, y_test))

    accuracy = (test_preds.argmax(dim=1) == y_test).float().mean()
```

```
#     test_accuracy_history.append(accuracy)
    print(accuracy)
```

Крім того можна проаналізувати як проходить навчання, записавши accuracy, і loss. Будемо зберігати наші втрати на тесті, в list "test_loss_history", а наші accuracy, відповідно, в list "test_accuracy_history".

Дуже цікаво такі графіки дивитись окремо для тренування і тесту для того, щоб побачити перенавчання.

```
plt.plot(test_accuracy_history)
plt.plot(test_loss_history)
```

**Практична частина**

1. **Зробіть покращення у нейронній мережі, наведеній у теоретичній частині та дайте відповідь на наступні питання:**
   a. Побудуйте на одному графіку loss для train і validation. Чи правда, що loss на train і validation падає однаково швидко і виходить на однакове значення, або ж у нас є перенавчання?
   b. Чи веде збільшення кількості епох (з 40 епох до 200 епох) до поліпшення метрик на валідації?
   c. Чи уповільнює torch.backends.cudnn.deterministic = True навчання на практиці? Якщо так, то наскільки?
   d. Спробуйте різні методи градієнтного спуску. Як вибір градієнтного спуску впливає на accuracy? Для впевненості краще проводити один експеримент 3-5 разів на різних random seed: так ви зрозумієте, чи дійсно позначається вплив методу або справа в випадковості.
   e. Реалізуйте функціональність torch.nn.Linear з нуля і порівняйте з оригіналом!

f. Знайдіть оптимальну кількість епох для навчання, аналізуючи графіки втрат.
g. Нехай у нас буде 1 об'єкт x на вході з двома компонентами. Його ми передаємо в повнозв'язний шар з 3-ма нейронами і отримаємо, відповідно, 3 виходи. Після цього напишіть цю ж функціональність за допомогою матричного множення. Та перевірте розрахунок похідної.

Шаблон коду:

```
import torch

# Створимо тензор x:
x = torch.tensor([[10., 20.]])

# Оригінальний повнозв’язний шар із 2-ма входами та 3-мя нейронами (виходами):
fc = torch.nn.Linear(2, 3)

# Ваги fc-шара зберігаються у fc.weight, а bias відповідно у fc.bias
# fc.weight и fc.bias за замовчування ініціалізуються випадковими числами

# Давайте задамо свої значення ваг та зміщень:
w = torch.tensor([[11., 12.], [21., 22.], [31., 32]])
fc.weight.data = w

b = torch.tensor([[31., 32., 33.]])
fc.bias.data = b

# Отримаємо вихід fc-шара:
```

*fc_out = fc(x)*

*# Просумуємо виходи із fc-шара, щоб отримати скаляр (адже у PyTorch backward можна розрахувати тільки від скалярної функції):*
*fc_out_summed = fc_out.sum()*

*# Порахуємо градієнти формули fc_out_summed:*
*fc_out_summed.backward()*
*weight_grad = fc.weight.grad*
*bias_grad = fc.bias.grad*

*# Тепер зробимо розрахунки які були вище але без fc-шару:*
*# Зазначимо, що у "w" та "b" необхідно розраховувати градієнти (для fc-шара це виходить автоматично):*
*w.requires_grad_(True)*
*b.requires_grad_(True)*

Спробуйте отримати аналогічні результати використовуючи матричне множення

*fc_out_alternative = # x * w^T + b*

Отримаємо вихід після сумування нашої формули:

*our_formula = ... # SUM{x * w^T + b}*

Зробіть backward для нашої формули:
...

Аналіз результатів:

*print(fc_out == fc_out_alternative)*
*print('fc_weight_grad:', weight_grad)*
*print('our_weight_grad:', w.grad)*
*print('fc_bias_grad:', bias_grad)*
*print('out_bias_grad:', b.grad)*

2. **Адаптувати розроблену модель для роботи з набором даних CIFAR 10.**

Зробіть аналогічні розрахунки та натренуйте модель для набору даних CIFAR 10. Вкажіть, що це за набір даних, особливості, кількість класів, отримані результати, тощо.

Завантажується він аналогічним чином:

*torchvision.datasets.CIFAR10*

Набір даних CIFAR-10 являє собою набір зображень, які зазвичай використовуються для навчання алгоритмами машинного навчання і комп'ютерного зору. Це один з найбільш широко використовуваних наборів даних для дослідження машинного навчання. Складається з 60000 кольорових (RGB, 3 канали) зображень розміром 32x32, які розбиті на 10 класів (6000 на кожен клас). 50000 картинок на навчання та 10000 тестових.

Враховуючи те, що зображення кольорові, то розмірність картинок буде вже 32 x 32 x 3. Тобто, для обробки такого зображення повнозв’язний нейрон у першому шарі має мати 32x32x3 = 3072 пікселів.

Кожен піксель кодується значеннями від 0 до 255 для кожного каналу.

З такими зображеннями можна працювати – є мережі, які дійсно

працюють з такими даними. Але ми, для зручності, нормуємо ці дані – розділимо кожен піксель, кожне його значення, на 255 і отримаємо, що в наших картинках будуть лежати значення від нуля до одиниці.

*X_train /= 255.*
*X_test /= 255.*

Подивіться які класи дає цей датасет

*print(CIFAR_train.classes)*

Ще одна особливість цього набору даних: як і у звичайних картинок, канал "колір" кодується в останній розмірності. Тобто, спочатку йде висота картинки, ширина, а після цього вже колір. Але pytorch вимагає, щоб цей канал йшов на першому місці. Тобто, ми маємо зараз 4-мірний тензор: "кількість картинок, висота, ширина і колір", а Pytorch хоче: "кількість картинок, кількість каналів, ширина, висота". Нам потрібно реорганізувати розмірність тензора таким чином, щоб колір йшов на другому місці – якраз після кількості картинок в наборі даних. Це робиться за допомогою методу "permute" з чотирма аргументами. Перший аргумент "0" відповідає за кількість картинок в нашому наборі даних. Ми не хочемо, щоб розмірність "кількість картинок" в наборі даних змінила позицію, відповідно ставимо нуль на перше місце. Далі варто поставити число "3" – це означає, що на це місце прийде розмірність, яка була під номером "3" в первісному тензорі, тобто це буде "кількість каналів". Далі йде "1", тобто на це місце прийде "висота картинки", а далі "2" – це значить – на це місце прийде "ширина картинки", і у нас, після виконання цієї операції, у всього датасета буде shape: 50 000 на 3 на 32 на 32:

*X_train = X_train.permute(0, 3, 1, 2)*

*X_test = X_test.permute(0, 3, 1, 2)*

## Висновки

У висновках обґрунтувати вибір оптимальної нейронної мережі, гіперпараметрів та методів оптимізації з точки зору точності роботи моделі для обох завдань.

## Контрольні запитання

4. Що являє собою повнозв’язна нейронна мережа?
5. Що являє собою набір даних CIFAR 10?

# 9: РОЗПІЗНАВАННЯ РУКОПИСНИХ ЧИСЕЛ ЗГОРТКОВОЮ НЕЙРОННОЮ МЕРЕЖЕЮ

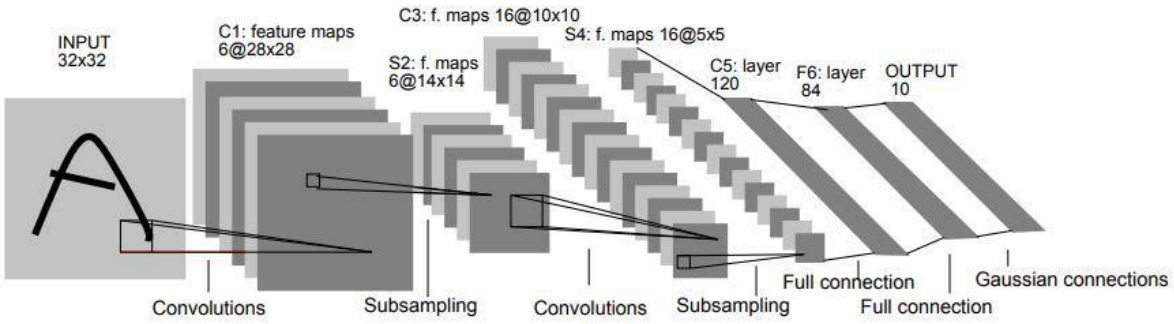
**Мета роботи:** засвоїти принцип роботи та програмної реалізації згорткових нейронних мереж.

## Теоретична частина

У минулій практичній роботі ми класифікували набор даних MNIST (рукописні цифри) за допомогою повнозв'язної нейронної мережі. Зараз спробуємо поліпшити наш результат, використавши згорткові нейронні мережі.

Згорткові нейронні мережі (ЗНМ) – це клас глибинних штучних нейронних мереж прямого поширення, який успішно застосовується до аналізу візуальних зображень. ЗНМ використовують різновид багатошарових перцептронів, розроблений так, щоби вимагати використання мінімального обсягу попередньої обробки. Вони відомі також як інваріантні відносно зсуву або просторово інваріантні штучні нейронні мережі, виходячи з їхньої архітектури спільних ваг та характеристик інваріантності відносно паралельного перенесення.

Розглянемо архітектуру LeNet, вона найперша в світі згорткових нейронних мереж (рисунок 9.1). Відповідно, вона не найкраща, але вона досить логічна.



| Layer | | Feature Map | Size | Kernel Size | Stride | Activation |
|---|---|---|---|---|---|---|
| Input | Image | 1 | 32x32 | - | - | - |
| 1 | Convolution | 6 | 28x28 | 5x5 | 1 | tanh |
| 2 | Average Pooling | 6 | 14x14 | 2x2 | 2 | tanh |
| 3 | Convolution | 16 | 10x10 | 5x5 | 1 | tanh |
| 4 | Average Pooling | 16 | 5x5 | 2x2 | 2 | tanh |
| 5 | Convolution | 120 | 1x1 | 5x5 | 1 | tanh |
| 6 | FC | - | 84 | - | - | tanh |
| Output | FC | - | 10 | - | - | softmax |

Рисунок 9.1 – архітектура мережі LeNet

Отже, спершу подається зображення розміром 32 на 32. У MNIST зображення 28 на 28, необхідно буде зробити деяку попередню обробку зображень. Це зображення проходить через згортку 5 на 5, з шістьма вихідними каналами, значить – у нас згортка 5 на 5 проходить по всьому зображенню. У неї нульові паддінг (padding), тобто вона не виходить за межі зображення, і з цієї причини вона обрізає два пікселя з кожного боку зображення, і вихідне зображення з розміру 32 на 32 перетворюється до зображення 28 на 28. Але тепер це вже не зображення а тензор глибиною 6, тому що ми використали згортку з шістьма вихідними каналами. Після цього тензор розміром 28 на 28 на 6 передається в average pooling (зараз вже майже всюди використовують max pooling), який обчислює середнє значення по квадрату 2 на 2 зі страйд (stride) 2 (тобто він бере непересічні ділянки 2 на 2), і обчислює середнє значення пікселів або чисел, які виявляються в цьому тензор. І на виході маємо один піксель, відповідно, все зображення стискується у два рази: було 28 на 28 на 6, стало 14 на 14 на 6. Далі повторюється точно така ж згортка, як в перший раз, тільки тепер у неї кількість вхідних каналів – 6, а вихідних каналів – 16. При цьому, з зображення 14 на 14, виходить зображення 10 на 10, тому що у нас по два пікселя «з'їлося» з кожного боку, тому що ми застосовували згортки без padding – без виходу за межі зображення. Далі знову пулінг (точно такий же,

2 на 2 average pooling), і далі у нас є два варіанти, що робити (в принципі вони аналогічні):

1) У оригінальному LeNet – беруть згортку 5 на 5, (тому що у нас виходить тензор 5 на 5 на 16) з кількістю вхідних каналів 16, а вихідних – 120. І ця згортка, вона перетворює глибокий тензор в один вектор розміру 1 на 1 на 120.

2) А ми зробимо трошки по-іншому: спочатку цей тензор 5 на 5 на 16 ми розвернемо в один вектор. Таким чином, у нас є готовий вектор, до якого можна застосувати повнозв'язний шар, який на виході буде мати 120 нейронів. Таким чином, в повнозв'язний шарі буде 120 нейронів, при цьому кількість ваг буде однаковою.

Після того, як ми отримали вектор довжиною 120 (неважливо, яким способом), ми застосовуємо до нього два повнозв'язних шара. Перший шар з 120 нейронів отримує 84 нейрона, а другий нам віддає відповідь, тобто він віддає нам 10 нейронів, які потім перетворюються в одну з цифр.

Після кожної згортки та пулінгу будемо застосовувати активацію – візьмемо як і в LeNet – гіперболічні тангенси (зараз наприклад – ReLU було б актуальніше; точно так же сьогодні ми б не застосовували згортки п'ять на п'ять – напевно, ми б застосовували скрізь згортки 3 на 3). Після останнього повнозв'язного шару який утискає розмір на виході до 10 нейронів застосовується софтмакс, який ми застосовувати не будемо, тому що нам потрібен тільки максимальний нейрон по активації. А якщо нам потрібні були б ймовірності, то ми б використовували софтмакс і отримали б деякі ймовірності впевненості мережі.

**Імпортування бібліотек та датасета.**

Як і в минулій практичній роботі ми імпортуємо ті ж самі бібліотеки, фіксуємо random seed, імпортуємо бібліотеку для завантаження набору даних. Імпортуємо точно так же MNIST, і отримуємо X_train, Y_train, X_test, Y_test.

Перша відмінність полягає в тому, що, на відміну від повнозв'язної мережі, яка бачила картинку як один довгий вектор, ми хочемо в згорткову мережу передавати картинку як тривимірний тензор, де перший канал – це глибина картинки, в чорно-білій картинці це 1 канал з яскравістю сірого пікселя. А в RGB зображенні будуть RGB канали. Відповідно, ми повинні нашу картинку, яка прийшла на вхід (вона розміром 28 на 28), розтиснути до 1 на 28 на 28. Це ми робимо за допомогою функції unsqueeze – X_train.unsqueeze(тут ставимо індекс: в якому ж вимірі ми хочемо розтиснути його). X_train у нас – тензор з 60 000 картинок 28 на 28, а ми хочемо щоб було 60 000 на 1 на 28 на 28, і те ж саме ми робимо з тестом:

```
import torch
import random
import numpy as np

random.seed(0)
np.random.seed(0)
torch.manual_seed(0)
torch.cuda.manual_seed(0)
torch.backends.cudnn.deterministic = True

import torchvision.datasets

MNIST_train = torchvision.datasets.MNIST('./', download=True, train=True)
MNIST_test = torchvision.datasets.MNIST('./', download=True, train=False)

X_train = MNIST_train.train_data
y_train = MNIST_train.train_labels
```

```
X_test = MNIST_test.test_data
y_test = MNIST_test.test_labels

print(len(y_train), len(y_test))

import matplotlib.pyplot as plt
plt.imshow(X_train[0, :, :])
plt.show()
print(y_train[0])

X_train = X_train.unsqueeze(1).float()
X_test = X_test.unsqueeze(1).float()

print(X_train.shape)
```

**Модель**

Далі запрограмуємо нейронну мережу. Назвемо її LeNet5, тому що так називалася мережа у ЛеКуна. LeNet5 – вона називається тому, що там 5 шарів для навчання – три згортки і два повнозв'язних шара, у нас буде дві згортки і три повнозв'язних шара.

Точно так же, як у ЛеКуна, у нас буде перший шар, який приймає один канал на вхід, тому що картинка у нас одномірна (там один канал) – не RGB, а grayscale, і на виході буде 6 каналів. Але на відміну від ЛеКуна, до якого приходила картинка 32 на 32, у нас картинка 28 на 28. І якщо ми зробимо точно так же, як в оригінальній мережі, тобто застосуємо згортку 5 на 5 без паддінгу, у нас вийде картинка 28 - 4 = 24 пікселя на 24. Ми б хотіли, щоб після першої згортки картинка була б 28 на 28. Якщо ви не хочете втрачати розмірність картинки, то вам потрібно встановити певний паддінг, щоб згортка виходила за межі картинки і тензор, або зображення, на виході були такого ж розміру. При використанні фільтрів 5 на 5 згортки повинні

виходити на 2 пікселя за розмір зображення, тоді підсумковий тензор матиме такий же розмір. Якби у нас були згортки 3 на 3, то нам потрібно було б виходити на один піксель, і тоді б кількість цих згорток, які ми прикладаємо до зображення, була б так само, як раз, розміру зображення. Поставимо паддінг "2" щоб зберегти розмір. Отже, нам потрібно застосувати наш перший згортковий шар, він називається "Conv2d", тому що він двомірний. Якби у нас були якісь тривимірні зображення, наприклад, мозку людини, і ми б мали третю координату, то у нас було б "Conv3d", який в PyTorch теж є. Але у нас картинка плоска, відповідно ми будемо ходити в двомірному просторі:

*self.conv1 = torch.nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5, padding=2)*

У цьому Conv2d є багато параметрів, ми повинні використовувати кількість вхідних каналів – обов'язково проставити, що їх буде "1". Вихідних каналів буде 6. Розмір ядра згортки – "5", якщо у вас раптом згортка не симетрична (бувають розтягнуті згортки), то тут можна вказати кортеж (tuple) з двох чисел, але у нас згортка буде квадратна, 5 на 5, відповідно, ми просто зазначимо "5". І далі ми повинні вказати паддінг "2", тому що у нас згортка повинна виходити за межі зображення.

Далі ми повинні застосувати активізацію. Активації в LeNet – гіперболічні тангенси, відповідно ми застосовуємо "torch.nn.Tanh". Після цього – перший пулінг, називається "pool1", це "average pooling". В середині мережі зазвичай використовується max pooling, але в силу традицій тут теж ми використовуємо average pooling 2d:

*self.pool1 = torch.nn.AvgPool2d(kernel_size=2, stride=2)*

У нього "kernel_size" – 2, тому що це пулінг 2 на 2, і "stride" – 2, тому що він застосовується без перетинів. Цей пулінг нам стисне зображення від 28 на 28 до 14 на 14.

Далі робимо приблизно все те ж саме: Conv2d – знову згортка, вхідних каналів 6, тому що було тут вихідних каналів 6, і вихідних каналів у цій згортці 16. Так було в оригінальній архітектурі – точно такий же kernel size 5, padding 0. Знову активація тангенсом, знову пулінг. З зображення, яке було 14 на 14, згортка зробить зображення 10 на 10, тому що з'їсть по 2 пікселя з кожного боку, а пулінг зробить зображення 5 на 5.

Далі розгортуємо зображення в один вектор (це ми робимо в функції "forward", а поки ми представимо, що ми вже розтягли все зображення в один вектор), і далі нам потрібні три повнозв'язних шари. Перший повнозв'язний шар на вхід приймає зображення розміром 5 на 5 і глибиною 6. Перемножуємо 5 на 5 на 6, отримуємо 400 – це розмір нашого вектора, і на виході ми хочемо вектор розміру 120.

*self.fc1 = torch.nn.Linear(5 * 5 * 16, 120)*

Знову тангенс, і знову повнозв'язний шар 120 на 84 (self.fc2 = torch.nn.Linear(120, 84)), знову тангенс, і нарешті-то fully-connected шар, який складається з 84 нейронів і робить 10 з відповідями (self.fc3 = torch.nn.Linear(84, 10)). А в функції forward ми повторюємо всю цю логіку, але тепер застосовуємо ці шари до деякого вхідного тензора X.

Вхідний тензор X – це, насправді, Батч з картинок. Застосовуємо згортку, активацію, пулінг, згортку, активацію, пулінг, і виконавши x = x.view(x.size(0), x.size(1) * x.size(2) * x.size(3)) розгортаємо тензор, який чотиривимірний, тому що перша розмірність відповідає за розмірність Батч. У PyTorch-тензорів є функція view, яка тензор перетворює до потрібної розмірності. Перша розмірність буде x.size [0] – це розмір Батч, а далі тензор буде одновимірний, відповідно три наступні розмірності ми повинні просто

перемножити і отримати 400. Далі – перший повнозв’язний шар, активація, другий повнозв’язний, активація, повнозв’язний шар.

Останнє це ініціалізація мережі, вона без параметрів (lenet5 = LeNet5()):

```
class LeNet5(torch.nn.Module):
  def __init__(self):
    super(LeNet5, self).__init__()

    self.conv1 = torch.nn.Conv2d(
      in_channels=1, out_channels=6, kernel_size=5, padding=2)
    self.act1  = torch.nn.Tanh()
    self.pool1 = torch.nn.AvgPool2d(kernel_size=2, stride=2)
    self.conv2 = torch.nn.Conv2d(
      in_channels=6, out_channels=16, kernel_size=5, padding=0)
    self.act2  = torch.nn.Tanh()
    self.pool2 = torch.nn.AvgPool2d(kernel_size=2, stride=2)
    self.fc1   = torch.nn.Linear(5 * 5 * 16, 120)
    self.act3  = torch.nn.Tanh()
    self.fc2   = torch.nn.Linear(120, 84)
    self.act4  = torch.nn.Tanh()
    self.fc3   = torch.nn.Linear(84, 10)

  def forward(self, x):

    x = self.conv1(x)
    x = self.act1(x)
    x = self.pool1(x)
    x = self.conv2(x)
    x = self.act2(x)
```

```
        x = self.pool2(x)
        x = x.view(x.size(0), x.size(1) * x.size(2) * x.size(3))
        x = self.fc1(x)
        x = self.act3(x)
        x = self.fc2(x)
        x = self.act4(x)
        x = self.fc3(x)
        return x

lenet5 = LeNet5()
```

**Навчання.**

Процес навчання, як в минулий раз, буде йти по Батчам. Батч буде розміру 100, кожну епоху ми будемо друкувати accuracy і накопичувати loss на даній епосі. А всередині кожного Батч будемо спочатку обнуляти градієнти, після цього обчислювати – які ж картинки підуть в поточний Батч, переносити поточний Батч на device (на CUDA, на GPU). Після цього пропускати Батч через мережу за допомогою функції forward, яку реалізували вище, після цього на prediction мережі рахувати loss (він без софтмаксу, тому що крос-ентропія приймає виходи без софтмаксу). Далі будемо рахувати градієнти і робити крок градієнтного спуску. Після закінчення розрахунків на Батчі можемо порахувати якість на відкладеній вибірці, на X_test. Тут ми знову робимо обрахування якості на X_test цілком. Точно так само ми можемо розрахувати accuracy на кожному з Батчів, а потім підсумувати та підрахувати сумарне accuracy.

І ще один момент, дуже важливий при використанні відеокартки може статися виток пам'яті – ми цього не помітили, а от на даному прикладі, так як у нас мережа стає трохи більше, ми починаємо відчувати, що пам’яті на відеокартці не вистачає – в якийсь момент обрахування може впасти. Чому так відбувається? Тому що результат обчислення втрат на тестових даних

ми клали в минулий раз безпосередньо в list – об'єкт, який зберігав весь граф обчислень, було test_loss_history.append(loss(test_preds, y_test).cpu())). Виходить, що в loss зберігається не просто число, а весь граф обчислень, який нам допомагає потім обрахувати градієнти. Так як, по-перше, це все зберігається на GPU (бо у нас loss зберігається на GPU, якщо ми його спеціально на CPU не перенесемо), і весь цей loss тепер зберігається у векторі – відповідно, не очищається пам'ять. Тобто на GPU накопичуються ці графи, за якими при бажанні можна обчислити градієнти, але насправді нам ці графи не потрібні. Нам потрібні самі числа, які відповідають нашій функції втрат: її чисельне представлення. Тому ми можемо взяти, і викинути всю інформацію про граф обчислення похідної, для цього можна просто сказати їй ".data", і залишиться тільки одне число – скаляр. Після цього потрібно ще її, по-хорошому, відправити на CPU, щоб вона не займала нам пам'ять.

Далі можемо порахувати якість. Точно так же, як в минулий раз, ми беремо той нейрон, у якого вихід найбільший. Порівнюємо номер цього нейрона (ми беремо спеціально argmax, щоб у нас був номер нейрона) з y_test, а там знаходиться номер тієї цифри яку ми хочемо передбачити. Далі перетворюємо відповідь до float, тому що саме у float можна взяти mean. Беремо mean у цього float, після цього теж переводимо все в одне число (на всякий випадок), і беремо від нього ".cpu" (тобто, переміщаємо його на CPU). І далі accuracy, точно так же, складуємо в test_accuracy_history:

```
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
lenet5 = lenet5.to(device)

loss = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(lenet5.parameters(), lr=1.0e-3)

batch_size = 100
```

```
test_accuracy_history = []
test_loss_history = []
X_test = X_test.to(device)
y_test = y_test.to(device)

for epoch in range(10000):
    order = np.random.permutation(len(X_train))
    for start_index in range(0, len(X_train), batch_size):
        optimizer.zero_grad()
        batch_indexes = order[start_index:start_index+batch_size]
        X_batch = X_train[batch_indexes].to(device)
        y_batch = y_train[batch_indexes].to(device)
        preds = lenet5.forward(X_batch)
        loss_value = loss(preds, y_batch)
        loss_value.backward()
        optimizer.step()
    test_preds = lenet5.forward(X_test)
    test_loss_history.append(loss(test_preds, y_test).data.cpu())
    accuracy = (test_preds.argmax(dim=1) == y_test).float().mean().data.cpu()
    test_accuracy_history.append(accuracy)
    print(accuracy)
lenet5.forward(X_test)
# plt.plot(test_accuracy_history)
plt.plot(test_loss_history)
```

## Практична частина

1. **Реалізуйте модель наведену у теоретичній частині, натренуйте та проаналізуйте результат та швидкість навчання** (для довідки,

найкращі результати на наборі даних MNIST мають похибку 0.03% тобто точність більше ніж 99.97)

2. **Спробуйте домогтися якості 0.992 на даному наборі даних** (в максимумі на валідації). Зверніть увагу на наступні моменти:
   a. Чи з'являється перенавчання при збільшенні кількості епох?
   b. Як додавання різних шарів впливає на швидкість навчання (які шари обробляються швидше: згорткові або повнозв'язні)?
   c. Проаналізуйте дисперсію цільової метрики від запуску до запуску.
   d. Скільки запусків досить зробити, перед тим як стверджувати, що одна архітектура краща за іншу?
3. **Спробуйте провести аналогічні розрахунки для набору даних CIFAR10**:
   a. Змінить мережу LeNet таким чином, щоб вона приймала зображення 32 на 32 і три канали на вході. Щоб передати їй три канали, потрібно в in_chanels першої ж згортки поставити "3" (раніше тут було "1"). А ось розмірність 32 на 32 у нас виходить з паддінга. Як ви пам'ятаєте, в оригінальному LeNet була розмірність 32 на 32 і там були нульові паддінги, тобто згортка не виходили за зображення. Ми навмисно в LeNet для набору даних MNIST це змінювали, щоб у нас перша згортка виходила за зображення і після першої згортки було 28 на 28. А тепер ми, навпаки, хочемо щоб паддінга не було, і з розмірності 32 на 32 виходила розмірність 28 на 28. Відповідно, необхідно поставити паддінг "0". Більше нічого не змінюється – процес навчання не змінюється, він залишається таким же, як і раніше.
   b. Знайдіть оптимальну кількість епох для навчання, аналізуючи графіки втрат
4. **Підберіть правильні параметри паддінгу (в коментарі після кожного завдання вказано, який розмір тензору на виході повинен бути):**

```
import torch
N = 4
C = 3
C_out = 10
H = 8
W = 16
x = torch.ones((N, C, H, W))
out = torch.nn.Conv2d(C, C_out, kernel_size=(3, 3), padding=...)(x)
print(out.shape)  # torch.Size([4, 10, 8, 16])

out = torch.nn.Conv2d(C, C_out, kernel_size=(5, 5), padding=...)(x)
print(out.shape)  # torch.Size([4, 10, 8, 16])

out = torch.nn.Conv2d(C, C_out, kernel_size=(7, 7), padding=...)(x)
print(out.shape)  # torch.Size([4, 10, 8, 16])

out = torch.nn.Conv2d(C, C_out, kernel_size=(9, 9), padding=...)(x)
print(out.shape)  # torch.Size([4, 10, 8, 16])

out = torch.nn.Conv2d(C, C_out, kernel_size=(3, 5), padding=...)(x)
print(out.shape)  # torch.Size([4, 10, 8, 16])

out = torch.nn.Conv2d(C, C_out, kernel_size=(3, 3), padding=...)(x)
print(out.shape)  # torch.Size([4, 10, 22, 30])

out = torch.nn.Conv2d(C, C_out, kernel_size=(4, 4), padding=...)(x)
print(out.shape)  # torch.Size([4, 10, 7, 15])

out = torch.nn.Conv2d(C, C_out, kernel_size=(2, 2), padding=...)(x)
print(out.shape)  # torch.Size([4, 10, 9, 17])
```

## Висновки

У висновках обґрунтувати вибір оптимальної згорткової нейронної мережі для всіх наведених завдань: кількість шарів, розмір згортки, тощо. Проаналізувати точність роботи моделі. Надати рекомендації, щодо можливості подальшого покращення.

## Контрольні запитання

1. Що являє собою згорткова нейронна мережа?
2. У чому суть операції згортки?
3. Для чого робиться операція pooling?

# 10: УДОСКОНАЛЕННЯ ЗГОРТКОВОЇ НЕЙРОННОЇ МЕРЕЖІ LENET5. BATCH NORMALIZATION

**Мета роботи:**

1. Закріпити принципи роботи та побудови згорткових нейронних мереж.

2. Засвоїти принцип роботи пакетної нормалізації.

## Теоретичні відомості

У цій роботі ми спробуємо дещо покращити LeNet щоб підвищити якість на валідації. У нас є деякі речі які непогано було б виправити:

1) По-перше, активації: зараз вже ніхто не використовує активації гіперболічний тангенс, тому що вони призводять до проблеми «загасання градієнта» – ви не можете побудувати дійсно глибоку мережу, щоб ваші помилки, пораховані в кінці мережі добре проходили до початку мережі. Тангенси або сигмоїди призводять до того, що сигнал дуже швидко згасає.

2) По-друге зараз мало хто використовує згортки 5 на 5. Практично всі сучасні мережі згортки 5 на 5 замінюють на дві згортки 3 на 3, які йдуть одна за одною. Чому так робиться? Тому що в 5 на 5 – 25 ваг, а ось в двох згортках 3 на 3 – в одній 9 ваг, в інший 9 ваг – виходить 18 ваг. При цьому, в принципі, вони мають аналогічний ефект (покриття, несуть однакову інформацію). Ваг буде менше – напевно, буде менше перенавчання.

3) По-третє у нас використовується average pooling, але average pooling вже ніде не використовується, крім як в самому кінці мережі. Зараз використовуються всюди max pooling.

4) І останній момент – це Батч нормалізація, яка покликана прискорювати навчання. Питання куди поставити Батч нормалізація: перед активацією, після неї – відкрите. Частіше зараз роблять активацію, а потім Батч нормалізацію.

5) У зв’язку з використанням Батч нормалізації ми повинні не забути додати в процес навчання деякий прапорець: або наша мережа знаходиться в стані тренування (і, відповідно, ми повинні проставити net.train()), або мережа знаходиться в стані "evaluation", (який проставляється за допомогою "net.eval()"). Навіщо це потрібно? Поки ми не використовували Батч-нормалізацію, нам не потрібно було проставляти ці прапорці – мережа знаходиться в "train" режимі, або в "evaluation". Але справа в тому, що Батч-нормалізація відбувається не в момент обчислення градієнтів (не в момент backward), а в момент forward. Тобто кожен раз, коли ми проходимо картинкою по мережі в forward-напрямку у нас заново навчаються / підганяються параметри математичного очікування і стандартного відхилення в batch-norm шарі. І, відповідно, щоб цього не відбувалося, ми повинні явно сказати batch-norm шару, що у нас мережа не навчається, а ми її тестуємо, щоб він залишав вірними ці параметри математичного очікування і стандартного відхилення. Інакше у нас буде сітка змінюватися, коли ми це випустимо в виробництво.

**Перепишемо код LeNet.**

Зміни більшою частиною будуть у визначенні відповідного класу. Залишимо де можливо минулий функціонал, там де модель не визначена будемо викликати raise NotImplementedError:

1) У нас з'явилися деякі змінні при визначенні нашої моделі (у конструкторі __init__): activation, pooling, conv_size і use_batch_norm. Вони визначають що – активація буде тангенсом або ReLU; pooling буде "average" або "max pooling"; "Conv_size = 5" – це означає, буде одна згортка 5 на 5, або, якщо ми поставимо сюди значення "3", то буде дві послідовних згортки 3 на 3; змінна use_batch_norm визначає, чи буде використана Батч-нормалізація, чи ні.

2) Функція forward тепер виглядає наступним чином. Якщо у нас "conv_size = 5", то ми використовуємо одну згортку 5 на 5; якщо у нас "conv_size = 3", то ми будемо використовувати спочатку одну згортка 3 на

3, і результат її передаємо в наступну згортку 3 на 3. Далі буде йти активація, відповідно – в ініціалізації ми сказали: act1 – це буде або ReLU, або тангенс. І якщо ми хочемо Батч-нормалізацію, ми можемо додатково, після згортки застосовувати шар Батч-нормалізації batchnorm1 або batchnorm2. Треба окремо сказати про шари Батч-нормалізації. Вони викликаються за допомогою шару torch.nn.BatchNorm2d, тому що ми маємо справу з картинками. Якби ми хотіли нормалізувати деякий вектор після, наприклад fully-connected шару, то ми б використовували torch.nn.BatchNorm1d. На вхід цього шару потрібно передати num_features, тобто ту кількість каналів, яку має картинка або тензор перед Батч-нормалізацією:

```
class LeNet5(torch.nn.Module):
    def __init__(self,
                 activation='tanh',
                 pooling='avg',
                 conv_size=5,
                 use_batch_norm=False):
        super(LeNet5, self).__init__()

        self.conv_size = conv_size
        self.use_batch_norm = use_batch_norm

        if activation == 'tanh':
            activation_function = torch.nn.Tanh()
        elif activation == 'relu':
            activation_function  = torch.nn.ReLU()
        else:
            raise NotImplementedError

        if pooling == 'avg':
```

```
    pooling_layer = torch.nn.AvgPool2d(kernel_size=2, stride=2)
elif pooling == 'max':
    pooling_layer  = torch.nn.MaxPool2d(kernel_size=2, stride=2)
else:
    raise NotImplementedError

if conv_size == 5:
    self.conv1 = torch.nn.Conv2d(
        in_channels=1, out_channels=6, kernel_size=5, padding=2)
elif conv_size == 3:
    self.conv1_1 = torch.nn.Conv2d(
        in_channels=1, out_channels=6, kernel_size=3, padding=1)
    self.conv1_2 = torch.nn.Conv2d(
        in_channels=6, out_channels=6, kernel_size=3, padding=1)
else:
    raise NotImplementedError

self.act1 = activation_function
self.bn1 = torch.nn.BatchNorm2d(num_features=6)
self.pool1 = pooling_layer

if conv_size == 5:
    self.conv2 = torch.nn.Conv2d(
        in_channels=6, out_channels=16, kernel_size=5, padding=0)
elif conv_size == 3:
    self.conv2_1 = torch.nn.Conv2d(
        in_channels=6, out_channels=16, kernel_size=3, padding=0)
    self.conv2_2 = torch.nn.Conv2d(
        in_channels=16, out_channels=16, kernel_size=3, padding=0)
else:
```

```
            raise NotImplementedError

        self.act2 = activation_function
        self.bn2 = torch.nn.BatchNorm2d(num_features=16)
        self.pool2 = pooling_layer

        self.fc1 = torch.nn.Linear(5 * 5 * 16, 120)
        self.act3 = activation_function

        self.fc2 = torch.nn.Linear(120, 84)
        self.act4 = activation_function

        self.fc3 = torch.nn.Linear(84, 10)

    def forward(self, x):
        if self.conv_size == 5:
            x = self.conv1(x)
        elif self.conv_size == 3:
            x = self.conv1_2(self.conv1_1(x))
        x = self.act1(x)
        if self.use_batch_norm:
            x = self.bn1(x)
        x = self.pool1(x)

        if self.conv_size == 5:
            x = self.conv2(x)
        elif self.conv_size == 3:
            x = self.conv2_2(self.conv2_1(x))
        x = self.act2(x)
        if self.use_batch_norm:
```

```
        x = self.bn2(x)
    x = self.pool2(x)

    x = x.view(x.size(0), x.size(1) * x.size(2) * x.size(3))
    x = self.fc1(x)
    x = self.act3(x)
    x = self.fc2(x)
    x = self.act4(x)
    x = self.fc3(x)

    return x
```

Виділимо окремо функцію train із параметрами (net – це наша модель, та набори даних):

```
def train(net, X_train, y_train, X_test, y_test):
    device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
    net = net.to(device)
    loss = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(net.parameters(), lr=1.0e-3)

    batch_size = 100

    test_accuracy_history = []
    test_loss_history = []

    X_test = X_test.to(device)
    y_test = y_test.to(device)

    for epoch in range(30):
```

```
        order = np.random.permutation(len(X_train))
        for start_index in range(0, len(X_train), batch_size):
            optimizer.zero_grad()
            net.train()                    # Ось тут ми вказуємо, що сітка навчається

            batch_indexes = order[start_index:start_index+batch_size]

            X_batch = X_train[batch_indexes].to(device)
            y_batch = y_train[batch_indexes].to(device)

            preds = net.forward(X_batch)

            loss_value = loss(preds, y_batch)
            loss_value.backward()

            optimizer.step()

        net.eval() # Тут ми вказуємо, що сітка тестується, використовуємо дані для нормалізації із тренування
        test_preds = net.forward(X_test)
        test_loss_history.append(loss(test_preds, y_test).data.cpu())

        accuracy = (test_preds.argmax(dim=1) == y_test).float().mean().data.cpu()
        test_accuracy_history.append(accuracy)

        print(accuracy)
    print('---------------')
    return test_accuracy_history, test_loss_history
```

Тепер можна викликати нейромережу та подивимось на процес навчання та тестування:

```
accuracies = {}
losses = {}
accuracies['tanh'], losses['tanh'] = \
    train(LeNet5(activation='tanh', conv_size=5),
          X_train, y_train, X_test, y_test)

accuracies['relu'], losses['relu'] = \
    train(LeNet5(activation='relu', conv_size=5),
          X_train, y_train, X_test, y_test)

accuracies['relu_3'], losses['relu_3'] = \
    train(LeNet5(activation='relu', conv_size=3),
          X_train, y_train, X_test, y_test)

accuracies['relu_3_max_pool'], losses['relu_3_max_pool'] = \
    train(LeNet5(activation='relu', conv_size=3, pooling='max'),
          X_train, y_train, X_test, y_test)

accuracies['relu_3_max_pool_bn'], losses['relu_3_max_pool_bn'] = \
    train(LeNet5(activation='relu', conv_size=3, pooling='max',
use_batch_norm=True),
          X_train, y_train, X_test, y_test)

for experiment_id in accuracies.keys():
    plt.plot(accuracies[experiment_id], label=experiment_id)
plt.legend()
plt.title('Validation Accuracy')
```

## Практична частина

1. **Перевірити на практиці твердження про затухання градієнта.** У документації PyTorch можна знайти такі функції активації: *ELU, Hardtanh, LeakyReLU, LogSigmoid, PReLU, ReLU, ReLU6, RReLU, SELU, CELU, Sigmoid, Softplus, Softshrink, Softsign, Tanh, Tanhshrink, Hardshrink.*

Вам належить знайти активацію, яка призводить до найменшого загасання градієнта. Для перевірки ми сконструюємо SimpleNet, яка буде мати всередині 3 fc-шару, по 1 нейрону в кожному без bias. Ваги цих нейронів ми ініціалізуємо одиницями. На вхід в цю мережу будемо подавати числа з нормального розподілу. Зробимо 200 запусків (NUMBER_OF_EXPERIMENTS) для чесного порівняння і порахуємо середнє значення градієнта в першому шарі.

Знайдіть таку функцію, яка буде давати максимальні значення градієнта в першому шарі. Всі функції активації потрібно форматувати з аргументами за замовчуванням (порожніми дужками).

```
import torch
import numpy as np

seed = int(input())
np.random.seed(seed)
torch.manual_seed(seed)

NUMBER_OF_EXPERIMENTS = 200

class SimpleNet(torch.nn.Module):
    def __init__(self, activation):
        super().__init__()
```

```
        self.activation = activation
        self.fc1 = torch.nn.Linear(1, 1, bias=False)  # one neuron without bias
        self.fc1.weight.data.fill_(1.)  # init weight with 1
        self.fc2 = torch.nn.Linear(1, 1, bias=False)
        self.fc2.weight.data.fill_(1.)
        self.fc3 = torch.nn.Linear(1, 1, bias=False)
        self.fc3.weight.data.fill_(1.)

    def forward(self, x):
        x = self.activation(self.fc1(x))
        x = self.activation(self.fc2(x))
        x = self.activation(self.fc3(x))
        return x

    def get_fc1_grad_abs_value(self):
        return torch.abs(self.fc1.weight.grad)

def get_fc1_grad_abs_value(net, x):
    output = net.forward(x)
    output.backward()  # no loss function. Pretending that we want to minimize output
                       # In our case output is scalar, so we can calculate backward
    fc1_grad = net.get_fc1_grad_abs_value().item()
    net.zero_grad()
    return fc1_grad

activation =  # Try different activations to get biggest gradient
```

```
        # ex.: torch.nn.Tanh()

    net = SimpleNet(activation=activation)

    fc1_grads = []
    for x in torch.randn((NUMBER_OF_EXPERIMENTS, 1)):
        fc1_grads.append(get_fc1_grad_abs_value(net, x))
    print(np.mean(fc1_grads))
```

2. **Натренуйте модель *LeNet5* надану в теоретичному описі. Проведіть наступні експерименти:**
    a. Зробіть кілька експериментів з параметрами, що надані у теоретичному описі. Проаналізуйте отриманий результат. При яких параметрах отримуємо найкращу якість? Які покращення дали результат? Спробуйте пояснити.
    b. Зробіть аналогічні дії на датасеті CIFAR10 (можете спробувати натренувати CIFAR100 – розширена версія цього набору даних, має 100 класів). Порівняйте результати із покращеннями, що дали найбільший внесок.
    c. Спробуйте додати фільтрів і подивитися – а якщо ми з зображення будемо отримувати більше інформації, яка буде точність моделі?
    d. Крім того, зробіть згортки, що виходять за розмір зображення, щоб цей розмір не мінявся до і після згортки (тобто паддінг = 1). Найчастіше – це гарна ідея, тому що таким чином мережа розуміє – в якому саме місці згортка знаходиться в даний момент, знаходиться вона в кутку зображення, зліва чи справа. З-за того, що пікселі, які виходять за розмір зображення заповнюються нулями: відповідно, можна навчити такі згортки,

які будуть реагувати на межі зображення і це може бути додатковою інформацією, яка раніше не була доступна.

е. Застосуйте Батч-нормалізацію до вихідного зображення (в самому началі мережі) – таку Батч-нормалізацію можна трактувати як нормування по яскравості і контрасту початкового зображення.

Загальна структура мережі та експериментів може буде наступною:

Батч-нормалізація (3 канали на вході)

Згортка(3 на 3, 16 каналів на виході, паддінг = 1).
активація ReLU
Батч-нормалізація (16)
max pooling

Згортка(16 каналів на вході, 3 на 3, 32 канали на виході, паддінг = 1).
активація ReLU
Батч-нормалізація (32)
max pooling

Згортка(32 каналів на вході, 3 на 3, 64 канали на виході, паддінг = 1).
активація ReLU
Батч-нормалізація (64)
max pooling

Перший повнозв'язний шар (прийме розтягнутий вектор зображення розміром 8 на 8 на 64. Це число – 4096: 8 на 8 обумовлено розміром тензора, який прийшов після останньої згортки, а 64 – це її кількість каналів, на виході вектор розміром 256).

активація тангенсом.

BatchNorm1d, тому що у нас 1D тензор. На вхід йому приходить 256

Другий повнозв'язний шар (на виході вектор розміром 64).

тангенс як активації

BatchNorm1d (64).

Третій повнозв'язний шар (на виході вектор розміром 10).

f. Проаналізуйте результат та знайдіть оптимальну кількість епох для навчання (точку зупинки, щоб уникнути перенавчання), аналізуючи графіки втрат.
g. Спробуйте додати ще такі шари як DropOut (nn.Dropout(), nn.Dropout2d (тут вказується частка нейронів, що буде вимкнена), перед згорткою (після нормалізації) або повнозв’язним шаром).
h. Спробуйте додати l2-регуляризацію. У PyTorch вона активується за допомогою параметра weight_decay в оптимізаторі (для всіх шарів одразу в автоматичному режимі). Значення зазвичай вибирають з [1e-3, 1e-4, 1e-5]. Значення 1e-2 ставити не варто, тому що мережа не зможе вчитися, а 1e-6 швидше за все просто не вплине на навчання (але краще перевірити це твердження самостійно). Приклад:

```
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4,
                             weight_decay=1e-5)
```

Або вручну це робиться так:

```
        loss_value = loss(preds, y_batch)
        L2 = 0
for p in net.parameters():
        L2 = L2 + (p**2).sum()
loss_value = loss_value + 2. / y_batch.size(0) * LAMBDA * L2

        loss_value.backward()
        optimizer.step()
```

## Висновки

У висновках обґрунтувати вибір оптимальної нейронної мережі та отримані результаті для обох завдань. Оцінити вплив пакетної нормалізації на підсумковий результат.

## Контрольні запитання

1. Що являє собою пакетна нормалізація?
2. Як впливає використання пакетної нормалізації на процес навчання нейронної мережі?
3. Які параметри використовуються для пакетної нормалізації під час тестування моделі?

# 11: СПОСОБИ ПОБУДОВИ МОДЕЛІ У PYTORCH

**Мета роботи –** засвоїти способи побудови моделі у PyTorch.

## Теоретичні відомості

У цій практичній роботі ми побачимо, як використовувати три основні будівельні блоки PyTorch для побудови нейронних мереж: Module, Sequences і ModuleList. Ми почнемо з прикладу і ітеративно зробимо його кращим. Усі ці три класи містяться в модулі torch.nn

```
import torch.nn as nn

# nn.Module
# nn.Sequential
# nn.Module
```

### 1. Module

nn.Module є основним будівельним блоком, він визначає базовий клас для всієї нейронної мережі, і ви обов’язково повинні успадкуватися від нього.

*Довідка*: PyTorch має functional аналоги кожного nn модуля. Під functional ми маємо на увазі "відсутність внутрішнього стану" або те, що "значення вихідних даних визначаються виключно і повністю аргументами введених значень". Дійсно, torch.nn.functional надає багато тих же модулів, які ви знайдете в nn, але з усіма можливими параметрами які переміщені, як аргумент виклику функції. Наприклад, функціональним аналогом nn.Linear є nn.functional.linear, який є функцією, яка приймає (вхід, вага, зміщення = None). Параметри ваги та зміщення є аргументами функції.

Створимо класичний класифікатор CNN, як приклад:

```
import torch.nn.functional as F

class MyCNNClassifier(nn.Module):
    def __init__(self, in_c, n_classes):
        super().__init__()
        self.conv1 = nn.Conv2d(in_c, 32, kernel_size=3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(32)

        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(32)

        self.fc1 = nn.Linear(32 * 28 * 28, 1024)
        self.fc2 = nn.Linear(1024, n_classes)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = self.bn2(x)
        x = F.relu(x)
        x = x.view(x.size(0), -1) # flat, робимо масив одновимірним перед повнозв'язним шаром
        x = self.fc1(x)
        x = F.sigmoid(x)
        x = self.fc2(x)
        return x
model = MyCNNClassifier(1, 10)
print(model)
```

*MyCNNClassifier(*

*(conv1): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))*

*(bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)*

*(conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))*

*(bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)*

*(fc1): Linear(in_features=25088, out_features=1024, bias=True)*

*(fc2): Linear(in_features=1024, out_features=10, bias=True)*

*)*

Це дуже простий класифікатор, який використовує два шари з 3x3 convs + batchnorm + relu та частину декодування з двома лінійними шарами. Тут є дві проблеми.

Якщо ми хочемо додати шар, нам доведеться знову записати багато коду у __init__ та у forward функцію. Також, якщо у нас є якийсь загальний блок, який ми хочемо використовувати в іншій моделі, наприклад 3x3 conv + batchnorm + relu, нам доведеться писати його ще раз.

**Sequential: stack and merge layers**

Sequential – це контейнер модулів, який можна складати разом (stacked together) і запускати одночасно. Ви можете помітити, що ми повинні зберігати в self все. Ми можемо використовувати Sequential для вдосконалення нашого коду:

```
class MyCNNClassifier(nn.Module):
  def __init__(self, in_c, n_classes):
    super().__init__()
    self.conv_block1 = nn.Sequential(
      nn.Conv2d(in_c, 32, kernel_size=3, stride=1, padding=1),
```

```
            nn.BatchNorm2d(32),
            nn.ReLU()
        )
        self.conv_block2 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU()
        )
        self.decoder = nn.Sequential(
            nn.Linear(32 * 28 * 28, 1024),
            nn.Sigmoid(),
            nn.Linear(1024, n_classes)
        )
    def forward(self, x):
        x = self.conv_block1(x)
        x = self.conv_block2(x)
        x = x.view(x.size(0), -1) # flat
        x = self.decoder(x)
        return x
model = MyCNNClassifier(1, 10)
print(model)

MyCNNClassifier(
  (conv_block1): Sequential(
    (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU()
  )
  (conv_block2): Sequential(
```

*(0): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))*

*(1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)*

*(2): ReLU()*

*)*

*(decoder): Sequential(*

*(0): Linear(in_features=25088, out_features=1024, bias=True)*

*(1): Sigmoid()*

*(2): Linear(in_features=1024, out_features=10, bias=True)*

*)*

*)*

Вже набагато краще. Зауважимо, що conv_block1 та conv_block2 виглядають майже однаково. Ми могли б створити функцію, яка перезаписує nn.Sequences, щоб ще більше спростити код:

```
def conv_block(in_f, out_f, *args, **kwargs):
    return nn.Sequential(
        nn.Conv2d(in_f, out_f, *args, **kwargs),
        nn.BatchNorm2d(out_f),
        nn.ReLU())
```

Тоді ми можемо просто викликати цю функцію в нашому Модулі:

```
class MyCNNClassifier(nn.Module):
    def __init__(self, in_c, n_classes):
        super().__init__()
        self.conv_block1 = conv_block(in_c, 32, kernel_size=3, padding=1)
        self.conv_block2 = conv_block(32, 64, kernel_size=3, padding=1)
        self.decoder = nn.Sequential(
            nn.Linear(32 * 28 * 28, 1024),
```

```
            nn.Sigmoid(),
            nn.Linear(1024, n_classes))
        def forward(self, x):
            x = self.conv_block1(x)
            x = self.conv_block2(x)
            x = x.view(x.size(0), -1) # flat
            x = self.decoder(x)
            return x
    model = MyCNNClassifier(1, 10)
    print(model)

    MyCNNClassifier(
      (conv_block1): Sequential(
        (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): ReLU()
      )
      (conv_block2): Sequential(
        (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): ReLU()
      )
      (decoder): Sequential(
        (0): Linear(in_features=25088, out_features=1024, bias=True)
        (1): Sigmoid()
        (2): Linear(in_features=1024, out_features=10, bias=True)
      )
    )
```

Вийшло ще краще. Тепер conv_block1 та conv_block2 майже однакові! Ми можемо об'єднати їх за допомогою nn.Sequences:

```
class MyCNNClassifier(nn.Module):
    def __init__(self, in_c, n_classes):
        super().__init__()
        self.encoder = nn.Sequential(
            conv_block(in_c, 32, kernel_size=3, padding=1),
            conv_block(32, 64, kernel_size=3, padding=1)   )

        self.decoder = nn.Sequential(
            nn.Linear(32 * 28 * 28, 1024),
            nn.Sigmoid(),
            nn.Linear(1024, n_classes)  )

    def forward(self, x):
        x = self.encoder(x)
        x = x.view(x.size(0), -1) # flat
        x = self.decoder(x)
        return x
model = MyCNNClassifier(1, 10)
print(model)

MyCNNClassifier(
  (encoder): Sequential(
    (0): Sequential(
      (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU()
```

*)*

*(1): Sequential(*

*(0): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))*

*(1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)*

*(2): ReLU()*

*)*

*)*

*(decoder): Sequential(*

*(0): Linear(in_features=25088, out_features=1024, bias=True)*

*(1): Sigmoid()*

*(2): Linear(in_features=1024, out_features=10, bias=True)*

*)*

*)*

self.encoder тепер вміщує обидва conv_block. Ми розв'язали логіку для нашої моделі та полегшуємо її читання та повторне використання. Нашу функцію conv_block можна тепер імпортувати та використовувати в іншій моделі.

**Dynamic Sequential: створити кілька шарів одночасно**

Що робити, якщо ми хочемо додати нові шари у self.encoder? Робити це «в лоб» не дуже зручно:

```
self.encoder = nn.Sequential(
        conv_block(in_c, 32, kernel_size=3, padding=1),
        conv_block(32, 64, kernel_size=3, padding=1),
        conv_block(64, 128, kernel_size=3, padding=1),
        conv_block(128, 256, kernel_size=3, padding=1),       )
```

Було б добре, якби ми могли визначити розміри як масив і автоматично створити всі шари, не записуючи кожен з них? На щастя, ми можемо створити масив і передати його у Sequential:

```
class MyCNNClassifier(nn.Module):
    def __init__(self, in_c, n_classes):
        super().__init__()
        self.enc_sizes = [in_c, 32, 64]

        conv_blocks = [conv_block(in_f, out_f, kernel_size=3, padding=1)
                       for in_f, out_f in zip(self.enc_sizes, self.enc_sizes[1:])]
        self.encoder = nn.Sequential(*conv_blocks)
        self.decoder = nn.Sequential(
            nn.Linear(32 * 28 * 28, 1024),
            nn.Sigmoid(),
            nn.Linear(1024, n_classes))
    def forward(self, x):
        x = self.encoder(x)
        x = x.view(x.size(0), -1) # flat
        x = self.decoder(x)
        return x
model = MyCNNClassifier(1, 10)
print(model)

MyCNNClassifier(
  (encoder): Sequential(
    (0): Sequential(
      (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

*(2): ReLU()*

*)*

*(1): Sequential(*

*(0): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))*

*(1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)*

*(2): ReLU()*

*)*

*)*

*(decoder): Sequential(*

*(0): Linear(in_features=25088, out_features=1024, bias=True)*

*(1): Sigmoid()*

*(2): Linear(in_features=1024, out_features=10, bias=True)*

*)*

*)*

Давайте розберемо що робиться. Ми створили масив self.enc_size, який містить розміри нашого кодувальника. Потім ми створюємо масив conv_blocks шляхом ітерування по розмірах. Оскільки ми повинні надати обидва значення (вхідний розмір і вихідний розмір для кожного шару), ми робимо zip масиву розмірів з самим собою, перемістивши його на один.

Щоб зрозуміти, подивіться наступний приклад:

```
sizes = [1, 32, 64]
for in_f,out_f in zip(sizes, sizes[1:]):
   print(in_f,out_f)
1 32
32 64
```

Далі, оскільки Sequential не приймає список, ми розкладаємо його за допомогою оператора *.

Тепер, якщо ми просто хочемо додати розмір, ми можемо легко додати нове число до списку. Це є загальна практика робити параметр розміру:

```
class MyCNNClassifier(nn.Module):
    def __init__(self, in_c, enc_sizes, n_classes):
        super().__init__()
        self.enc_sizes = [in_c, *enc_sizes]
        conv_blokcs = [conv_block(in_f, out_f, kernel_size=3, padding=1)
                       for in_f, out_f in zip(self.enc_sizes, self.enc_sizes[1:])]
        self.encoder = nn.Sequential(*conv_blokcs)

        self.decoder = nn.Sequential(
            nn.Linear(32 * 28 * 28, 1024),
            nn.Sigmoid(),
            nn.Linear(1024, n_classes)       )
    def forward(self, x):
        x = self.encoder(x)
        x = x.view(x.size(0), -1) # flat
        x = self.decoder(x)
        return x
model = MyCNNClassifier(1, [32,64, 128], 10)
print(model)

MyCNNClassifier(
  (encoder): Sequential(
    (0): Sequential(
      (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
      (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU()
    )
    (1): Sequential(
      (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU()
    )
    (2): Sequential(
      (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU()
    )
  )
  (decoder): Sequential(
    (0): Linear(in_features=25088, out_features=1024, bias=True)
    (1): Sigmoid()
    (2): Linear(in_features=1024, out_features=10, bias=True)
  )
)
```

Ми можемо зробити те  саме із частиною decoder:

```
def dec_block(in_f, out_f):
    return nn.Sequential(
        nn.Linear(in_f, out_f),
        nn.Sigmoid()    )
```

```
class MyCNNClassifier(nn.Module):
    def __init__(self, in_c, enc_sizes, dec_sizes,  n_classes):
        super().__init__()
        self.enc_sizes = [in_c, *enc_sizes]
        self.dec_sizes = [32 * 28 * 28, *dec_sizes]
        conv_blokcs = [conv_block(in_f, out_f, kernel_size=3, padding=1)
                for in_f, out_f in zip(self.enc_sizes, self.enc_sizes[1:])]
        self.encoder = nn.Sequential(*conv_blokcs)
        dec_blocks = [dec_block(in_f, out_f)
                for in_f, out_f in zip(self.dec_sizes, self.dec_sizes[1:])]

        self.decoder = nn.Sequential(*dec_blocks)

        self.last = nn.Linear(self.dec_sizes[-1], n_classes)

    def forward(self, x):
        x = self.encoder(x)
        x = x.view(x.size(0), -1) # flat
        x = self.decoder(x)
        return x
model = MyCNNClassifier(1, [32,64], [1024, 512], 10)
print(model)

MyCNNClassifier(
  (encoder): Sequential(
    (0): Sequential(
      (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU()
```

```
    )
    (1): Sequential(
      (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU()
    )
  )
  (decoder): Sequential(
    (0): Sequential(
      (0): Linear(in_features=25088, out_features=1024, bias=True)
      (1): Sigmoid()
    )
    (1): Sequential(
      (0): Linear(in_features=1024, out_features=512, bias=True)
      (1): Sigmoid()
    )
  )
  (last): Linear(in_features=512, out_features=10, bias=True)
)
```

Дотримуючись того самого шаблону, створюємо новий блок для частини decoding, linear + sigmoid, і передаємо масив з розмірами. Нам довелося додати self.last, оскільки ми не хочемо активувати вихід.

Тепер ми можемо навіть розбити свою модель надвоє! Encoder + Decoder:

```
class MyEncoder(nn.Module):
    def __init__(self, enc_sizes):
        super().__init__()
```

```
            self.conv_blokcs = nn.Sequential(*[conv_block(in_f, out_f,
kernel_size=3, padding=1)
                    for in_f, out_f in zip(enc_sizes, enc_sizes[1:])])
            def forward(self, x):
                return self.conv_blokcs(x)

        class MyDecoder(nn.Module):
            def __init__(self, dec_sizes, n_classes):
                super().__init__()
                self.dec_blocks = nn.Sequential(*[dec_block(in_f, out_f)
                    for in_f, out_f in zip(dec_sizes, dec_sizes[1:])])
                self.last = nn.Linear(dec_sizes[-1], n_classes)

            def forward(self, x):
                return self.dec_blocks()

        class MyCNNClassifier(nn.Module):
            def __init__(self, in_c, enc_sizes, dec_sizes,  n_classes):
                super().__init__()
                self.enc_sizes = [in_c, *enc_sizes]
                self.dec_sizes = [32 * 28 * 28, *dec_sizes]

                self.encoder = MyEncoder(self.enc_sizes)

                self.decoder = MyDecoder(dec_sizes, n_classes)

            def forward(self, x):
                x = self.encoder(x)
                x = x.flatten(1) # flat
                x = self.decoder(x)
```

```
        return x
    model = MyCNNClassifier(1, [32,64], [1024, 512], 10)
    print(model)
    MyCNNClassifier(
      (encoder): MyEncoder(
        (conv_blokcs): Sequential(
          (0): Sequential(
            (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): ReLU()
          )
          (1): Sequential(
            (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): ReLU()
          )
        )
      )
      (decoder): MyDecoder(
        (dec_blocks): Sequential(
          (0): Sequential(
            (0): Linear(in_features=1024, out_features=512, bias=True)
            (1): Sigmoid()
          )
        )
        (last): Linear(in_features=512, out_features=10, bias=True)
      )
    )
```

Майте на увазі, що MyEncoder і MyDecoder також можуть бути функціями, які повертають nn.Sequences. Краще використовувати перший патерн  для моделей, а другий для будівельних блоків. Зануривши наш модуль у підмодулі, легше поділитися кодом, налагодити його та протестувати його.

**ModuleList : коли нам потрібно ітеруватися**

ModuleList дозволяє зберігати Module як список. Це може бути корисно, коли вам потрібно перебирати шари і зберігати / використовувати деяку інформацію, наприклад, в U-net.

Основна відмінність від Sequential полягає в тому, що ModuleList не має методу forward, тому внутрішні шари не з'єднані. Припускаючи, що нам потрібен кожен вихід кожного шару в декодері, ми можемо зберігати його:

```
class MyModule(nn.Module):
    def __init__(self, sizes):
        super().__init__()
        self.layers = nn.ModuleList([nn.Linear(in_f, out_f) for in_f, out_f in zip(sizes, sizes[1:])])
        self.trace = []

    def forward(self,x):
        for layer in self.layers:
            x = layer(x)
            self.trace.append(x)
        return x
model = MyModule([1, 16, 32])
import torch

model(torch.rand((4,1)))
```

*[print(trace.shape) for trace in model.trace]*

*torch.Size([4, 16])*

*torch.Size([4, 32])*

*[None, None]*

**ModuleDict: коли нам потрібно вибирати**

Що робити, якщо ми хочемо перейти на LeakyRelu в нашому conv_block? Ми можемо використовувати ModuleDict для створення словника модуля і динамічно перемикати модуль, коли нам потрібно:

```
def conv_block(in_f, out_f, activation='relu', *args, **kwargs):
   activations = nn.ModuleDict([
            ['lrelu', nn.LeakyReLU()],
            ['relu', nn.ReLU()]    ])
   return nn.Sequential(
      nn.Conv2d(in_f, out_f, *args, **kwargs),
      nn.BatchNorm2d(out_f),
      activations[activation]
   )
print(conv_block(1, 32,'lrelu', kernel_size=3, padding=1))
print(conv_block(1, 32,'relu', kernel_size=3, padding=1))
Sequential(
  (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (2): LeakyReLU(negative_slope=0.01)
)
Sequential(
  (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

*(1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)*

*(2): ReLU()*

*)*

Остаточна реалізація. Давайте все обернемо у модулі:

```
def conv_block(in_f, out_f, activation='relu', *args, **kwargs):
    activations = nn.ModuleDict([
            ['lrelu', nn.LeakyReLU()],
            ['relu', nn.ReLU()]
    ])

    return nn.Sequential(
        nn.Conv2d(in_f, out_f, *args, **kwargs),
        nn.BatchNorm2d(out_f),
        activations[activation]
    )

def dec_block(in_f, out_f):
    return nn.Sequential(
        nn.Linear(in_f, out_f),
        nn.Sigmoid()
    )

class MyEncoder(nn.Module):
    def __init__(self, enc_sizes, *args, **kwargs):
        super().__init__()
        self.conv_blokcs = nn.Sequential(*[conv_block(in_f, out_f, kernel_size=3, padding=1, *args, **kwargs)
```

```
                for in_f, out_f in zip(enc_sizes, enc_sizes[1:])])

        def forward(self, x):
            return self.conv_blokcs(x)

    class MyDecoder(nn.Module):
        def __init__(self, dec_sizes, n_classes):
            super().__init__()
            self.dec_blocks = nn.Sequential(*[dec_block(in_f, out_f)
                    for in_f, out_f in zip(dec_sizes, dec_sizes[1:])])
            self.last = nn.Linear(dec_sizes[-1], n_classes)
        def forward(self, x):
            return self.dec_blocks()
    class MyCNNClassifier(nn.Module):
        def __init__(self, in_c, enc_sizes, dec_sizes, n_classes,
activation='relu'):
            super().__init__()
            self.enc_sizes = [in_c, *enc_sizes]
            self.dec_sizes = [32 * 28 * 28, *dec_sizes]
            self.encoder = MyEncoder(self.enc_sizes, activation=activation)
            self.decoder = MyDecoder(dec_sizes, n_classes)
        def forward(self, x):
            x = self.encoder(x)
            x = x.flatten(1) # flat
            x = self.decoder(x)
            return x
    model = MyCNNClassifier(1, [32,64], [1024, 512], 10, activation='lrelu')
    print(model)
    MyCNNClassifier(
      (encoder): MyEncoder(
```

*(conv_blokcs): Sequential(*

*(0): Sequential(*

*(0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))*

*(1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)*

*(2): LeakyReLU(negative_slope=0.01)*

*)*

*(1): Sequential(*

*(0): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))*

*(1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)*

*(2): LeakyReLU(negative_slope=0.01)*

*)*

*)*

*)*

*(decoder): MyDecoder(*

*(dec_blocks): Sequential(*

*(0): Sequential(*

*(0): Linear(in_features=1024, out_features=512, bias=True)*

*(1): Sigmoid()))*

*(last): Linear(in_features=512, out_features=10, bias=True)))*

Отже, підсумовуючи:

• Використовуйте Module, коли у вас є великий блок, який складається з декількох менших блоків

• Використовуйте Sequential, коли ви хочете створити невеликий блок із шарів

• Використовуйте ModuleList, коли вам потрібно переглядати деякі шари або будівельні блоки та робити щось

• Використовуйте ModuleDict, коли вам потрібно параметризувати деякі блоки вашої моделі, наприклад функцію активації

## Практична частина

1. Адаптуйте код із практичної роботи № 10, використовуючи модульний підхід PyTorch. Ви можете просто взяти дану в теоретичній частині нейромережу, та адаптувати вхід та вихід (розміри) під необхідні набори даних.

## Висновки

У висновках порівняйте результат роботи нейронної мережі із минулої практичної роботи та даної, розробленої з використання модельного підходу PyTorch.

## Контрольні запитання

1. Що являє собою блок Module PyTorch?
2. Що являє собою ModuleDict, Sequential та ModuleList у PyTorch?

# 12: TRANSFER LEARNING (ЧАСТИНА І). АРХІТЕКТУРА RESNET18, RESNET20, CIFARNET

**Мета роботи –** ознайомитися з архітектурами згорткових нейронних мереж та засвоїти концепцію transfer learning.

## Теоретичні відомості

Отже, ми зрозуміли базові принципи побудови власних архітектур, однак зараз рідко хто трудиться над конструюванням нової архітектури для популярних завдань. Переможець конкурсу класифікації на ImageNet щороку змінюється, проте, найпопулярніший в побутовому використанні варіант – це ResNet.

Модифікацій даної мережі досить багато: ResNet13, ResNet18, ResNet34, ResNet50, ResNet101, ResNet152, які були створені для ImageNet. А так же є ResNet20, ResNet32, ResNet44, ResNet56, ResNet110, ResNet1202 для датасета CIFAR10. Пропонуємо перевірити ці нейромережі на міцність:

Почнемо із підготовки даних для CIFAR10 (ми його трохи перепишемо)

**Обробка даних.** Завантажимо необхідні бібліотеки

```
import torch
import torchvision.datasets
import random
import numpy as np
import time
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import seaborn as sns
```

Завантажимо набор даних CIFAR10:

```
CIFAR_train = torchvision.datasets.CIFAR10('./', download=True, train=True)
CIFAR_test = torchvision.datasets.CIFAR10('./', download=True, train=False)

X_train = torch.FloatTensor(CIFAR_train.data)
y_train = torch.LongTensor(CIFAR_train.targets)
X_test = torch.FloatTensor(CIFAR_test.data)
y_test = torch.LongTensor(CIFAR_test.targets)
Нормуємо наші дані
X_train /= 255.
X_test /= 255.
```

Подивимося на наші дані:

```
import matplotlib.pyplot as plt
plt.figure(figsize=(20,2))
for i in range(10):
    plt.subplot(1, 10, i+1)
    plt.imshow(X_train[i])
    print(y_train[i], end=' ')
```

Реорганізуємо наш тензор даних для роботи у pytorch. Переведемо його до виду [Число зображень, число каналів, висота зображення, ширина зображення].

```
X_train = X_train.permute(0, 3, 1, 2)
X_test = X_test.permute(0, 3, 1, 2)
```

**Процес навчання**

Зазначимо, що зараз ми передаємо на вхід функції train всі необхідні змінні: batch_size – розмір батчу, epoch_num – кількість епох навчання, epoch_info_show – через скільки епох виводити точність роботи моделі на екран якщо флаг verbose встановлено, save_net_state – чи повертати із функції навчені стани моделі, weight_decay значення l2 нормалізації. Таким чином, наша функція буде універсальною для різних архітектур, які будуть змінюватися першим параметром.

Далі, як завжди, йдуть вибір пристрою, визначення функції втрат та встановлення оптимізатору. Переводимо все, що можна для розрахунку на пристрої. Вводимо змінну t для виміру часу роботи. Визначаємо списки для зберігання точності та значення втрат на тесті. Запускаємо цикл по епохам та Батчам. Робимо відповідні кроки навчання, включивши train state. Вмикаємо eval state перед розрахунками на тесті. Обгортка "with torch.no_grad ()" тимчасово встановила всі прапори req_grad на false. Ми використовуємо torch.no_grad, щоб вказати PyTorch, що ми не слід відстежувати, обчислювати чи змінювати градієнти під час тестування.

Можна відмітити, що model.eval () та torch.no_grad () роблять дві різні функції:

1) model.eval() сповістить всі ваші шари, що ви перебуваєте в режимі eval, таким чином шари batchnorm або dropout працюватимуть в режимі eval замість тренувального режиму.

2) torch.no_grad() впливає на двигун автоградієнта та вимикає його. Це зменшить використання пам'яті та пришвидшить обчислення, але ви не зможете робити резервну копію (чого не потрібно в сценарії eval).

Робимо forward на тесті, записуємо точність та втрати до списків. Отримуємо стани мережі, якщо нам потрібно буде їх повернути (за прапором save_net_state) - net_state = net.state_dict(). Видаляємо мережу – звільняємо пам’ять. (del net) Повертаємо результат. Підготовлюємо

словники для різних мереж. Таким чином, код навчання виглядає наступним чином:

```
def train(net, X_train, y_train, X_test, y_test, batch_size=256, epoch_num=50, epoch_info_show=10,\
        weight_decay=0, save_net_state=False, verbose=True):
    device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
    net = net.to(device)
    loss = torch.nn.CrossEntropyLoss().to(device)
    optimizer = torch.optim.Adam(net.parameters(), lr=1.0e-3, \
                    weight_decay=weight_decay)
    t = time.time()
    test_accuracy_history = []
    test_loss_history = []
    X_test = X_test.to(device)
    y_test = y_test.to(device)

    for epoch in range(1, epoch_num+1):
      order = np.random.permutation(len(X_train))
      for start_index in range(0, len(X_train), batch_size):
        optimizer.zero_grad()
        net.train()
        batch_indexes = order[start_index:start_index+batch_size]
        X_batch = X_train[batch_indexes].to(device)
        y_batch = y_train[batch_indexes].to(device)

        preds = net.forward(X_batch)
        loss_value = loss(preds, y_batch)
        loss_value.backward()
        optimizer.step()
      net.eval()
```

```
            with torch.no_grad():
              test_preds = net.forward(X_test)
              loss_value = loss(test_preds, y_test).item()
              test_loss_history.append(loss_value)
              accuracy = (test_preds.argmax(dim=1) == y_test).float().mean().item()
              test_accuracy_history.append(accuracy)
              if verbose:
                if epoch % epoch_info_show == 0:
                if device.type == 'cuda:0':
                        print('Train Epoch: {} Time: {} Accuracy: {}, GPU_Mem_alloc: {} GPU_Mem_cashed: {}'.format(epoch, time.strftime("%H:%M:%S", time.gmtime(time.time() - t)), accuracy, torch.cuda.memory_allocated(), torch.cuda.memory_cached()))
                else:
                        print('Train Epoch: {} Time: {} Accuracy: {}'.format(epoch, time.strftime("%H:%M:%S", time.gmtime(time.time() - t)), accuracy))
            net_state = net.state_dict()
            del net
            if save_net_state:
              return test_accuracy_history, test_loss_history, net_state
            else:
              return test_accuracy_history, test_loss_history
        accuracies = {}
        losses = {}
```

**Побудова графіків**

У змінній net_list будуть записані назви всіх зроблених експериментів (мереж). Потім, використовуючи її ми будуємо точність та втрати з відповідним словників:

```
sns.set_style("darkgrid")
sns.set(rc={'figure.figsize':(15, 6)})

def acc_loss_graph(accuracies, losses, net_list, save_file_name='plot.png', download=False):
  fig, (ax1, ax2) = plt.subplots(1, 2)

  for experiment_id in net_list:
    ax1.plot(accuracies[experiment_id], label=experiment_id)
  ax1.legend()
  ax1.set_title('Validation Accuracy')
  fig.tight_layout()

  for experiment_id in net_list:
    ax2.plot(losses[experiment_id], label=experiment_id)
  ax2.legend()
  ax2.set_title('Validation Loss');

  fig.tight_layout()

  if download:
   fig.savefig(save_file_name)
```

**CIFARNet**

Для початку відтворимо клас CIFARNet з минулої практичної роботи:

```
class CIFARNet(torch.nn.Module):
  def __init__(self):
    super(CIFARNet, self).__init__()
```

```
        self.batch_norm0 = torch.nn.BatchNorm2d(3)

        self.conv1 = torch.nn.Conv2d(3, 16, 3, padding=1)
        self.act1  = torch.nn.ReLU()
        self.batch_norm1 = torch.nn.BatchNorm2d(16)
        self.pool1 = torch.nn.MaxPool2d(2, 2)

        self.conv2 = torch.nn.Conv2d(16, 32, 3, padding=1)
        self.act2  = torch.nn.ReLU()
        self.batch_norm2 = torch.nn.BatchNorm2d(32)
        self.pool2 = torch.nn.MaxPool2d(2, 2)

        self.conv3 = torch.nn.Conv2d(32, 64, 3, padding=1)
        self.act3  = torch.nn.ReLU()
        self.batch_norm3 = torch.nn.BatchNorm2d(64)

        self.fc1   = torch.nn.Linear(8 * 8 * 64, 256)
        self.act4  = torch.nn.Tanh()
        self.batch_norm4 = torch.nn.BatchNorm1d(256)

        self.fc2   = torch.nn.Linear(256, 64)
        self.act5  = torch.nn.Tanh()
        self.batch_norm5 = torch.nn.BatchNorm1d(64)

        self.fc3   = torch.nn.Linear(64, 10)

    def forward(self, x):
        x = self.batch_norm0(x)
        x = self.conv1(x)
        x = self.act1(x)
```

```
        x = self.batch_norm1(x)
        x = self.pool1(x)

        x = self.conv2(x)
        x = self.act2(x)
        x = self.batch_norm2(x)
        x = self.pool2(x)

        x = self.conv3(x)
        x = self.act3(x)
        x = self.batch_norm3(x)

        x = x.view(x.size(0), x.size(1) * x.size(2) * x.size(3))
        x = self.fc1(x)
        x = self.act4(x)
        x = self.batch_norm4(x)
        x = self.fc2(x)
        x = self.act5(x)
        x = self.batch_norm5(x)
        x = self.fc3(x)

        return x
```

Отримавши CIFARNet, навчимо її:

```
accuracies['cifar_net'], losses['cifar_net'] = \
    train(CIFARNet(), X_train, y_train, X_test, y_test)
```

**Transfer learning (TL)**

Transfer Learning може використовувати накопичений при вирішенні однієї задачі досвід для вирішення іншої, аналогічної проблеми. Нейромережа спочатку навчається на більших об'ємах даних, а потім – на цільовому наборі донавчається (рисунок 12.1).

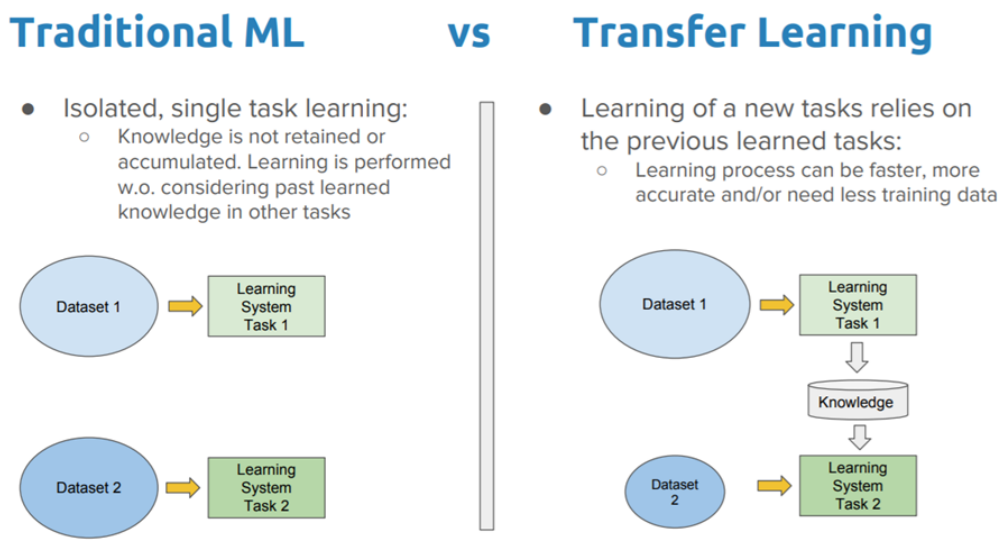



Рисунок 12.1 – Порівняння звичайного навчання та transfer learning

Якщо перед нами постає завдання розпізнавання зображень, можна скористатися готовим сервісом. Однак, якщо потрібно навчити модель на власному наборі даних, то доведеться робити це самостійно.

Для таких типових задач, як класифікація зображень, можна скористатися готовою архітектурою (AlexNet, VGG, Inception, ResNet і т.д.) і навчити нейромережу на своїх даних. Реалізації таких мереж за допомогою різних фреймворків (в PyTorch також) вже існують, так що на даному етапі можна використовувати одну з них як чорний ящик, не вникаючи глибоко в принцип її роботи.

Однак, глибокі нейронні мережі вимогливі до великих обсягів даних для збіжності навчання. І часто в нашій задачі недостатньо даних для того, щоб добре натренувати всі шари нейромережі. Transfer Learning вирішує цю проблему.

Нейронні мережі, які використовуються для класифікації, як правило, містять N вихідних нейронів в останньому шарі, де N – це кількість класів. Такий вихідний вектор трактується як набір ймовірностей приналежності до класу. У Вашій задачі кількість класів може відрізнятися від того, яке було у датасеті тієї архітектури, яку Ви хочете використати. У такому випадку нам доведеться повністю викинути цей останній шар і поставити новий, з потрібною кількістю вихідних нейронів. Якщо ж ви хочете використати архітектуру на тому ж датасеті, то нічого змінювати не потрібно (рисунок 12.2).

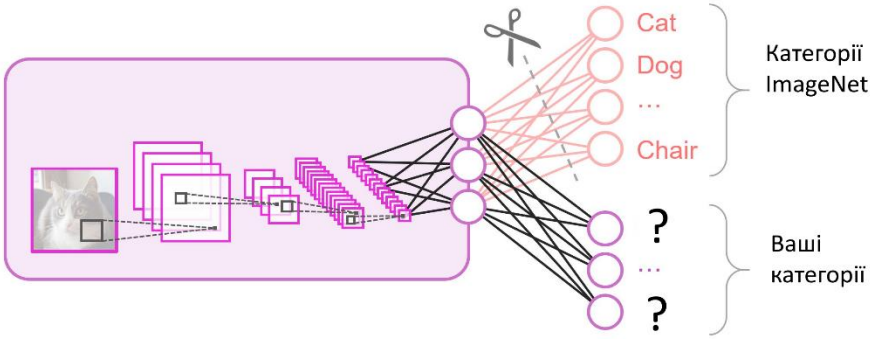



Рисунок 12.2 – Базовий принцип transfer learning

Найчастіше в кінці класифікаційних мереж використовуються повнозв'язні шари. Так як ми замінили цей шар, використовувати попередньо навчені ваги для нього вже не вийде. Доведеться тренувати його з нуля, ініціалізувати його ваги випадковими значеннями. Ваги для всіх інших шарів ми завантажуємо з попередньо навченої архітектури.

Загалом виділяють 2 базові стратегії донавчання моделі.

1) end-to-end – коли тренують всю (або вибіркові шари) мережу з кінця в кінець, а попередньо навчені ваги не фіксують, щоб дати їм трохи скоректуватися і підлаштуватися під наші дані. Такий процес називається тонким налаштуванням (fine-tuning або Fine Tuning Off-the-shelf Pre-trained Models).

2) Off-the-shelf Pre-trained Models as Feature Extractors – коли тренують тільки останній повнозв'язний шар, котрий ми змінили. Основна ідея – просто скористатися натренованими вагами шарів попередньо підготовленої моделі для отримання ознак, але не оновлювати ваги шарів моделі під час тренінгу з новими даними для нового завдання.

Для вирішення завдання методом TL нам знадобляться наступні структурні компоненти (рисунок 12.3):

1) Опис моделі нейромережі
2) «Пайплайн» (загальна структура) навчання
3) «Інференс пайплайн»
4) Натреновані ваги для цієї моделі
5) Дані для навчання та валідації

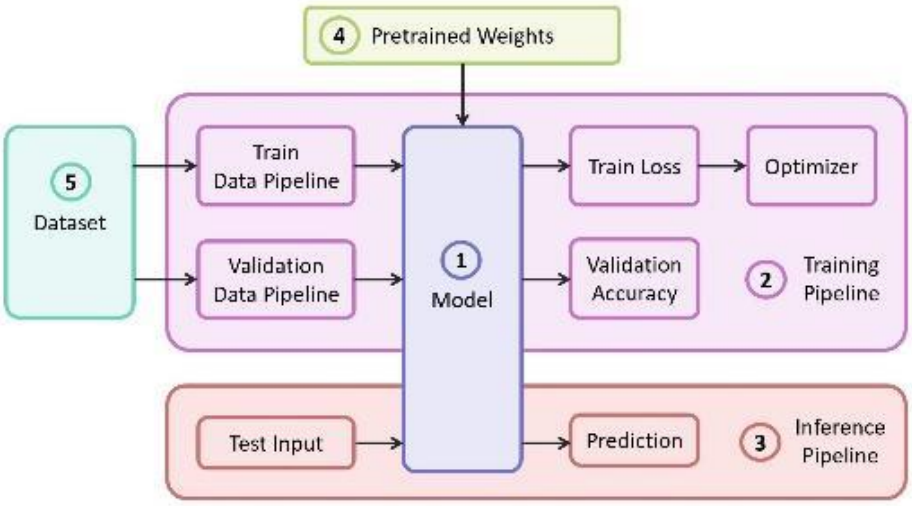



Рисунок 12.3 – Компоненти transfer learning

**Види глибокого TL**

Література про донавчання пережила багато ітерацій, і терміни, пов'язані з цим, використовувались вільно і часто взаємозамінно. Отже, іноді незрозуміло як розрізняти *transfer learning, domain adaptation ma multi-task learning*. Все це пов'язані речі і вони намагаються вирішити подібні проблеми. Загалом, ви завжди повинні думати про transfer learning як

загальну концепцію чи принцип, де ми спробуємо вирішити цільову задачу, використовуючи знання із відповідного домену.

**Адаптація домену (domain adaptation)**

Адаптація домену – це зазвичай сценарії, коли граничні ймовірності між вихідним та цільовим доменами різні, наприклад, $P(X_s) \neq P(X_t)$. У розподілі даних джерела та цільових доменів існує властивий зсув або дрейф, який вимагає налаштування для передачі навчання. Наприклад, набір даних відгуків про фільми, позначений як позитивний чи негативний, буде відрізнятися від набору даних відгуків про продукт. Класифікатор, навчений на даних відгуків про перегляд фільмів, побачив би інший розподіл, якщо його використовувати для класифікації оглядів продуктів. Таким чином, різні методи адаптації домену використовуються при TL в цих сценаріях.

**Плутанина домену (domain confusion)**

Варто зауважити, що різні шари в глибокій навчальній мережі захоплюють різні набори функцій. Ми можемо скористатися цим фактом, щоб дізнатись про функції, що інваріантні домену та покращити їх передачу в різні домени. Замість того, щоб дозволяти моделі вивчати будь-яке представлення, ми підштовхуємо представлення обох доменів буди максимально схожими. Цього можна досягти, застосувавши певні етапи попередньої обробки безпосередньо до самих представлень – основна ідея цієї методики – додати ще одну мету до вихідної моделі, заохочувати подібність, плутаючи сам домен.

**Багатозадачне навчання (multi-task learning)**

Багатозадачне навчання – дещо інший напрям у TL. У випадку багатозадачного навчання кілька завдань вивчаються одночасно, не розрізняючи джерело та цілі. У цьому випадку “учень” отримує інформацію про декілька завдань одночасно, порівняно з TL, де “учень” спочатку не має уявлення про цільове завдання (12.4).

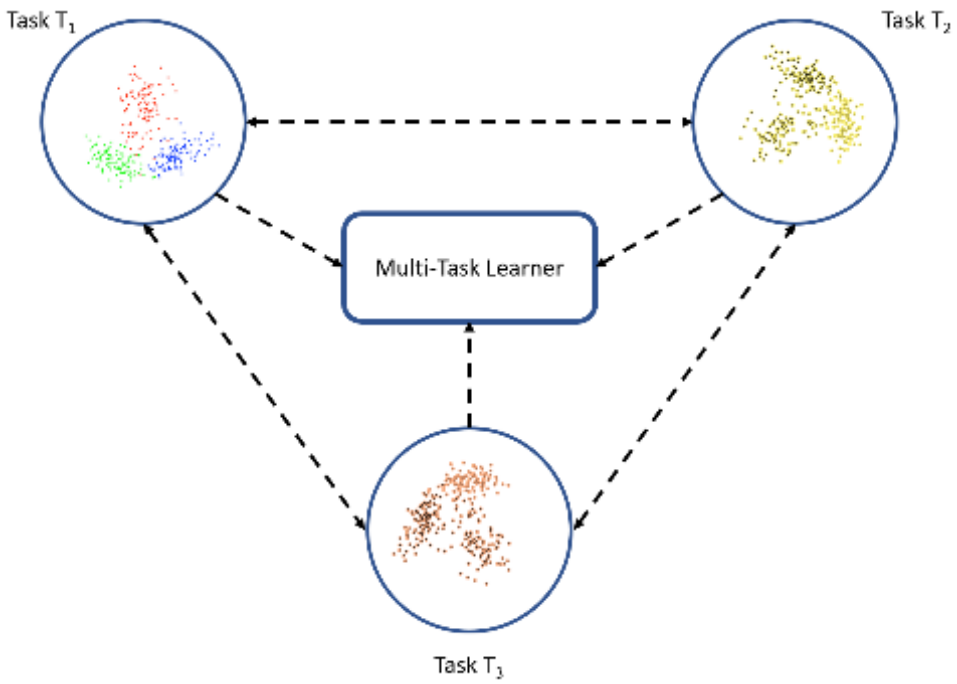



Рисунок 12.4 – Схематичне представлення багатозадачного навчання

**One-shot Learning**

Системи глибокого навчання за своєю природою є «голодними» до даних (data-hungry), тому їм потрібно багато прикладів для навчання, щоб вивчити ваги. Це один із обмежуючих аспектів глибоких нейронних мереж, хоча такого не відбувається у навчанні людини. Наприклад, щойно дитині показують, як виглядає яблуко, вона може легко визначити різні сорти яблук (з одним або кількома прикладами навчання); це не так з ML та алгоритмами глибокого навчання. Одноразове навчання – це варіант трансферного навчання (TL), де ми намагаємось зробити необхідний результат на основі лише одного чи кількох прикладів навчання. Це по суті корисно в реальних сценаріях, де неможливо мати мічені дані для кожного можливого класу (якщо це завдання класифікації), а також у сценаріях, коли нові класи можна додавати часто. Базова стаття ‘One Shot Learning of Object Categories’ ввела термін «One-shot Learning». У цій статті було представлено варіацію Bayesian framework для навчання для категоризації об'єктів. З цього часу цей підхід удосконалено та застосовується за допомогою систем глибокого навчання.

**Zero-shot Learning.**

Навчання з нуля – це ще один крайній варіант TL, який спирається на не позначені приклади для вивчення завдання. Це може здатися неймовірним, особливо, коли навчання з використанням прикладів – це саме те, що стосується більшості алгоритмів навчання з викладачем. Методи навчання з нульовими даними або методи нульового короткого навчання, вносять розумні корективи на самому етапі навчання, щоб використовувати додаткову інформацію для розуміння невидимих даних. У своїй книзі про глибоке навчання (Deep Learning) Goodfellow та їх співавтори представляють нульове навчання як сценарій, коли вивчаються три змінні, такі як традиційна вхідна змінна, x, традиційна вихідна величина, y та додаткова випадкова змінна, яка описує завдання, T. Модель готується до вивчення умовного розподілу ймовірностей P (y|x, T). Навчання з нуля стане в нагоді в таких сценаріях, як машинний переклад, де ми можемо навіть не мати міток цільовою мовою.

**Застосування transfer learning**

Глибоке навчання, безумовно, є однією з конкретних категорій алгоритмів, яка була використана для того, щоб дуже успішно отримати переваги transfer learning. Нижче наведено кілька прикладів:

1) Transfer learning для NLP (natural language processing): Текстові дані представляють усілякі проблеми, коли мова йде про ML та глибоке навчання. Зазвичай вони трансформуються або векторизуються за допомогою різних методик. Вставки (Embedding), такі як Word2vec та FastText, були підготовлені за допомогою різних наборів навчальних даних. Вони використовуються в різних завданнях, таких як аналіз настроїв та класифікація документів, шляхом передачі знань із вихідних завдань. Окрім цього, новіші моделі, такі як Universal Sentence Encoder і BERT, безумовно, представляють безліч можливостей для майбутнього.

2) Transfer learning для аудіо / мови: Подібно до доменів, таких як NLP та Computer Vision, глибоке навчання успішно використовується для

завдань на основі аудіо даних. Наприклад, моделі автоматичного розпізнавання мови (ASR), розроблені для англійської мови, успішно використовуються для поліпшення продуктивності розпізнавання мовлення для інших мов, наприклад, німецької. Також автоматизована ідентифікація динаміків – ще один приклад, коли transfer learning значно допомогло.

3) Transfer learning для комп'ютерного зору: глибоке навчання досить успішно використовується для різних завдань комп'ютерного зору, таких як розпізнавання та ідентифікація об'єктів, використовуючи різні архітектури CNN. У своїй роботі "Як передаються функції в глибоких нейронних мережах", (https://arxiv.org/abs/1411.1792) автори представляють свої висновки щодо того, як нижні шари діють як звичайні екстрактори з комп’ютерним зором, такі як крайові детектори, в той час як кінцеві шари працюють на особливості функції.

Для прикладу використання підготовлених моделей, завантажимо модель Resnet18 (архітектура створена для даних ImageNet), використаємо, отримаємо точність та втрати:

```
from torchvision.models import resnet18
accuracies['resnet18'], losses['resnet18'] = \
  train(resnet18(), X_train, y_train, X_test, y_test)
```

**ResNet власноруч.**

ResNet – це скорочена назва для Residual Network (дослівно – «залишкова мережа»), але що таке residual learning («залишкове навчання»)?

Глибокі згорткові нейронні мережі перевершили людський рівень класифікації зображень в 2015 році. Глибокі мережі витягають низько-, середньо-і високорівневі ознаки наскрізним багатошаровим способом, а збільшення кількості stacked layers може збагатити «рівні» ознак.

Коли глибша мережа починає згортатися, виникає проблема: зі збільшенням глибини мережі точність спочатку збільшується, а потім

швидко погіршується. Зниження точності навчання показує, що не всі мережі легко оптимізувати.

**З'єднання швидкого доступу**

Щоб подолати цю проблему, Microsoft ввела глибоку «залишкову» структуру навчання. Замість того, щоб сподіватися на те, що кожна кілька stacked layers безпосередньо відповідає бажаному основному представленню, вони явно дозволяють цим шарам відповідати «залишковому». Формулювання $F(x)+x$ може бути реалізоване за допомогою нейронних мереж з з'єднаннями для швидкого доступу.

*З'єднання швидкого доступу* (shortcut connections) пропускають один або кілька шарів і виконують співставлення ідентифікаторів. Їх виходи додаються до виходів stacked layers. Використовуючи ResNet, можна вирішити безліч проблем, таких як:

1) ResNet відносно легко оптимізувати: «прості» мережі (які просто складають разом багато шарів) показують велику помилку навчання, коли глибина збільшується.

2) ResNet дозволяє відносно легко збільшити точність завдяки збільшенню глибини, чого з іншими мережами домогтися складніше.

**Residual Blocks**

Давайте зосередимось на локальній нейронній мережі, яку зображено на рисунку 12.5. Позначимо вхід x. Ми припускаємо, що ідеальне відображення, яке ми хочемо отримати шляхом навчання, є f(x), яке буде використовуватися як вхід до функції активації. Частина в полі пунктирною лінією на лівому зображенні повинна безпосередньо відповідати відображенню f(x). Це може бути складним завданням, якщо нам не потрібен цей конкретний шар, і ми значно краще збережемо вхідний x. Частина у вікні пунктирної лінії на правому зображенні тепер лише повинна параметризувати відхилення від ідентичності (identity), оскільки ми повертаємо $x+f(x)$. На практиці відображення залишків (residuals mappings)

часто простіше оптимізувати. Нам потрібно лише встановити f(x) = 0. Праве зображення на рисунку 12.5 ілюструє основний Залишковий блок ResNet.

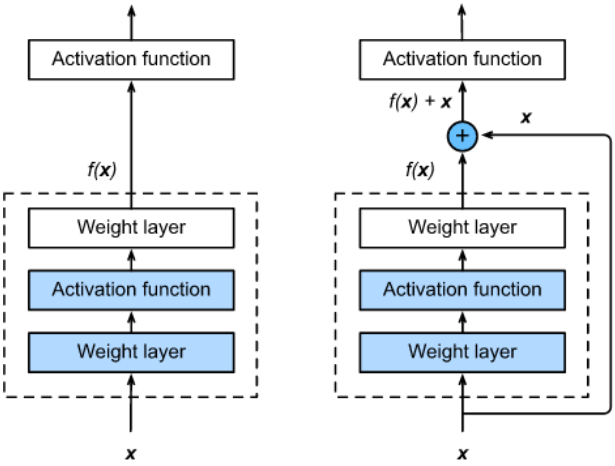



Рисунок 12.5 – Різниця між звичайним блоком (зліва) і Residual Block (праворуч). В останньому випадку ми можемо замикати згортки.

Логічна схема базового будівельного блоку для ResNet наведена на рисунку 12.6.

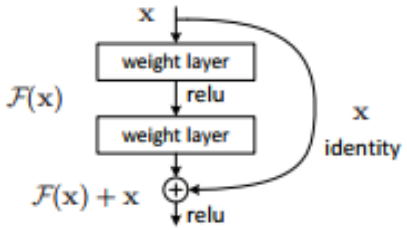



Рисунок 12.6 – Логічна схема базового будівельного блоку для ResNet

**Архітектурні конфігурації для ImageNet.**

Будівельні блоки показані в дужках з номерами блоків ResNet, які виконують повний 3 на 3 згортковий дизайн VGG (рисунок 12.7, 12.8). Залишковий блок має два конвеєрні шари 3×3 з однаковою кількістю вихідних каналів. За кожним згортковим шаром слідує шар батч нормалізації та функція активації ReLU. Потім ми пропускаємо ці дві операції згортання і додаємо вхід безпосередньо перед остаточною функцією активації ReLU. Цей вид конструкції вимагає, щоб вихід двох згорткових шарів був такої ж форми, що і вхід, щоб вони могли бути додані

разом. Якщо ми хочемо змінити кількість каналів або stride, нам потрібно ввести додатковий згортковий шар 1 × 1 для перетворення вводу в потрібну форму для операції додавання (рисунок 12.7).

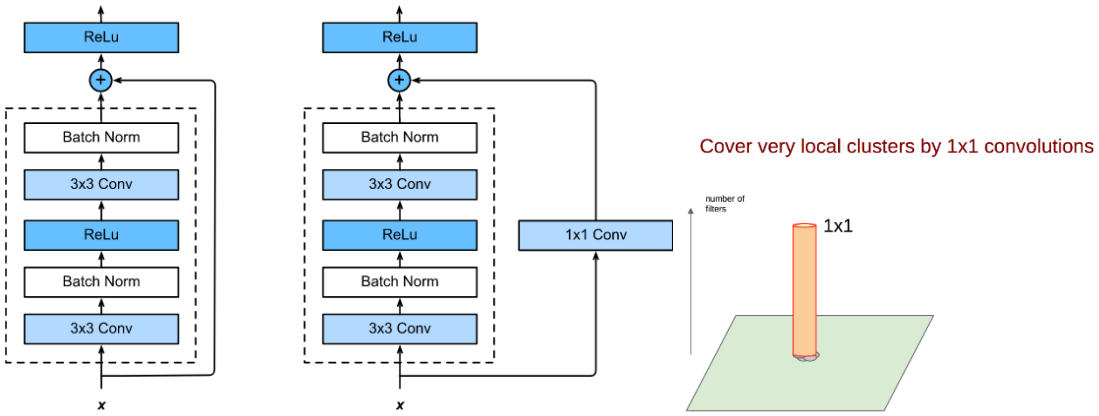



Рисунок 12.7 – Зліва: звичайний блок ResNet; Праворуч: блок ResNet з згорткою 1x1

**Проста мережа VGG та 34 layer plain** (рисунок 12.8): Прості базові лінії (в центрі) в основному натхненні філософією мереж VGG (зліва). Згорткові шари в основному мають фільтри 3 × 3 і дотримуються двох простих правил:

1) Для однієї і тієї ж вихідної карти об'єктів шари мають однакову кількість фільтрів;

2) Якщо розмір карти об'єктів зменшується вдвічі, число фільтрів подвоюється, щоб зберегти тимчасову складність кожного шару.

Варто відзначити, що модель ResNet має менше фільтрів і складність менше, ніж мережі VGG.

**ResNet (рисунок 12.8)**: на основі описаної простої мережі додано швидке з'єднання (праворуч), яке перетворює мережу в її залишкову версію. Ідентифікаційні швидкі з'єднання F(x(W) + x) можуть використовуватися безпосередньо, коли вхід і вихід мають однакові розмірності (швидкі з'єднання представлені суцільними лініями на рисунку 12.8). Коли розмірності збільшуються (пунктирні лінії на рисунку), розглядається два варіанти:

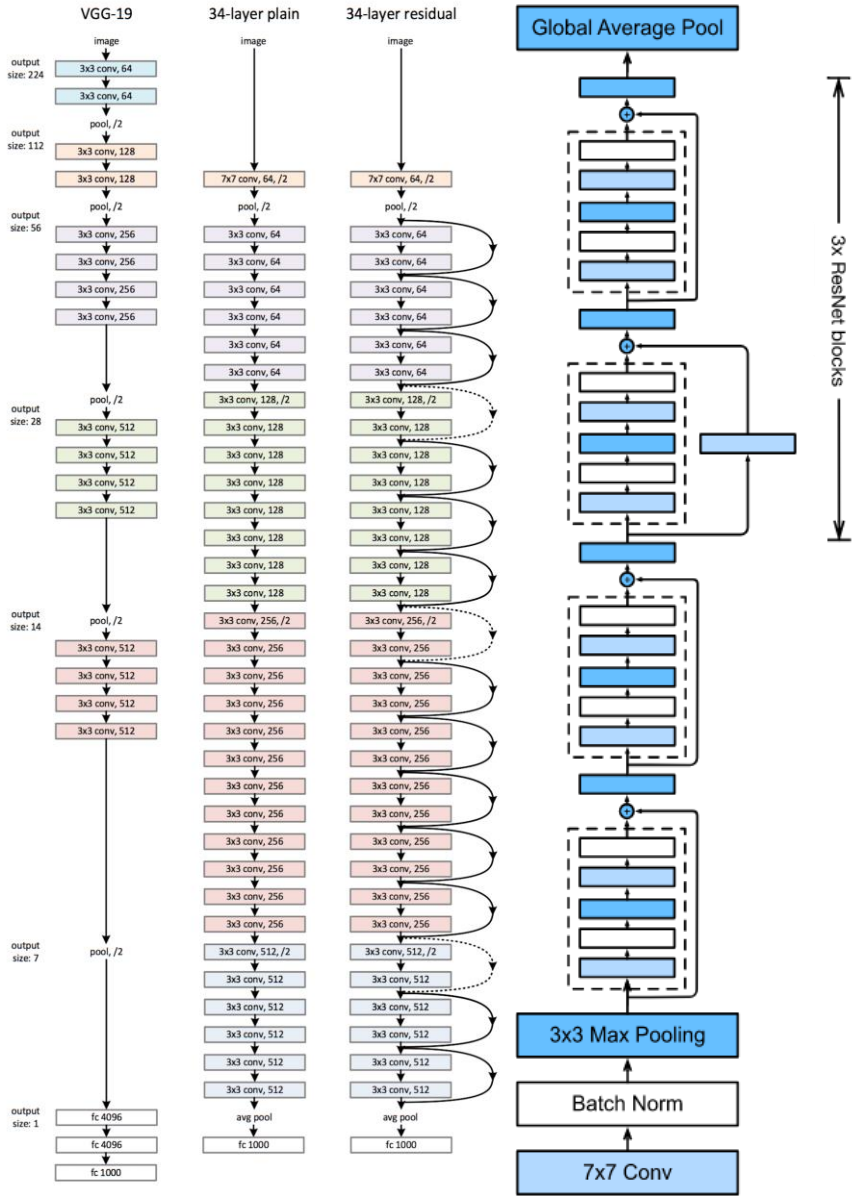



Рисунок 12.8 – Приклад архітектури для ImageNet. Зліва: модель VGG-19 (19,6 млрд. FLOP) як еталон. Посередині: проста мережу з 34 шарами (3,6 млрд. FLOP). Справа: ResNet з 34 шарами (3,6 мільярда FLOP). Пунктирні швидкі з'єднання збільшують розмірність.

1) Швидке з'єднання виконує зіставлення ідентифікаторів з додатковими нулями, доданими для збільшення розмірності. Ця опція не вводить ніяких додаткових параметрів.

2) Проекція швидкого з'єднання в F(x(W) + x) використовується для зіставлення розмірності (виконано за допомогою згорток 1 × 1).

Для будь-якої з опцій, якщо швидкі з'єднання йдуть по картах об'єктів двох розмірностей, вони виконуються з кроком 2.

Кожен блок ResNet має два рівня глибини (використовується в невеликих мережах, таких як ResNet 18, 34) або 3 рівня (ResNet 50, 101, 152).

**Схеми ResNet**

Порівняльний аналіз різних моделей ResNet наведений на рисунку 12.9.

| layer name | output size | 18-layer | 34-layer | 50-layer | 101-layer | 152-layer |
|---|---|---|---|---|---|---|
| conv1 | 112×112 | 7×7, 64, stride 2 | | | | |
| conv2_x | 56×56 | 3×3 max pool, stride 2<br>$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$ | $\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$ | $\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$ | $\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$ | $\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$ |
| conv3_x | 28×28 | $\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$ | $\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$ | $\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$ | $\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$ | $\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$ |
| conv4_x | 14×14 | $\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$ | $\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$ | $\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$ | $\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$ | $\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$ |
| conv5_x | 7×7 | $\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$ | $\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$ | $\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$ | $\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$ | $\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$ |
| | 1×1 | average pool, 1000-d fc, softmax | | | | |
| FLOPs | | $1.8\times10^9$ | $3.6\times10^9$ | $3.8\times10^9$ | $7.6\times10^9$ | $11.3\times10^9$ |

Рисунок 12.9 – Архітектури ResNet

**ResNet18**

У кожному модулі є 4 згорткових шари (за винятком згорткового шару 1 × 1 (вперше такий шар був використаний Google у Inception Model), 1 × 1 згортки експлуатуються для збільшення і зменшення розмірності, як і заповідав гугл (у GoogleNet)). Разом з першим згортковим шаром і останнім повністю з'єднаним шаром загалом налічується 18 шарів. Тому ця модель широко відома як ResNet-18. Конфігуруючи в модулі різну кількість каналів та залишкових блоків, ми можемо створити різні моделі ResNet, такі як глибший 152-шаровий ResNet-152. Хоча основна архітектура ResNet схожа на структуру GoogLeNet, структура ResNet є простішою та легшою для зміни.

**ResNet50**

**50-шарова ResNet**: кожен 3-шаровий блок замінюється в 34-шаровій мережі 3-шаровим вузьким місцем, в результаті виходить 50-шарова ResNet

(рисунок 12.9). Використовується варіант 2 для збільшення розмірності. Ця модель має 3,8 мільярда FLOPs.

**ResNet101 і 152**

ResNet з 101 і 152 шарами: створюється, використовуючи більше 3-шарових блоків. Навіть після збільшення глибини 152-шарова ResNet (11,3 мільярда FLOP) має меншу складність, ніж мережі VGG-16/19 (15,3 / 19,6 мільярда FLOPs (Floating Point Operations Per Second)).

**ResNet з нуля.**

Створимо ResNet власноруч (оригінальна стаття може бути знайдена за посиланням: https://arxiv.org/pdf/1512.03385.pdf) для вирішення завдання CIFAR10. ResNet20 складається із 9 Residual Blocks (3 + 3 + 3), ResNet110 складається із 54 Residual Blocks (18 + 18 + 18).

*Особливості***:**

1) BatchNorm вже включає додавання терміну зміщення (bias) = гамма * нормалізація(x) + зміщення. Тож немає необхідності (і це не має сенсу) додавати ще один термін зміщення у шар згортки перед шаром BatchNorm, отже використовуємо torch.nn.Conv2d(…,bias=False). Простіше кажучи, BatchNorm зміщує активацію на їх середні значення. Отже, будь-яка константа буде скасована. Навіть якщо ви реалізуєте ConvWithBias + BatchNorm, він буде вести себе як ConvWithoutBias + BatchNorm. Це те саме, що кілька повністю пов'язаних шарів без функції активації (з лінійною функцією активації) будуть поводитись як один.

2) Class BasicBlock – описує Residual Block + forward pass в цьому блоці.

2.1) Вхідний та вихідний розміри, та stride передаються на вхід функції.

2.2.) Ще раз базова концепція. Основна концепція глибокого залишкового (residual) навчання полягає в тому, що замість того, щоб очікувати, що кожен з декількох складених шарів безпосередньо буде відповідати базовому відображенню, автори явно дозволяють цим шарам

відповідати залишковому відображенню. Нехай H(x) – бажане основне відображення. Ми намагаємось зробити так, щоб складені нелінійні шари підходили до іншого відображення F(x):=H(x)−x. Тоді оригінальне відображення може бути перероблено у вигляді F(x)+x. Автори стверджують, що оптимізувати залишкове відображення простіше, ніж оптимізувати оригінальне відображення.

Ось де вступають у гру короткострокові з'єднання (Shortcut connections). Shortcut – це з'єднання, які пропускають один або кілька шарів (рисунок 12.6).

Так, залишковий блок приймає вхід з in_channels (in_planes), застосовує деякі блоки згорткових шарів, щоб зменшити його до out_channels (planes) і додати його до оригінального вхідного зображення. Якщо їх розміри не відповідають (це може статися, якщо in_planes != planes або stride != 1 в обох випадках вхідне зображення буде мати інший розмір ніж вихідне після двох послідовних згорток і ми просто не зможемо додати вхід до виходу із двох згорток без приведення розміру), то можливо зробити два варіанти:

a) Shortcut все ще робить identity mapping з додатковими нульовими паддінгами для збільшення розмірів. При цьому не вводиться зайвих параметрів;
b) Використовується Projection shortcut (H(x) = F(x)+Wx за рахунок згортки 1 × 1). При цьому із згорткою 1на1 іде шар BatchNorm. Крім того у ResNet кожен блок має параметр expansion, щоб збільшити канали out_channels, якщо потрібно.

   Для обох варіантів Shortcut виконуються з кроком (stride) 2.

```
class LambdaLayer(torch.nn.Module):
   def __init__(self, lambd):
      super(LambdaLayer, self).__init__()
      self.lambd = lambd
```

```
        def forward(self, x):
            return self.lambd(x)

    class BasicBlock(torch.nn.Module):
        expansion = 1
        def __init__(self, in_planes, planes, stride=1, option='A',
                     use_batch_norm=True, use_drop_out=False, d_out_p=0.5):
            super(BasicBlock, self).__init__()
            self.use_batch_norm = use_batch_norm
            self.use_drop_out = use_drop_out
            self.d_out_p = d_out_p
            self.act  = torch.nn.ReLU()

            self.conv1 = torch.nn.Conv2d(in_planes, planes, kernel_size=3,
stride=stride, padding=1, bias=False)
            self.bn1 = torch.nn.BatchNorm2d(planes)
            self.d_out1 = torch.nn.Dropout2d(d_out_p)
            self.conv2 = torch.nn.Conv2d(planes, planes, kernel_size=3,
stride=1, padding=1, bias=False)
            self.bn2 = torch.nn.BatchNorm2d(planes)
            self.d_out2 = torch.nn.Dropout2d(d_out_p)

            self.shortcut = torch.nn.Sequential()
            if stride != 1 or in_planes != planes:
                if option == 'A':
                    self.shortcut = LambdaLayer(lambda x:
torch.nn.functional.pad(x[:, :, ::2, ::2], \
                                            (0, 0, 0, 0, planes//4, planes//4), "constant", 0))
                elif option == 'B':
                    self.shortcut = torch.nn.Sequential(
```

```
                    torch.nn.Conv2d(in_planes, self.expansion * planes,
kernel_size=1, stride=stride, bias=False),
                    torch.nn.BatchNorm2d(self.expansion * planes)
                )

        def forward(self, x):
            out = self.conv1(x)
            if self.use_batch_norm:
              out = self.bn1(out)
            if self.use_drop_out:
              out = self.d_out1(out)
            out = self.act(out)
            out = self.conv2(out)

            if self.use_batch_norm:
              out = self.bn2(out)
            if self.use_drop_out:
              out = self.d_out2(out)

            out += self.shortcut(x)
            out = self.act(out)
            return out

        class ResNet(torch.nn.Module):
            def __init__(self, block, num_blocks, num_classes=10,
                    use_batch_norm=True, use_drop_out=False, d_out_p=0.5):
            super(ResNet, self).__init__()
            self.use_batch_norm = use_batch_norm
            self.use_drop_out = use_drop_out
            self.d_out_p = d_out_p
```

```
        self.in_planes = 16
        self.act  = torch.nn.ReLU()

        self.conv1 = torch.nn.Conv2d(3, 16, kernel_size=3, stride=1,
padding=1, bias=False)
        self.bn1 = torch.nn.BatchNorm2d(16)
        self.d_out1 = torch.nn.Dropout2d(d_out_p)
        self.layer1 = self._make_layer(block, 16, num_blocks[0], stride=1)
        self.layer2 = self._make_layer(block, 32, num_blocks[1], stride=2)
        self.layer3 = self._make_layer(block, 64, num_blocks[2], stride=2)
        self.linear = torch.nn.Linear(64, num_classes)

    def _make_layer(self, block, planes, num_blocks, stride):
        strides = [stride] + [1]*(num_blocks-1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_planes, planes, stride,
                                use_batch_norm=self.use_batch_norm,
                                use_drop_out=self.use_drop_out,
                                d_out_p=self.d_out_p))
            self.in_planes = planes * block.expansion
        return torch.nn.Sequential(*layers)
    def forward(self, x):
        out = self.conv1(x)
        if self.use_batch_norm:
          out = self.bn1(out)
        if self.use_drop_out:
          out = self.d_out1(out)
        out = self.act(out)
        out = self.layer1(out)
```

```
        out = self.layer2(out)
        out = self.layer3(out)
        out = torch.nn.functional.avg_pool2d(out, out.size()[3])
        out = out.view(out.size(0), -1)
        out = self.linear(out)
        return out
```

Та модифікації мережі які можуть бути корисними (вони різняться тільки тим, чи використовуємо ми Батч нормалізацію та/або Dropout):

```
def resnet110():
    return ResNet(BasicBlock, [18, 18, 18])

def resnet110_no_bn():
    return ResNet(BasicBlock, [18, 18, 18], use_batch_norm=False)

def resnet110_d_out15():
    return ResNet(BasicBlock, [18, 18, 18], use_drop_out=True, d_out_p=0.15)

def resnet20():
    return ResNet(BasicBlock, [3, 3, 3])

def resnet20_d_out5():
    return ResNet(BasicBlock, [3, 3, 3], use_drop_out=True)

def resnet20_d_out3(**kwargs):
    return ResNet(BasicBlock, [3, 3, 3], use_drop_out=True, d_out_p=0.3)

def resnet20_d_out8():
```

```
    return ResNet(BasicBlock, [3, 3, 3], use_drop_out=True, d_out_p=0.8)

def resnet20_d_out15():
    return ResNet(BasicBlock, [3, 3, 3], use_drop_out=True, d_out_p=0.15)
```

**Навчимо тепер ResNet20.**

```
accuracies['resnet20'], losses['resnet20'] = \
    train(resnet20(), X_train, y_train, X_test, y_test)
```

Порівняємо результати всіх трьох мереж: ResNet20, Resnet18 і CIFARNet.

```
acc_loss_graph(accuracies, losses, ['resnet20', 'resnet18', 'cifar_net'])
```

**ResNet110 і Batch normalization.**

Тепер навчимо ResNet110 в початковому варіанті і у варіанті з відключенням BatchNorm шарів. Для навчання знизимо розмір Батч до 64.

Можна очікувати, що зміна розміру Батч вплине на результат роботи мережі. Щоб порівняння було релевантним навчимо ResNet20 використовуючи такий же розмір Батч (batch_size = 64):

```
accuracies['resnet110'], losses['resnet110'] = \
    train(resnet110(), X_train, y_train, X_test, y_test, batch_size=64)

accuracies['resnet110_no_bn'], losses['resnet110_no_bn'] = \
    train(resnet110_no_bn(), X_train, y_train, X_test, y_test, batch_size=64)
```

*accuracies['resnet20 bs=64'], losses['resnet20 bs=64'] = *

*train(resnet20(), X_train, y_train, X_test, y_test, batch_size=64)*

*acc_loss_graph(accuracies, losses, ['resnet20','resnet20 bs=64', 'resnet110', 'resnet110_no_bn'])*

**DropOut**

Подивимося як буде вести себе мережу ResNet20, якщо після кожного шару BatchNorm додати шар Dropout с різними значеннями параметра p.

*accuracies['resnet20_d_out p=0.5'], losses['resnet20_d_out p=0.5'] = *

*train(resnet20_d_out5(), X_train, y_train, X_test, y_test)*

*accuracies['resnet20_d_out p=0.3'], losses['resnet20_d_out p=0.3'] = *

*train(resnet20_d_out3(), X_train, y_train, X_test, y_test)*

*accuracies['resnet20_d_out p=0.8'], losses['resnet20_d_out p=0.8'] = *

*train(resnet20_d_out8(), X_train, y_train, X_test, y_test)*

*accuracies['resnet20_d_out p=0.15'], losses['resnet20_d_out p=0.15'] = *

*train(resnet20_d_out15(), X_train, y_train, X_test, y_test)*

*acc_loss_graph(accuracies, losses, ['resnet20', 'resnet20_d_out p=0.15', 'resnet20_d_out p=0.3', 'resnet20_d_out p=0.5', 'resnet20_d_out p=0.8' ])*

**L2-регулярізація**

Подивимося як вплине на роботу ResNet20 застосування l2-регуляризації.

*accuracies['resnet20 wd=1e-5'], losses['resnet20 wd=1e-5'] = *

*train(resnet20(), X_train, y_train, X_test, y_test, weight_decay=1e-5)*

*accuracies['resnet20 wd=1e-4'], losses['resnet20 wd=1e-4'] = *
*train(resnet20(), X_train, y_train, X_test, y_test, weight_decay=1e-4)*

*accuracies['resnet20 wd=1e-3'], losses['resnet20 wd=1e-3'] = *
*train(resnet20(), X_train, y_train, X_test, y_test, weight_decay=1e-3)*

*accuracies['resnet20 wd=1e-2'], losses['resnet20 wd=1e-2'] = *
*train(resnet20(), X_train, y_train, X_test, y_test, weight_decay=1e-2)*

*acc_loss_graph(accuracies, losses, ['resnet20', 'resnet20 wd=1e-2', 'resnet20 wd=1e-3','resnet20 wd=1e-4', 'resnet20 wd=1e-5'])*

*acc_loss_graph(accuracies, losses, ['resnet20','resnet20 wd=1e-4', 'resnet20 wd=1e-5'])*

Теж ж саме для ResNet110:

*accuracies['resnet110_d_out15'], losses['resnet110_d_out15'], resnet110_d_out15_state = *
*train(resnet110_d_out15(), X_train, y_train, X_test, y_test, batch_size=64, save_net_state=True)*
*accuracies['resnet20_d_out p=0.15 bs=64'], losses['resnet20_d_out p=0.15 bs=64'], resnet20_d_out15_state = *
*train(resnet20_d_out15(), X_train, y_train, X_test, y_test, batch_size=64, save_net_state=True)*
*acc_loss_graph(accuracies, losses, ['resnet110_d_out15','resnet20_d_out p=0.15 bs=64', 'resnet110', 'cifar_net'])*

## Практична частина

1. Реалізуйте нейронні мережі наведені в теоретичному матеріалі та проаналізуйте отриманий результат.
2. Спробуйте власні оптимізації, спробуйте покращити результат: модифікуйте архітектуру, додайте аугментацію, можете створити ансамбль із декількох мереж, тощо.
3. Спробуйте знайти краще рішення для датасету CIFAR10, яка мережа повинна бути використана.
4. Спробуйте інші вбудовані в PyTorch моделі. Повний список моделей можна знайти у документації https://pytorch.org/docs/stable/torchvision/models.html.
5. Спробуйте додати bottleneck до residual blocks.
6. З бібліотеки torchvision (ставиться разом з pytorch), можна імпортувати ResNet18 командою "from torchvision.models import resnet18". Порівняйте результати resnet18, або інших мереж із Вашою мережею. Яка мережа дає кращий результат?
7. Спробуйте, користуючись описом архітектури з оригінальної статті (https://arxiv.org/pdf/1512.03385.pdf), написати власну реалізацію ResNet20. Якщо виникнуть сумніви, можна звіритися з кодом з https://github.com/akamaster/pytorch_resnet_cifar10. Чи вдалося побити resnet18?
8. Реалізуйте ResNet110 (можливо, доведеться зменшити розмір batch'a). Перевірте твердження, що ResNet110 не піддаються навчанню (або навчається в 10% випадків), якщо відключити BatchNorm.
9. Проаналізуйте результат та зробіть висновки щодо впливу параметру *p* та шару Dropout взагалі.
10. Проаналізуйте як впиває BatchNorm на навчання. Як вплинуло зменшення розміру Батч? Що буде при наступному зменшенні розміру Батч? Чому?
11. Проаналізуйте, як впливає застосування l2-регуляризації?

## Висновки

У висновках обґрунтувати вибір оптимальної архітектури нейронної мережі та гіперпараметрів. Проаналізувати результат, та надати відповіді на питання із практичної частини.

## Контрольні запитання

1. Що являє собою transfer learning?
2. Які архітектури згорткових нейронних мереж ви знаєте, в чому їх особливість?

# 13: АРХІТЕКТУРА GOOGLE NET

**Мета роботи** – засвоїти принцип роботи та побудови архітектури Google Net.

## Теоретичні відомості

Переможець ILSVRC 2014 – архітектура GoogLeNet також відома як Inception Module. Вона досягла 6,67% помилок у змаганні (рисунок 13.1-13.3).

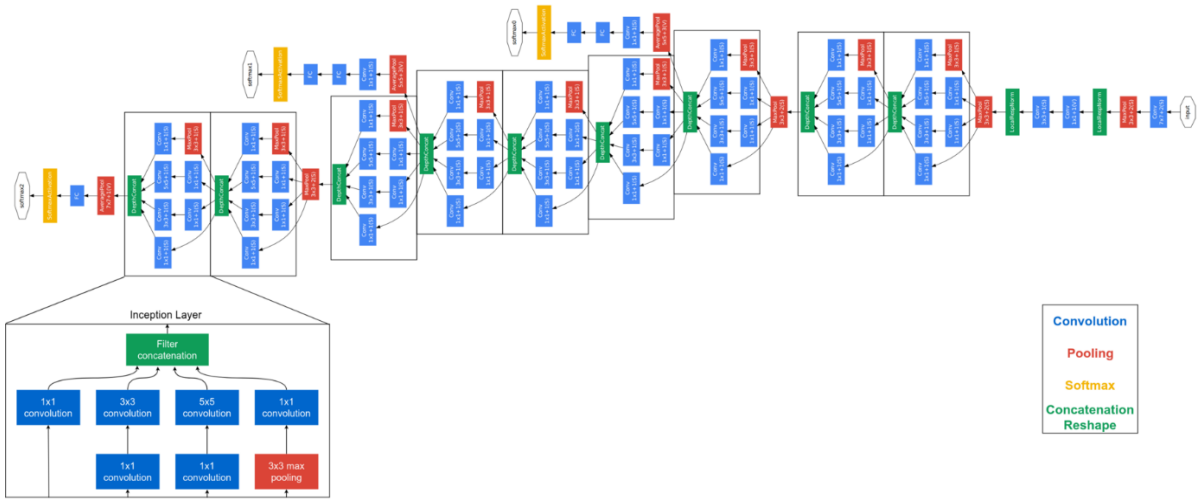



Рисунок 13.1 – Архітектура GoogleNet

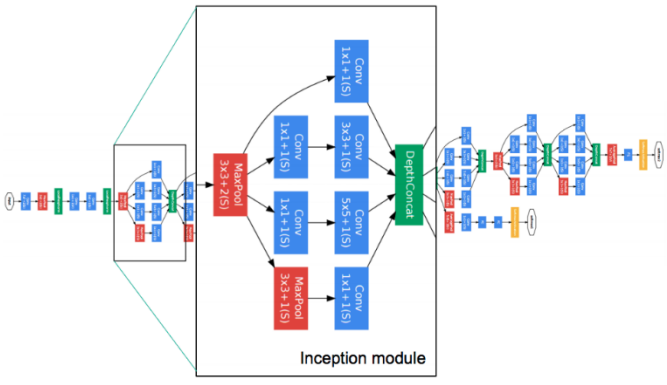



Рисунок 13.2 – Архітектура Inception Block

| type | patch size/ stride | output size | depth | #1×1 | #3×3 reduce | #3×3 | #5×5 reduce | #5×5 | pool proj | params | ops |
|---|---|---|---|---|---|---|---|---|---|---|---|
| convolution | 7×7/2 | 112×112×64 | 1 | | | | | | | 2.7K | 34M |
| max pool | 3×3/2 | 56×56×64 | 0 | | | | | | | | |
| convolution | 3×3/1 | 56×56×192 | 2 | | 64 | 192 | | | | 112K | 360M |
| max pool | 3×3/2 | 28×28×192 | 0 | | | | | | | | |
| inception (3a) | | 28×28×256 | 2 | 64 | 96 | 128 | 16 | 32 | 32 | 159K | 128M |
| inception (3b) | | 28×28×480 | 2 | 128 | 128 | 192 | 32 | 96 | 64 | 380K | 304M |
| max pool | 3×3/2 | 14×14×480 | 0 | | | | | | | | |
| inception (4a) | | 14×14×512 | 2 | 192 | 96 | 208 | 16 | 48 | 64 | 364K | 73M |
| inception (4b) | | 14×14×512 | 2 | 160 | 112 | 224 | 24 | 64 | 64 | 437K | 88M |
| inception (4c) | | 14×14×512 | 2 | 128 | 128 | 256 | 24 | 64 | 64 | 463K | 100M |
| inception (4d) | | 14×14×528 | 2 | 112 | 144 | 288 | 32 | 64 | 64 | 580K | 119M |
| inception (4e) | | 14×14×832 | 2 | 256 | 160 | 320 | 32 | 128 | 128 | 840K | 170M |
| max pool | 3×3/2 | 7×7×832 | 0 | | | | | | | | |
| inception (5a) | | 7×7×832 | 2 | 256 | 160 | 320 | 32 | 128 | 128 | 1072K | 54M |
| inception (5b) | | 7×7×1024 | 2 | 384 | 192 | 384 | 48 | 128 | 128 | 1388K | 71M |
| avg pool | 7×7/1 | 1×1×1024 | 0 | | | | | | | | |
| dropout (40%) | | 1×1×1024 | 0 | | | | | | | | |
| linear | | 1×1×1000 | 1 | | | | | | | 1000K | 1M |
| softmax | | 1×1×1000 | 0 | | | | | | | | |

Рисунок 13.3 – Детальна архітектура GoogleNet

Ця архітектура складається з 22 шарів в глибину. Це зменшує кількість параметрів з 60 мільйонів (AlexNet) до 4 мільйонів.

Також в GoogleNet немає повнозв'язних шарів на виході із моделі.

У складі GoogleNet є невелика підмережа – Stem Network (prenet). Вона складається з трьох згорткових шарів з двома pooling-шарами і розташовується в самому початку архітектури.

На схемі нейромережі можна побачити невеликі проміжні «відростки» – це допоміжні класифікаційні виходи для введення додаткового градієнта на початкових шарах (для оцінки якості при навчанні після 4-го та 7-го inception block (AvgPool – Conv(1,1) зі страйдом 1-FC-FC-Softmax)).

Ідея основного модуля Inception полягає в тому, що він сам по собі є невеликою локальною мережею. Вся його робота полягає в паралельному застосуванні декількох фільтрів на вихідне зображення. Дані фільтрів об'єднуються, і створюється вихідний сигнал, який переходить на наступний шар.

Нижче наведена реалізація базового класу Inception блоку, InceptionAux блоку (додаткові виходи, якщо потрібно), та GoogleNet. Звернемо увагу, якщо хочете використовувати додаткові виходи при навчанні, необхідно трохи модифікувати train функцію, наприклад змінивши/додавши наступний код у правильному місці:

```
if net.__class__.__name__ == "GoogLeNet" and net.aux_logits:
    preds = net.forward(X_batch)
    loss_value1 = loss(preds[0], y_batch)
    loss_value2 = loss(preds[1], y_batch)
    loss_value3 = loss(preds[2], y_batch)
    loss_value = loss_value1 + 0.3 * loss_value2 + 0.3 * loss_value3
else:
    preds = net.forward(X_batch)
    loss_value = loss(preds, y_batch)
```

Коефіцієнт 0.3 перед втратами додаткових виходів можете змінюватися.

```
class Inception(nn.Module):
    def __init__(self, in_planes, n1x1, n3x3red, n3x3, n5x5red, n5x5, pool_planes):
        super(Inception, self).__init__()
        # 1x1 conv branch
        self.b1 = nn.Sequential(
            nn.Conv2d(in_planes, n1x1, kernel_size=1),
            nn.BatchNorm2d(n1x1),
            nn.ReLU(True),
        )
```

```
# 1x1 conv -> 3x3 conv branch
self.b2 = nn.Sequential(
    nn.Conv2d(in_planes, n3x3red, kernel_size=1),
    nn.BatchNorm2d(n3x3red),
    nn.ReLU(True),
    nn.Conv2d(n3x3red, n3x3, kernel_size=3, padding=1),
    nn.BatchNorm2d(n3x3),
    nn.ReLU(True),
)

# 1x1 conv -> 5x5 conv branch
self.b3 = nn.Sequential(
    nn.Conv2d(in_planes, n5x5red, kernel_size=1),
    nn.BatchNorm2d(n5x5red),
    nn.ReLU(True),
    nn.Conv2d(n5x5red, n5x5, kernel_size=3, padding=1),
    nn.BatchNorm2d(n5x5),
    nn.ReLU(True),
    nn.Conv2d(n5x5, n5x5, kernel_size=3, padding=1),
    nn.BatchNorm2d(n5x5),
    nn.ReLU(True),
)

# 3x3 pool -> 1x1 conv branch
self.b4 = nn.Sequential(
    nn.MaxPool2d(3, stride=1, padding=1),
    nn.Conv2d(in_planes, pool_planes, kernel_size=1),
    nn.BatchNorm2d(pool_planes),
    nn.ReLU(True),
)
```

```
    def forward(self, x):
        y1 = self.b1(x)
        y2 = self.b2(x)
        y3 = self.b3(x)
        y4 = self.b4(x)
        return torch.cat([y1,y2,y3,y4], 1)

class InceptionAux(nn.Module):
    def __init__(self, in_planes, num_classes):
        super(InceptionAux, self).__init__()

        self.conv = nn.Conv2d(in_planes, 128, kernel_size=1)
        self.fc1 = nn.Linear(2048, 1024)
        self.fc2 = nn.Linear(1024, num_classes)

    def forward(self, x):
        # aux1: N x 512 x 14 x 14, aux2: N x 528 x 14 x 14
        x = torch.nn.functional.adaptive_avg_pool2d(x, (4, 4))
        # aux1: N x 512 x 4 x 4, aux2: N x 528 x 4 x 4
        x = self.conv(x)
        # N x 128 x 4 x 4
        x = torch.flatten(x, 1)
        # N x 2048
        x = torch.nn.functional.relu(self.fc1(x), inplace=True)
        # N x 1024
        x = torch.nn.functional.dropout(x, 0.7, training=self.training)
        # N x 1024
        x = self.fc2(x)
        # N x 1000 (num_classes)
        return x
```

```
class GoogLeNet(nn.Module):
    def __init__(self, num_classes=1000, aux_logits=True):
        super(GoogLeNet, self).__init__()
        self.pre_layers = nn.Sequential(
            # N x 3 x 224 x 224
            nn.Conv2d(3, 64, kernel_size=7, padding=3, stride=2),
            # N x 64 x 112 x 112
            nn.MaxPool2d(3, stride=2, ceil_mode=True),
            # N x 64 x 56 x 56
            nn.Conv2d(64, 64, kernel_size=1),
            # N x 64 x 56 x 56
            nn.Conv2d(64, 192, kernel_size=3, padding=1),
            # N x 192 x 56 x 56
            nn.MaxPool2d(3, stride=2, ceil_mode=True),
            # N x 192 x 28 x 28
            # Or just use one conv layer
            # nn.Conv2d(3, 192, kernel_size=3, padding=1),
            # nn.BatchNorm2d(192),
            # nn.ReLU(True),
        )
        self.aux_logits = aux_logits

        self.a3 = Inception(192,  64,  96, 128, 16, 32, 32)
        self.b3 = Inception(256, 128, 128, 192, 32, 96, 64)

        self.maxpool = nn.MaxPool2d(3, stride=2, padding=1)

        self.a4 = Inception(480, 192,  96, 208, 16,  48,  64)
        self.b4 = Inception(512, 160, 112, 224, 24,  64,  64)
        self.c4 = Inception(512, 128, 128, 256, 24,  64,  64)
```

```
        self.d4 = Inception(512, 112, 144, 288, 32,  64,  64)
        self.e4 = Inception(528, 256, 160, 320, 32, 128, 128)
        self.a5 = Inception(832, 256, 160, 320, 32, 128, 128)
        self.b5 = Inception(832, 384, 192, 384, 48, 128, 128)

        if aux_logits:
            self.aux1 = InceptionAux(512, num_classes)
            self.aux2 = InceptionAux(528, num_classes)

        # self.avgpool = nn.AvgPool2d(8, stride=1)
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.dropout = nn.Dropout(0.2)
        self.linear = nn.Linear(1024, num_classes)

    def forward(self, x):
        # N x 3 x 224 x 224
        out = self.pre_layers(x)
        # N x 192 x 28 x 28
        out = self.a3(out)
        # N x 256 x 28 x 28
        out = self.b3(out)
        # N x 480 x 28 x 28
        out = self.maxpool(out)
        # N x 480 x 14 x 14
        out = self.a4(out)
        # N x 512 x 14 x 14
        aux_defined = self.training and self.aux_logits
        if aux_defined:
            aux1 = self.aux1(out)
        else:
```

```
    aux1 = None
out = self.b4(out)
# N x 512 x 14 x 14
out = self.c4(out)
# N x 512 x 14 x 14
out = self.d4(out)
# N x 528 x 14 x 14
if aux_defined:
    aux2 = self.aux2(out)
else:
    aux2 = None
out = self.e4(out)
# N x 832 x 14 x 14
out = self.maxpool(out)
# N x 832 x 7 x 7
out = self.a5(out)
# N x 832 x 7 x 7
out = self.b5(out)
# N x 1024 x 7 x 7
out = self.avgpool(out)
# N x 1024 x 1 x 1
out = out.view(out.size(0), -1)  # or use x = torch.flatten(x, 1)
# N x 1024
x = self.dropout(x)
out = self.linear(out)
# N x 1000 (num_classes)
if aux_defined:
    return out, aux2, aux1
else:
    return out
```

## Практична частина

1. Реалізуйте архітектуру GoogleNet та порівняйте з вбудованою заздалегідь навченою архітектурою, та архітекторами ResNet. Яка архітектура дає кращий результат?
2. Спробуйте власні оптимізації, щоб покращити результат: модифікуйте архітектуру, додайте аугментацію, можете створити ансамбль із декількох мереж, тощо.

## Висновки

У висновках обґрунтувати вибір оптимальної нейронної мережі. Порівняти результати отримані мережею GoogleNet та ResNet з минулої практичної роботи.

## Контрольні запитання

1. Які особливості архітектури Google Net?
2. Що являє собою Inception block?

# 14: TRANSFER LEARNING (ЧАСТИНА II). ЗБЕРЕЖЕННЯ МОДЕЛІ, ЗАВАНТАЖЕННЯ МОДЕЛІ, ЗМІНА ЗАВАНТАЖЕНОЇ МОДЕЛІ

**Мета роботи** – засвоїти методику збереження та завантаження моделей у PyTorch.

## Теоретичні відомості

Ця практична робота пропонує рішення для різних випадків використання, щодо збереження та завантаження моделей PyTorch.

Що стосується збереження та завантаження моделей, слід ознайомитись з трьома основними функціями:

1) **torch.save**: зберігає «серіалізований» об’єкт на диску. Ця функція використовує утиліту pickle (https://docs.python.org/3/library/pickle.html - реалізує двійкові протоколи для «серіалізації» та «десеріалізації» структури об’єкта Python. "Pickling " – це процес, при якому ієрархія об'єкта Python перетворюється в байтовий потік, а "unpickling" – це зворотна операція, при якій потік байтів (з бінарного файлу або байтового об'єкта) перетворюється назад в ієрархію об'єктів Python для «серіалізації». За допомогою цієї функції можна зберегти моделі, тензори та словники всіх видів об’єктів.

2) **torch.load**: Використовує засоби pickle’s unpickling, щоб «десеріалізувати» зібрані об’єкти в пам'ять. Ця функція також полегшує завантаження даних на пристрій.

3) **torch.nn.Module.load_state_dict**: завантажує словник параметрів моделі за допомогою «десеріалізованого» state_dict.

**Що таке state_dict?**

У PyTorch, параметри, що засвоюються (тобто ваги та зміщення) моделі torch.nn.Module, містяться в параметрах моделі (можна отримати через model.parameters()). State_dict – це просто словник Python, який відображає кожен шар у свій тензор параметрів. Зауважте, що лише шари з параметрами, які навчаються (згорткові шари, лінійні шари тощо) та

зареєстровані буфери (batchnorm's running_mean) містять записи в state_dict. Об'єкти оптимізатора (torch.optim) також мають state_dict, який містить інформацію про стан оптимізатора, а також використовувані гіпер-параметри.

Оскільки об'єкти state_dict є словниками Python, їх можна легко зберігати, оновлювати, змінювати та відновлювати, додаючи таким чином модульність для моделей та оптимізаторів PyTorch.

Давайте наприклад подивимося на state_dict з наступної простої моделі:

```
import torch
import torch.nn as nn
from torch.autograd import Variable
from torch.optim import Adam
import numpy as np
import torch.nn.functional as F
from torch import optim

# Define model
class TheModelClass(nn.Module):
    def __init__(self):
        super(TheModelClass, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
```

```
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# Initialize model
model = TheModelClass()

# Initialize optimizer
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

# Print model's state_dict
print("Model's state_dict:")
for param_tensor in model.state_dict():
    print(param_tensor, "\t", model.state_dict()[param_tensor].size())

print("Model's state_dict:")
for var_name in model.state_dict():
    print(var_name, "\t", model.state_dict()[var_name])

# Print optimizer's state_dict
print("Optimizer's state_dict:")
for var_name in optimizer.state_dict():
    print(var_name, "\t", optimizer.state_dict()[var_name])

print("Model's state_dict:")
for var_name in model.state_dict():
```

*print(var_name, "\t", model.state_dict()[var_name])*

*Результат:*

*Model's state_dict:*
*conv1.weight torch.Size([6, 3, 5, 5])*
*conv1.bias torch.Size([6])*
*conv2.weight torch.Size([16, 6, 5, 5])*
*conv2.bias torch.Size([16])*
*fc1.weight torch.Size([120, 400])*
*fc1.bias torch.Size([120])*
*fc2.weight torch.Size([84, 120])*
*fc2.bias torch.Size([84])*
*fc3.weight torch.Size([10, 84])*
*fc3.bias torch.Size([10])*

*Значення параметрів:*

*Optimizer's state_dict:*
*state {}*
*param_groups [{'lr': 0.001, 'momentum': 0.9, 'dampening': 0, 'weight_decay': 0, 'nesterov': False, 'params': [2318702728896, 2318702728968, 2318702729040, 2318702729112, 2318702729184, 2318702729256, 2318702729328, 2318702729400, 2318702729472, 2318702729544]}]*

**Збереження та завантаження моделі для тесту.**

1. Зберегти / завантажити state_dict (рекомендується).

Зберігаючи модель, потрібно лише зберегти вивчені параметри навченої моделі. Збереження state_dict моделі за допомогою функції torch.save() надає вам найбільшу гнучкість для відновлення моделі пізніше, тому це рекомендований метод для збереження моделей.

Поширена умова PyTorch – це збереження моделей, використовуючи розширення файлу .pt або .pth.

Пам’ятайте, що ви повинні визвати model.eval(), щоб встановити шари dropout та batch normalization в режим оцінювання (тесту) перед запуском тесту. Якщо цього не зробити, це дасть суперечливі результати.

Зауважте, що функція load_state_dict() бере об’єкт словника, а не шлях до збереженого об'єкта. Це означає, що ви повинні скасувати «десеріалізацію» збереженого state_dict, перш ніж передати його функції load_state_dict(). Наприклад, ви НЕ МОЖЕТЕ завантажити, використовуючи model.load_state_dict (PATH).

```
# Save model
torch.save(model.state_dict(), "model_test.pt")
# Load model
model = TheModelClass()

print("Model's state_dict before load:")
# Should be new random coefficients
for var_name in model.state_dict():
   print(var_name, "\t", model.state_dict()[var_name])

model.load_state_dict(torch.load("model_test.pt"))
model.eval()

print("Model's state_dict:")
# Should be coefficients from saved model
```

```
for var_name in model.state_dict():
    print(var_name, "\t", model.state_dict()[var_name])

# Should be new random coefficients
for var_name in model.state_dict():
    print(var_name, "\t", model.state_dict()[var_name])

model.load_state_dict(torch.load("model_test.pt"))
model.eval()

print("Model's state_dict:")
# Should be coefficients from saved model
for var_name in model.state_dict():
    print(var_name, "\t", model.state_dict()[var_name])
```

2. Зберегти / завантажити цілу модель

Цей процес збереження / завантаження використовує інтуїтивний синтаксис і включає найменшу кількість коду. Якщо зберегти модель таким чином, ви зможете зберегти весь модуль за допомогою модуля pickle Python. Недоліком такого підходу є те, що «серіалізовані» дані прив’язані до конкретних класів та точної структури каталогів, які використовуються при збереженні моделі. Причиною цього є те, що pickle не зберігає сам клас моделі. Скоріше, це зберігає шлях до файлу, що містить клас, який використовується під час завантаження. Через це ваш код може порушуватися різними способами при використанні в інших проектах або після зміни.

```
# Save
torch.save(model, PATH)
```

*# Load*

*# Model class must be defined somewhere*

*model = torch.load(PATH)*

*model.eval()*

3. Збереження та завантаження загальної контрольної точки для виводу (тесту) та / або відновлення навчання

Зберігаючи загальну контрольну точку (значення на якийсь ітерації), яка буде використовуватися для виведення (тестування) або відновлення тренувань, ви повинні зберегти більше, ніж просто state_dict моделі. Важливо також зберегти state_dict оптимізатора, оскільки він містить буфери та параметри, які оновлюються в процесі навчання моделі. Інші елементи, які ви можете зберегти – це епоха, на якій ви зупинилися, останнє записане значення втрат, зовнішні шари torch.nn.Embedding, тощо.

Щоб зберегти кілька компонентів, організуйте їх у словнику та використовуйте torch.save() для «серіалізації» словника. Загальне правило PyTorch полягає в збереженні цих контрольних точок за допомогою розширення файлу .tar.

Щоб завантажити елементи, спочатку ініціалізуйте модель та оптимізатор, а потім завантажте словник локально, використовуючи torch.load (). Звідси ви можете легко отримати доступ до збережених елементів, просто запитуючи словник.

Пам’ятайте, що ви повинні визвати model.eval(), щоб встановити шари dropout та batch normalization в режим оцінювання (evaluation) перед запуском. Якщо цього не зробити, це дасть суперечливі результати. Якщо ви хочете відновити навчання, треба викликати model.train(), щоб переконатися, що ці шари перебувають у режимі тренувань.

*# Model Save particular checkpoint*

*torch.save({*

```
            'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'loss': loss,
            ...
            }, PATH)

model = TheModelClass(*args, **kwargs)
optimizer = TheOptimizerClass(*args, **kwargs)

# Model Load Particular Checkpoint
checkpoint = torch.load(PATH)
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
epoch = checkpoint['epoch']
loss = checkpoint['loss']
model.eval()
# - or -
model.train()
```

4. Збереження декількох моделей в одному файлі.

Зберігаючи модель, що складається з декількох модулів torch.nn.Modules, таких як GAN, sequence-to-sequence модель або ансамбль моделей, необхідно дотримуватись того ж підходу, що і під час збереження загальної контрольної точки. Іншими словами, збережіть словник state_dict кожної моделі та відповідного оптимізатора. Як згадувалося раніше, ви можете зберегти будь-які інші предмети, які можуть допомогти вам відновити навчання, просто додавши їх до словника.

Щоб завантажити моделі, спочатку ініціалізуйте моделі та оптимізатори, а потім завантажте словник локально, використовуючи torch.load(). Звідси ви можете легко отримати доступ до збережених елементів, просто запитуючи словник.

```
# Save
torch.save({
        'modelA_state_dict': modelA.state_dict(),
        'modelB_state_dict': modelB.state_dict(),
        'optimizerA_state_dict': optimizerA.state_dict(),
        'optimizerB_state_dict': optimizerB.state_dict(),
        ...
        }, PATH)
# Load

modelA = TheModelAClass(*args, **kwargs)
modelB = TheModelBClass(*args, **kwargs)
optimizerA = TheOptimizerAClass(*args, **kwargs)
optimizerB = TheOptimizerBClass(*args, **kwargs)

checkpoint = torch.load(PATH)
modelA.load_state_dict(checkpoint['modelA_state_dict'])
modelB.load_state_dict(checkpoint['modelB_state_dict'])
optimizerA.load_state_dict(checkpoint['optimizerA_state_dict'])
optimizerB.load_state_dict(checkpoint['optimizerB_state_dict'])

modelA.eval()
modelB.eval()
# - or -
modelA.train()
```

*modelB.train()*

5. Запуск моделі за допомогою параметрів з іншої моделі.

Часткове завантаження моделі або завантаження часткової моделі є загальними сценаріями при transfer learning або навчанні нової складної моделі. Використання натренованих параметрів, навіть якщо лише декілька з них корисні, допоможе швидко запустити тренувальний процес і, можливо, допоможе моделі зійтися швидше, ніж тренування з нуля.

Незалежно від того, завантажуєте ви з часткового state_dict, в якому відсутні деякі ключі, або завантажуєте state_dict з більшою кількістю ключів, ніж модель, в яку ви завантажуєте, ви можете встановити аргумент strict=False у функції load_state_dict (), щоб ігнорувати невідповідні ключі.

Якщо ви хочете завантажити параметри з одного шару в інший, але деякі ключі не збігаються, просто змініть ім’я ключів параметрів у state_dict, який ви завантажуєте, щоб вони відповідали ключам у моделі, в яку ви завантажуєте.

```
# Save
torch.save(modelA.state_dict(), PATH)
# Load
modelB = TheModelBClass(*args, **kwargs)
modelB.load_state_dict(torch.load(PATH), strict=False)
```

**Збереження та завантаження моделі на всіх пристроях**

1. Збережіть на GPU, завантажте на CPU (процесор)

Завантажуючи модель на процесор, яка була підготовлена з використанням графічного процесора (GPU), передайте torch.device ('cpu') аргумент у map_location у функції torch.load(). У цьому випадку сховища,

що лежать в основі тензорів, динамічно переставляються на пристрій процесора за допомогою аргументу map_location.

```
# Save
torch.save(model.state_dict(), PATH)

# Load
device = torch.device('cpu')
model = TheModelClass(*args, **kwargs)
model.load_state_dict(torch.load(PATH, map_location=device))
```

2. Збережіть на GPU, завантажте на GPU

Завантажуючи модель на GPU, яка була підготовлена та збережена на GPU, просто перетворіть ініціалізовану модель в оптимізовану модель CUDA за допомогою model.to(torch.device ('cuda')). Також обов'язково використовуйте функцію .to (torch.device ('cuda')) на всіх входах моделі для підготовки даних для моделі. Зауважте, що виклик my_tensor.to (пристрій) повертає нову копію my_tensor в GPU. Він НЕ переписує my_tensor. Тому не забудьте перезаписати тензори вручну: my_tensor = my_tensor.to (torch.device ('cuda')).

```
# Save
torch.save(model.state_dict(), PATH)

# Load
device = torch.device("cuda")
model = TheModelClass(*args, **kwargs)
model.load_state_dict(torch.load(PATH))
model.to(device)
```

*# Make sure to call input = input.to(device) on any input tensors that you feed to the model*

3. Збережіть на CPU , завантажте на GPU

Під час завантаження моделі в GPU, яка була підготовлена і збережена на процесорі, встановіть аргумент map_location у функції torch.load() на cuda: device_id. Це завантажує модель на даний пристрій GPU. Далі обов’язково треба викликати model.to(torch.device ('cuda')) для перетворення тензорів параметрів моделі в тензори CUDA. Нарешті, обов'язково використовуйте функцію .to (torch.device ('cuda')) на всіх входах моделі, щоб підготувати дані для оптимізованої моделі CUDA. Зауважте, що виклик my_tensor.to (пристрій) повертає нову копію my_tensor в GPU. Він НЕ переписує my_tensor. Тому не забудьте перезаписати тензори вручну: my_tensor = my_tensor.to (torch.device ('cuda')).

*# Save*

*torch.save(model.state_dict(), PATH)*

*# Load*

*device = torch.device("cuda")*

*model = TheModelClass(*args, **kwargs)*

*model.load_state_dict(torch.load(PATH, map_location="cuda:0")) # Choose whatever GPU device number you want*

*model.to(device)*

*# Make sure to call input = input.to(device) on any input tensors that you feed to the model*

4. Збереження моделей torch.nn.DataParallel

torch.nn.DataParallel – це обгортка моделі, яка дозволяє паралельно використовувати GPU. Щоб зберегти модель DataParallel в цілому,

збережіть model.module.state_dict(). Таким чином, ви маєте можливість завантажувати модель будь-яким способом на будь-який потрібний вам пристрій.

```
# Save
torch.save(model.module.state_dict(), PATH)
# Load
# Load to whatever device you want
```

**Виявлення малярії за допомогою обробки зображень та машинного навчання.**

Розглянемо тепер для прикладу задачу виявлення малярії за допомогою обробки зображень. Скористаємося Transfer learning підходом (рисунок 14.1).

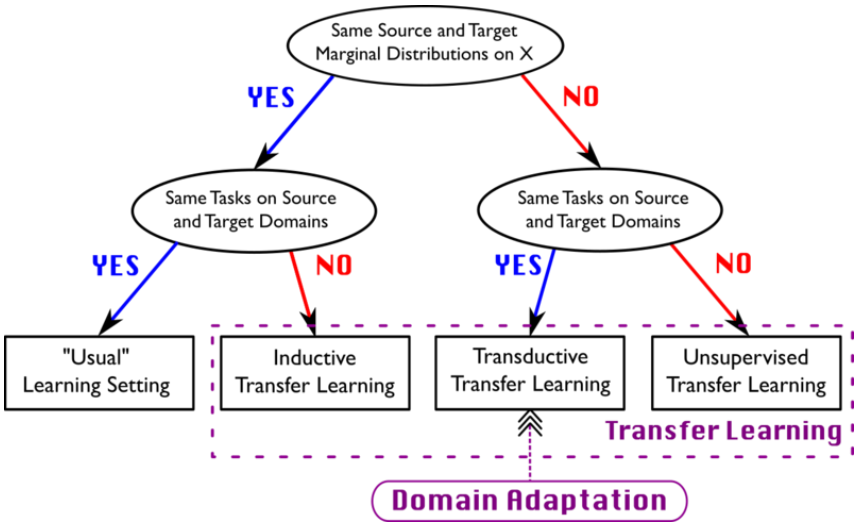



Рисунок 14.1 – Схематичне пояснення Transfer Learning підходу

Будемо використовувати Transfer learning для створення класифікатора зображень для виявлення малярії за зображенням мазків крові.

Набор даних має два класи, які ми збираємося класифікувати. Або зображення є інфікованим або незараженим. Набір даних, який ми будемо

використовувати, можна завантажити за наступним посиланням: https://www.kaggle.com/iarunava/cell-images-for-detecting-malaria або з іншого ресурсу: https://ceb.nlm.nih.gov/repositories/malaria-datasets/ (тут є оригінальні дані про цей датасет з описом та посиланням на статтю) або, у крайньому випадку, за наступним посиланням з гугл диску: https://drive.google.com/open?id=16DbIOMCtCuRuMdYF64MPv3iLqpSG6tfv. Заздалегідь підготовлена мережа проходила навчання на ImageNet, який містить 1,2 мільйона зображень з 1000 категорій, який доступний у моделях torchvision.models, які ми вже знаємо, він загалом має більше 6 різних архітектур, якими ми можемо користуватися.

**Підготовка до навчання**

Якщо у вас немає графічного процесора ви можете використовувати безкоштовні графічні процесори Google, запропоновані через Google Colab, для навчання вашої моделі. За цим посиланням ви можете знайти чудовий посібник із налаштуваннями Colab (https://hackernoon.com/getting-started-with-pytorch-in-google-collab-with-free-gpu-61a5c70b86a).

Отже, імпортуємо всі необхідні пакети та бібліотеки, які нам знадобляться для цієї програми виявлення малярії:

```
from matplotlib import pyplot as plt
import torch
from torch import nn
import torch.nn.functional as F
from torch import optim
from torch.autograd import Variable
from torchvision import datasets, transforms, models
from PIL import Image
import numpy as np
import os
```

```
from torch.utils.data.sampler import SubsetRandomSampler
import pandas as pd
```

Візуалізуємо деякі дані, які ми маємо: ми вказуємо шлях до каталогу (img_dir), що містить наші набори зображень. Вкажіть свій шлях до зображень. Давайте спочатку розглянемо, як виглядатиме паразитоване зображення:

```
img_dir='D:/шлях до зображень на вашому локальному комп'ютері'
def imshow(image):
    """Display image"""
    plt.figure(figsize=(6, 6))
    plt.imshow(image)
    plt.axis('off')
    plt.show()
# Example image
x = Image.open(img_dir + '/Parasitized/C100P61ThinF_IMG_20150918_144104_cell_165.png')
np.array(x).shape
imshow(x)
```

**Визначення необхідних перетворень та завантаження даних**

Трансформація – це процес, за допомогою якого одна фігура, вираз чи функція перетворюються на іншу. Давайте визначимо кілька перетворень для даних навчання, тестування та перевірки (train, test, validation). Ми повинні мати на увазі, що в деяких категоріях може бути обмежена кількість зображень. Таким чином, щоб збільшити кількість зображень, що розпізнаються в мережі, ми виконуємо те, що називається збільшенням даних (data augmentation).

Під час навчання ми випадковим чином обрізаємо, змінюємо розмір та обертаємо зображення, щоб кожну епоху (один прохід через набір даних) мережа бачила різні варіанти одного і того ж зображення. Це врешті-решт призведе до кращої точності на ваших тестових даних. Зауважте, що для тестових даних ми не проводимо збільшення даних, а просто робимо правильний розмір та обрізання по центру. Це тому, що ми хочемо, щоб наші дані валідації були схожими або виглядали як ваші кінцеві вхідні дані (поза вибірковими даними / тестовими даними).

Бібліотека torchvision.transforms дає можливість робити велику кількість трансформацій із зображеннями для аугментації даних. Проаналізуйте опис цієї бібліотеки за наступним посиланням: https://pytorch.org/docs/stable/torchvision/transforms.html, та спробуйте додати власні трансформації.

```
# Define your transforms for the training, validation, and testing sets
train_transforms = transforms.Compose([transforms.RandomResizedCrop(size=256, scale=(0.8, 1.0)),
                                       transforms.RandomRotation(degrees=15),
                                       transforms.ColorJitter(),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.CenterCrop(size=224),  # Image net standards
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.485, 0.456, 0.406],
                                                            [0.229, 0.224, 0.225])
                                       ])

test_transforms = transforms.Compose([transforms.Resize(256),
                                      transforms.CenterCrop(224),
```

```
                transforms.ToTensor(),
                transforms.Normalize([0.485, 0.456, 0.406],
                                     [0.229, 0.224, 0.225])])

validation_transforms = transforms.Compose([transforms.Resize(256),
                                transforms.CenterCrop(224),
                                transforms.ToTensor(),
                                transforms.Normalize([0.485, 0.456, 0.406],
                                                     [0.229, 0.224, 0.225])])
```

За допомогою визначених перетворень ми повинні завантажувати набір даних. Найпростішим способом завантаження даних зображень є використання набору dataset.ImageFolder від torchvision, який приймає як вхід шлях до зображень та перетворень.

З завантаженим imageFolder давайте розділимо дані на 20% – на валідацію та 10% на тестовий набір; потім передаємо його у DataLoader, який приймає набір даних, який ви отримаєте з ImageFolder, і повертає партії зображень та їх відповідні мітки (параметр shuffling може бути встановлено на True для введення змін протягом епох).

```
#Loading in the dataset
train_data = datasets.ImageFolder(img_dir,transform=train_transforms)
# number of subprocesses to use for data loading
num_workers = 0
# percentage of training set to use as validation
valid_size = 0.2

test_size = 0.1

# obtain training indices that will be used for validation
```

```
num_train = len(train_data)
indices = list(range(num_train))
np.random.shuffle(indices)
valid_split = int(np.floor((valid_size) * num_train))
test_split = int(np.floor((valid_size+test_size) * num_train))
valid_idx, test_idx, train_idx = indices[:valid_split],
indices[valid_split:test_split], indices[test_split:]

print(len(valid_idx), len(test_idx), len(train_idx))

# define samplers for obtaining training and validation batches
train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)
test_sampler = SubsetRandomSampler(test_idx)

# prepare data loaders (combine dataset and sampler)
train_loader = torch.utils.data.DataLoader(train_data, batch_size=32,
   sampler=train_sampler, num_workers=num_workers)
valid_loader = torch.utils.data.DataLoader(train_data, batch_size=32,
   sampler=valid_sampler, num_workers=num_workers)
test_loader = torch.utils.data.DataLoader(train_data, batch_size=32,
   sampler=test_sampler, num_workers=num_workers)
```

**Навчання моделі**.

Кроки, які ми будемо використовувати для нашої навченої моделі, такі:

1) Завантаження в попередньо підготовлену модель
2) Заморожування згорткових шарів
3) Заміна повністю пов'язаних шарів спеціальним класифікатором
4) Навчання спеціального класифікатора для конкретного завдання

Тепер ми можемо завантажити наші підготовлені дані в одну з попередньо підготовлених моделей, наприклад будемо використовувати densenet121, який має досить високу точність на наборі даних ImageNet. Назва говорить нам, що ця модель складається із 121 шару. Завантажимо модель та подивимось на її архітектуру:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
#pretrained=True will download a pretrained network for us
model = models.densenet121(pretrained=True)
print(model)
```

Враховуючи нашу модель, нам потрібно навчити класифікатор. Однак зараз ми використовуємо дійсно глибоку нейронну мережу. Тому майте на увазі, що якщо ви не маєте графічного процесору, це може зайняти багато часу.

Тут ми будемо використовувати GPU для розрахунків для пришвидшення процесу (скористайтеся Google Colab). Обчислення лінійної алгебри проводяться паралельно на графічному процесорі, що призводить до 100-кратного збільшення швидкості навчання. Також можна тренуватися на декількох графічних процесорах, зменшуючи час навчання.

PyTorch, поряд з усіма іншими фреймворками глибокого навчання, використовує CUDA для ефективного обчислення forward та backwards passes на GPU. У PyTorch ви переміщуєте параметри моделі та інших тензорів до пам'яті GPU за допомогою model.cuda(). Ви можете перемістити їх з графічного процесора за допомогою model.cpu(), що зазвичай робиться, коли вам потрібно працювати на мережевому виході за межами PyTorch.

Заморожування згорткових шарів та заміна повністю з’єднаних шарів спеціальним класифікатором:

```
# Freezing model parameters and defining the fully connected network to be attached to the model, loss function and the optimizer.
# We there after put the model on the GPUs

for param in model.parameters():
    param.require_grad = False

# Create our own classifier to predict two classes ouput
fc = nn.Sequential(
    nn.Linear(1024, 460),
    nn.ReLU(),
    nn.Dropout(0.4),

    nn.Linear(460, 2),
    nn.LogSoftmax(dim=1))

# Replace densenet121 classifier to our classifier (in fact we just replace last layer and we will train the coefficients
# only for this layer, while all other coefficients in layers stay unchanged)
model.classifier = fc
criterion = nn.NLLLoss()
# Over here we want to only update the parameters of the classifier so
optimizer = torch.optim.Adam(model.classifier.parameters(), lr=0.003)
model.to(device)
```

Заморожування параметрів моделі по суті дозволяє нам зберегти ваги попередньо підготовленої моделі для ранніх згорткових шарів – метою яких є вилучення особливостей.

Потім ми визначаємо нашу повнозв'язну мережу, яка матиме 1024 вхідний нейрон (це залежить від попередньо підготовлених вхідних нейронів моделі), так і спеціальний прихований шар.

Ми також визначаємо функцію активації, яка буде використовуватися, і dropout, що допоможе уникнути перенавчання, випадковим чином вимикаючи нейрони в шарі, щоб змусити інформацію ділитися між іншими вузлами.

Після того, як ми визначили нашу власну повнозв'язну мережу, ми підключаємо її до повнозв'язної мережі заздалегідь підготовленої моделі відповідно до проблеми, яку ми хочемо вирішити. Визначимо функцію втрат (NLLLoss – negative log likelihood), оптимізатор і підготуємо модель для навчання, перемістивши її до device.

**Навчання спеціального класифікатора для конкретного завдання**

Під час тренінгу ми робимо ітерації через DataLoader для кожної епохи. Для кожної партії (batch) втрати обчислюються за допомогою функції втрат. Градієнти втрат щодо параметрів моделі розраховуються за допомогою методу loss.backward ().

Optimizer.zero_grad () відповідає за очищення накопичених градієнтів, оскільки ми будемо обчислювати градієнти знову і знову. optimizer.step() оновлює параметри моделі, використовуючи Stochastic Gradient Descent з моментом (Adam).

Для запобігання перенавчання ми будемо використовуємо потужну ідею, яка називається раннє припинення (early stopping). Ідея дуже проста – припинити навчання, коли продуктивність на тестовому наборі даних починає знижуватися:

*# Training the model and saving checkpoints of best performances. That is lower validation loss and higher accuracy*
*epochs = 10*
*valid_loss_min = np.Inf*

```
import time

for epoch in range(epochs):

    start = time.time()

    # scheduler.step()
    model.train()

    train_loss = 0.0
    valid_loss = 0.0

    for inputs, labels in train_loader:
        # Move input and label tensors to the default device
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()

        logps = model(inputs)
        loss = criterion(logps, labels)
        loss.backward()
        optimizer.step()
    train_loss += loss.item()

    model.eval()

    with torch.no_grad():
        accuracy = 0
        for inputs, labels in valid_loader:
            inputs, labels = inputs.to(device), labels.to(device)
```

```
            logps = model.forward(inputs)
            batch_loss = criterion(logps, labels)

        valid_loss += batch_loss.item()
        # Calculate accuracy
        ps = torch.exp(logps)
        top_p, top_class = ps.topk(1, dim=1)
        equals = top_class == labels.view(*top_class.shape)
        accuracy += torch.mean(equals.type(torch.FloatTensor)).item()

        # calculate average losses
        train_loss = train_loss / len(train_loader)
        valid_loss = valid_loss / len(valid_loader)
        valid_accuracy = accuracy / len(valid_loader)

        # print training/validation statistics
        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f} \tValidation
Accuracy: {:.6f}'.format(
            epoch + 1, train_loss, valid_loss, valid_accuracy))

        if valid_loss <= valid_loss_min:
            print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.format(
                valid_loss_min,
                valid_loss))
            model_save_name = "Malaria.pt"
            path = F"/content/drive/My Drive/{model_save_name}"
            torch.save(model.state_dict(), path)
            valid_loss_min = valid_loss

        print(f"Time per epoch: {(time.time() - start):.3f} seconds")
```

Після навчання збережемо контрольні точки кращих параметрів моделі, завантажимо контрольну точку і перевіримо працездатність моделі на відкладених даних (дані з тесту).

Завантаження збереженої моделі з диска:

```
model.load_state_dict(torch.load('Malaria.pt'))
```

Тестування завантаженої моделі на відкладених даних:

```
def test(model, criterion):
    # monitor test loss and accuracy
    test_loss = 0.
    correct = 0.
    total = 0.

    for batch_idx, (data, target) in enumerate(test_loader):
        # move to GPU
        if torch.cuda.is_available():
            data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the loss
        loss = criterion(output, target)
        # update average test loss
        test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data - test_loss))
        # convert output probabilities to predicted class
        pred = output.data.max(1, keepdim=True)[1]
        # compare predictions to true label
```

```
        correct                                                    +=
np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu().numpy())
        total += data.size(0)

    print('Test Loss: {:.6f}\n'.format(test_loss))
    print('\nTest Accuracy: %2d%% (%2d/%2d)' % (100. * correct / total, correct,
total))
test(model, criterion)
```

Тепер, коли ми маємо впевненість у своїй моделі, саме час зробити деякі прогнози та візуалізувати результати:

```
def load_input_image(img_path):
    image = Image.open(img_path)
    prediction_transform  =  transforms.Compose([transforms.Resize(size=(224,
224)),
                                transforms.ToTensor(),
                                transforms.Normalize([0.485, 0.456, 0.406],
                                                     [0.229, 0.224, 0.225])])

    # discard the transparent, alpha channel (that's the :3) and add the batch
dimension
    image = prediction_transform(image)[:3,:,:].unsqueeze(0)
    return image

def predict_malaria(model, class_names, img_path):
    # load the image and return the predicted breed
    img = load_input_image(img_path)
    model = model.cpu()
    model.eval()
```

```
    idx = torch.argmax(model(img))
    return class_names[idx]

from glob import glob
from PIL import Image
from termcolor import colored

class_names=['Parasitized','Uninfected']
inf = np.array(glob(img_dir + "/Parasitized/*"))
uninf = np.array(glob(img_dir + "/Uninfected/*"))
for i in range(3):
    img_path=inf[i]
    img = Image.open(img_path)
    if predict_malaria(model, class_names, img_path) == 'Parasitized':
        print(colored('Parasitized', 'green'))
    else:
        print(colored('Uninfected', 'red'))
    plt.imshow(img)
    plt.show()
for i in range(3):
    img_path=uninf[i]
    img = Image.open(img_path)
    if predict_malaria(model, class_names, img_path) == 'Uninfected':
        print(colored('Uninfected', 'green'))
    else:
        print(colored('Parasitized', 'red'))
    plt.imshow(img)
    plt.show()
```

Таким чином,  ми змогли створити додаток для класифікації малярії, який міг би (з певним доопрацюванням, звичайно) не тільки врятувати життя, але й допомогти пришвидшити процеси у лабораторіях та допомогти медичним працівникам.

## Практична частина

1. Спробуйте покращити отримані результати в задачі про виявлення малярії (модель наведена у теоретичному матеріалі), використовуючи різні моделі (архітектури), підходи, технології. Спробуйте внести зміни в оптимізатор, попередньо підготовлену модель та функцію втрат. Ви можете додати більше перетворень або навіть додати більше шарів до повністю з'єднаних шарів.
2. Спробуйте вирішити завдання аналізу рентгенівського знімку легень та класифікувати їх на класи: норма та пневмонія. Завантажте датасет за посиланням та проаналізуйте додаткову інформацію: https://www.kaggle.com/paultimothymooney/chest-xray-pneumonia

## Висновки

У висновках обґрунтувати вибір оптимальної нейронної мережі для задачі виявлення малярії та рентгенівських знімків легень. Зробити відповідні висновки, щодо отриманих результатів

## Контрольні запитання

1. Як проводиться збереження коефіцієнтів моделі у PyTorch?
2. Як завантажити коефіцієнти попередньо збереженої моделі у PyTorch?

# 15. CONDITIONAL RANDOM FIELDS, VITERBY

**Мета роботи** – засвоїти принцип роботи ймовірнісних моделей: CRF, Viterby.

## Теоретичні відомості

Припустимо, у Вас є два однакові кубики, але один кубик справедливий (стандартний, де кожне число має однакову ймовірність випасти), а другий перезважений таким чином, що число 6 з'являється з 80% -вою ймовірністю, тоді як числа 1–5 однаково ймовірні з 4%-вою ймовірністю. Якби Вам дали послідовності з 15 кидків кубиків, змогли б Ви передбачити, який кубик використовувався для кожної послідовності?

Проста модель може передбачити, що використовувався зважений кубик щоразу, коли з'явиться 6, і сказати, що використовувався «справедливий» для всіх інших чисел.

Насправді, якби ми з однаковою ймовірністю використовували будь-які кубики для будь-якої послідовності, то це просте правило – найкраще, що можна зробити.

Але що робити, якщо після використання правильного кубика у Вас є 90% ймовірності, що буде використаний «зважений» кубик на наступному кидку кубика? Якщо наступний кидок кубика дав 3, ваша модель передбачить, що ми використовували «правильний» кубик, тоді яка насправді «зважений» кубик є більш вірогідним вибором. Ми можемо перевірити це за допомогою теореми Байєса – ймовірність, що кубик «правильний, $y_i$ = Fair», при умові, що попередній кубик був «правильний, $y_{i-1}$ = Fair» і зараз випало число 3 за теоремою Байєса дорівнює 0.32:

$$P\left(y_i = Fair|_{y_{i-1}=Fair}, x_i = 3\right) =$$

$$\frac{P(x_i=3|_{y_i=Fair})P\left(y_i=Fair|_{y_{i-1}=Fair}\right)}{P\left(x_i=3|_{y_i=Fair}\right)P\left(y_i=Fair|_{y_{i-1}=Fair}\right)+P\left(x_i=3|_{y_i=Biased}\right)P\left(y_i=Biased|_{y_{i-1}=Fair}\right)} =$$

$$\frac{0.1*1/6}{0.1*1/6+0.9*0.04} = 0.32$$

де $y_i$ – випадкова величина типу кубика для кидку і. $x_i$ – значення кубика для кидку і.

У чисельнику маємо добуток ймовірності, що у «правильному» кубику випаде «3» – це 1/6 на ймовірність, що зараз «правильний» кубик, тоді як минулий також був правильний: 0.1 = 1/6 * 0.1 = 0.017

У чисельнику маємо суму добутків: 1) той же добуток, що і в чисельнику (0.017); 2) ймовірність, що 3 випало у «зваженого» кубика на ймовірність (це 0.04), що після «правильного кубика» буде «зважений» – це 0.9 = 0.04 * 0.9 = 0.036: 1)+2) = 0.036 + 0.017 = 0.053

Отже ймовірність, що використовувався «правильний кубик» дорівнює всього 0.017/0.053 = 0.32 = 32%. Тобто «зважений» кубик більш вірогідний.

Таким чином, робити найбільш вірогідний вибір на кожному етапі – це лише життєздатна стратегія, коли ми однаковою мірою використовуємо будь-які кубики. За набагато більш імовірним сценарієм, коли попередній вибір кубиків впливає на майбутній вибір, вам доведеться враховувати взаємозалежність кидків, щоб досягти успіху.

Умовне випадкове поле (conditional random field, CRF) – це стандартна модель для прогнозування найбільш ймовірної послідовності міток, яка відповідає послідовності входів. CRF є темою в більш широкій і глибшій темі, яка називається ймовірнісними графічними моделями (probabilistic graphical models). Тому теоретичне висвітлення цього питання достатньо складне, та потребує цілого окремого курсу по ймовірнісним моделям.

Ціль цієї роботи – висвітлити достатньо теорії, щоб ви змогли зануритися в ресурси 1-ї категорії з уявленням, чого очікувати, і показати,

як реалізувати CRF на простій проблемі, яку ви можете реалізувати на власному ноутбуці. Це повинно забезпечити вам інтуїцію, необхідну для адаптації цієї простої іграшки CRF для більш складної проблеми.

Три наступні широкі категорії застосовуються до будь-якої статистичної моделі, навіть до простої логістичної регресії, тому, в цьому сенсі, у CRF нема нічого особливого:

1) визначення параметрів моделі

2) як оцінити ці параметри

3) використовування цих параметрів для прогнозування.

Але це не означає, що CRF так само проста, як і наприклад модель логістичної регресії. Справи будуть дещо складнішими, коли ми зрозуміємо той факт, що ми робимо послідовність прогнозів на відміну від одного прогнозу.

**Визначення параметрів моделі**

У цій простій проблемі єдиними параметрами, про які ми повинні турбуватися, є витрати, пов'язані з переходом від однієї кістки до іншої в послідовних кидках. Ми маємо турбуватися про шість чисел, і ми збережемо їх у матриці 2x3 під назвою матриця переходу:

$$\begin{bmatrix} C(\text{stay with fair dice}) & C(\text{switch to fair from biased}) & C(\text{first dice in sequence is fair}) \\ C(\text{switch to biased from fair dice}) & C(\text{stay with biased dice}) & C(\text{first dice in sequence is biased}) \end{bmatrix}$$

Перший стовпець відповідає переходам від «правильного» кубика у попередньому кидку до «правильного» кубика (значення у рядку 1) та від «зваженого» кубика до «правильного» (значення у рядку 2) у поточній спробі. Отже, перший запис у першому стовпчику кодує вартість прогнозування, що ми використовуємо «правильні» кубики на наступному кидку, враховуючи, що ми використовували «правильні» кубики на поточному кидку. Якщо дані показують, що ми навряд чи використовуємо «правильні» кубики в послідовних спробах, модель дізнається, що ціна повинна бути високою і навпаки. Ця ж логіка стосується і другого стовпця.

Перший і другий стовпці матриці передбачають, що ми знаємо, які кістки ми використовували в попередньому кидку. Тому ми маємо трактувати першу спробу як особливий випадок. Ми збережемо відповідні витрати у третьому стовпці.

**Оцінка параметрів**

Скажімо, я даю вам набір значень кубиків X та їх відповідні позначки Y (справедливий або зважений). Ми знайдемо матрицю переходу T, яка мінімізує negative log likelihood (негативну логарифмічну ймовірність) за всіма навчальними даними. Давайте розглянемо, що являють собою вірогідність та негативна логарифмічна вірогідність для однієї послідовності кидків кубиків. Щоб отримати їх для всього набору даних, ви маєте усереднити цей показник для всіх послідовностей.

$$Likelihood(\text{dice labels Y}|\text{rolls X and matrix T}) = \frac{\sum_{i=0}^{n} P(x_i|y_i)T(y_i|y_{i-1})}{\sum_{y'}\sum_{i=0}^{n} P(x_i|y'_i)T(y'_i|y'_{i-1})}$$

$$NegLogLikelihood(\text{dice labels Y}|\text{rolls X and matrix T}) = \sum_{y'}\sum_{i=0}^{n} Log(P(x_i|y'_i)T(y'_i|y'_{i-1})) - \sum_{i=0}^{n} Log(P(x_i|y_i)T(y_i|y_{i-1}))$$

де P ($x_i|y_i$) – це ймовірність випадання заданого значення кубика ($x_i$) з урахуванням поточної мітки кубика ($y_i$). Для прикладу, P($x_i$ | $y_i$) = 1/6, якщо $y_i$ = кістки «правильні». Інший член, T ($y_i$ | $y_{i-1}$) – це вартість переходу з попередньої метки кубика до поточної. Ми можемо просто прочитати цю вартість з перехідної матриці.

Зверніть увагу, як у знаменнику ми обчислюємо суму по всіх можливих послідовностях міток y`. У традиційній логістичній регресії для проблеми класифікації двох класів у знаменнику ми будемо мати два члени. Але зараз ми маємо справу з послідовностями, і для послідовності довжиною 15 існує всього $2^{15}$ можливих послідовностей міток, тому кількість членів у знаменнику насправді величезна. "Особливість" CRF полягає в тому, що він експлуатує те, як поточна мітка кубика залежить тільки від попередньої для того, щоб обчислити цю величезну суму ефективно. Цей «особливий» алгоритм називається алгоритмом «вперед-назад» (forward-backward algorithm*).

**Прогнозування послідовності**

Як тільки ми оцінимо нашу матрицю переходу, ми можемо використовувати її для пошуку найбільш вірогідної послідовності міток для кубиків для заданої послідовності кидків кубиків. Наївний спосіб зробити це – обчислити ймовірність всіх можливих послідовностей, але це нерозв'язне завдання навіть для послідовностей середньої довжини. Як і для оцінки параметрів, нам доведеться використовувати спеціальний алгоритм, щоб знайти найбільш ймовірну послідовність. Цей алгоритм тісно пов'язаний з алгоритмом вперед-назад, і його називають **алгоритмом Вітербі**.

Хоча ми не займаємось глибоким навчанням, бібліотека автоматичного знаходження похідної PyTorch допоможе нам навчити нашу модель CRF за допомогою градієнтного спуску без необхідності обчислювати будь-які градієнти вручну. Це заощадить багато роботи. Використання PyTorch змусить нас реалізувати forward частину алгоритму “вперед-назад” та алгоритм Вітербі, що є добре з повчальної точки зору, ніж використання спеціалізованого Python пакету CRF.

Для початку почнемо з того, як повинен виглядати результат. Нам потрібен метод для обчислення логарифмічної ймовірності для довільної послідовності спроб з урахуванням міток на кубиків. Ось один із способів, як це могло виглядати:

```
def neg_log_likelihood(self, rolls, states):
    """Compute neg log-likelihood for a given sequence.
    Input:
        rolls: numpy array, dim [1, n_rolls]. Integer 0-5 showing value on dice.
        states: numpy array, dim [1, n_rolls]. Integer 0, 1. 0 if dice is fair.
    """
    loglikelihoods = self._data_to_likelihood(rolls)
```

*states = torch.LongTensor(states)*

*sequence_loglik = self._compute_likelihood_numerator(loglikelihoods, states)*

*denominator = self._compute_likelihood_denominator(loglikelihoods)*

*return denominator - sequence_loglik*

Цей метод робить три основні речі:

1) порівнює значення на кубиках до ймовірності (метод _data_to_likelihood)

2) обчислює чисельник члену логарифмічної правдоподібності.

3) обчислює знаменник члену логарифмічної правдоподібності.

Спершу розглянемо метод _data_to_likelihood, який допоможе нам зробити крок 1. Що ми зробимо, це створити матрицю розміром 6 x 2, де перший стовпець – це ймовірність спроб 1–6 для «правильного» кубика, і другий стовпець – це ймовірність спроб 1–6 для «зваженого» кубика. Ось як виглядає ця матриця для нашого завдання:

*array([[-1.79175947, -3.21887582],*
*[-1.79175947, -3.21887582],*
*[-1.79175947, -3.21887582],*
*[-1.79175947, -3.21887582],*
*[-1.79175947, -3.21887582],*
*[-1.79175947, -0.22314355]])*

Тепер, якщо ми бачимо спробу кубика з 4, ми можемо просто вибрати четвертий ряд матриці. Перший запис цього вектора – це ймовірність четвірки для «правильних» кубиків (log(1/6)), а другий запис – це ймовірність четвірки для «зважених» кубиків (log (0,04)). Ось як виглядає код:

```
def _data_to_likelihood(self, rolls):
    """Converts a numpy array of rolls (integers) to log-likelihood.
    self.loglikeligood is a matrix of 6 x 2 in our case.
    Input is one [1, n_rolls]
    """
    log_likelihoods = self.loglikelihood[rolls]
    return Variable(torch.FloatTensor(log_likelihoods), requires_grad=False)
```

Далі напишемо методи обчислення чисельника та знаменника логарифмічної правдоподібності.

```
def _compute_likelihood_numerator(self, loglikelihoods, states):
    """Computes numerator of likelihood function for a given sequence.
    We'll iterate over the sequence of states and compute the sum
    of the relevant transition cost with the log likelihood of the observed
    roll.
    Input:
        loglikelihoods: torch Variable. Matrix of n_obs x n_states.
                        i,j entry is loglikelihood of observing roll i given state j
        states: sequence of labels
    Output:
        score: torch Variable. Score of assignment.
    """

    prev_state = self.n_states
    score = Variable(torch.Tensor([0]))
    for index, state in enumerate(states):
```

```
            score += self.transition[state, prev_state] + loglikelihoods[index, state]
            prev_state = state
        return score

    def _compute_likelihood_denominator(self, loglikelihoods):
        """Implements the forward pass of the forward-backward algorithm.
        We loop over all possible states efficiently using the recursive
        relationship: alpha_t(j) = \sum_i alpha_{t-1}(i) * L(x_t | y_t) * C(y_t | y{t-1} = i)
        Input:
            loglikelihoods: torch Variable. Same input as _compute_likelihood_numerator.
                            This algorithm efficiently loops over all possible state sequences
                            so no other imput is needed.
        Output:
            torch Variable.
        """
        # Stores the current value of alpha at timestep t
        prev_alpha = self.transition[:, self.n_states] + loglikelihoods[0].view(1, -1)
        for roll in loglikelihoods[1:]:
            alpha_t = []
            # Loop over all possible states
            for next_state in range(self.n_states):
                # Compute all possible costs of transitioning to next_state
                feature_function = self.transition[next_state, :self.n_states].view(1, -1) + \
                                   roll[next_state].view(1, -1).expand(1, self.n_states)
```

```
            alpha_t_next_state = prev_alpha + feature_function
            alpha_t.append(self.log_sum_exp(alpha_t_next_state))
        prev_alpha = torch.cat(alpha_t).view(1, -1)
    return self.log_sum_exp(prev_alpha)
```

Тепер у нас є весь код, який нам потрібен, щоб почати вивчати нашу матрицю переходу. Але якщо ми хочемо робити прогнози після тренування нашої моделі, нам доведеться закодувати алгоритм Вітербі:

```
def _viterbi_algorithm(self, loglikelihoods):
    """Implements Viterbi algorithm for finding most likely sequence of labels.
    Very similar to _compute_likelihood_denominator but now we take the maximum
    over the previous states as opposed to the sum.
    Input:
        loglikelihoods: torch Variable. Same input as _compute_likelihood_denominator.
    Output:
        tuple. First entry is the most likely sequence of labels. Second is
            the loglikelihood of this sequence.
    """
    argmaxes = []
    # prev_delta will store the current score of the sequence for each state
    prev_delta = self.transition[:, self.n_states].view(1, -1) + \
                 loglikelihoods[0].view(1, -1)

    for roll in loglikelihoods[1:]:
        local_argmaxes = []
        next_delta = []
```

```
            for next_state in range(self.n_states):
                feature_function = self.transition[next_state,
:self.n_states].view(1, -1) + \
                                   roll.view(1, -1) + \
                                   prev_delta
                most_likely_state = self.argmax(feature_function)
                score = feature_function[0][most_likely_state]
                next_delta.append(score)
                local_argmaxes.append(most_likely_state)
            prev_delta = torch.cat(next_delta).view(1, -1)
            argmaxes.append(local_argmaxes)

        final_state = self.argmax(prev_delta)
        final_score = prev_delta[0][final_state]
        path_list = [final_state]

        # Backtrack through the argmaxes to find most likely state
        for states in reversed(argmaxes):
            final_state = states[final_state]
            path_list.append(final_state)
        return np.array(path_list), final_score
```

Давайте оцінимо модель за деякими імітованими даними, використовуючи наступні ймовірності:

1) P(перший кубик в послідовності є «правильний») = 0,5

2) P(поточні кубики «правильні»|попередні кубики «правильні») = 0,8

3) P (поточні кубики «зважені»|попередні кубики «зважені») = 0,35

Зібравши весь код до купи, отримаємо наступну реалізацію (оригінал коду можна знайти за наступним посиланням:

https://github.com/freddyalfonsoboulton/crf_tutorial/blob/master/utils.py):

```
import torch
import torch.nn as nn
from torch.autograd import Variable
from torch.optim import Adam
import numpy as np

class CRF(nn.Module):
    # Class conditional random field (CRF)
    def __init__(self, n_dice, log_likelihood):
        super(CRF, self).__init__()
        self.n_states = n_dice
        self.transition =
torch.nn.init.normal(nn.Parameter(torch.randn(n_dice, n_dice + 1)), -1, 0.1)
        self.loglikelihood = log_likelihood

    def to_scalar(self, var):
        return var.view(-1).data.tolist()[0]

    def argmax(self, vec):
        _, idx = torch.max(vec, 1)
        return self.to_scalar(idx)

    # numerically stable log sum exp
    # Source:
http://pytorch.org/tutorials/beginner/nlp/advanced_tutorial.html
    def log_sum_exp(self, vec):
        max_score = vec[0, self.argmax(vec)]
```

```
        max_score_broadcast = max_score.view(1, -1).expand(1,
vec.size()[1])
        return max_score + \
            torch.log(torch.sum(torch.exp(vec - max_score_broadcast)))

    def _data_to_likelihood(self, rolls):
        """Converts a numpy array of rolls (integers) to log-likelihood.
        Input is one [1, n_rolls]
        """
        return Variable(torch.FloatTensor(self.loglikelihood[rolls]),
requires_grad=False)

    def _compute_likelihood_numerator(self, loglikelihoods, states):
        """Computes numerator of likelihood function for a given sequence.
        We'll iterate over the sequence of states and compute the sum
        of the relevant transition cost with the log likelihood of the observed
        roll.
        Input:
            loglikelihoods: torch Variable. Matrix of n_obs x n_states.
                        i,j entry is loglikelihood of observing roll i given state j
            states: sequence of labels
        Output:
            score: torch Variable. Score of assignment.
        """
        prev_state = self.n_states
        score = Variable(torch.Tensor([0]))
        for index, state in enumerate(states):
            score += self.transition[state, prev_state] + loglikelihoods[index,
state]
            prev_state = state
```

```
        return score

    def _compute_likelihood_denominator(self, loglikelihoods):
        """Implements the forward pass of the forward-backward algorithm.
        We loop over all possible states efficiently using the recursive
        relationship: alpha_t(j) = \sum_i alpha_{t-1}(i) * L(x_t | y_t) * C(y_t
| y{t-1} = i)
        Input:
          loglikelihoods: torch Variable. Same input as
_compute_likelihood_numerator.
                          This algorithm efficiently loops over all possible state
sequences
                          so no other imput is needed.
        Output:
          torch Variable.
        """

        # Stores the current value of alpha at timestep t
        prev_alpha = self.transition[:, self.n_states] +
loglikelihoods[0].view(1, -1)
        for roll in loglikelihoods[1:]:
          alpha_t = []
          # Loop over all possible states
          for next_state in range(self.n_states):
            # Compute all possible costs of transitioning to next_state
            feature_function = self.transition[next_state,
:self.n_states].view(1, -1) + \
                               roll[next_state].view(1, -1).expand(1, self.n_states)

            alpha_t_next_state = prev_alpha + feature_function
```

```
                alpha_t.append(self.log_sum_exp(alpha_t_next_state))
            prev_alpha = torch.stack(alpha_t).view(1, -1)
        return self.log_sum_exp(prev_alpha)

    def _viterbi_algorithm(self, loglikelihoods):
        """Implements Viterbi algorithm for finding most likely sequence of labels.
        Very similar to _compute_likelihood_denominator but now we take the maximum
        over the previous states as opposed to the sum.
        Input:
            loglikelihoods: torch Variable. Same input as _compute_likelihood_denominator.
        Output:
            tuple. First entry is the most likely sequence of labels. Second is
                the loglikelihood of this sequence.
        """
        argmaxes = []
        # prev_delta will store the current score of the sequence for each state
        prev_delta = self.transition[:, self.n_states].contiguous().view(1, -1) + \
                     loglikelihoods[0].view(1, -1)

        for roll in loglikelihoods[1:]:
            local_argmaxes = []
            next_delta = []
            for next_state in range(self.n_states):
                feature_function = self.transition[next_state, :self.n_states].view(1, -1) + \
                                   roll.view(1, -1) + \
```

```
                        prev_delta
                    most_likely_state = self.argmax(feature_function)
                    score = feature_function[0][most_likely_state]
                    next_delta.append(score)
                    local_argmaxes.append(most_likely_state)
                prev_delta = torch.stack(next_delta).view(1, -1)
                argmaxes.append(local_argmaxes)
            final_state = self.argmax(prev_delta)
            final_score = prev_delta[0][final_state]
            path_list = [final_state]
            # Backtrack through the argmaxes to find most likely state
            for states in reversed(argmaxes):
                final_state = states[final_state]
                path_list.append(final_state)
            return np.array(path_list), final_score

        def neg_log_likelihood(self, rolls, states):
            """Compute neg log-likelihood for a given sequence.
            Input:
                rolls: numpy array, dim [1, n_rolls]. Integer 0-5 showing value on
dice.
                states: numpy array, dim [1, n_rolls]. Integer 0, 1. 0 if dice is fair.
            """
            loglikelihoods = self._data_to_likelihood(rolls)
            states = torch.LongTensor(states)
            sequence_loglik                                          =
self._compute_likelihood_numerator(loglikelihoods, states)
            denominator                                              =
self._compute_likelihood_denominator(loglikelihoods)
            return denominator - sequence_loglik
```

```
    def forward(self, rolls):
        loglikelihoods = self._data_to_likelihood(rolls)
        return self._viterbi_algorithm(loglikelihoods)


def crf_train_loop(model, rolls, targets, n_epochs, learning_rate=0.01):
    # Train of CRF with Adam optimizer
    optimizer = Adam(model.parameters(), lr=learning_rate,
                     weight_decay=1e-4)
    for epoch in range(n_epochs):
        batch_loss = []
        N = rolls.shape[0]
        model.zero_grad()
        for index, (roll, labels) in enumerate(zip(rolls, targets)):
            # Forward Pass
            neg_log_likelihood = model.neg_log_likelihood(roll, labels)
            batch_loss.append(neg_log_likelihood)

            if index % 50 == 0:
                ll = torch.cat(batch_loss).mean()
                ll.backward()
                optimizer.step()
                print("Epoch {}: Batch {}/{} loss is {:.4f}".format(epoch, index //
50, N // 50, ll.data.numpy()))
                batch_loss = []
        return model

# INPUT DATA
# two dice one is fair, one is loaded
fair_dice = np.array([1/6]*6)
```

```
loaded_dice = np.array([0.04,0.04,0.04,0.04,0.04,0.8])
probabilities = {'fair': fair_dice, 'loaded': loaded_dice}
# if dice is fair at time t, 0.8 chance we stay fair, 0.2 chance it is loaded at time 2
transition_mat = {'fair': np.array([0.8, 0.2, 0.0]),
                  'loaded': np.array([0.35, 0.65, 0.0]),
                  'start': np.array([0.5, 0.5, 0.0])}
states = ['fair', 'loaded', 'start']
state2ix = {'fair': 0,
            'loaded': 1,
            'start': 2}

log_likelihood = np.hstack([np.log(fair_dice).reshape(-1,1),
                            np.log(loaded_dice).reshape(-1,1)])

def simulate_data(n_timesteps):
    data = np.zeros(n_timesteps)
    prev_state = 'start'
    state_list = np.zeros(n_timesteps)
    for n in range(n_timesteps):
        next_state = np.random.choice(states, p=transition_mat[prev_state])
        state_list[n] = state2ix[next_state]
        next_data = np.random.choice([0,1,2,3,4,5], p=probabilities[next_state])
        data[n] = next_data
        prev_state = next_state
    return data, state_list
n_obs = 15
rolls = np.zeros((5000, n_obs)).astype(int)
targets = np.zeros((5000, n_obs)).astype(int)
```

```
for i in range(5000):
    data, dices = simulate_data(n_obs)
    rolls[i] = data.reshape(1, -1).astype(int)
    targets[i] = dices.reshape(1, -1).astype(int)
model = CRF(2, log_likelihood)
model = crf_train_loop(model, rolls, targets, 10, 0.001)
torch.save(model.state_dict(), "./checkpoint.hdf5")
model.load_state_dict(torch.load("./checkpoint.hdf5"))
data, dices = simulate_data(15)
test_rolls = data.reshape(1, -1).astype(int)
test_targets = dices.reshape(1, -1).astype(int)
print(test_rolls[0])
print(model.forward(test_rolls[0])[0])
print(test_targets[0])
print(np.exp(list(model.parameters())[0].data.numpy()))
data, dices = simulate_data(15)
test_rolls = data.reshape(1, -1).astype(int)
test_targets = dices.reshape(1, -1).astype(int)
print(test_rolls[0])
print(model.forward(test_rolls[0])[0])
print(test_targets[0])
```

CRF використовуються в:

1) NLP (natural language processing), наприклад, для таких речей, як розпізнавання назв, тощо. Це має сенс у цьому випадку, тому що від того, чи належить слово "New" до назви, сильно залежить, якщо, наприклад, наступне слово "York". Але CRF має бути корисним у будь-якому сценарії, коли ви вивчаєте послідовність спостережень.

2) Обробка зображень – в після обробці при сегментації зображень, щоб згладити результати прогнозування.

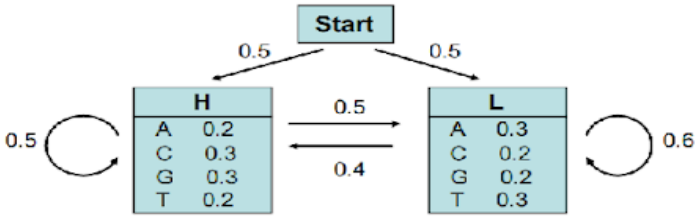
## Практична частина

1. **Реалізуйте модель надану у теоретичній частині та проведіть відповідний експеримент.**
2. **Як виглядає орієнтовна матриця переходу?** Проаналізуйте отримані результати. Спробуйте збільшити тривалість навчання, як зміниться результат? Зробіть висновки та вкажіть це у звіті.
3. **Задача генетики. Створіть модель та поясніть результат.**

   Розглянемо наступну просту модель. Ця модель складається з двох станів, H (високий вміст GC) та L (низький вміст GC). Для довідки, GC: У генетиці, вміст гуаніну і цітозину (часто скорочується як «вміст ГЦ» або «вміст GC», іноді навіть як GC%) – характеристика геному будь-якого даного організму або будь-якого іншого шматка ДНК або РНК. Ми можемо, наприклад, припустити, що стан H характеризує кодоване ДНК тоді як L характеризує некодоване ДНК.

   Таким чином, модель може бути використана для прогнозування області кодування ДНК із заданої послідовності.

   *Задані ймовірності:*




   *Розглянемо послідовність: S = GGCACTGAA.*

   *Існує кілька шляхів через приховані стани (H і L), які ведуть до заданої послідовності S.*

   *Наприклад: P = LLHHHHLLL*

## Висновки

У висновках обґрунтувати та проаналізувати отриманий результат. Надати відповіді на поставлені запитання у практичній частині.

## Контрольні запитання

1. У чому суть Conditional Random Fields?
2. Що являє собою алгоритм Viterbi?

# 16: РЕКУРЕНТНІ МЕРЕЖІ (RNN, GRU, LSTM)

**Мета роботи:** засвоїти принцип роботи та побудови рекурентних нейронних мереж.

## Теоретична частина

Ідея RNN полягає у використанні послідовної інформації. У традиційній нейронній мережі ми припускаємо, що всі входи (і виходи) незалежні один від одного. Але для багатьох завдань це дуже погана ідея. Якщо ви хочете, наприклад, передбачити наступне слово у реченні, то краще знати, які слова були перед ним. RNN називається рекурентною, оскільки вона виконує одне і те ж завдання для кожного елемента послідовності, при цьому вихід залежить від попередніх обчислень, і у них є "пам'ять", яка фіксує інформацію про попередні обчислення (рисунок 16.1-16.2).

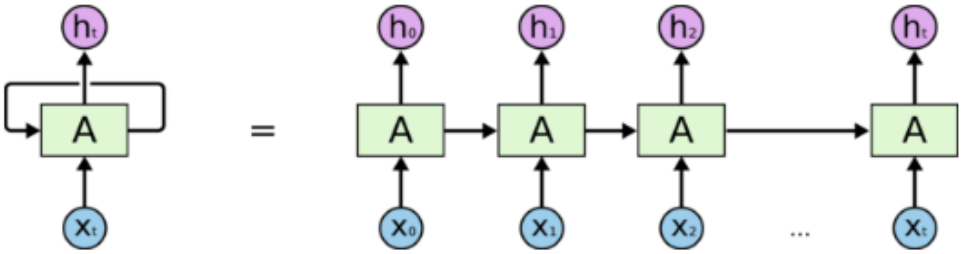



Рисунок 16.1 – Розгорнута модель RNN

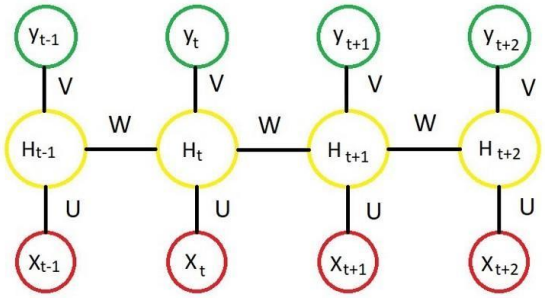



Рисунок 16.2 – Базові рівняння RNN: U – вектор ваг для прихованого шару; V – вектор ваг для вихідного шару; W – однаковий вектор ваг для різних часових кроків; X – вхідний вектор; y – вихідний вектор;

$$H_t = \sigma(U * X_t + W * H_{t-1});\ y_t = Softmax(V * H_t)$$

**Різні типи RNN**

Основна причина того, що рекурентні мережі є більш захоплюючими, полягає в тому, що вони дозволяють нам працювати над послідовностями векторів: Послідовності на вході, виході або в самому загальному випадку і на вході і на виході (рисунок 16.3)

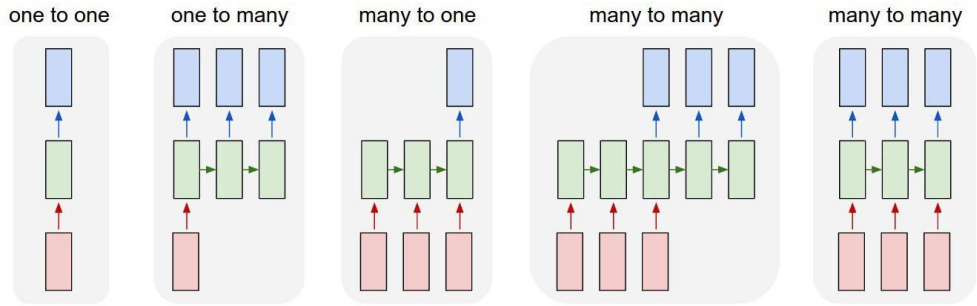



Рисунок 16.3 – Типи RNN моделей

Різні типи рекурентних нейронних мереж (рисунок 16.3 зліва направо): (1) one to one – звичайна глибока нейронна мережа прямого поширення (feed forward networks), один вхід – один вихід, RNN у цьому випадку не потрібні; вхід та вихід не залежать від попередньої інформації / виводу, наприклад класифікація зображення. (2) one to many – виведення послідовності (наприклад, субтитри зображення приймають зображення та виводять речення зі слів; на рисунку 16.4 – наведено приклад поєднання CNN та RNN мережі для такого типу завдання). (3) many to one – послідовний вхід (наприклад, аналіз настроїв, коли певне речення класифікується як вираження позитивних чи негативних настроїв). (4) many to many – послідовний вхід та вихід (наприклад, машинний переклад: RNN читає речення англійською мовою, а потім виводить речення французькою мовою). (5) many to many – синхронізована послідовність входу та виходу (наприклад, класифікація відео, де ми хочемо позначити кожен кадр відео). Зауважте, що в будь-якому випадку заздалегідь не визначено обмежень щодо довжин послідовностей, оскільки повторне перетворення (зелене) є фіксованим і може застосовуватися стільки разів, скільки нам подобається.

Кожен прямокутник на рисунку 16.3 представляє вектори, а стрілки представляють функції. Вхідні вектори червоного кольору, вихідні вектори синього та зеленого кольорів утримують стан RNN.

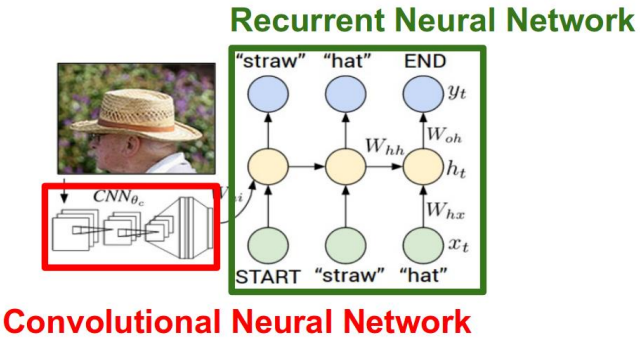



Рисунок 16.4 – Поєднання CNN та RNN мережі для задачі опису зображення

**Глибокий погляд на RNN**

У простій нейромережі ви можете бачити блок вводу, приховані одиниці та вихідні блоки, які обробляють інформацію незалежно, не маючи відношення до попередньої інформації. Також у простих мережах ми надавали різну вагу та зміщення прихованим одиницям, не даючи можливості запам'ятати будь-яку інформацію (рисунок 16.5). Ваги всередині RNN мережі наведені на рисунку 16.6, а узагальненні формули для розрахунків на рисунку 16.7.

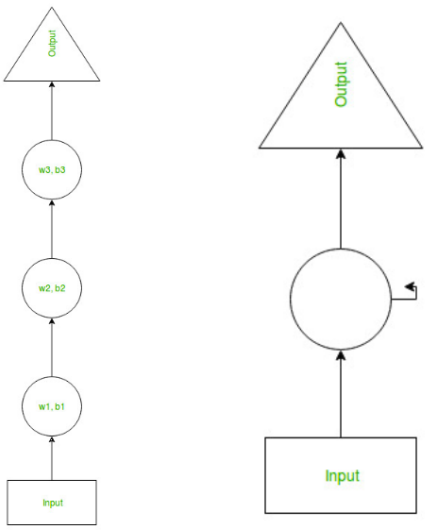



Рисунок 16.5 – Проста нейронна мережа (зліва) проти RNN (праворуч)

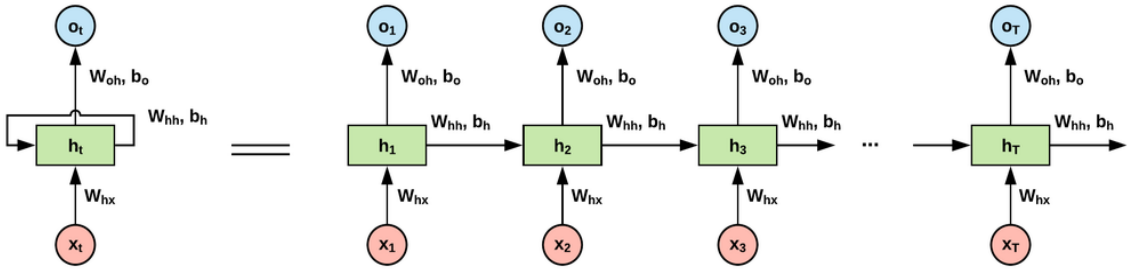



Рисунок 16.6 – Ваги RNN мережі

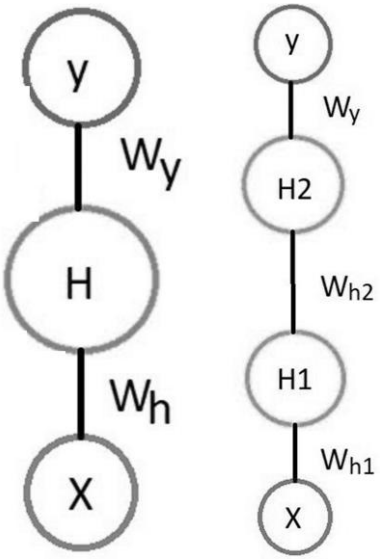



Рисунок 16.7 – Розрахунки всередині RNN по шарам: $H = \sigma(W_h * X);\ y = Softmax(W_y * H), \sigma -$ функція активації, $H_1 = \sigma(W_{h1} * X), H_2 = \sigma(W_{h2} * X), y = Softmax(W_y * H_2)$

**Поточна мітка часу**

*Поточний стан:*

$h_t = f(h_{t-1}, x_t)$ – це функція, що використовує результат на попередньому часовому кроці і вхід, дає поточний стан; $O_t$ – вихідний стан, $h_t$ – поточна мітка часу, $h_{t-1}$ – це попередня мітку часу, а $x_t$ передається як стан входу.

Після застосування функції активації маємо:

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

де W – ваги, h – одиничний прихований вектор, Whh – вага при попередньому прихованому стані, Whx – вага при поточному вхідному

стані, tanh – це функція активації, вона реалізує нелінійність, що притискає активацію до діапазону [-1.1]. Вихід: $y_t = W_{hy}h_t$, $y_t$ – вихідний стан, $W_{hy}$ – вага у вихідному стані. Повний розрахунок наданий на рисунку 16.8.

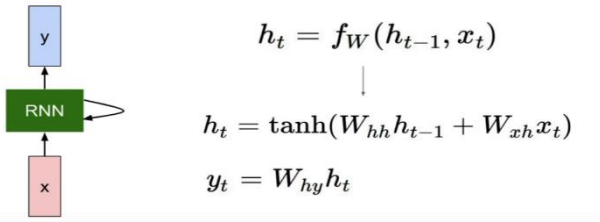



Рисунок 16.8 – Розрахунок всередині RNN комірки.

**Приклад: Мовна модель на рівні символів**

Надамо RNN величезний фрагмент тексту і попросимо його моделювати розподіл ймовірностей наступного символу в послідовності, заданої послідовністю попередніх символів (рисунок 16.9).

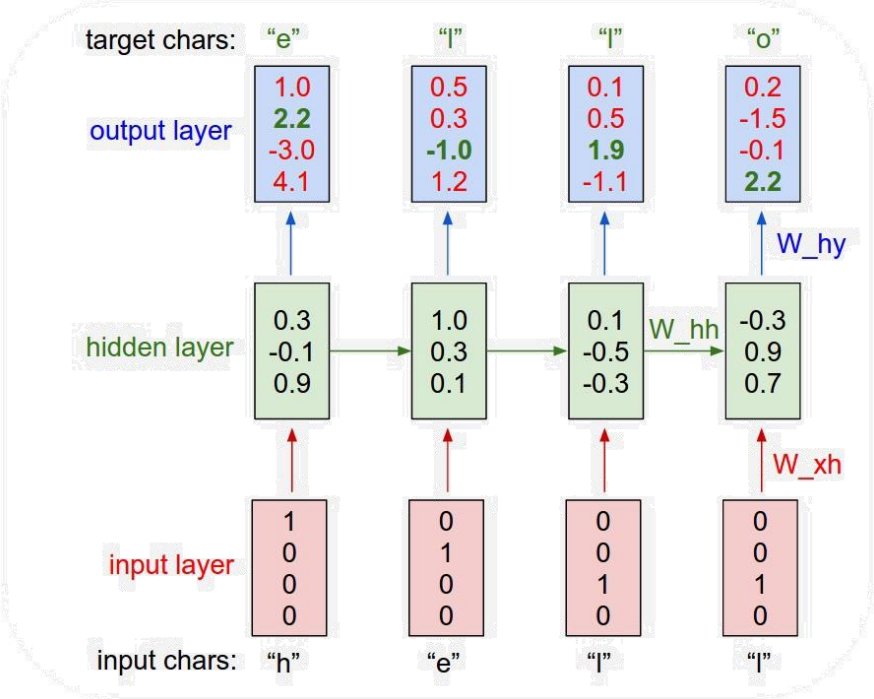



Рисунок 16.9 –Мовна модель на рівні символів

В якості прикладу роботи, припустимо, ми мали лише лексику з чотирьох можливих букв "helo", і хотіли навчити RNN на тренувальній послідовності "hello". Ця послідовність тренувань насправді є джерелом

чотирьох окремих навчальних прикладів: 1. Ймовірність "e" повинна бути ймовірною, враховуючи контекст "h", 2. "l", ймовірно, в контексті "he", 3. "l" також, ймовірно, слід враховувати контекст "hel" і, нарешті, 4. "o", ймовірно, повинен бути наданий контексту "hell" (посилання на оригінальну постановку задачі: http://karpathy.github.io/2015/05/21/rnn-effectiveness/ та https://www.analyticsvidhya.com/blog/2017/12/introduction-to-recurrent-neural-networks/).

**BPTT (зворотне поширення похибки у часі)**

Щоб зрозуміти та візуалізувати Backpropogation, розгорнемо мережу в усі часові позначки, щоб ми могли бачити, як оновлюються ваги. Повернення назад у кожен часовий крок, коли треба міняти / оновлювати ваги, називається Backpropogate через час. Приклад розрахунків наведений на рисунку 16.10.

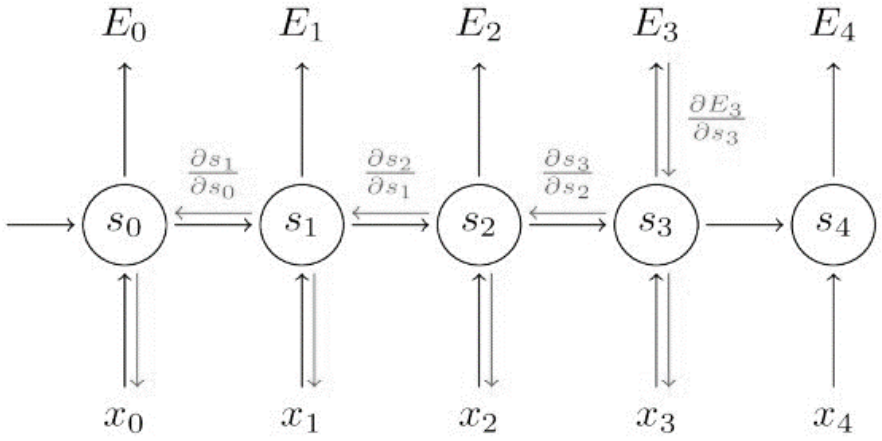



Рисунок 16.10 – Приклад розрахунку Backpropogation:

$$\frac{\partial E}{\partial W} = \sum_i \frac{\partial E_t}{\partial W}; \frac{\partial E_3}{\partial W} = \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial s_3}\frac{\partial s_3}{\partial W};\ s_3 = tanh\ tanh\ (Ux_t + Ws_2)$$

$s_3$ залежить від $s_2$, який залежить від $W$ та $s_1$ і так далі

$$\frac{\partial E_3}{\partial W} = \sum_i \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial s_3}\frac{\partial s_3}{\partial s_k}\frac{\partial s_k}{\partial W}$$

Ми зазвичай трактуємо повну послідовність (слово) як один навчальний приклад, тому загальна помилка – це лише сума помилок на

кожному кроці часу (символу). Ваги, як ми бачимо, однакові на кожному кроці.

Таким чином, що потрібно робити для Backpropogate через час:

1) Помилка перехресної ентропії спочатку обчислюється за допомогою поточного виходу та фактичного виходу.

2) Пам'ятайте, що мережа “розкручена” протягом усіх етапів часу.

3) Для “розкрученої” мережі градієнт обчислюється для кожного кроку часу щодо параметра ваги.

4) Тепер, коли вага однакова для всіх часових кроків, градієнти можна комбінувати разом для всіх часових кроків.

5) Потім ваги оновлюються як для повторюваних нейронів, так і для щільних шарів.

*Примітка*. Повернення до попереднього часу та оновлення його ваг – це дуже повільний процес. Це вимагає як обчислювальної ваги, так і часу.

Під час Backpropogate через час ви можете отримати 2 типи проблем.

1) Зникнення градієнта

2) Вибух градієнта

**Зникнення градієнта**: помилка – це різниця фактичної та прогнозованої моделі. А що робити, якщо часткова похідна похибки щодо ваги набагато менша за 1 (рисунок 16.11)?

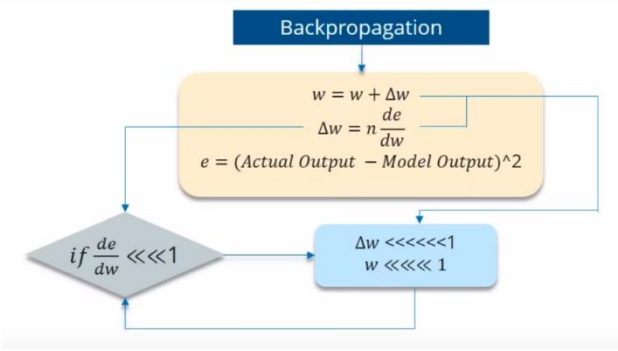



Рисунок 16.11 – Проблема зникнення градієнта

Якщо часткова похідна помилки менше 1, то коли її множать на коефіцієнт навчання, який також дуже малий, то результат не буде великою зміною порівняно з попередньою ітерацією.

Наприклад: скажімо, значення зменшилось у наступній послідовності: 0,863 → 0,532 → 0,356 → 0,192 → 0,111 → 0,086 → 0,023 → 0,019. Ми бачимо, що в останніх 3 ітераціях не сталося багато змін. Це зникнення градієнту називається зникаючим градієнтом (рисунок 16.12).

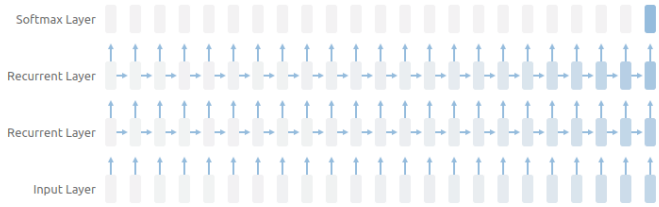



Рисунок 16.12 – Зникаючий градієнт, стан, де внесок попередніх етапів стає незначним у градієнті для RNN-комірки.

Також це призводить до того, що довготривалі залежності ігноруються під час тренувань. Ви можете візуалізувати цю проблему градієнта, що зникає, у режимі реального часу за наступним посиланням (https://distill.pub/2019/memorization-in-rnns/).

Протягом багатьох років було запропоновано кілька рішень проблеми зникаючого градієнта. Найпопулярнішими є архітектури LSTM та GRU.

**Вибух градієнта:** ми говоримо про вибухонебезпечні градієнти, коли алгоритм надає дуже велике значення вагам, без особливих причин. Але, на щастя, цю проблему можна легко вирішити, якщо обрізати або збити градієнти (рисунок 16.13).

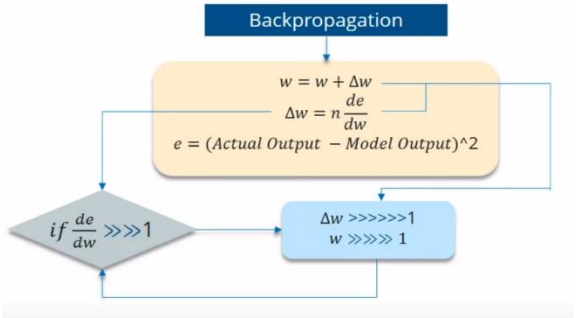


Рисунок 16.13 – Проблема вибуху градієнта

Отже, як можна подолати проблему зникаючого градієнту та вибухового градієнта?

Зникаючий градієнт: 1) Функція активації Relu; 2) LSTM, GRU.

Вибуховий градієнт: 1) Урізаний (truncated) BTT (замість того, щоб починати зворотне поширення помилки із останньою часу, ми можемо вибрати аналогічну позначку часу, яка є безпосередньо перед нею); 2) Обрізання градієнту за порогом (gradient clipping); RMSprop для регулювання швидкості навчання.

**Переваги RNN**

1) Основна перевага RNN над ANN полягає в тому, що RNN може моделювати послідовність даних (тобто часовий ряд), так що кожен зразок може вважатись залежним від попередніх.

2) RNN використовується навіть із згортковими шарами для розширення ефективного сусідства пікселів.

**Недоліки RNN**

1) Градієнт, що зникає і вибухає.

2) Навчання RNN – дуже складне завдання.

3) Він не може обробляти дуже довгі послідовності, якщо використовувати tanh або relu в якості функції активації.

**Побудова моделі**

У цій практичній роботі ми навчимося будувати рекурентну нейронну мережу (RNN) для класифікації зображень. Ви знову ж таки можете використовувати Google Colab для використання GPU. Модель RNN без використання вбудованих бібліотек PyTorch буде реалізована у спрощеному вигляді, вона дещо відрізняється від класичної моделі RNN.

Спочатку імпортуємо необхідні бібліотеки, які ми будемо використовувати:

*import torch*

*import torch.nn as nn*

*import torch.nn.functional as F*

*import os*

*import numpy as np*

**RNN з одиничним нейроном**

Спочатку побудуємо обчислювальний граф для одношарової RNN. Для наочності архітектура, яку ми будуємо представлена на рисунку 16.14.

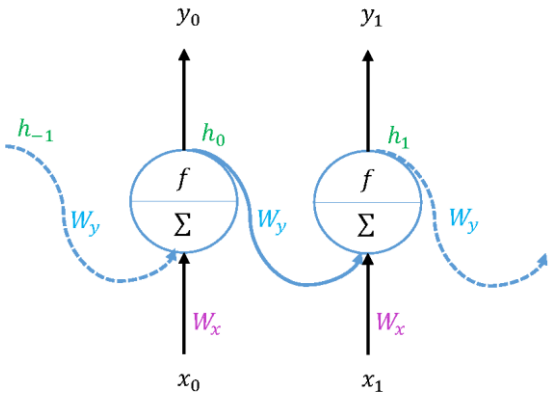



Рисунок 16.14 – RNN з одиничним нейроном

А це код під неї:

```
class SingleRNN(nn.Module):
    def __init__(self, n_inputs, n_neurons):
        super(SingleRNN, self).__init__()
        self.Wx = torch.randn(n_inputs, n_neurons)  # 4 X 1
        self.Wy = torch.randn(n_neurons, n_neurons)  # 1 X 1
        self.b = torch.zeros(1, n_neurons)  # 1 X 4

    def forward(self, X0, X1):
        self.Y0 = torch.tanh(torch.mm(X0, self.Wx) + self.b)  # 4 X 1
        self.Y1 = torch.tanh(torch.mm(self.Y0, self.Wy) +
                             torch.mm(X1, self.Wx) + self.b)  # 4 X 1
```

*return self.Y0, self.Y1*

У наведеному вище коді реалізовано простий один шар, один нейрон RNN. Ініціалізовані дві матриці ваг, Wx і Wy зі значеннями з нормального розподілу. Wx містить ваги з'єднання для входів поточного етапу часу, тоді як Wy містить ваги з'єднання для виходів попереднього етапу часу. Ми також додали зміщення b. Функція forward розраховує два виходи – по одному на кожен часовий крок (два в цілому). Зауважте, що ми використовуємо tanh як нелінійність (функція активації) через torch.tanh (...). Ви маєте можливість змінити її на сигмоїду.

Щодо входу, ми надаємо 4 екземпляри, причому кожен екземпляр містить дві вхідні послідовності.

Приклад подання даних на вхід в модель RNN наведений на рисунку 16.15.

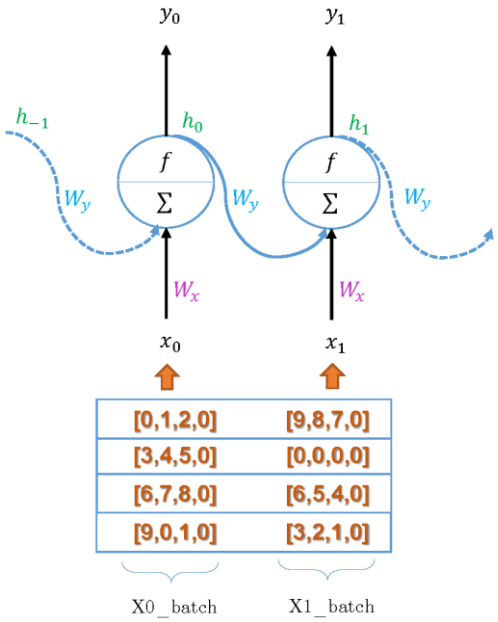



Рисунок 16.15 – Приклад подання даних на вхід RNN

Код для тестування моделі:

*N_INPUT = 4*

*N_NEURONS = 1*

```
X0_batch = torch.tensor([[0,1,2,0], [3,4,5,0],
                         [6,7,8,0], [9,0,1,0]],
                         dtype = torch.float) #t=0 => 4 X 4

X1_batch = torch.tensor([[9,8,7,0], [0,0,0,0],
                         [6,5,4,0], [3,2,1,0]],
                         dtype = torch.float) #t=1 => 4 X 4

model = SingleRNN(N_INPUT, N_NEURONS)
Y0_val, Y1_val = model(X0_batch, X1_batch)
```

Після того, як ми подали вхідні дані в обчислювальний граф, ми отримуємо виходи для кожного етапу часу (Y0, Y1), кожен розміром 4x1, який відображає розмір партії та приховані одиниці відповідно.

**Збільшення нейронів у шарі RNN**

Далі узагальнимо RNN, який ми тільки що створили, щоб один шар підтримував n кількість нейронів. З точки зору архітектури, насправді нічого не змінюється, оскільки ми вже параметризували кількість нейронів у створеному нами графі обчислень. Однак розмір виводу змінюється, оскільки ми змінили розмір кількості одиниць (тобто нейронів) у шарі RNN (рисунок 16.16).

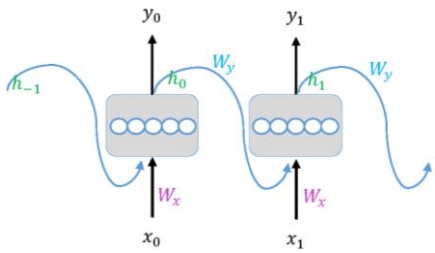



Рисунок 16.16 – Приклад RNN із декількома нейронами

Нам потрібно тільки змінити кількість нейронів N_NEURONS, наприклад поставимо

```
N_NEURONS = 5
```

Також змінимо трохи вхід до мережі

```
N_INPUT = 3 # number of features in input
N_NEURONS = 5 # number of units in layer

X0_batch = torch.tensor([[0,1,2], [3,4,5],
                         [6,7,8], [9,0,1]],
                         dtype = torch.float) #t=0 => 4 X 3

X1_batch = torch.tensor([[9,8,7], [0,0,0],
                          [6,5,4], [3,2,1]],
                         dtype = torch.float) #t=1 => 4 X 3

model = SingleRNN(N_INPUT, N_NEURONS)
Y0_val, Y1_val = model(X0_batch, X1_batch)
print(Y0_val)
print(Y1_val)
```

Тепер, коли ми друкуємо результати, отримані для кожного етапу часу, він має розмір (4x5), що представляє розмір партії та кількість нейронів відповідно.

**Вбудована комірка RNN у PyTorch**

Побудований нами граф обчислення має суттєвий недолік. Що робити, якщо ми намагаємося побудувати архітектуру, яка підтримує надзвичайно великі входи та виходи. Спосіб його побудови вимагає від нас індивідуального обчислення результатів для кожного кроку, збільшуючи

рядки коду, необхідні для реалізації потрібного графа обчислень. Нижче ми спробуємо використовувати вбудований модуль RNNCell для скорочення коду.

Спершу спробуємо реалізувати це неформально, без класу, для аналізу ролі, яку відіграє RNNCell (nn.RNNCell – вбудована функція для RNN у PyTorch):

```
rnn = nn.RNNCell(3, 5) # n_input X n_neurons
X_batch = torch.tensor([[[0,1,2], [3,4,5],
                         [6,7,8], [9,0,1]],
                        [[9,8,7], [0,0,0],
                         [6,5,4], [3,2,1]]
                        ], dtype = torch.float) # X0 and X1

hx = torch.randn(4, 5) # m X n_neurons
output = []
# for each time step
for i in range(2):
    hx = rnn(X_batch[i], hx)
    output.append(hx)
print(output)
```

За допомогою наведеного вище коду ми в основному реалізували ту саму модель, що була реалізована в SingleRNN класі. torch.RNNCell (...) виконує всю «магію» створення та підтримки необхідних для нас ваг (W) та зміщень (b). torch.RNNCell приймає тензор як вхід і видає наступний прихований стан для кожного елемента в пакеті. Детальніше про цей модуль можна прочитати за наступним посилання: https://pytorch.org/docs/stable/nn.html?highlight=rnncell#torch.nn.RNNCell.

Тепер давайте формально побудуємо граф обчислення, використовуючи ту саму інформацію, яку ми використовували вище, тобто обернемо код у клас:

```
class CleanBasicRNN(nn.Module):
    def __init__(self, batch_size, n_inputs, n_neurons):
        super(CleanBasicRNN, self).__init__()
        rnn = nn.RNNCell(n_inputs, n_neurons)
        self.hx = torch.randn(batch_size, n_neurons)  # initialize hidden state
    def forward(self, X):
        output = []
        # for each time step
        for i in range(2):
            self.hx = rnn(X[i], self.hx)
            output.append(self.hx)

        return output, self.hx

FIXED_BATCH_SIZE = 4  # our batch size is fixed for now
N_INPUT = 3
N_NEURONS = 5

X_batch = torch.tensor([[[0, 1, 2], [3, 4, 5],
                         [6, 7, 8], [9, 0, 1]],
                        [[9, 8, 7], [0, 0, 0],
                         [6, 5, 4], [3, 2, 1]]
                        ], dtype=torch.float)  # X0 and X1

model = CleanBasicRNN(FIXED_BATCH_SIZE, N_INPUT, N_NEURONS)
```

*output_val, states_val = model(X_batch)*

*print(output_val)  # contains all output for all timesteps*

*print(states_val)  # contains values for final state or final timestep, i.e., t=1*

Ви можете бачити, що код більш зрозумілий, оскільки нам не потрібно явно робити розрахунки на вагах, як показано в попередньому фрагменті коду – Pytorch обробляє все неявно та красномовно.

**RNN для класифікації зображень на прикладі датасету MNIST.**

В минулих роботах ми використовували повнозв'язну та згорткову мережу для класифікації зображень. Тепер спробуємо створити класифікатор зображення на основі рекурентної мережі за допомогою набору даних MNIST. Нагадую, що набір даних MNIST складається з зображень, що містять рукописні номери від 1–10. По суті, ми хочемо побудувати класифікатор для прогнозування чисел, відображених набором зображень. Це звучить дивно, але ви будете здивовані тим, наскільки добре RNN виконують цю задачу класифікації зображень.

Крім того, ми також будемо використовувати модуль RNN замість модуля RNNCell, оскільки ми хочемо узагальнити граф обчислень, щоб він також міг підтримувати n кількість шарів. Ми будемо використовувати лише один шар у наступному графі обчислень, але згодом ви можете експериментувати з кодом, додавши більше шарів.

**Імпорт набору даних**

Перш ніж будувати граф обчислення на основі RNN, давайте імпортуємо набір даних MNIST, розділимо його на тестові та навчальні частини, зробимо декілька перетворень та далі вивчимо його. Для завантаження та імпорту набору даних MNIST вам, як і в минулому разі, знадобляться такі бібліотеки Pytorch та рядки коду, також ми одразу вказали параметри (Hyper-parameters) нашої мережі

*import torch*

```
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms

# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
# Hyper-parameters
sequence_length = 28
input_size = 28
hidden_size = 128
num_layers = 2
num_classes = 10
batch_size = 64
num_epochs = 1
learning_rate = 0.01

# MNIST dataset
train_dataset = torchvision.datasets.MNIST(root='../../data/',
                                           train=True,
                                           transform=transforms.ToTensor(),
                                           download=True)

test_dataset = torchvision.datasets.MNIST(root='../../data/',
                                          train=False,
                                          transform=transforms.ToTensor())

# Data loader
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)
```

```
test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)
```

Код вище завантажує і готує набір даних для подачі в граф обчислень, який ми будемо будувати згодом. Зауважте, що нам потрібно ввести batch size. Це тому, що trainloader testloader – це ітератори, що полегшить роботу, коли ми перебираємо набір даних і навчаємо нашу модель RNN із minibatches.

**Модель**

Побудуємо обчислювальний граф нашої моделі на основі RNN та код для її реалізації (рисунок 16.17)

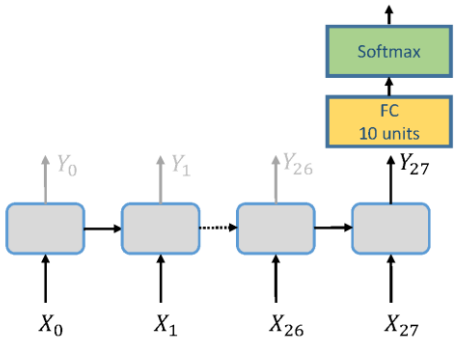



Рисунок 16.17 – Схематичне представлення моделі, яку будуємо

```
# Recurrent neural network (many-to-one)
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, num_classes):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.gru = nn.GRU(input_size, hidden_size, num_layers,
batch_first=True)
        self.fc = nn.Linear(hidden_size, num_classes)
```

```
  def forward(self, x):
    # x shape (batch, time_step, input_size)
    # out shape (batch, time_step, output_size)
    # h0 shape (n_layers, batch, hidden_size)
    # c0 shape (n_layers, batch, hidden_size)
        # Set initial hidden and cell states

        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(device)

        # c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(device)
        #  Forward propagate LSTM
        # out, _ = self.lstm(x, (h0, c0))  # out: tensor of shape (batch_size, seq_length, hidden_size)
        out, _ = self.gru(x, h0)  # out: tensor of shape (batch_size, seq_length, hidden_size)

        # Decode the hidden state of the last time step
        out = self.fc(out[:, -1, :])
        return out
```

**Модель RNN робить наступне:**

1) Функція ініціалізації __init __ (...) оголошує декілька змінних, а потім базовий RNN-шар, а потім повністю пов'язаний шар FC. Можете змінити RNN на GRU або LSTM шар. Ви можете змінювати параметри шарів, додати ключ dropout=True or False, якщо хочете його використати, чи bidirectional=True or False для використання двонаправленої мережі.

2) Функція forward приймає вхід розміру (X batch_size, n_steps, input_size). Дані протікають через рівень RNN, а потім через повністю пов'язаний шар.

3) batch_first=True – говорить, що вхід та вихід буде мати першим розміром розмір батчу (batch, time_step, input_size)

4) В self.gru(x, h0) замість позначення скритого шару h0, можна просто вказати None – self.gru(x, None), це означає нульові значення скритого/тих шарів.

Вихідні дані представляють log probabilities (ймовірності) моделі. Якщо хочете використовувати LSTM модель, Вам потрібно додати ще один скритий шар (cell memory – c0 та додати його на вхід шару (коментований код)).

Приклад представлення картинки як часової послідовності наведений на рисунку 16.18.

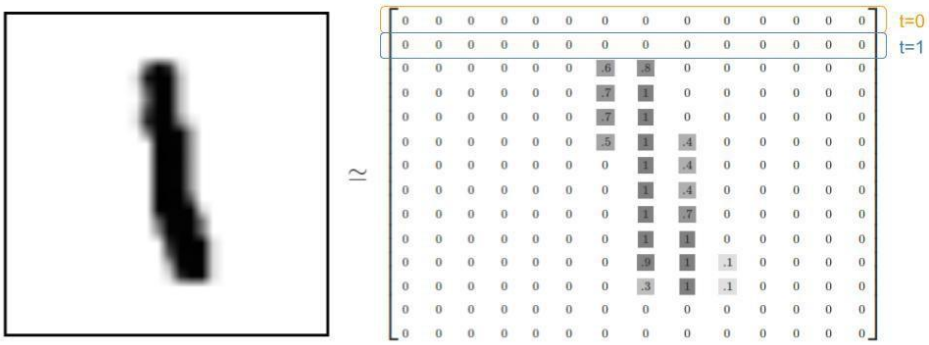



Рисунок 16.18 – Картинка як числова послідовність

**Навчання**

Тепер давайте розглянемо код для підготовки моделі класифікації зображень, навчання, тестування, та збереження моделі в кінці (він майже нічим не відрізняється від того, що ми робили раніше):

*model = RNN(input_size, hidden_size, num_layers, num_classes).to(device)*

*# Loss and optimizer*

```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
# Train the model
total_step = len(train_loader)
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        model.train()
        images = images.reshape(-1, sequence_length, input_size).to(device)
        labels = labels.to(device)
        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)
        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        if (i + 1) % 100 == 0:
            model.eval()
            with torch.no_grad():
                correct = 0
                total = 0
                for images, labels in test_loader:
                    images = images.reshape(-1, sequence_length, input_size).to(device)
                    labels = labels.to(device)
                    outputs = model(images)
                    _, predicted = torch.max(outputs.data, 1)
                    total += labels.size(0)
                    correct += (predicted == labels).sum().item()
                print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}, Test accuracy: {}'
```

```
                .format(epoch + 1, num_epochs, i + 1, total_step, loss.item(),
100 * correct / total))

    # Test the model
    model.eval()
    with torch.no_grad():
        correct = 0
        total = 0
        for images, labels in test_loader:
            images = images.reshape(-1, sequence_length, input_size).to(device)
            labels = labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
        print('Test Accuracy of the model on the 10000 test images: {}
%'.format(100 * correct / total))
        # print 10 predictions from test data
        print(predicted[:10], 'prediction numbers')
        print(labels[:10], 'real numbers')

    # Save the model checkpoint
    torch.save(model.state_dict(), 'model.ckpt')
```

1) Створюємо екземпляр моделі RNN (...) з відповідними параметрами.

2) Критерій являє собою функцію, яку ми будемо використовувати для обчислення втрат моделі. Функція nn.CrossEntropyLoss () в основному застосовує log софтмакс з подальшою операцією negative log likelihood loss

(негативна логарифмічна ймовірність) над виходом моделі. Для обчислення функції втрат потрібні як логарифмічні ймовірності та і цільові значення.

3) Для навчання нам також потрібен алгоритм оптимізації, який допомагає оновлювати ваги на основі поточних втрат. Це досягається функцією оптимізації optim.Adam, яка вимагає параметрів моделі та рівень навчання. Крім того, ви можете також використовувати optim.SGD або будь-який інший доступний алгоритм оптимізації.

4) Кожні 100 кроків ми будемо виводи, втрати та точність на тесті.

5) В кінці навчання знову протестуємо весь тестовий датасет та покажемо 10 передбачень для прикладу та збережемо модель.

Таким чином ви можете реалізувати базовий клас RNN в PyTorch. Ви також навчилися застосовувати RNN для вирішення проблеми класифікації зображень у реальному світі.

Наприкінці на рисунку 16.19 наведемо приклад порівняння рекурентних комірок.

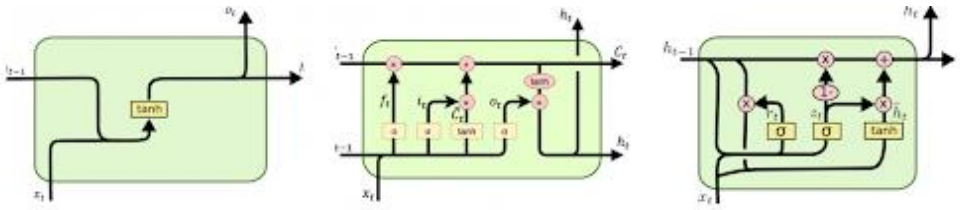



Рисунок 16.19 – Порівняння класичної RNN (зліва), LSTM (у центрі) та GRU (праворуч)

На рисунку 16.20 наведено базові рівняння для LSTM комірки (зліва) та GRU (справа).

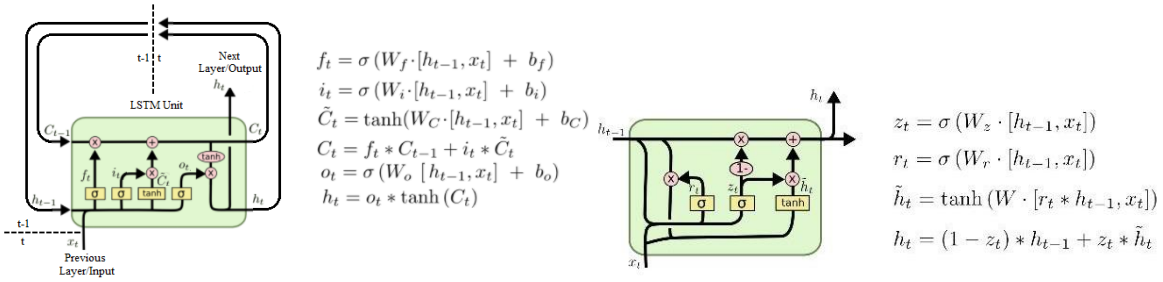



Рисунок 16.20 – Базові рівняння LSTM (зліва) та GRU (справа) комірок

## Практична частина

1. Зберіть код наведений у теоретичній частині, натренуйте модель, та проаналізуйте результат. Спробуйте його покращити методами, які вам вже знайомі. Наведіть найкращий результат на тесті.

2. Спробуйте інші архітектури рекурентних мереж, такі як GRU та LSTM (nn.GRU та nn.LSTM). Для GRU достатньо просто замінити комірку. А для LSTM необхідно подати 2 скритих стани lstm_out, (h_n, hc) = self.lstm(X, None). Проаналізуйте результат. Чи покращилися значення? Яка модель дала кращий результат?

3. Спробуйте реалізувати двонаправлену RNN модель (додати ключ bidirectional=True) в nn.RNN(…, bidirectional=True). Як треба змінити вхід та розміри? Спробуйте реалізувати та отримати результат.

4. Додайте шар нормалізації та dropout (можна просто вказати ключ True в середині nn.RNN()). Чи поліпшився результат?

5. Спробуйте адаптувати модель із цього завдання для вирішення набору даних CIFAR10 та CIFAR100. Проаналізуйте результат. Чи краще він ніж отриманий із згортковими мережами та повнозв’язними?

## Висновки

У висновках обґрунтувати вибір оптимальних параметрів нейронної мережі та навчання. Проаналізувати отримані результати, надати відповіді на поставлені питання у практичній частині.

## Контрольні запитання

1. Що являє собою рекурентна нейронна мережа?
2. Які різновиди рекурентних нейронних мереж існують?

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. Ian Pointer, Programming PyTorch for Deep Learning, O'Reilly Media, Inc., 2019.
2. Eli Stevens, Luca Antiga, Thomas Viehmann. Deep learning with PyTorch, Manning, Shelter Island, 2020, 522p.
3. Zhang, X. A Mathematical Model of a Neuron with Synapses based on Physiology. Nat Prec (2008). https://doi.org/10.1038/npre.2008.1703.1
4. Sima J. “Introduction to Neural Networks,” Technical Report No. V 755, Institute of Computer Science, Academy of Sciences of the Czech Republic, 1998.
5. Kröse B., and van der Smagt P. An Introduction to Neural Networks. (8th ed.) University of Amsterdam Press, University of Amsterdam, 1996.
6. Gurney K. An Introduction to Neural Networks. (1st ed.) UCL Press, London EC4A 3DE, UK, 1997.
7. Paplinski A.P. Neural Nets. Lecture Notes, Dept. of Computer Sciences and Software Eng., Manash Universtiy, Clayton-AUSTRALIA
8. Rosenblatt, F. (1958). "The Perceptron: A Probabilistic Model For Information Storage And Organization in the Brain". Psychological Review. 65 (6): 386–408. doi:10.1037/h0042519
9. Stephan Dreiseitl, Lucila Ohno-Machado. “Logistic regression and artificial neural network classification models: a methodology review,” Journal of Biomedical Informatics, 2002, pp. 352-359, https://doi.org/10.1016/S1532-0464(03)00034-0.
10. Ripley, B. D. (1994). Neural Networks and Related Methods for Classification. Journal of the Royal Statistical Society. Series B (Methodological), 56(3), 409–456. http://www.jstor.org/stable/2346118
11. Haykin, Simon. Neural networks and learning machines / Simon Haykin.- 3rd ed., Pearson Education Ltd., 2008, 936p.

12. Y. LeCun et al., “Backpropagation Applied to Handwritten Zip Code Recognition,” in Neural Computation, vol. 1, no. 4, pp. 541-551, Dec. 1989, doi: 10.1162/neco.1989.1.4.541.

13. J. Weng, N. Ahuja and T. S. Huang, “Cresceptron: a self-organizing neural network which grows adaptively,” [Proceedings 1992] IJCNN International Joint Conference on Neural Networks, 1992, pp. 576-581 vol.1, doi: 10.1109/IJCNN.1992.287150.

14. J. J. Weng, N. Ahuja and T. S. Huang, “Learning recognition and segmentation of 3-D objects from 2-D images,” 1993 (4th) International Conference on Computer Vision, 1993, pp. 121-128, doi: 10.1109/ICCV.1993.378228.

15. J. Weng, N. Ahuja and T.S. Huang. “Learning Recognition and Segmentation Using the Cresceptron,” International Journal of Computer Vision 25, 109–143 (1997). https://doi.org/10.1023/A:1007967800668.

16. Hinton, Geoffrey E. et al. “The "wake-sleep" algorithm for unsupervised neural networks,” Science, vol. 268 (5214), 1995, pp. 1158-61. doi:10.1126/science.7761831.

17. John F. Kolen; Stefan C. Kremer, “Gradient Flow in Recurrent Nets: The Difficulty of Learning LongTerm Dependencies,” in A Field Guide to Dynamical Recurrent Networks, IEEE, 2001, pp. 237-243, doi: 10.1109/9780470544037.ch14.

18. Morgan, Nelson; Bourlard, Hervé; Renals, Steve; Cohen, Michael and Franco, Horacio. “Hybrid neural network/hidden markov model systems for continuous speech recognition,” International Journal of Pattern Recognition and Artificial Intelligence, 1993, № 07 (4), pp. 899–916. doi:10.1142/s0218001493000455.

19. Rossi, Richard J. (2018). Mathematical Statistics: An Introduction to Likelihood Based Inference. New York: John Wiley & Sons. p. 227

20. Ward, Michael Don; Ahlquist, John S. (2018). Maximum Likelihood for Social Science : Strategies for Analysis. New York: Cambridge University Press

21. Press, W.H.; Flannery, B.P.; Teukolsky, S.A.; Vetterling, W.T. (1992). "Least Squares as a Maximum Likelihood Estimator". Numerical Recipes in FORTRAN: The Art of Scientific Computing (2nd ed.). Cambridge: Cambridge University Press. pp. 651–655.

22. Schwallie, Daniel P. (1985). "Positive definite maximum likelihood covariance estimators". Economics Letters. 17 (1–2): 115–117. doi:10.1016/0165-1765(85)90139-9.

23. Bottou, Léon (1998). "Online Algorithms and Stochastic Approximations". Online Learning and Neural Networks. Cambridge University Press

24. Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting,” JMLR, vol. 15(56), pp. 1929-1958, 2014.

25. Sergey Ioffe, Christian Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” PMLR, vol. 37, pp. 448-456, 2015.

26. Bilmes, Jeff; Asanovic, Krste; Chin, Chee-Whye; Demmel, James (April 1997). "Using PHiPAC to speed error back-propagation learning". 1997 IEEE International Conference on Acoustics, Speech, and Signal Processing. ICASSP. Munich, Germany: IEEE. pp. 4153-4156 vol.5. doi:10.1109/ICASSP.1997.604861

27. Hinton, Geoffrey. “Lecture 6e rmsprop: Divide the gradient by a running average of its recent magnitude,” p. 26, 2020.

28. Duchi, John; Hazan, Elad; Singer, Yoram (2011). “Adaptive subgradient methods for online learning and stochastic optimization,” JMLR, vol. 12, pp. 2121–2159.

29. Zeiler, Matthew D. (2012). "ADADELTA: An adaptive learning rate method". arXiv:1212.5701.

30. Ruder S (2016) An overview of gradient descent optimization algorithms. arXiv. Available online at: https://arxiv.org/pdf/1609.04747.pdf.

31. Kingma, Diederik; Ba, Jimmy (2014). "Adam: A Method for Stochastic Optimization". arXiv:1412.6980.

32. Spall, J. C. (2000). "Adaptive Stochastic Approximation by the Simultaneous Perturbation Method". IEEE Transactions on Automatic Control. 45 (10): 1839−1853. doi:10.1109/TAC.2000.880982.

33. Polyak, Boris T.; Juditsky, Anatoli B. (1992). "Acceleration of stochastic approximation by averaging" (PDF). SIAM J. Control Optim. 30 (4): 838–855. doi:10.1137/0330046

34. Yoshua Bengio. “Practical Recommendations for Gradient-Based Training of Deep Architectures.” Neural Networks: Tricks of the Trade (2012). doi:10.1007/978-3-642-35289-8_26.

35. Domínguez A. “A History of the Convolution Operation,” IEEE Pulse. — 2015. — Vol. 6, no. 1. — P. 38—49

36. Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," in Proceedings of the IEEE, vol. 86, no. 11, pp. 2278-2324, Nov. 1998, doi: 10.1109/5.726791.

37. Yamashita, R., Nishio, M., Do, R.K.G. et al. Convolutional neural networks: an overview and application in radiology. Insights Imaging 9, 611–629 (2018). https://doi.org/10.1007/s13244-018-0639-9

38. LeCun Y, Bengio Y, Hinton G. Deep learning. Nature. 2015 May 28;521(7553):436-44. doi: 10.1038/nature14539. PMID: 26017442.

39. Yasaka K, Akai H, Abe O, Kiryu S (2018) Deep learning with convolutional neural network for differentiation of liver masses at dynamic contrast-enhanced CT: a preliminary study. Radiology 286:887–896

40. Hubel DH, Wiesel TN (1968) Receptive fields and functional architecture of monkey striate cortex. J Physiol 195:215–243

41. Dan C. Cireşan, Alessandro Giusti, Luca M. Gambardella, and Jürgen Schmidhuber. “Deep neural networks segment neuronal membranes in electron microscopy images,” In Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'12), 2012. Curran Associates Inc., Red Hook, NY, USA, 2843–2851. doi: https://dl.acm.org/doi/10.5555/2999325.2999452.

42. A. Krizhevsky, I. Sutskever, and G. E. Hinton. “ImageNet classification with deep convolutional neural networks,” Commun. ACM, vol. 60, 6 (June 2017), 84–90. https://doi.org/10.1145/3065386.

43. Cireşan, D.C., Giusti, A., Gambardella, L.M., Schmidhuber, J. “Mitosis Detection in Breast Cancer Histology Images with Deep Neural Networks,” In: Mori, K., Sakuma, I., Sato, Y., Barillot, C., Navab, N. (eds) Medical Image Computing and Computer-Assisted Intervention – MICCAI 2013. MICCAI 2013. Lecture Notes in Computer Science, vol 8150. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-40763-5_51.

44. Hof, Robert D. “Is Artificial Intelligence Finally Coming into Its Own?” MIT Technology Review, MIT Technology Review, 29 Mar. 2016. [Режим доступу]: www.technologyreview.com/s/513696/deep-learning/.

45. Ciresan, Dan C. et al. “Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition.” ArXiv abs/1003.0358, 2010. [Режим доступу]: https://arxiv.org/pdf/1003.0358.pdf

46. Sermanet, Pierre et al. “OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks.” CoRR abs/1312.6229, 2014. [Режим доступу]: https://arxiv.org/pdf/1312.6229.pdf

47. Simonyan, Karen and Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition.” CoRR abs/1409.1556, 2015. [Режим доступу]: https://arxiv.org/abs/1409.1556

48. Lin, Min et al. “Network In Network.” CoRR abs/1312.4400, 2014. [Режим доступу]: https://arxiv.org/abs/1312.4400

49. Szegedy, Christian et al. “Going deeper with convolutions.” 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2015): 1-9. [Режим доступу]: https://arxiv.org/abs/1409.4842

50. Szegedy, Christian et al. “Rethinking the Inception Architecture for Computer Vision.” 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 2818-2826. doi:10.1109/CVPR.2016.308

51. He, Kaiming et al. “Deep Residual Learning for Image Recognition.” 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2016): 770-778. doi:10.1109/cvpr.2016.90

52. P. Sermanet and Y. LeCun, "Traffic sign recognition with multi-scale Convolutional Networks," The 2011 International Joint Conference on Neural Networks, 2011, pp. 2809-2813, doi: 10.1109/IJCNN.2011.6033589.

53. Szegedy, Christian et al. “Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning.” AAAI (2017). doi:10.1609/aaai.v31i1.11231

54. Iandola, Forrest N. et al. “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size.” ArXiv abs/1602.07360, 2016. [Режим доступу]: https://arxiv.org/abs/1602.07360

55. Paszke, Adam et al. “ENet: A Deep Neural Network Architecture for Real-Time Semantic Segmentation.” ArXiv abs/1606.02147, 2016. [Режим доступу]: https://arxiv.org/abs/1606.02147

56. Chollet, François. “Xception: Deep Learning with Depthwise Separable Convolutions,” 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2017): 1800-1807. doi: DOI:10.1109/CVPR.2017.195

57. Howard, Andrew G. et al. “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications.” ArXiv abs/1704.04861, 2017. [Режим доступу]: https://arxiv.org/abs/1704.04861

58. Larsson, Gustav et al. “FractalNet: Ultra-Deep Neural Networks without Residuals.” ArXiv abs/1605.07648, 2017. [Режим доступу]: https://arxiv.org/abs/1605.07648

59. Ronneberger, Olaf et al. “U-Net: Convolutional Networks for Biomedical Image Segmentation.” ArXiv abs/1505.04597, 2015. [Режим доступу]: https://arxiv.org/abs/1505.04597

60. Wan, Alvin et al. “NBDT: Neural-Backed Decision Trees.” ArXiv abs/2004.00221, 2020.[Режим доступу]: https://arxiv.org/abs/2004.00221

61. T. Robinson, "A real-time recurrent error propagation network word recognition system," [Proceedings] ICASSP-92: 1992 IEEE International Conference on Acoustics, Speech, and Signal Processing, 1992, pp. 617-620, vol.1, doi: 10.1109/ICASSP.1992.225833.

62. A. Waibel, T. Hanazawa, G. Hinton, K. Shikano and K. J. Lang, "Phoneme recognition using time-delay neural networks," in IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. 37, no. 3, pp. 328-339, March 1989, doi: 10.1109/29.21701.

63. J. Baker, Li Deng, Jim Glass, S. Khudanpur, C.-H. Lee, N. Morgan, D. O'Shaughnessy. “Research Developments and Directions in Speech Recognition and Understanding, Part 1” IEEE Signal Processing Magazine. № 26 (3), pp. 75–80. doi:10.1109/msp.2009.932166.

64. Y. Bengio. Artificial Neural Networks and their Application to Speech / Sequence Recognition, McGill University Ph.D. thesis. 1991. [Режим доступу]:https://escholarship.mcgill.ca/concern/file_sets/kw52j887f?locale=en

65. L. Deng, K. Hassanein, M. Elmasry. “Analysis of correlation structure for a neural predictive model with applications to speech recognition,” Neural Networks, vol. 7 (2), 1994, pp. 331–339. doi:10.1016/0893-6080(94)90027-2.

66. Machine Learning I Week 14: Sequence Learning Introduction, Alex Graves, Technische Universitaet Muenchen. [Режим доступу]: http://archive.www6.in.tum.de/www6/pub/Main/TeachingWs2009MachineLearning/Slides_09.pdf

67. CSC2535 2013: Advanced Machine Learning, Lecture 10: Recurrent neural networks, G. Hinton, University of Toronto. [Режим доступу]: https://www.cs.toronto.edu/~hinton/csc2535/notes/lec10new.pdf

68. CS224d Deep NLP, Lecture 8: Recurrent Neural Networks, Richard Socher, Stanford University. [Режим доступу]: https://cs224d.stanford.edu/lectures/CS224d-Lecture8.pdf

69. Supervised Sequence Labelling with Recurrent Neural Networks, Alex Graves, Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigten Dissertation. [Режим доступу]:

https://www.cs.toronto.edu/~graves/phd.pdf

70. The Unreasonable Effectiveness of Recurrent Neural Networks, Andrej Karpathy, blog About Hacker's guide to Neural Networks. [Режим доступу]: http://karpathy.github.io/2015/05/21/rnn-effectiveness/

71. Understanding LSTM Networks, Christopher Olah [Режим доступу]: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

72. Ryan Kiros, Yukun Zhu, Ruslan Salakhutdinov, Richard S. Zemel, Antonio Torralba, Raquel Urtasun, and Sanja Fidler, “Skip-thought vectors,” In Proceedings of the 28th International Conference on Neural Information Processing Systems, Vol. 2 (NIPS'15), MIT Press, Cambridge, MA, USA, 3294–3302. https://dl.acm.org/doi/10.5555/2969442.2969607

73. Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier. “Language modeling with gated convolutional networks,” In Proceedings of the 34th International Conference on Machine Learning, vol. 70 (ICML'17), JMLR.org, pp. 933–941.

https://dl.acm.org/doi/10.5555/3305381.3305478

74. Karol Gregor, Ivo Danihelka, Alex Graves, Danilo Rezende, Daan Wierstra. “DRAW: A recurrent neural network for image generation,” Proceedings of the 32nd International Conference on Machine Learning, PMLR 37:1462-1471, 2015. [Режим доступу]: https://proceedings.mlr.press/v37/gregor15.html.

75. Hang Chu, Raquel Urtasun, and Sanja Fidler. “Song From PI: A Musically Plausible Network for Pop Music Generation,” arXiv preprint arXiv:1611.03477, 2016. [Режим доступу]: https://arxiv.org/pdf/1611.03477.pdf

76. Zheng, Shuai, et al. “Conditional random fields as recurrent neural networks,” Proceedings of the IEEE International Conference on Computer Vision. 2015, pp. 1529-1537. doi:10.1109/ICCV.2015.179.

77. Ramanathan, Vignesh, et al. “Detecting events and key actors in multi-person videos,” arXiv preprint arXiv:1511.02917, 2015. [Режим доступу]: https://arxiv.org/abs/1511.02917

78. Shah, Rajiv, and Rob Romijnders. “Applying Deep Learning to Basketball Trajectories,” arXiv preprint arXiv:1608.03793, 2016. [Режим доступу]: https://arxiv.org/pdf/1608.03793.pdf

79. Moez Baccouche, Franck Mamalet, Christian Wolf, Christophe Garcia, Atilla Baskurt. “Action Classification in Soccer Videos with Long Short-Term Memory Recurrent Neural Networks,” 20th International Conference on Artificial Neural Networks (ICANN), Sep 2010, Thessaloniki, Greece. pp.154-159, doi:10.1007/978-3-642-15822-3_20.

80. Chung, Junyoung, et al. “Empirical evaluation of gated recurrent neural networks on sequence modeling,” arXiv preprint arXiv:1412.3555 (2014). [Режим доступу]: https://arxiv.org/abs/1412.3555

81. Greff, Klaus et al. “LSTM: A Search Space Odyssey,” IEEE Transactions on Neural Networks and Learning Systems, vol. 28, 2017, pp. 2222-2232 doi:10.1109/tnnls.2016.2582924.

82. A. Graves, M. Liwicki, S. Fernandez, R. Bertolami, H. Bunke, J. Schmidhuber. “A Novel Connectionist System for Improved Unconstrained Handwriting Recognition,” IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 31, no. 5, 2009. doi: 10.1109/TPAMI.2008.137.

83. F. Gomez, J. Schmidhuber, R. Miikkulainen. “Accelerated Neural Evolution through Cooperatively Coevolved Synapses,” Journal of Machine

Learning Research (JMLR), vol. 9, pp. 937-965, 2008. doi: https://dl.acm.org/doi/10.5555/1390681.1390712.

84. J. Schmidhuber, D. Wierstra, M. Gagliolo, F. Gomez. “Training Recurrent Networks by Evolino,” Neural Computation, vol. 19(3), pp. 757-779, 2007. doi: https://doi.org/10.1162/neco.2007.19.3.757

85. F. Gers, N. Schraudolph, J. Schmidhuber. “Learning precise timing with LSTM recurrent networks,” Journal of Machine Learning Research, vol. 3, pp. 115-143, 2002. doi: https://doi.org/10.1162/153244303768966139.

86. J. Schmidhuber, F. Gers, D. Eck. J. Schmidhuber, F. Gers, D. Eck. “Learning nonregular languages: A comparison of simple recurrent networks and LSTM,” Neural Computation, vol. 14(9), pp. 2039-2041, 2002. doi: https://doi.org/10.1162/089976602320263980.

87. F. A. Gers and E. Schmidhuber, "LSTM recurrent networks learn simple context-free and context-sensitive languages," in IEEE Transactions on Neural Networks, vol. 12, no. 6, pp. 1333-1340, Nov. 2001, doi: 10.1109/72.963769.

88. F. A. Gers and J. Schmidhuber and F. Cummins. “Learning to Forget: Continual Prediction with LSTM,” Neural Computation, vol. 12(10), pp. 2451-2471, 2000, doi: 10.1049/cp:19991218.

89. S. Hochreiter and J. Schmidhuber. “Long Short-Term Memory,” Neural Computation, vol. 9(8), pp. 1735-1780, 1997. doi: https://doi.org/10.1162/neco.1997.9.8.1735

90. Natekin A and Knoll A. “Gradient boosting machines, a tutorial,” Front. Neurorobot. Vol. 7:21. 2013. doi: 10.3389/fnbot.2013.00021

91. Cho, Kyunghyun et al. “Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation.” EMNLP, 2014. doi:10.3115/v1/D14-1179

92. Yao, Kaisheng et al. “Depth-Gated LSTM.” ArXiv abs/1508.03790, 2015. [Режим доступу]: https://arxiv.org/abs/1508.03790

93. Koutník, Jan et al. “A Clockwork RNN.” ICML, 2014. [Режим доступу]: https://arxiv.org/abs/1402.3511?context=cs.NE

94. Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. 2015. An empirical exploration of recurrent network architectures. In Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15). JMLR.org, 2342–2350. doi: https://dl.acm.org/doi/10.5555/3045118.3045367

95. Mao, Junhua et al. “Deep Captioning with Multimodal Recurrent Neural Networks (m-RNN).” arXiv: Computer Vision and Pattern Recognition, 2015. [Режим доступу]: https://arxiv.org/abs/1412.6632

96. J. Langr and V. Bok. GAN in Action: Deep Learning with Generative Adversarial Networks, Manning, 2019, 276p.

97. Ramponi, Giorgia et al. “T-CGAN: Conditional Generative Adversarial Network for Data Augmentation in Noisy Time Series with Irregular Sampling.” ArXiv abs/1811.08295, 2018. [Режим доступу]: https://arxiv.org/abs/1811.08295

98. Mirza, Mehdi and Simon Osindero. “Conditional Generative Adversarial Nets.” ArXiv abs/1411.1784, 2014. [Режим доступу]: https://arxiv.org/abs/1411.1784

99. Radford, Alec et al. “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks.” CoRR abs/1511.06434, 2016. [Режим доступу]: https://arxiv.org/abs/1511.06434.

100. H. Zhang et al., "StackGAN: Text to Photo-Realistic Image Synthesis with Stacked Generative Adversarial Networks," 2017 IEEE International Conference on Computer Vision (ICCV), 2017, pp. 5908-5916, doi: 10.1109/ICCV.2017.629.

101. Emily Denton, Soumith Chintala, Arthur Szlam, and Rob Fergus. 2015. Deep generative image models using a Laplacian pyramid of adversarial networks. In Proceedings of the 28th International Conference on Neural

Information Processing Systems - Volume 1 (NIPS'15). MIT Press, Cambridge, MA, USA, 1486–1494. doi: https://dl.acm.org/doi/10.5555/2969239.2969405

102. A. Arnab et al., "Conditional Random Fields Meet Deep Neural Networks for Semantic Segmentation: Combining Probabilistic Graphical Models with Deep Learning for Structured Prediction," in IEEE Signal Processing Magazine, vol. 35, no. 1, pp. 37-52, Jan. 2018, doi: 10.1109/MSP.2017.2762355

103. [Електронний ресурс] Conditional Random Field Tutorial in PyTorch, [Режим доступу]: https://towardsdatascience.com/conditional-random-field-tutorial-in-pytorch-ca0d04499463

104. Sutton, Charles and Andrew McCallum. “An Introduction to Conditional Random Fields.” Found. Trends Mach. Learn, vol. 4, 2012, pp. 267-373.

105. [Електронний ресурс] Advanced: Making dynamic decisions and the Bi-LSTM CRF [Режим доступу]: https://pytorch.org/tutorials/beginner/nlp/advanced_tutorial.html

106. [Електронний ресурс] PyTorch tutorials [Режим доступу]: https://pytorch.org/tutorials/#

107. [Електронний ресурс] Deep Learning with PyTorch Step-by-Step [Режим доступу]: https://github.com/dvgodoy/PyTorchStepByStep

108. [Електронний ресурс] Daniel Godoy. Understanding PyTorch with an example: a step-by-step tutorial [Режим доступу]: https://towardsdatascience.com/understanding-pytorch-with-an-example-a-step-by-step-tutorial-81fc5f8c4e8e

109. [Електронний ресурс] Neural networks and computer vision [Режим доступу]: https://stepik.org/course/50352/promo

**Реєстр. № НП 22/23-132. Обсяг 8,85 авт. арк.**

**Національний технічний університет України**

**«Київський політехнічний інститут імені Ігоря Сікорського»**
**проспект Перемоги, 37, м. Київ, 03056**
**https://kpi.ua**
**Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.**