МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний аерокосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»

К. Ю. Дергачов, Л. О. Краснов, А. В. Шостак

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОЕКТУВАННЯ ТЕХНІЧНИХ СИСТЕМ

Частина 1

ОСНОВИ ПОБУДОВИ Й ВИКОРИСТАННЯ НЕЙРОННИХ МЕРЕЖ

Навчальний посібник

Харків «ХАІ» 2021

УДК 004.421
К78

Рецензенти: канд. техн. наук С. М. Флерко,
канд. техн. наук В. О. Кочура

Краснов, Л. О.

К78 Об'єктно-орієнтоване проектування технічних систем [Текст] : навч. посіб. Ч.1. Основи побудови й використання нейронних мереж / К. Ю. Дергачов, Л. О. Краснов, А. В. Шостак. – Харків : Нац. аерокосм. ун-т ім. М. Є. Жуковського «Харків. авіац. ін-т», 2021. – 168 с.

ISBN 978-966-662-811-7

Подано матеріали для практичного вивчення методів побудови, оптимізації архітектури й навчання сучасних нейронних мереж для різних додатків. Проведено досить детальний огляд сучасних методів та алгоритмів побудови й навчання глибинних нейронних мереж DNN (Deep Neural Network). Розглянуто ідею ієрархічного узагальнення й керування наявними ресурсами з допомогою пакета функцій та алгоритмів для керування параметрами мереж глибокого навчання в бібліотеці OpenCV.

Для студентів, що навчаються за напрямами підготовки «Авіоніка», «Аеронавігація» та «Системна інженерія» спеціальностей «Системи керування літальними апаратами та комплексами», «Комп`ютеризовані системи управління та автоматика», «Аеронавігаційні системи і системи аеронавігаційного обслуговування».

Іл. 106. Табл. 6. Бібліогр.: 38 назв

УДК 004.421

© Дергачов К. Ю., Краснов Л. О., Шостак А. В., 2021
© Національний аерокосмічний
університет ім. М. Є. Жуковського
«Харківський авіаційний інститут», 2021

ISBN 978-966-662-811-7

ПЕРЕДМОВА

Цим навчальним посібником ми починаємо серію публікацій з проблеми створення й використання нейронних мереж, в основному орієнтованих на системи класифікації й оброблення зображень, системи керування безпілотними літальними апаратами, автомобілями та іншими транспортними засобами.

Бурхливий розвиток комп'ютерних технологій (істотне підвищення швидкодії, значне збільшення обсягів пам'яті, створення нових операційних систем і мов програмування, апаратна мініатюризація та енергетична незалежність) природно привів до масового впровадження інноваційних методів та алгоритмів оброблення даних, що базуються на використанні штучного інтелекту. Особливо яскраво ця тенденція виявляється в області комп'ютерного зору при вирішенні широкого кола завдань розпізнавання образів і побудови сучасних систем технічного зору (СТЗ).

Методи штучного інтелекту набули практичного застосування в завданнях з організації високоякісного відеоспостереження, у робототехніці, при оснащенні безпілотних літальних апаратів, автомобільного транспорту та ін. Імовірно, одним з найбільш актуальних і важливих завдань СТЗ є розпізнавання облич (face recognition). Практичні потреби у вирішенні таких завдань у найрізноманітніших ситуаціях постійно зростають. Це забезпечення безпеки і face-control у сегменті масових громадських заходів і розваг, пошук потенційно небезпечних відвідувачів, підозрюваних у терористичних намірах, верифікація банківських карт і online-платежі, а також багато інших актуальних і корисних додатків.

Основною метою цієї роботи є формування у читачів ясних початкових теоретичних уявлень про основні види нейронних мереж, принципи їх роботи та головні області застосування. Крім того, ставиться завдання отримання практичних навичок самостійного створення відносно простих нейронних мереж без використання складних бібліотек. Для написання програмних кодів використовується мова програмування Python.

У роботі також проведено досить детальний огляд сучасних методів та алгоритмів побудови й навчання глибинних нейронних мереж DNN (Deep Neural Network). На прикладі завдання детектування облич (face detection) викладено різні варіанти побудови нейронних мереж з цією метою. Подано посилання на сучасні ресурси для вирішення цих завдань (бібліотеки OpenCV, TensorFlow, Keras) і проаналізовано супутні науково-технічні питання побудови реально діючих систем технічного зору.

Наведені відомості безсумнівно будуть корисними при створенні й експлуатації сучасних технічних проектів різного призначення, що використовують нейромережні технології.

Надалі планується опублікувати цикл лабораторних робіт, які, безсумнівно, допоможуть закріпити отримані знання й будуть корисними для самостійного проектування різних нейронних мереж.

1. ОСНОВИ ПОБУДОВИ Й НАВЧАННЯ НЕЙРОННИХ МЕРЕЖ

1.1. Загальна класифікація нейронних мереж

Спочатку творці систем штучного інтелекту заклали в основу їх роботи штучні нейронні мережі (ШНМ). Базова модель ШНМ є програмною реалізацією нейронної мережі структур головного мозку людини. Біологічна модель такої мережі − це деяка послідовність шарів нейронів, з'єднаних між собою синапсами. Нейрон − електрично збудлива клітина, призначена для приймання, оброблення, зберігання, передання та виведення інформації з допомогою електричних і хімічних сигналів. Синапс (з'єднання, зв'язок) − місце контакту між двома нейронами або між нейроном і клітиною, що одержує сигнал. Він передає нервовий імпульс між двома клітинами, причому під час синаптичного передання амплітуда й частота сигналу можуть змінюватися (посилюватися або послаблюватися) і регулюватися. Нерідко як еквівалентну заміну терміна «синапс» у ШНМ використовують термін «перцептрон».

Безумовно, програмна імітація роботи біологічних нейронних мереж дає дуже грубий опис оброблення інформації людським мозком, оскільки він містить велику кількість взаємозв'язаних нейронів, а сигнал, що передається одним нейроном, може передаватися в тисячі інших. Навчання відбувається через повторну активацію деяких нейронних з'єднань. При цьому збільшується ймовірність виведення потрібного результату при відповідній вхідній інформації (сигналах). Цей вид навчання використовує зворотний зв'язок − при правильному результаті нейронні зв'язки, які виводять його, стають більш щільними.

До теперішнього часу розроблено й запропоновано до використання в різних додатках безліч різновидів і типів нейронних мереж, тому для успішного вивчення роботи нейромережних технологій доречно провести їх детальну класифікацію. Результати класифікації за сукупністю інформаційних параметрів, моделями побудови й методами навчання показано на рис. 1.1.

Розглянемо детальніше структурні моделі нейронних мереж. Кожна мережа містить перший (вхідний) шар нейронів, який не виконує будь-яких перетворень та обчислень, а приймає й розподіляє вхідні сигнали по інших нейронах. Цей шар є загальним для всіх типів нейронних мереж. Для практичного використання можна виділити нейронні мережі з одношаровою і багатошаровою структурою.

Мережі з одношаровою структурою − структура взаємодії нейронів, у якій сигнали зі вхідного шару відразу прямують у вихідний шар. У ньому вони перетворюються, і відразу ж формується відповідь нейронної мережі. Як уже зазначалося, перший вхідний шар тільки приймає і розподіляє сигнали, а потрібні обчислення відбуваються вже в другому шарі.

Вхідні нейрони є об'єднаними з основним шаром з допомогою синапсів з різними вагами, що забезпечують відповідну якість зв'язків.

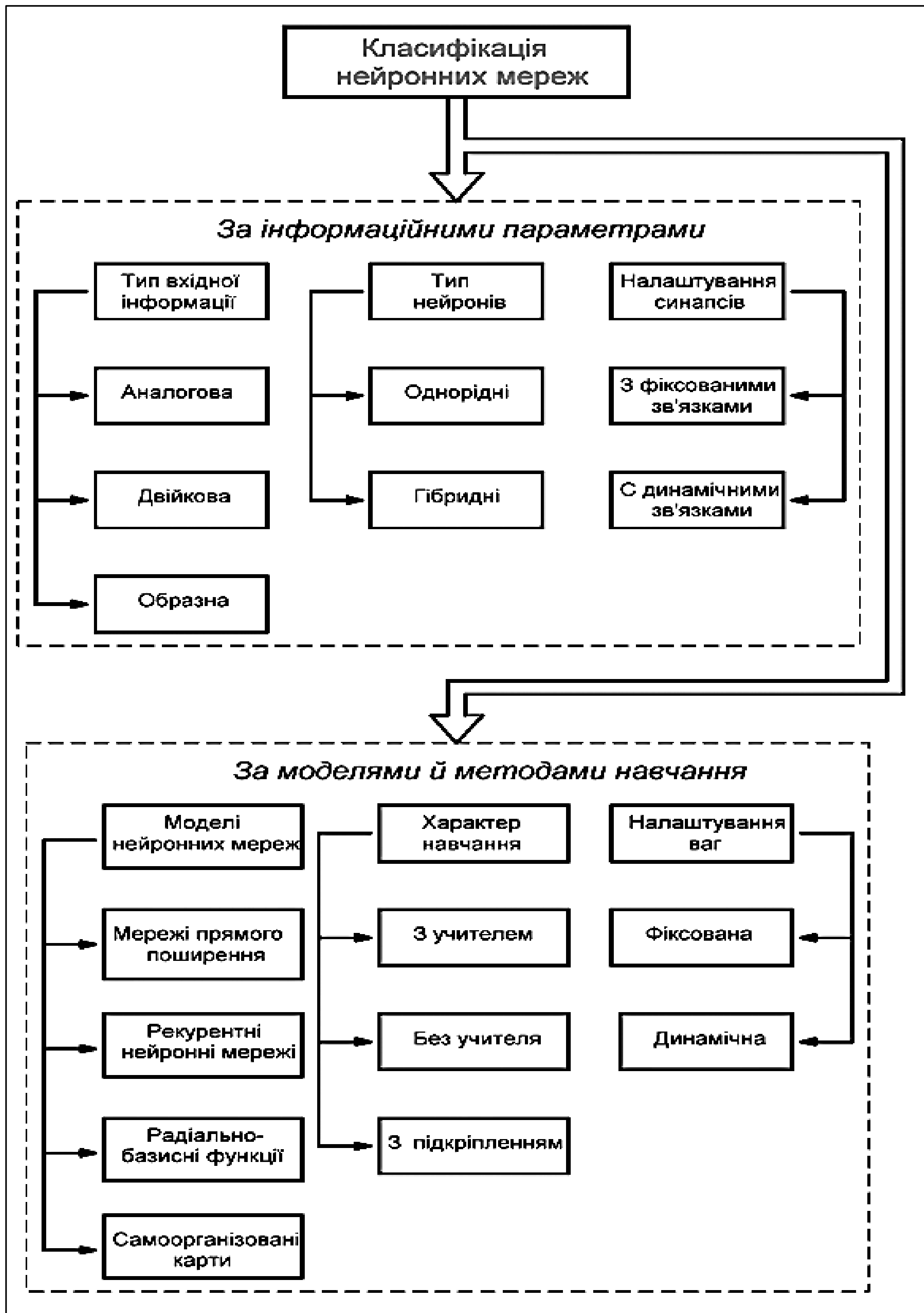


Рис. 1.1. Класифікація основних властивостей і структури нейронних мереж

Багатошарові нейронні мережі крім вихідного і вхідного шарів нейронів містять ще й кілька прихованих (проміжних) шарів. Їх кількість може бути різною залежно від складності мережі. Такі мережі більше відповідають структурі біологічних нейронних мереж, мають більші можливості порівняно з одношаровими, але потребують і більш істотних обчислювальних ресурсів.

Нейронні мережі також прийнято класифікувати за напрямком розподілу інформації по перцептронах між нейронами.

Нейронні мережі прямого поширення (односпрямовані). У цій структурі сигнал переміщається строго в напрямку від вхідного шару до вихідного. Рух сигналу в зворотному напрямку не здійснюється і в принципі є неможливим. Такі проекти застосовуються для вирішення завдань розпізнавання образів, прогнозування та кластеризації.

Рекурентні нейронні мережі (зі зворотними зв'язками). Тут сигнал рухається як в прямому, так і в зворотному напрямку, унаслідок чого результат з виходу мережі може повертатися на вхід. Вихід нейрона визначається ваговими характеристиками і вхідними сигналами, доповнюється попередніми виходами, що знову повернулися на вхід. Ці нейронні мережі виконують функцію короткочасної пам'яті, тому сигнали відновлюються й доповнюються під час їх оброблення.

Згорткові нейронні мережі (СНС, англ. CNN − Convolutional Neural Network) − основний інструмент для класифікації й розпізнавання об'єктів. Є безліч варіантів застосування CNN, такі як Deep Convolutional Neural Network (DCNN), Region-CNN (R-CNN), Fully Convolutional Neural Networks (FCNN), Mask R-CNN та ін. Згорткові нейронні мережі застосовуються досить широко в різних областях. Це, першою чергою, класифікація зображень і сигналів. Класифікації з допомогою CNN активно застосовуються в медицині − можна навчити нейронну мережу класифікації хвороб або симптомів, наприклад, для МРТ-діагностики. Одна з найбільш затребуваних областей застосування − розпізнавання облич. Це дає можливість виділяти обличчя на зображеннях, а потім з допомогою нейронних мереж CNN розпізнавати обличчя конкретної людини. CNN активно використовуються для вирішення завдань комп'ютерного зору, хоча можуть застосовуватися для роботи з аудіо- і будь-якими іншими даними, які можна подати у вигляді матриць.

Зазначимо, що тут наведено лише один з можливих варіантів класифікації нейронних мереж. Можливі й інші варіанти, але прийнятий підхід є цілком придатним для успішного початкового навчання основам технологій нейронних мереж.

1.2. Штучний нейрон

У штучних нейронних мережах за аналогією з біологічними нейронними мережами функція активації визначає вихідне значення нейрона залежно від результату зваженої суми входів і значення зсуву. Функція актива-

ції має властивості «вмикача/вимикача» («on/off»). Це означає, що якщо вхідні дані є більшими, ніж деяке значення, то вихід повинен змінювати стан, наприклад, з 0 на 1 або з -1 на +1. Це відповідає «увімкненню» нейрона.

Бажаними властивостями функції активації є:

- нелінійність;
- неперервна диференційовність;
- монотонність;
- гладкість функції і монотонність її похідної;
- апроксимованість тотожною функцією близько початку координат.

Існує велика кількість функцій активації нейронних мереж, які можуть використовуватися в різних завданнях, але найпопулярнішими є такі:

- функція одиничного стрибка;
- сигмоїдальна функція;
- гіперболічний тангенс;
- лінійний випрямляч (функція ReLU, rectified linear unit).

Функція одиничного стрибка. Рівняння для функції одиничного стрибка (ступінчастої функції, binary step function) має вигляд

$$\boldsymbol{f}(\boldsymbol{x}) = \begin{cases} \mathbf{0}, & \boldsymbol{x} < 0; \\ \mathbf{1}, & \boldsymbol{x} \geq \mathbf{0}. \end{cases}$$

Програмний код для візуалізації цієї функції має такий вигляд:

```
import matplotlib.pylab as plt
import numpy as np
x = np.arange(-8, 8, 0.1)
f = [] # ступінчаста функція
for i in x:
     if i<0:
          f.append(0)
     else:
          f.append(1)
plt.plot(x, f)
plt.xlabel('x')
plt.ylabel('f(x)')
plt.show()
```

Цю функцію графічно зображено на рис. 1.2.

Ступінчаста функція є пороговою функцією активації, тобто якщо $\boldsymbol{x}$ більше або менше деякого значення, то нейрон стає активованим. Така функція відмінно працює для бінарної класифікації. Але вона не працює, коли для класифікації потребується більша кількість нейронів і кількість можливих класів, більша від двох.

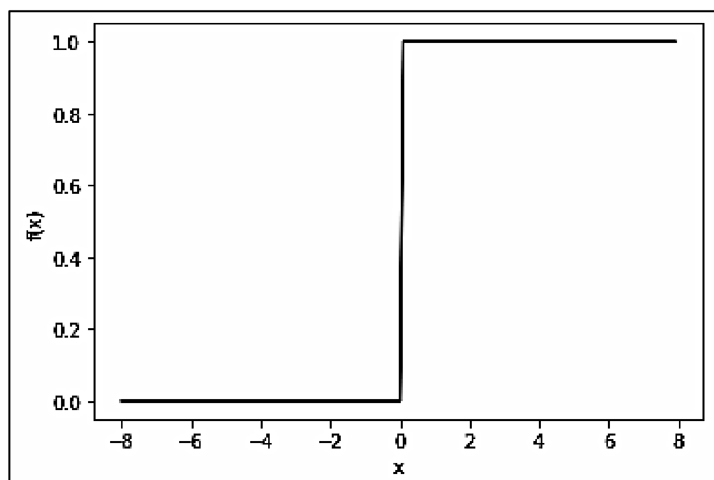


Рис. 1.2. Ступінчаста активаційна функція нейрона

Сигмоїдальна активаційна функція (логістична функція або гладка сходинка) має вигляд

$$\boldsymbol{f(x) = sigma(x) = \frac{1}{1 + \exp(-x)}}.$$

Програмний код для візуалізації цієї функції має такий вигляд:

```
import matplotlib.pylab as plt
import numpy as np
x = np.arange(-8, 8, 0.1)
f = 1/(1 + np.exp(-x)) # сигмоїдальна функція
plt.plot(x, f)
plt.xlabel('x')
plt.ylabel('f(x)')
plt.show()
```

Графічно цю функцію зображено на рис. 1.3.

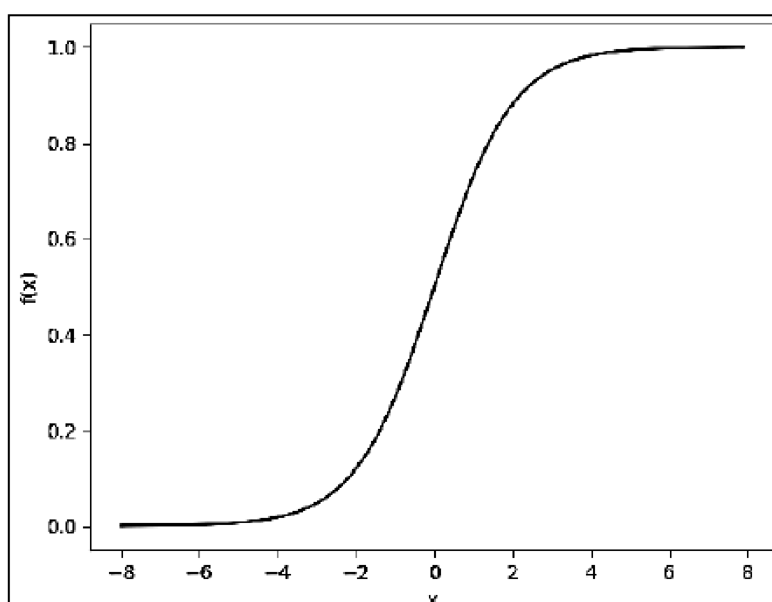


Рис. 1.3. Сигмоїдальна активаційна функція нейрона

На графіку видно, що активаційна функція зростає від 0 до 1 з кожним збільшенням значення x. Сигмоїдальна функція є гладкою, монотонно зростаючою нелінійною функцією і має похідну, що дуже важливо для алгоритму навчання.

Оскільки ця функція є нелінійною, то її можна використовувати в нейронних мережах з великою кількістю шарів, а також навчати ці мережі методом зворотного поширення помилки. Сигмоїда обмежена двома горизонтальними асимптотами $\boldsymbol{y = 1}$ і $\boldsymbol{y = 0}$, що дає нормалізацію вихідного значення кожного нейрона. Крім того, для сигмоїдальної функції характерним є гладкий градієнт, який запобігає «стрибкам» при підрахунку вихідного значення. Крім того, ця функція має ще одну перевагу: при значеннях $\boldsymbol{x} > 2$ і $\boldsymbol{x} < -2$ графік функції "притискається" до однієї з асимптот, що дає змогу робити чіткі прогнози щодо класів.

Незважаючи на безліч сильних сторін, сигмоїдальна функція має значний недолік. Похідна такої функції є дуже малою у всіх точках, крім порівняно невеликого проміжку. Це сильно ускладнює процес поліпшення ваг з допомогою градієнтного спуску. Більш того, ця проблема посилюється в разі, якщо модель містить багато шарів. Цю проблему називають проблемою зникаючого градієнта.

Що стосується використання сигмоїдальної функції, то її перевага полягає в нормалізації вихідного значення. Іноді це буває вкрай необхідно, наприклад, коли підсумкове значення шару має відображати ймовірність випадкової величини. Крім того, цю функцію зручно застосовувати під час класифікації завдяки властивості "притискання" до асимптот.

Функція гіперболічного тангенса має вигляд

$$\boldsymbol{f(x) = tanh(x) = \frac{2}{1 + \exp(-2x)} - 1}.$$

Програмний код для візуалізації цієї функції має такий вигляд

```
import matplotlib.pylab as plt
import numpy as np
x = np.arange(-8, 8, 0.1)
f = 2 / (1 + np.exp(-2*x))-1 # гіперболічний тангенс
plt.plot(x, f)
plt.xlabel('x')
plt.ylabel('f(x)')
plt.show()
```

Графічно цю функцію зображено на рис. 1.4.

Ця функція є скоригованою сигмоїдальною функцією

$$\boldsymbol{tanh(x) = 2sigma(2x) - 1.}$$

Вона має ті ж переваги й недоліки, але вже для діапазону значень (-1; 1).

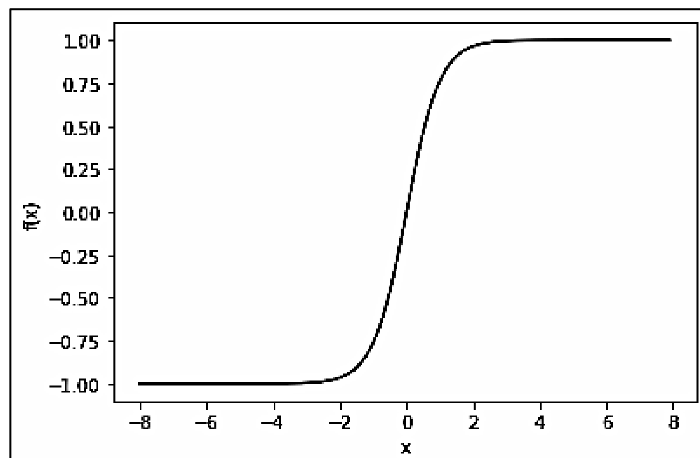


Рис. 1.4. Функція гіперболічного тангенса

Зазвичай гіперболічний тангенс переважає сигмоїдальну функцію у випадках, коли немає необхідності в нормалізації. Це відбувається через те, що область визначення цієї функції активації є центрованою відносно нуля, що знімає обмеження при підрахунку градієнта для переміщення в певному напрямку. Крім того, похідна гіперболічного тангенса є значно вищою поблизу нуля, даючи велику амплітуду градієнтного спуску, а отже, і більш швидку збіжність.

Функція лінійного випрямлення (***ReLU***) − це функція активації, що найчастіше використовується при глибокому навчанні. Вона повертає 0, якщо аргумент є від'ємним, у разі ж додатного аргумента функція повертає саме число, тобто її можна записати як

$$\boldsymbol{f(x) = max(0, x)}.$$

Програмний код для візуалізації цієї функції має такий вигляд:

```
import matplotlib.pylab as plt
import numpy as np
x = np.arange(-8, 8, 0.1)
f = []  # Функція ReLu
for i in x:
    f.append(max(0, i))
plt.plot(x, f)
plt.xlabel('x')
plt.ylabel('f(x)')
plt.show()
```

Цю функці графічно зображено на рис. 1.5.

Функцію ReLU можна використовувати в нейронних мережах з великою кількістю шарів. Вона має декілька переваг перед сигмоїдальною функцією і гіперболічним тангенсом:

- дуже швидко й просто розраховується похідна: для від'ємних значень − 0, для додатних – 1;
- розрідженість активації. У мережах з дуже великою кількістю нейронів використання сигмоїдальної функції або гіперболічного тангенса

як активаційної функції спричиняє активацію багатьох нейронів, що може позначитися на продуктивності навчання моделі. Якщо ж використовувати ***ReLU***, то кількість включених нейронів стане меншою внаслідок характеристик функції, і сама мережа стане легшою.

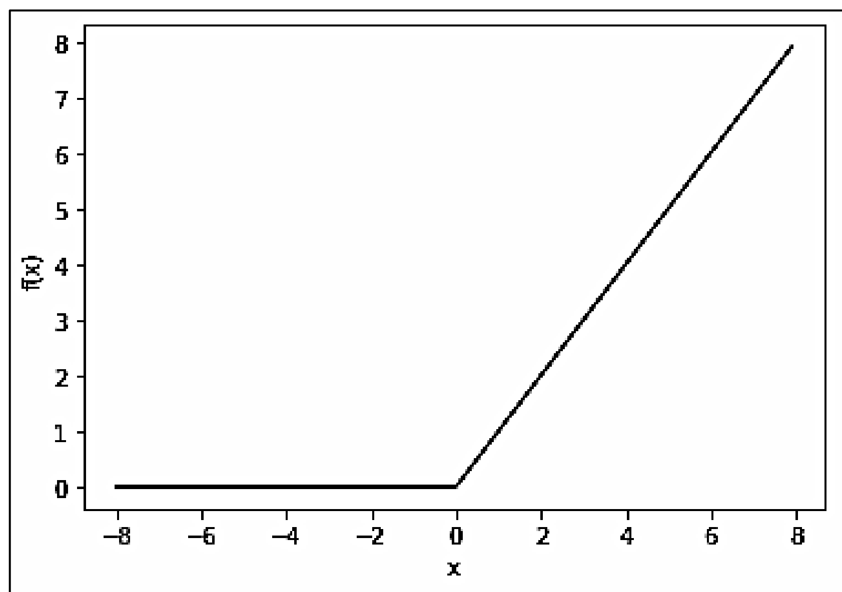


Рис. 1.5. Функція ReLU

Ця функція має один недолік, який називають проблемою зникаючого градієнта. Оскільки частина похідної функції дорівнює нулю, то й градієнт для неї буде нульовим, а це означає, що ваги не будуть змінюватися під час спуску по градієнту і нейронна мережа припинить учитися.

Функцію активації ***ReLU*** слід використовувати, якщо немає особливих вимог до вихідного значення нейрона на зразок необмеженої області визначення. Але якщо після навчання моделі результати вийшли не оптимальними, то слід перейти до інших функцій, які можуть дати кращий результат.

1.3. Перцептрони (вузли)

Біологічні нейрони ієрархічно з'єднуються в мережі, де вихід одних нейронів є входом для інших нейронів. Такі мережі можна подати у вигляді шарів, з'єднаних з вузлами. Кожен вузол приймає зважений вхід, формує активаційну функцію для суми входів і генерує вихід. Ці вузли також називають перцептронами (MLP, multilayer perceptron). Їх структуру показано на рис. 1.6. Вузол зображено у вигляді шару, у якому проводиться зважене підсумовування вхідних даних і введення результатів зваженого підсумовування в активаційну функцію. Процедура вагового підсумовування визначається співвідношенням

$$\boldsymbol{x_1 w_1 + x_2 w_2 + x_3 w_3 + b},$$

де $\boldsymbol{x_i}$ − значення вхідних даних $\boldsymbol{i}$-го каналу, а $\boldsymbol{w_i}$ − його ваговий коефіцієнт. За допомогою коефіцієнта $\boldsymbol{b}$ здійснюється зміщення елементів входу на 1.

Іншими словами, вихід активаційної функції визначається функцією передання $\boldsymbol{h_{w,b}(x)}$.

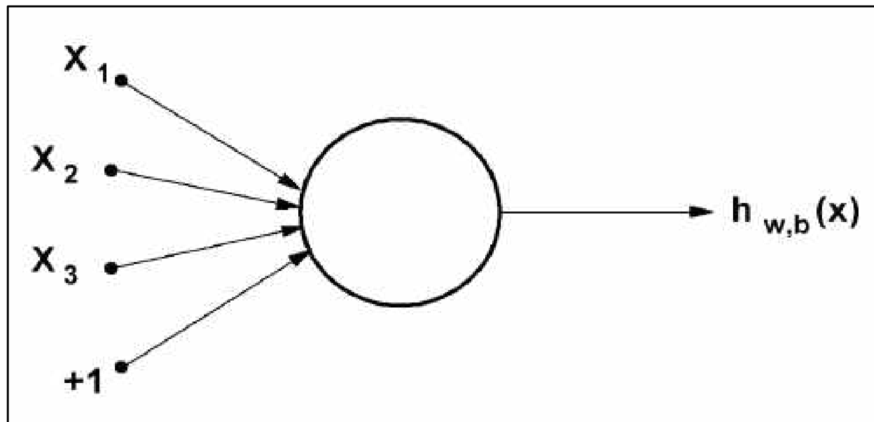


Рис. 1.6. Схематичне зображення перцептрона

Найпростішим прикладом роботи перцептрона є його використання в одноканальній схемі передання вхідних даних, як це показано на рис. 1.7.

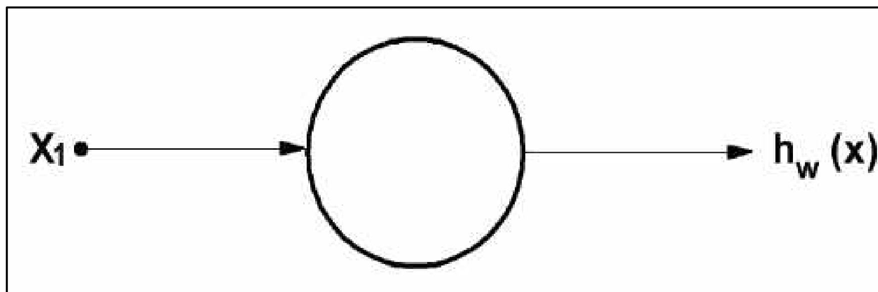


Рис. 1.7. Найпростіший одноканальний перцептрон

Вхід для активаційної функції такого вузла в цьому прикладі визначається як $\boldsymbol{x_1 w_1}$. Запишемо програмний код сигмоїдальної активаційної функції такого перцептрона для різних значень вагових коефіцієнтів $\boldsymbol{w_i}$ (від $\boldsymbol{w = 0.5}$ до $\boldsymbol{w = 2.0}$), як показано на рис. 1.8:

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

x = np.arange(-8, 8, 0.1)
w1 = 0.5
w2 = 1.0
w3 = 2.0
l1 = 'w = 0.5'
l2 = 'w = 1.0'
l3 = 'w = 2.0'
for w, l in [(w1, l1), (w2, l2), (w3, l3)]:
    f = 1 / (1 + np.exp(-x * w))
plt.plot(x, f, label = l)
plt.xlabel('x')
```

```
plt.ylabel('h_w(x)')
plt.legend(loc = 2)
plt.show()
```

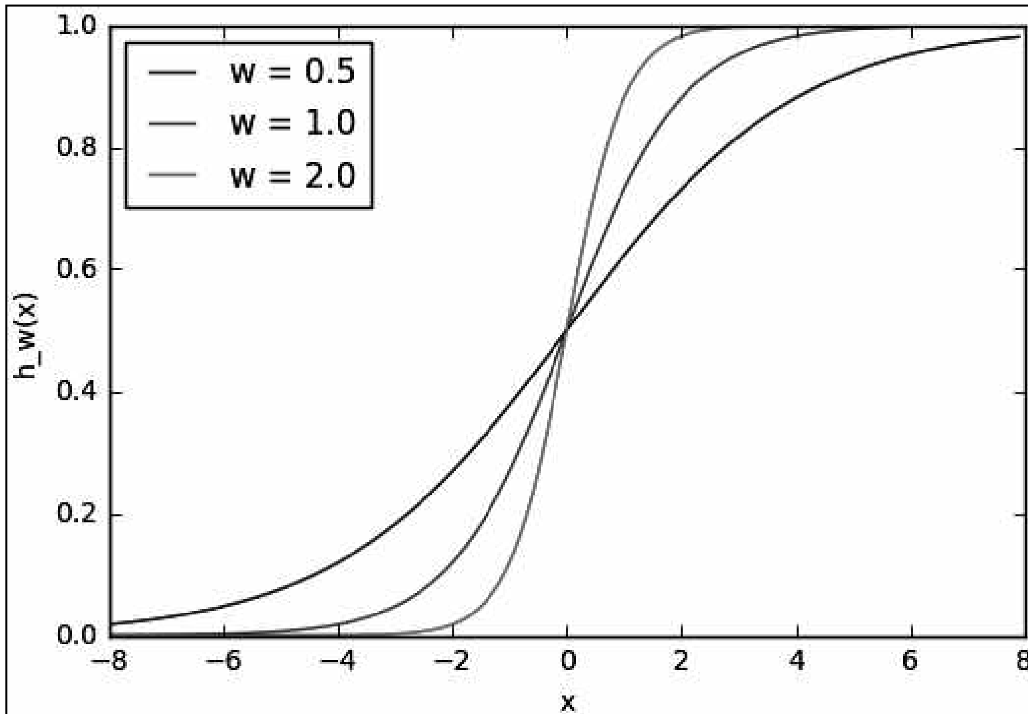


Рис. 1.8. Змінення активаційної функції при різних значеннях коефіцієнта $\boldsymbol{w}$

Очевидно, що змінення вагового коефіцієнта $\boldsymbol{w}$ впливає на рівень нахилу графіка активаційної функції. Така модель є корисною при виборі різної щільності взаємозв'язків між входами і виходами.

Якщо необхідно, щоб вихід змінювався тільки при $\boldsymbol{x} > 1$, то потрібно забезпечити зміщення. Такі нейрони отримали назву «нейрони зміщення». Вони потрібні для того, щоб мати можливість отримувати вихідний результат шляхом зміщення графіка функції активації вправо або вліво. Розглянемо елемент такої мережі зі зміщенням на вході (рис. 1.9).

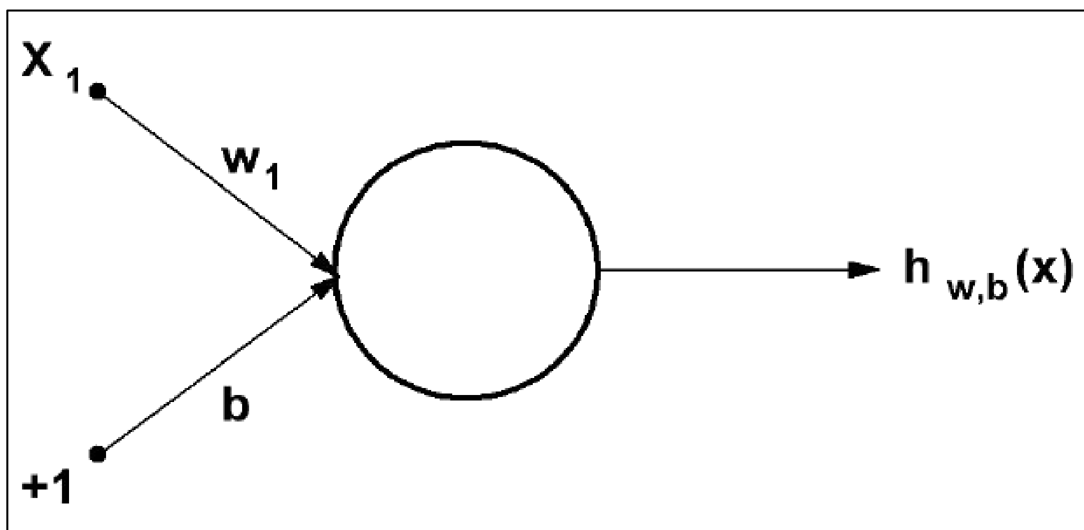


Рис. 1.9. Мережа зі зміщенням на вході

Код для активаційної функції в мережі зі зміщенням на вході має такий вигляд:

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

x = np.arange(-8, 8, 0.1)
w = 5.0
b1 = -8.0
b2 = 0.0
b3 = 8.0
l1 = 'b = -8.0'
l2 = 'b = 0.0'
l3 = 'b = 8.0'
for b, l in [(b1, l1), (b2, l2), (b3, l3)]:
  f = 1 / (1 + np.exp(-(x * w + b)))
plt.plot(x, f, label = l)
plt.xlabel('x')
plt.ylabel('h_wb(x)')
plt.legend(loc = 2)
plt.show()
```

Графік цієї функції показано на рис. 1.10. Очевидно, що, змінюючи "вагу" зміщення b, можна змінювати рівень запуску вузла. Яким чином в архітектуру нейронної мережі вводити вузли зміщення, розглянемо далі більш докладно.

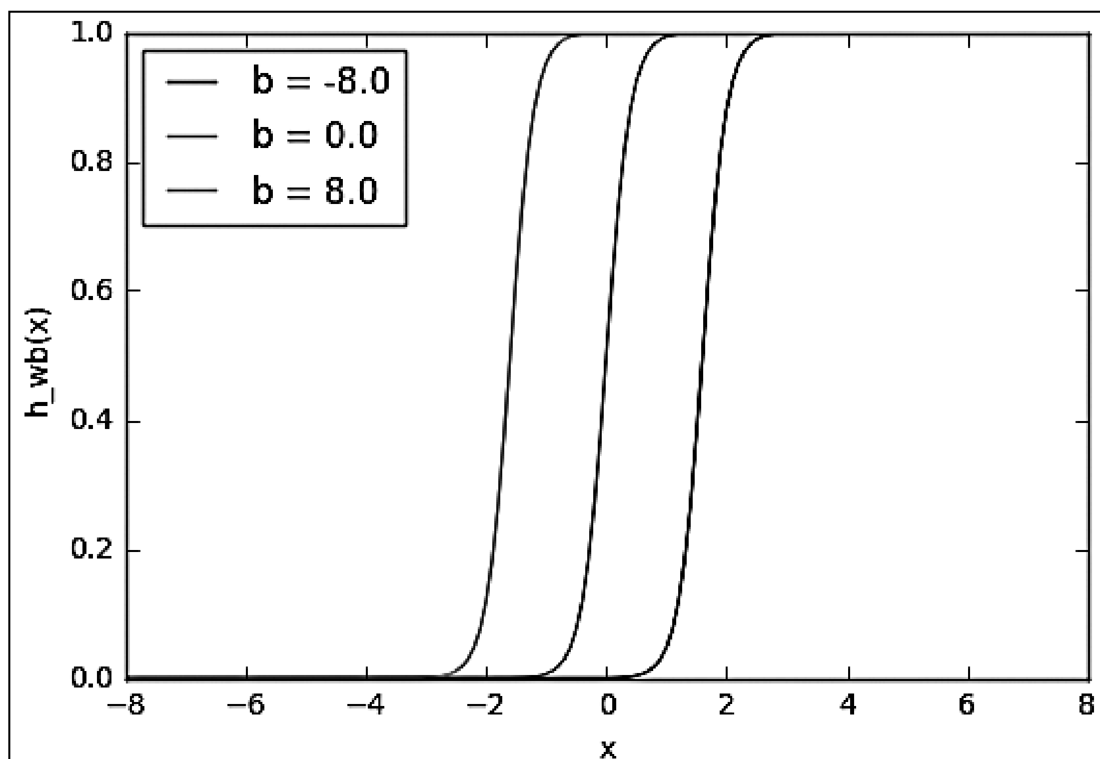


Рис. 1.10. Змінення рівнів запуску перцептрона з допомогою коефіцієнта зміщення b

1.4. Узагальнені кількісні характеристики нейронних мереж

Узагальнену архітектуру нейронних мереж різного призначення показано на рис. 1.11. Кожна така мережа містить деяку кількість вхідних вузлів, приховані і вихідні шари.

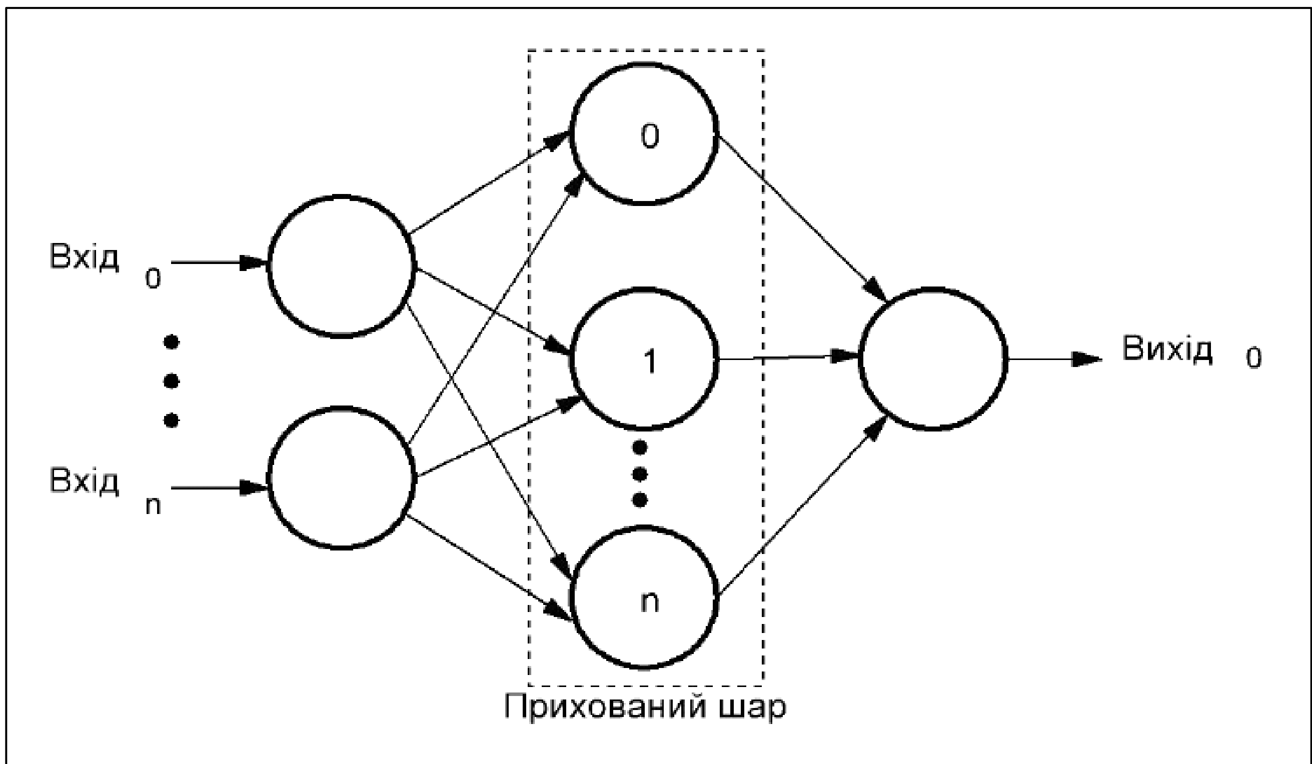


Рис. 1.11. Узагальнена архітектура нейронної мережі

Розглянемо більш докладно основні характеристики окремих елементів мережі:

1. Кількість вхідних вузлів у різних мережах є різною і має відповідати розмірності вхідних даних.

2. Зазвичай у нейронній мережі наявними є кілька прихованих шарів. Їх кількість не визначено. Однак зазначимо, що для забезпечення надійної класифікації вхідних даних цілком достатньо навіть одного прихованого шару.

3. Кількість вузлів в окремому прихованому шарі є змінною величиною. Оптимальна кількість прихованих вузлів визначається експериментально.

4. Кількість вихідних вузлів може бути більшою від одиниці, але це ускладнює програму формування нейронної мережі.

5. Функція активації для прихованих і вихідних вузлів зазвичай вибирається у вигляді стандартної «логістичної» сигмоїдальної функції

$$\boldsymbol{f(x) = \frac{1}{1 + e^{-x}}}.$$

Її графік показано на рис. 1.12.

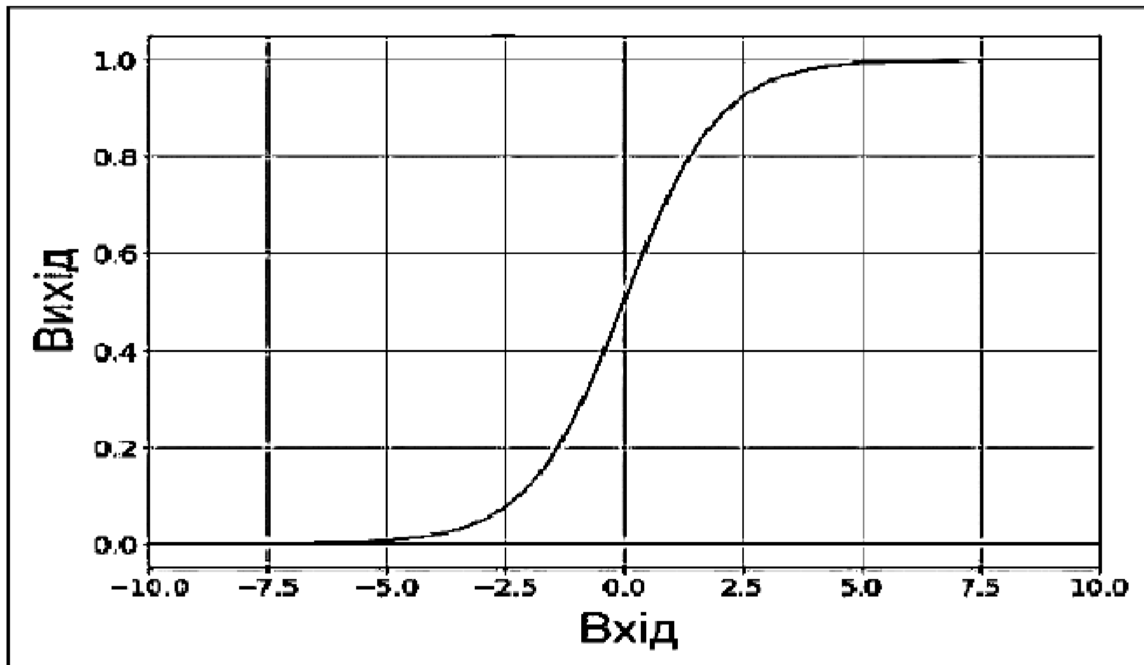


Рис. 1.12. Графік логістичної функції

1.5. Використання вузлів зміщення

Вузли зміщення можуть бути включені у вхідний або прихований шар, або в обидва шари одночасно. Використання вузлів зміщення − важливий фактор при формуванні коду нейронної мережі. Числові значення, зв'язані з вузлами зміщення, є константами, які вибираються розробником. Це дає змогу легко змінювати кількість вхідних або прихованих вузлів та їх розмірність. Така можливість забезпечує більшу гнучкість при налаштуванні нейронних мереж. Типову архітектуру нейронної мережі з вбудованими вузлами зміщення зображено на рис. 1.13.

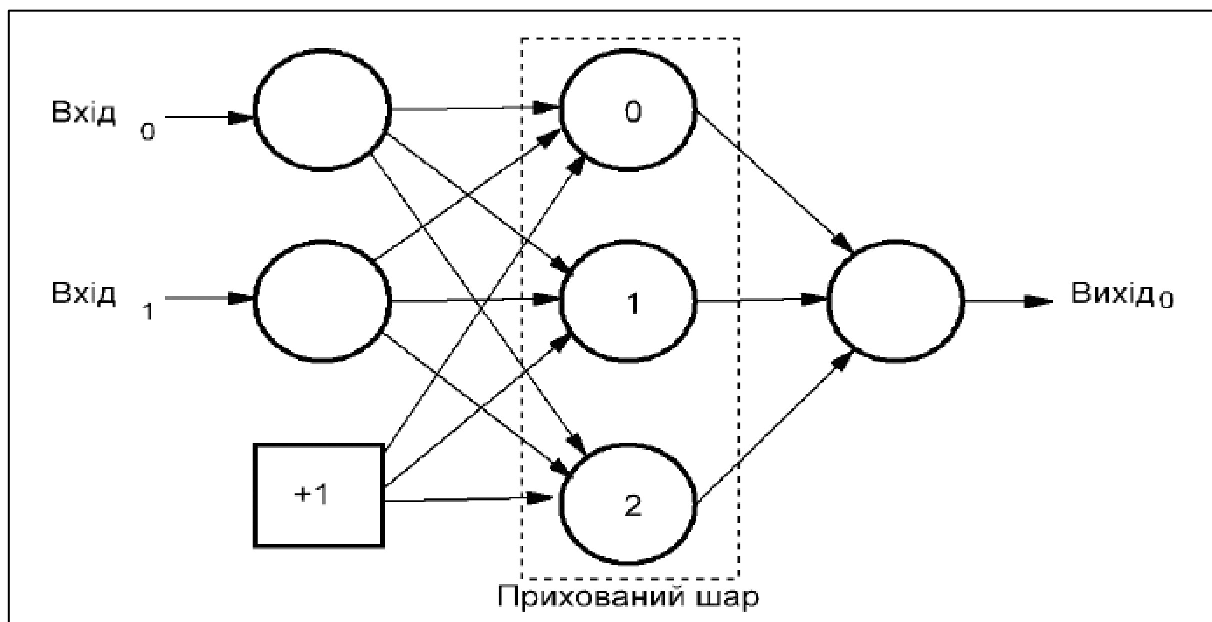


Рис. 1.13. Нейронна мережа з вузлами зміщення

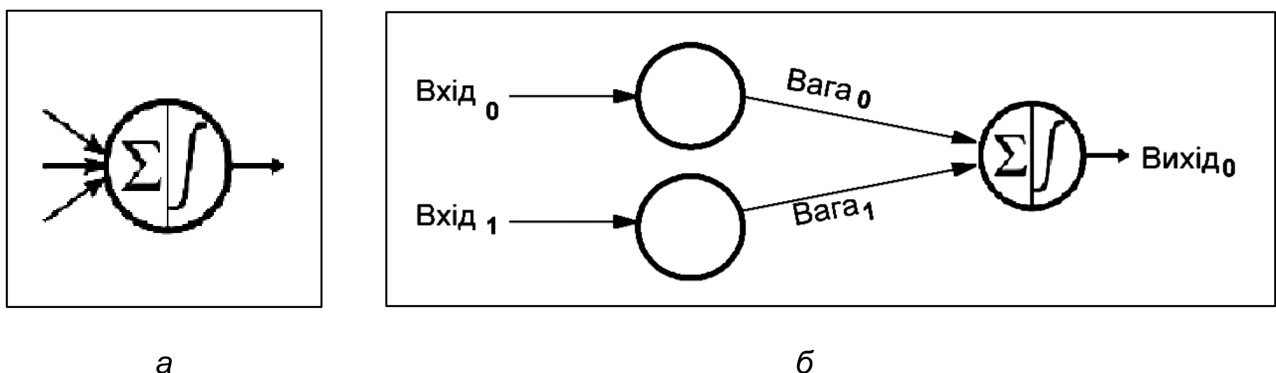
Одним з головних питань при побудові нейронних мереж є створення обчислювальної процедури, що дає змогу точно налаштувати вагові коефіцієнти багатошарового перцептрона для точної класифікації вхідних вибірок. Також важливою є й концепція «зворотного поширення». Це не менш важливий аспект проектування. Методи вирішення цих питань далі розглянемо більш докладно.

1.6. Приклад нейронної мережі «перцептрон» для класифікації даних

Наведемо простий приклад створення нейронної мережі «перцептрон» для класифікації даних. Розглянемо послідовно всі етапи реалізації проекту в узагальненому і дещо спрощеному вигляді. Це необхідно для ясного розуміння змісту і зв'язку всіх етапів синтезу нейронної мережі й методів тестування якості її роботи.

Одношаровий перцептрон. У попередніх розділах показано, що нейронна мережа складається з взаємозв'язаних вузлів, розташованих у шарах. Вузли у вхідному шарі розподіляють дані, а вузли в інших шарах виконують обчислення (підсумовування з подальшим використанням функції активації). З'єднання між цими вузлами є зваженими, це означає, що кожне з'єднання перемножує передані дані на скалярне значення. На рис. 1.14, *а* показано умовне позначення одношарового перцептрона (суматор і функція активації), а на рис 1.14, *б* графічно зображено алгоритм оброблення даних у нейронній мережі.

Цю конфігурацію називають одношаровим перцептроном. Він містить два шари (вхідний і вихідний), але тільки один шар містить обчислювальні вузли.



а *б*

Рис. 1.14. Умовне позначення вузла «перцептрон» (*а)*; оброблення даних у нейронній мережі (*б*)

Класифікація з допомогою перцептрона. Далі розглянемо роботу перцептрона, використовуючи нейронну мережу, показану на рис. 1.15.

У цьому прикладі розмірність вхідних даних дорівнює трьом, отже, з допомогою такого перцептрона можна вирішувати завдання в тривимірному просторі. Наприклад, якщо точка в тривимірному просторі розташована нижче осі X, то вона відповідає недійсним даним. Якщо точка знаходиться

на осі x або вище, то вона відповідає дійсним даним, які необхідно зберегти для подальшого аналізу. У цьому випадку нейронна мережа повинна класифікувати дані з вихідним значенням 1, що вказує на дійсний набір даних, і значенням 0, що вказує на неправильний набір даних.

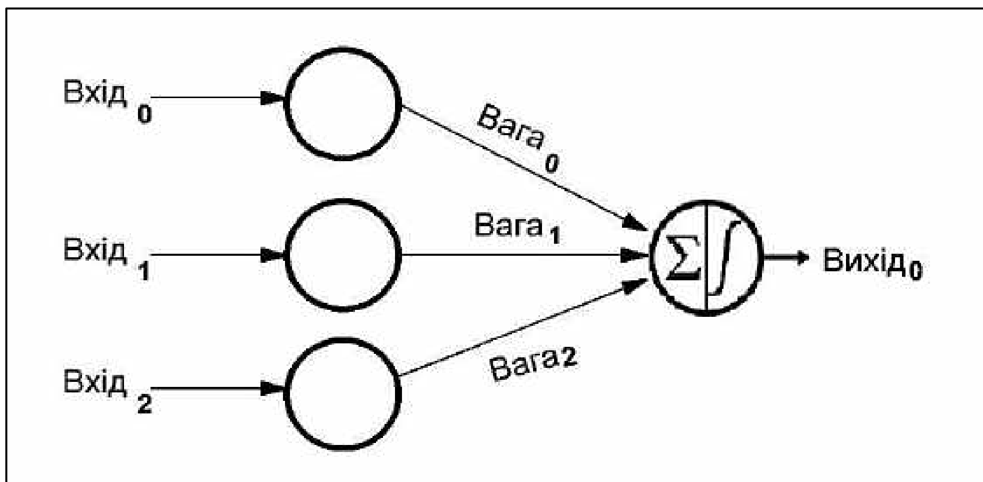


Рис. 1.15. Приклад нейтронної мережі

Для формалізації завдання необхідно перенести тривимірні координати у вхідний вектор. У цьому прикладі $\text{вхід}_{\mathbf{0}}$ – це компонента $\boldsymbol{x}$, $\text{вхід}_{\mathbf{1}}$ – компонента $\boldsymbol{y}$, а $\text{вхід}_{\mathbf{2}}$ – компонента $\boldsymbol{z}$. Потім потрібно визначити вагу. Цей приклад є настільки простим, що немає необхідності навчати мережу. Можна просто призначити будь-які необхідні ваги (рис. 1.16).

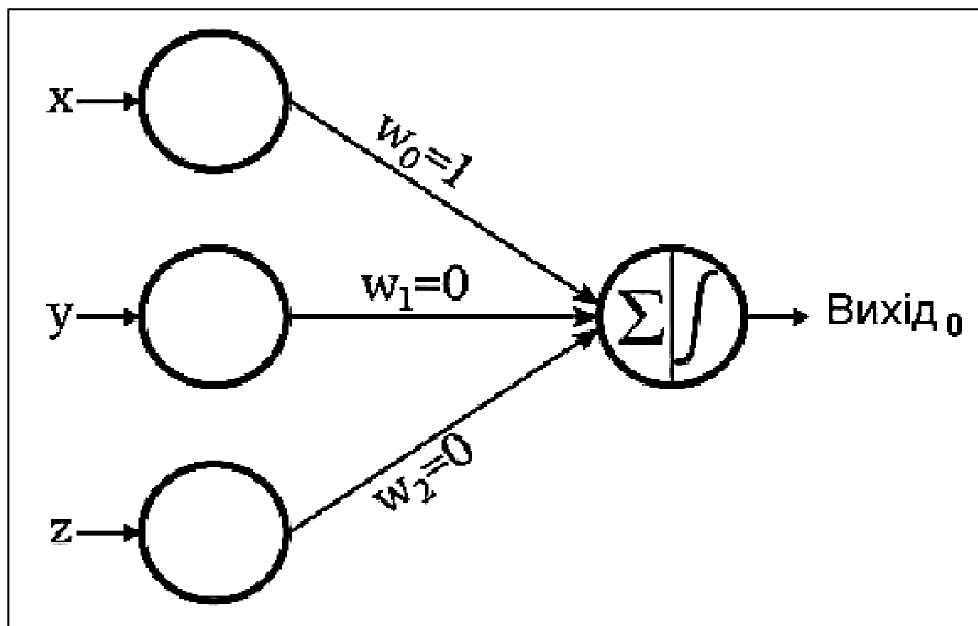


Рис. 1.16. Призначення ваг

Будемо вважати, що функція активації вихідного вузла являє собою одиничну сходинку, виражену таким чином:

$$\boldsymbol{f}(\boldsymbol{x}) = \begin{cases} \mathbf{0}, & \boldsymbol{x} < 0\,; \\ \mathbf{1}, & \boldsymbol{x} \geq \mathbf{1}\,. \end{cases}$$

Перцептрон (див. рис. 1.16) працює таким чином: оскільки $\boldsymbol{w_1} = \mathbf{0}$ і $\boldsymbol{w_2} = \mathbf{0}$, компоненти $\boldsymbol{y}$ і $\boldsymbol{z}$ не дають вкладу в результат підсумовування, що генерується вихідним вузлом. Єдиним вхідним значенням, яке впливає на результат підсумовування, є компонента $\boldsymbol{x}$. Вона доставляється на вихідний вузол без змін, тому що $\boldsymbol{w_0} = \mathbf{1}$. Якщо точка в тривимірному просторі знаходиться нижче осі $\boldsymbol{x}$, то результат підсумовування вихідного вузла буде від’ємним. Тоді функція активації перетворює це від'ємне значення на $\text{вихід}_{\mathbf{0}} = \mathbf{0}$. Якщо ж точка в тривимірному просторі знаходиться на осі $\boldsymbol{x}$ або вище, то сума дорівнює нулю або є більшою від нуля. При цьому функція активації перетворює це значення на $\text{вихід}_{\mathbf{0}} = \mathbf{1}$.

Вирішення завдань з допомогою перцептрона. У попередньому прикладі перцептрон описано як інструмент для вирішення завдань. Однак його було використано неповною мірою − завдання вирішено при призначених вагах.

У наведеному прикладі проблему класифікації даних на дійсні/недійсні вирішено дуже швидко, оскільки зв'язок між вхідними даними і шуканими вихідними значеннями є дуже простим. Однак у багатьох реальних ситуаціях надзвичайно важко сформулювати математичний зв'язок між вхідними даними і вихідними значеннями. Можна отримувати вхідні дані й записувати або виробляти відповідні вихідні значення, але математичного маршруту від входу до виходу немає.

Корисним прикладом є розпізнавання рукописного тексту. Нехай є зображення рукописних символів, і необхідно класифікувати ці зображення як «a», «b», «c» і т. д. Необхідно перетворити рукописний текст на звичайний комп'ютерний текст. Для цього треба генерувати вхідні зображення, потім призначати правильні категорії для кожного зображення. Таким чином, збір вхідних даних і відповідних вихідних значень не становить труднощів. Однак дуже складно сформулювати математичний алгоритм, який би правильно перетворював вхідні зображення у вихідну категорію.

Тому розпізнавання рукописного тексту і безліч інших завдань оброблення сигналів неможливо вирішити без допомоги складних інструментів. Нейронні мережі і є таким інструментом. Вони дають змогу вирішувати ці складні завдання завдяки швидкому й багаторазовому обчисленню з використанням величезної кількості числових даних.

Навчання нейронної мережі. Процес, протягом якого нейронна мережа може створювати математичний маршрут від входу до виходу, називають навчанням. Дані для навчання мережі складаються з вхідних значень і відповідних вихідних значень. До цих значень застосовується фіксована математична процедура. Метою цієї процедури є поступове змінення ваг мережі таким чином, щоб мережа могла розраховувати правильні вихідні значення навіть з вхідними даними, які вона ніколи раніше не бачила. По суті, це пошук шаблонів у навчальних даних і генерація ваг, які дають

змогу отримати корисний результат шляхом застосування цих шаблонів до нових даних.

На рис. 1.17 показано розглянутий вище класифікатор даних на дійсні/недійсні, але ваги є різними. Ці ваги генеруються при тренуванні перцептрона з допомогою 1000 точок даних. Такий процес навчання дає змогу автоматично апроксимувати необхідні математичні зв'язки перцептроном.

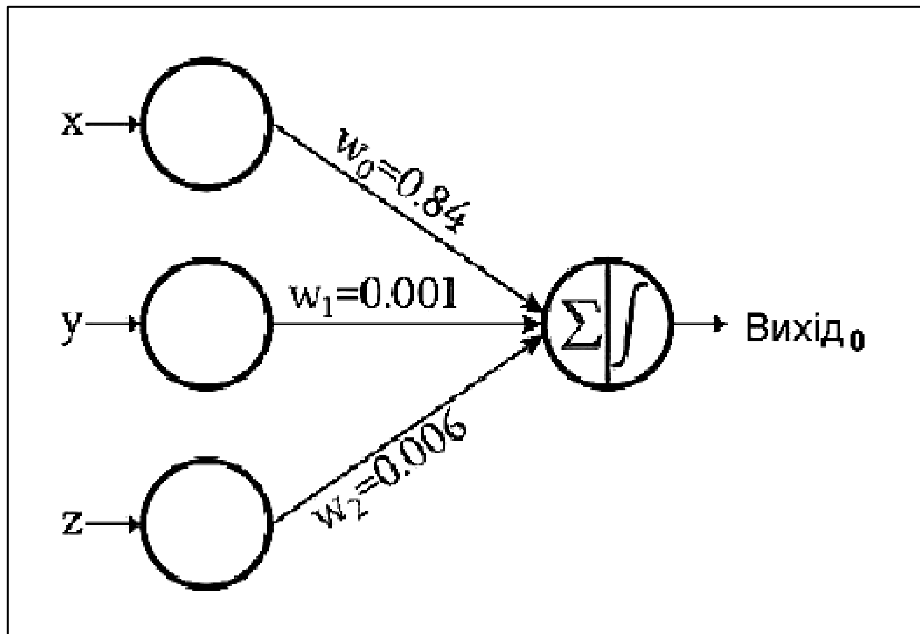


Рис. 1.17. Результати навчання нейронної мережі «перцептрон»

Тут показано результати навчання цього перцептрона, але немає даних про те, як їх було отримано. Далі опишемо коротку програму на Python, яка реалізує одношарову нейронну мережу «перцептрон», і методику навчання.

1.7. Навчання простої нейронної мережі «перцептрон»

Програмний код на Python дає змогу автоматично генерувати ваги для простої нейронної мережі:

```
import pandas
import numpy as np

input_dim = 3

learning_rate = 0.01

Weights = np.random.rand(input_dim)

#Weights[0] = 0.5
#Weights[1] = 0.5
#Weights[2] = 0.5

Training_Data = pandas.read_excel("3D_data.xlsx")
```

```
Expected_Output = Training_Data.output

Training_Data = Training_Data.drop(['output'], axis=1)

Training_Data = np.asarray(Training_Data)

training_count = len(Training_Data[:,0])

for epoch in range(0,5):

    for datum in range(0, training_count):

        Output_Sum = np.sum(np.multiply(Training_Data[datum,:], 
Weights))

        if Output_Sum < 0:
            Output_Value = 0
        else:
            Output_Value = 1

        error = Expected_Output[datum] - Output_Value

        for n in range(0, input_dim):
            Weights[n] = Weights[n] + learn-
ing_rate*error*Training_Data[datum,n]

print("w_0 = %.3f" %(Weights[0]))

print("w_1 = %.3f" %(Weights[1]))

print("w_2 = %.3f" %(Weights[2]))
```

Розглянемо інструкції цього коду більш докладно. Уточнимо налаштування мережі й організацію даних.

Розмірність вхідних даних визначається в такому рядку коду:

```
input_dim = 3
```

Розмірність вихідних даних у принципі можна змінювати. Однак для визначення положення точки в тривимірному просторі необхідно й достатньо наявності трьох вхідних вузлів. Тому програма не підтримує великої кількості вихідних вузлів.

Швидкість навчання нейронної мережі (`learning_rate`) описано в такому рядку:

```
learning_rate = 0.01
```

Питання збільшення швидкості навчання мереж розглянемо далі більш детально.

Створення вагових коефіцієнтів мережі. Цю процедуру забезпечує такий фрагмент програмного коду:

```
Weights = np.random.rand(input_dim)

#Weights[0] = 0.5
```

```
#Weights[1] = 0.5
#Weights[2] = 0.5
```

Ваги зазвичай не започатковано випадковими значеннями. Функція `numpy random.rand ()` генерує масив завдовжки `input_dim`, заповнений випадковими значеннями, розподіленими в інтервалі (0, 1). Однак початкові значення ваги впливають на кінцеві значення, отримані внаслідок процедури навчання. Якщо ж необхідно оцінити вплив інших змінних (розмір навчального набору або швидкість навчання), то можна усунути фактор, що заважає. Для цього всі ваги встановлюються у вигляді констант замість випадково згенерованих чисел.

Дані навчання викликаються з допомогою командного рядка

```
Training_Data = pandas.read_excel("3D_data.xlsx")
```

Бібліотека `pandas` використовується для імпорту навчальних даних з електронної таблиці `Excel`. Більш докладно формування тренувальних даних розглянемо в одному з наступних розділів.

Очікуваний результат навчання (`Expected_Output`) визначається такими інструкціями:

```
Expected_Output = Training_Data.output

Training_Data = Training_Data.drop(['output'], axis=1)
```

Набір навчальних даних містить вхідні дані й відповідні вихідні значення. Перша інструкція виокремлює вихідні значення й зберігає їх в окремому масиві, а така інструкція видаляє вихідні значення з вихідного набору навчальних даних:

```
Training_Data = np.asarray(Training_Data)

training_count = len(Training_Data[:,0])
```

Розрахунок вихідних значень проводиться багаторазовим проходженням та усередненням усіх даних, що використовуються для навчання (епохи навчальних даних):

```
for epoch in range(0,5):
```

Тривалість одного тренування визначається кількістю доступних навчальних даних. Проте, можна продовжити оптимізацію ваг, навчаючи нейронну мережу кілька разів з використанням того самого набіру даних, − переваги навчання не зникають просто тому, що нейронна мережа вже бачила ці навчальні дані. Кожен повний прохід через увесь навчальний набір називають епохою.

Тренувальний розрахунок (`training_count`):

```
for datum in range(0, training_count):
```

Процедура, яка міститься в цьому циклі, виконується один раз для кожного рядка в навчальному наборі, де рядок належить до групи значень вхідних даних і відповідає певному значенню (у цьому випадку вхідна група складається з трьох чисел, що являють собою компоненти $\boldsymbol{x}$, $\boldsymbol{y}$ і $\boldsymbol{z}$ точки у тривимірному просторі).

Робота вихідного вузла регламентується такою інструкцією:

```
Output_Sum =
np.sum(np.multiply(Training_Data[datum,:],Weights))
```

Вихідний вузол має підсумувати значення, отримані трьома вхідними вузлами. У програмному коді спочатку виконується поелементне множення масиву `Training_Data` на масив `Weights`, а потім обчислюється сума елементів у масиві, отриманому цим множенням:

```
if Output_Sum < 0:
    Output_Value = 0
else:
    Output_Value = 1
```

Оператор `if-else` застосовує одиничну ступінчасту функцію активації: якщо сума менше нуля, то значення, згенероване вихідним вузлом, дорівнюватиме 0; якщо сума дорівнює або більше нуля, то вихідне значення дорівнюватиме 1.

Оновлення ваг. Після завершення першого розрахунку вихідних значень виникають значення ваг, які не дають змоги провести правильну класифікацію, тому що вони були згенеровані випадковим чином. Перетворити нейронну мережу на ефективну систему класифікації можна шляхом багаторазового змінення ваг таким чином, щоб вони поступово відображали математичне співвідношення між вхідними даними й шуканими вихідними значеннями. Змінити вагу можна за таким правилом навчання для кожного рядка в навчальному наборі:

$$\boldsymbol{w}_{\text{новий}} = \boldsymbol{w} + \left(\boldsymbol{\alpha} \times \left(\text{вихід}_{\text{очікуваний}} - \text{вихід}_{\text{розрахований}}\right) \times \text{вхід}\right).$$

Символом $\boldsymbol{\alpha}$ позначено швидкість навчання. Для того щоб обчислити нове значення ваги, множимо відповідне вхідне значення на швидкість навчання і на різницю між очікуваним вихідним значенням (яке забезпечується навчальним набором) і розрахованим вихідним значенням. Потім результат цього множення додаємо до поточного значення ваги. Якщо визначати дельту ($\boldsymbol{\delta}$) як $\left(\text{вихід}_{\text{очікуваний}} - \text{вихід}_{\text{розрахований}}\right)$, то можна переписати формулу як

$$\boldsymbol{w}_{\text{новий}} = \boldsymbol{w} + (\boldsymbol{\alpha} \times \boldsymbol{\delta} \times \text{вхід}).$$

Помилка навчання нейронної мережі на Python визначається за інструкцією:

```
error = Expected_Output[datum] - Output_Value

for n in range(0,input_dim):
    Weights[n] = Weights[n] + learn-
ing_rate*error*Training_Data[datum,n]
```

Цей програмний код можна використовувати для навчання одношарового перцептрона з одним вихідним вузлом. Більш докладно теорію й практику навчання нейронних мереж різного призначення і конфігурації викладемо далі.

1.8. Підготовка навчальних даних для перцептрона

Метою навчання є надання даних, які дають змогу нейронній мережі збігатися на надійних математичних зв'язках між входом і виходом. У попередньому прикладі математичний зв'язок був простим: якщо компонента x точки у тривимірному просторі менше нуля, то вихідний результат дорівнює нулю (це означає, що ця точка даних є недійсною і не потребує подальшого аналізу); якщо ж компонента x дорівнює або більше нуля, то вихідний сигнал дорівнює одиниці (що вказує на дійсну точку даних).

Використання таблиць Excel. У випадках, коли відомо математичні зв'язки, можна створювати навчальні дані в програмах для роботи з електронними таблицями. У цьому випадку було використано Excel.

Компоненти, x, y і z були створені функцією `RANDBETWEEN()`. На рис. 1.18, *а* всі випадкові значення являють собою цілі числа від `-10` до `+10`. В іншій частині навчального набору (рис. 1.18, *б*) використано формулу `RANDBETWEEN(-10,10)/10`, щоб отримати нецілі компоненти x, y і z в інтервалі `[-1, +1]`.

D2 fx =IF(A2<0, 0, 1)

	A	B	C	D	E
1	x	y	z	output	
2	-3	8	-10	0	
3	-1	-6	-1	0	
4	-9	-2	-5	0	
5	-10	6	-5	0	
6	0	-2	7	1	
7	6	-5	-1	1	
8	9	-2	3	1	
9	3	10	-7	1	
10	5	2	4	1	
11	8	-7	-6	1	
12	-2	2	-9	0	
13	-5	7	9	0	
14	3	-6	-2	1	
15	-8	-9	8	0	
16	-2	9	-9	0	

а

Рис. 1.18. Цілочислові навчальні дані в Excel (*а*) і навчальні дані в інтервалі [-1, +1] (*б*)

A509 | =RANDBETWEEN(-10, 10)/10

	A	B	C	D	E	F
509	-0.7	-1	0.3	0		
510	0.3	0.1	0.3	1		
511	0.6	-0.1	-1	1		
512	0.9	-0.9	-0.3	1		
513	0.8	-0.8	-0.1	1		
514	-0.5	0.5	-0.1	0		
515	0.3	1	0.1	1		
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]		

б

Рис. 1.18. Закінчення

Генерація навчальних даних в електронній таблиці − хороший і доступний спосіб експериментувати з нейронними мережами, а це дуже важлива частина ознайомлення з розробленням і реалізацією нейронних мереж. Однак у реальних додатках навчальні дані таким чином не створюються. Сенс використання нейронної мережі полягає в тому, щоб створити алгоритм, який ви ще не знаєте і не можете легко визначити.

Забезпечення достатньості навчальних даних. Нейронна мережа не навчається через розуміння й критичне мислення. Це суто математична система, і вона дуже поступово наближається до складних зв'язків вхід-вихід. Таким чином, великі обсяги даних допомагають нейронній мережі продовжувати уточнювати свої ваги і тим самим досягати найбільшої кінцевої ефективності.

Диверсифікація даних. Важливо застосовувати для навчання різнома-нітні дані. Ми хочемо, щоб перцептрон апроксимував дійсний узагальнений взаємозв'язок між входом і виходом, а не неправильний або занадто спрощений взаємозв'язок, що існує між вхідними й вихідними значеннями недостатньо повного й різноманітного навчального набору. Необхідно зро-зуміти, що відбувається з нейронною мережею, коли ви навчаєте її з допомогою даних, які не відображають різноманітності реального завдання оброблення.

Відповідність навчальних даних робочим даним. При навчанні потрібно подавати дані, які нейронна мережа буде намагатися класифікувати. Зрештою, мета навчання полягає в тому, щоб дати можливість мережі ефективно обробляти деяку інформацію з реального життя. Наприклад, у простій системі класифікації, яку розглядали раніше, можна припустити, що реальні вхідні дані будуть змінюватися в межах інтервалу (-5, +5). У цьому випадку необхідно підготувати навчальний набір, що містить безліч цілих і нецілих чисел у діапазоні від -5 до +5.

Перетасування. У попередньому прикладі було показано, що навчальний набір може оброблятися кілька разів, при цьому кожний повний прохід через набір називається однією епохою. Рекомендується після кожної епохи перетасовувати вибірки в навчальному наборі, щоб порядок подання вибірок негативно не впливав на нейронну мережу. Тут можливі

різні варіанти: перший − реалізувати цю функцію в програмному забезпеченні нейронних мереж, а другий − продублювати навчальний набір в електронній таблиці, а потім випадковим чином змінити порядок вибірок у дубльованих наборах.

Перенавчання. Нейронні мережі можуть реагувати на надзвичайно складні зв'язки між входом і виходом, і, отже, мережа може задіяти деталі зв'язку між входом і виходом, які є специфічними для навчального набору і не мають відношення до реального завдання класифікації. Це називають перенавчанням (або перетренуванням).

Перенавчання нейронної мережі. Діаграма, зображена на рис. 1.19, ефективно ілюструє концепцію перенавчання. Це зв'язок «вхід-вихід», який є менш узагальненим і, отже, менш придатним. Червоні й сині точки являють собою навчальні вибірки, які класифікує нейронна мережа. Чорна лінія являє собою хорошу стратегію класифікації: вона прямує за загальним шаблоном, який відокремлює червоний колір від синього, і, отже, вона, імовірно, приведе до найнижчої помилки в реальних даних. Зелена лінія − це перенавчена стратегія класифікації. Ця стратегія дуже добре відображає навчальні дані та ідеально класифікує навчальні вибірки, які вона створила для реальних даних.

Інший спосіб візуалізації перетренування зображено на рис. 1.20. Тут необхідно розглядати нейронну мережу як систему, яка генерує математичну функцію, що є апроксимацією зв'язку між входом і виходом, виявленою навчальними вибірками. Недонавченість тут є неприпустимою − функція не дає змоги точно апроксимувати тренд у навчальних даних, але й перенавчання також є шкідливим − тренд спотворюється через надмірну деталізацію в навчальному наборі.

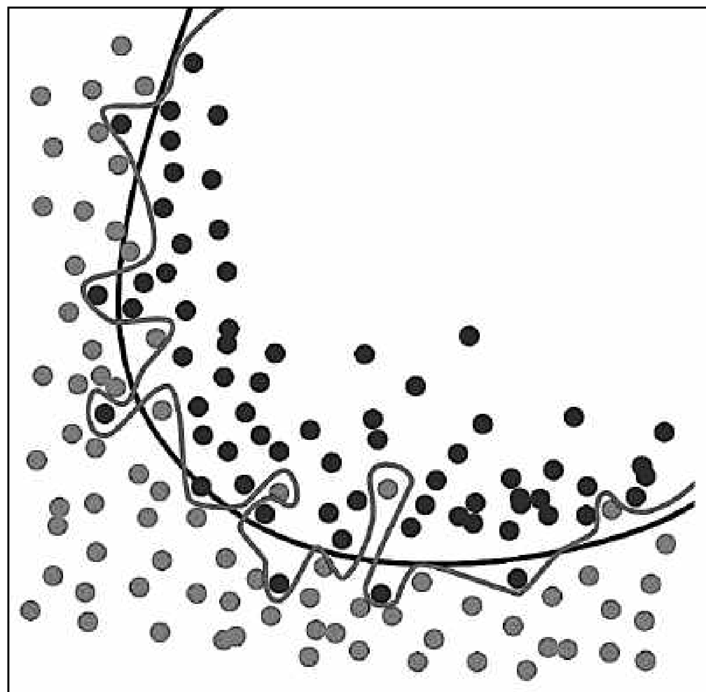


Рис. 1.19. Ілюстрація концепції перенавчання

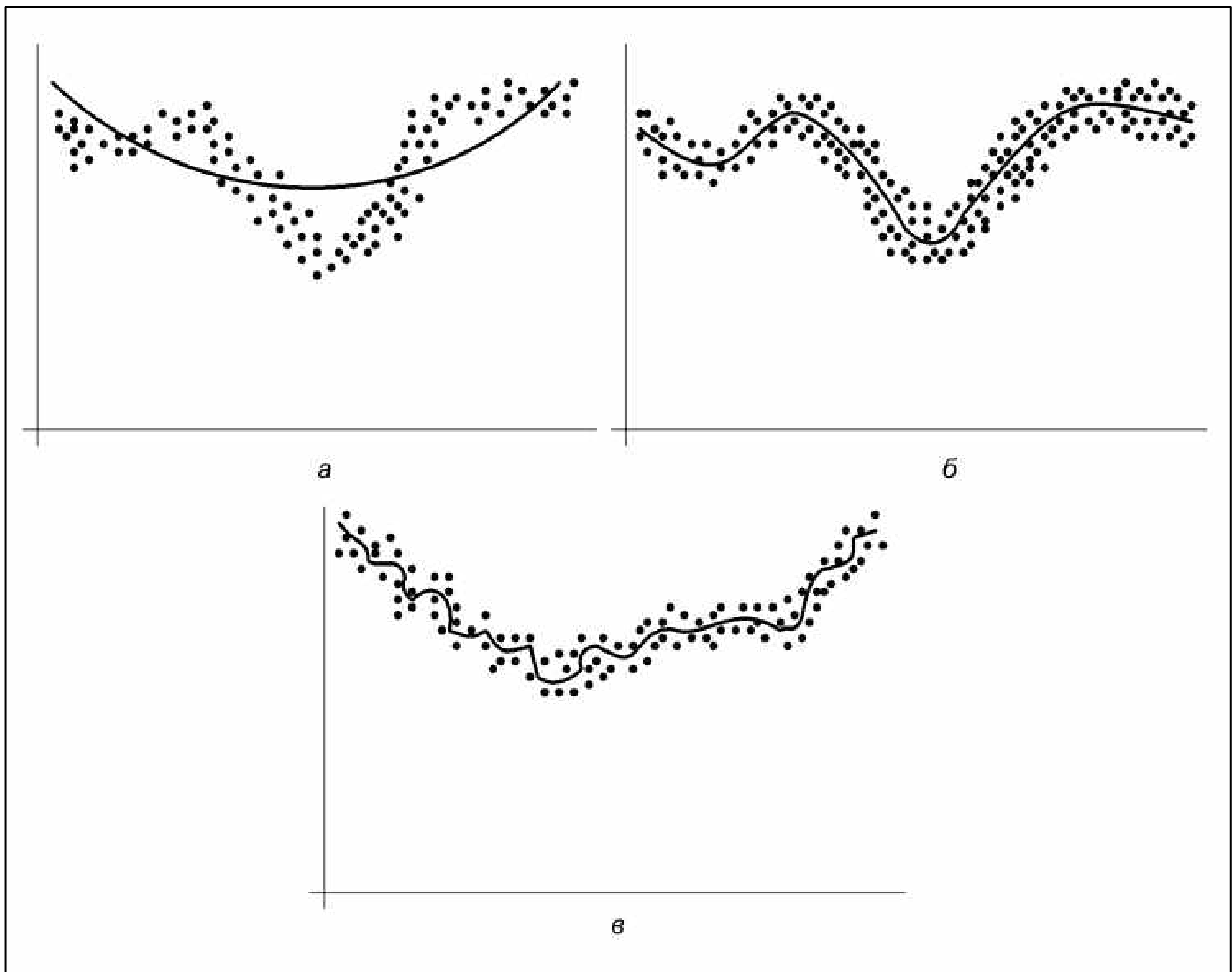


Рис. 1.20. Недостатньо навчена (*а*), добре навчена (*б*) і перенавчена (*в*) нейромережі

1.9. Вступ до теорії навчання нейронних мереж

Розглянемо далі теорію й практику навчання нейронних мереж і концепцію функції помилки.

Навчання − протиріччя між теорією і практикою. На перший погляд, навчання нейронної мережі здається досить простим. При роботі з простою мережею (такою, як одношаровий перцептрон на рис. 1.21) математика, необхідна для навчання, є досить простою, а мережа може бути реалізована у відносно короткій програмі, написаній на поширених мовах, таких як C або Python. Процес навчання не потребує надмірно великого обчислювального часу.

Крім того, загальна концепція нагадує дію стабілізації, пов'язаної з негативним зворотним зв'язком: на основі вхідних даних отримують вихідні дані, порівнюють отримані вихідні дані з очікуваним вихідним сигналом і передають цю інформацію назад у нейронну мережу таким чином, щоб ваги могли поступово збігатися на значеннях, які є прийнятними для поставленого завдання.

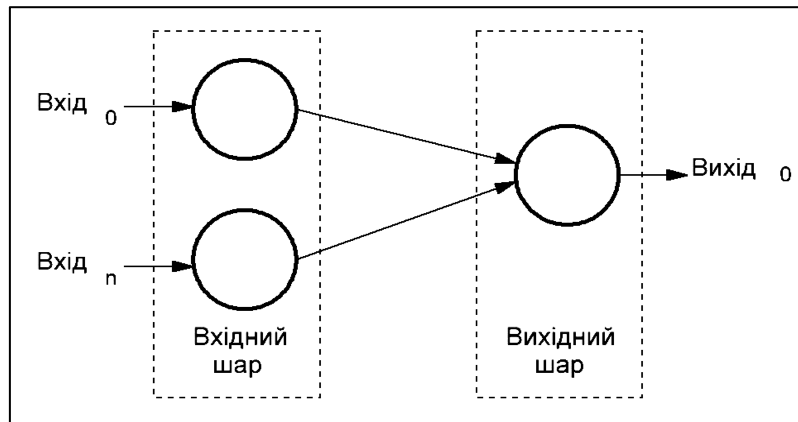


Рис. 1.21. Одношарова нейронна мережа «перцептрон»

Однак існує досить багато теорій, пов'язаних з навчанням штучних нейронних мереж − достатньо виконати пошук «neural network training» у Google Scholar, і можна отримати велику вибірку досліджень, які були проведені в цій області. Однак необхідно зрозуміти головну концепцію − методи мінімізації помилок.

Перцептрон як універсальний апроксиматор. Нейронна мережа може виконувати класифікацію, тому що вона автоматично знаходить і реалізує (за допомогою навчання) математичні зв'язки між вхідними даними й вихідними значеннями. У математичній термінології використовується слово «функція», щоб ідентифікувати зв'язок «вхід-вихід», і часто виражають функції в символах: $f(x)$. Таким чином, x являє собою вхідні дані, а функція $f(x)$ працює з вхідними даними й формує вихідні значення.

Як уже зазначалося раніше, перцептрон, що містить додаткові шари вузлів (більше, ніж просто вхідний і вихідний шари), називають багатошаровим перцептроном, або MLP (multilayer perceptron). З цих вузлів складається прихований шар, тому що їх не «видно» безпосередньо з боку входу або виходу. Таку архітектуру показано на рис. 1.22.

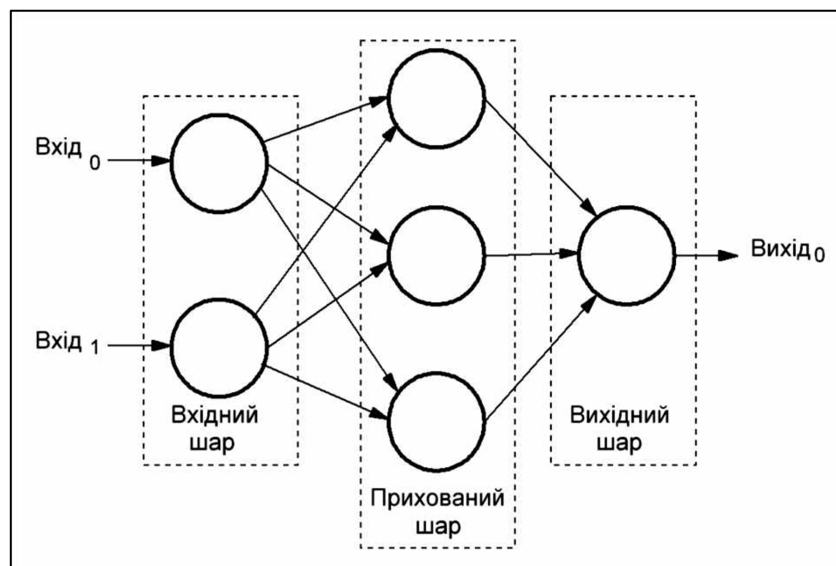


Рис. 1.22. Багатошарова нейронна мережа «перцептрон»

Багатошаровий перцептрон уважається універсальним апроксиматором. Існує кілька нюансів, обумовлених цією концепцією, але загальна ідея полягає в тому, що математика, яка виконується нейронною мережею, забезпечує гнучкість у підсумковій функції − зв'язку між входом і виходом.

Коли мережа вперше починає навчання, вагам присвоюються випадкові значення, і, отже, функції $\boldsymbol{f}(\boldsymbol{x})$ нейронної мережі (назвемо це $\boldsymbol{f}_{\text{нм}}(\boldsymbol{x})$) зовсім не відповідає реальний зв'язок $\boldsymbol{f}_{\text{реал}}(\boldsymbol{x})$ між входом і виходом. Під час навчання нейронна мережа генерує корисні коригування значень ваг, переглядаючи інформацію про помилки, що повертається з вихідних даних. Поступово $\boldsymbol{f}_{\text{нм}}(\boldsymbol{x})$ стає все ближчою до $\boldsymbol{f}_{\text{реал}}(\boldsymbol{x})$. У цьому випадку символ $\boldsymbol{x}$ не являє собою одну змінну, наприклад, $\boldsymbol{x}$ може бути вектором з 50 елементів.

Функція помилки. Розглянемо нейронну мережу з двома зваженими зв’язками, які ведуть до одного вихідного вузла. При навчанні такої нейронної мережі відомо правильне вихідне значення. Отже, можна обчислити помилку, що створюється цим вихідним вузлом. Крім того, можна візуалізувати помилку, використовуючи тривимірний графік: два входи відповідають осі $\boldsymbol{x}$ і осі $\boldsymbol{y}$, а помилка відповідає осі $\boldsymbol{z}$.

Основна ідея полягає в тому, що кожна комбінація вхідних ваг і вихідної помилки подібна точці в тривимірному просторі. При зміні ваг компоненти $\boldsymbol{x}$ та $\boldsymbol{y}$ точки змінюються, компонента $\boldsymbol{z}$ також змінюється. Зміна ваг призводить до зміни помилки. Коли вагові коефіцієнти поліпшуються, помилка зменшується і прямує до нуля. Цю закономірність показано у вигляді тривимірної функції помилки (рис. 1.23).

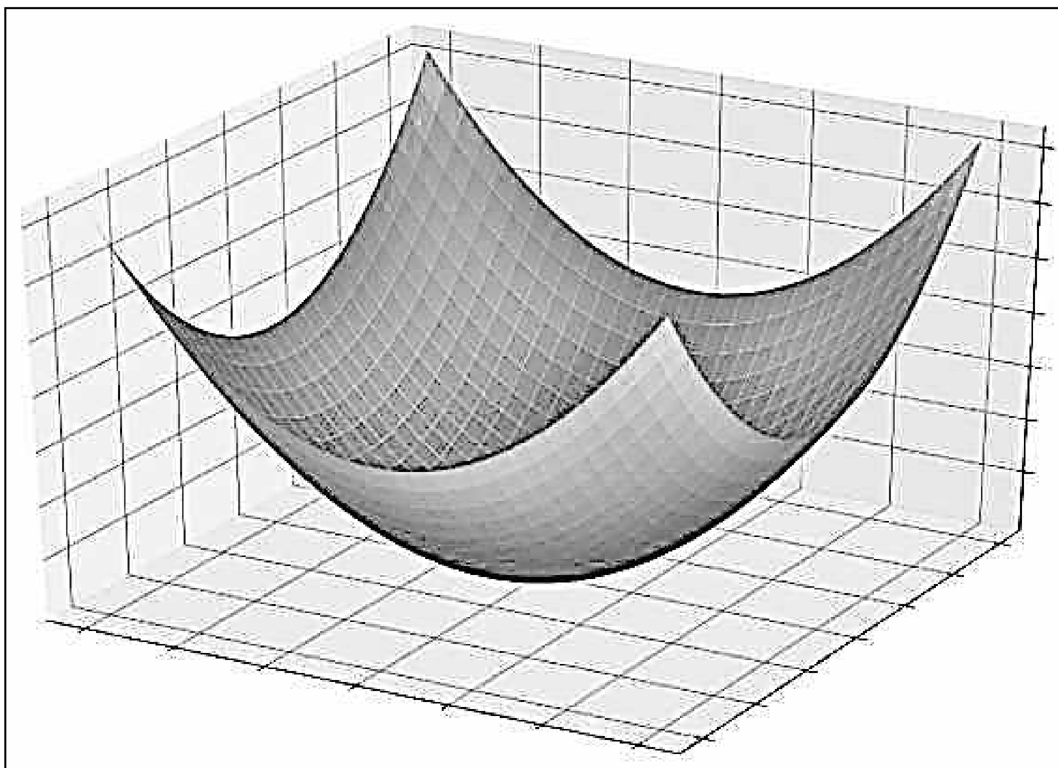


Рис. 1.23. Функція помилки

Процедура навчання − це, по суті, прагнення до мінімуму цієї функції, де $z = 0$, якщо вихідні дані, вироблені вузлом, дорівнюють очікуваним вихідним значенням. При поступовому коригуванні під час навчання точка, яка визначається двома вагами й помилкою, послідовно наближається до мінімального значення.

Спуск до мінімальної помилки відбувається при зміні ваг, і зміни ваг також роблять $f_{\text{нм}}(x)$ все більш схожою на $f_{\text{реал}}(x)$. Навчання змушує нейронну мережу змінювати свої ваги так, щоб це приводило до мінімізації функції помилки і наближення до адекватного математичного зв'язку між входом і виходом.

1.10. Швидкість навчання в нейронних мережах

Обговоримо докладніше концепцію швидкості навчання і з'ясуємо, як вона може вплинути на характеристики навчання нейронної мережі.

Швидкість навчання нейронної мережі − це здатність апроксимувати зв'язок вхідних і вихідних даних шляхом зміни ваг до досягнення мінімальної помилки їх неузгодженості за мінімальну кількість кроків (ітерацій).

Вплив швидкості навчання. Швидкість навчання впливає на розмір кроків, які ведуть до мінімізації помилки. Перейдемо до двовимірного подання залежності вихідної помилки від вхідної ваги. Візуалізація такої залежності (двовимірної функції помилки) показано на рис. 1.24, *а*.

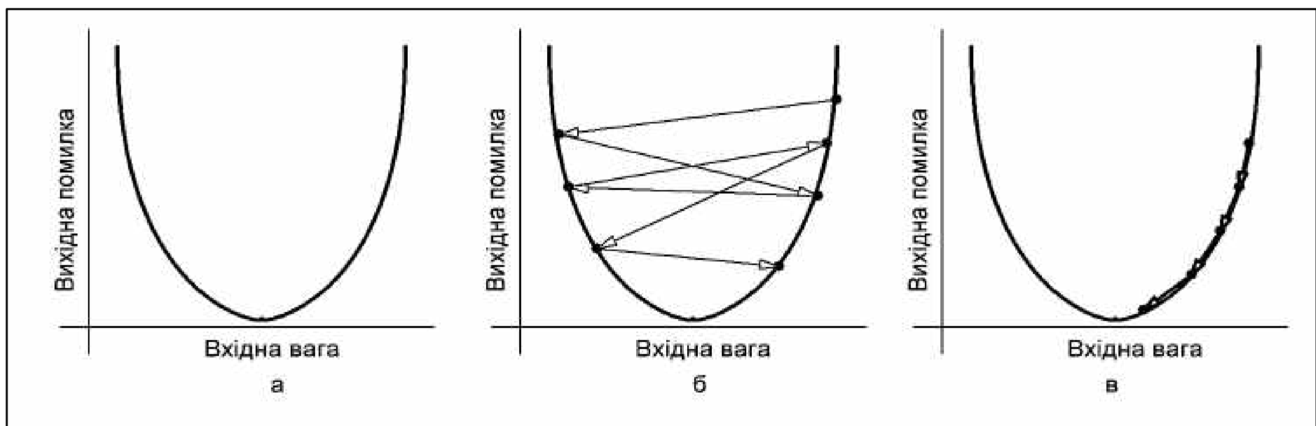


Рис. 1.24. Двовимірне подання функції помилки (*а*); великі стрибки вагових коефіцієнтів (*б*); прийнятна швидкість навчання (*в*)

У попередньому прикладі для поновлення ваг було використано таке кількісне правило навчання:

$$w_{\text{новий}} = w + (\alpha \times \delta \times \text{вхід}),$$

де α − швидкість навчання, δ − різниця між очікуваним вихідним сигналом і розрахованим вихідним сигналом (помилка).

Кожен раз, коли застосовується це правило навчання, вага переходить у нову точку на кривій помилки. Якщо помілка δ є великою, то стрибки

також можуть бути досить великими, і нейронна мережа може навчатися неефективно, тому що ваги не збігаються поступово до мінімальної помилки. Замість цього вони хаотично стрибають, як показано на рис. 1.24, *б*.

Оскільки, перш ніж модифікація застосовується до ваги, $\boldsymbol{\delta}$ множиться на швидкість навчання, можна зменшити розмір стрибків, вибравши $\boldsymbol{\alpha} < \mathbf{1}$. Мета − використовувати швидкість навчання для забезпечення помірно швидкої й послідовної збіжності. Корисний тип навчання може мати такий вигляд, як це показано на рис. 1.24, *в*.

Вибір швидкості навчання. На жаль, не існує універсального правила вибору швидкості навчання і немає навіть точного способу визначення оптимальної швидкості навчання для заданого додатка. Навчання − це складний і мінливий процес, а швидкість навчання вибирається експериментально.

Якщо нейронна мережа може швидко обробляти навчальні дані, можна вибрати кілька різних швидкостей навчання й порівняти отримані ваги (якщо відомо, якими вони мають бути) або ввести «свіжі» дані й оцінити взаємозв'язок між швидкістю навчання і точністю класифікації.

Більш складний підхід, який був би більш практичним для нейронних мереж, що потребують тривалого часу навчання, полягає в аналізі змін у помилках під час навчання мережі. Помилка має змінюватися в напрямку мінімуму, а зміни в помилці мають бути досить малими, щоб уникнути «стрибкоподібної» поведінки, показаної на рис. 24, *б*, але не настільки малими, щоб нейронна мережа навчалася вкрай повільно.

Графік швидкості навчання. Швидкість навчання не обов'язково має бути постійною протягом усієї процедури навчання. Швидкість навчання змінюється щоразу, коли з допомогою правила навчання оновлюються ваги.

Коли нейронна мережа починає навчання, помилка, імовірно, буде великою. Більш висока швидкість навчання допомагає нейронній мережі робити великі кроки в напрямку мінімальної помилки. Однак, коли мережа наближається до нижньої частини кривої помилки, ці довгі кроки можуть перешкоджати збіжності. Коли швидкість навчання зменшується, довгі кроки зменшуються, і, урешті-решт, нейронна мережа починає працювати при мінімальному рівні помилки.

1.11. Машинне навчання з багатошаровим перцептроном

Далі з'ясуємо, чому для високопродуктивних нейронних мереж необхідним є додатковий «прихований» шар обчислювальних вузлів. Раніше детально роз'яснювалася робота одношарового перцептрона, що складається з вхідного і вихідного шарів. Термін «одношаровий» використовувався тому, що ця конфігурація містить тільки один шар обчислювальних активних вузлів, тобто вузлів, які змінюють дані шляхом підсумовування, а

потім застосовують функцію активації. Вузли вхідного шару просто розподіляють дані.

Одношаровий перцептрон простий, і процедура його навчання також проста. На жаль, він не має функціональності, що потребується для складних реальних додатків.

Перцептрон як нейромережний логічний елемент. Повернімося до конфігурації нейронної мережі, зображеної на рис. 1.14, *б*. Перцептрон у ній містить два шари (вхідний і вихідний), але тільки в одному шарі (вихідному) є обчислювальний вузол.

Загальний вигляд цього перцептрона (рис. 1.25) нагадує логічний елемент. При навчанні такої нейронної мережі з допомогою вибірок елементів вхідного вектора, що складаються з нулів та одиниць, вихідне значення дорівнює одиниці, тільки якщо обидва вхідних значення дорівнюють одиниці. У цьому випадку нейронна мережа класифікує вхідний вектор способом, аналогічним роботі логічного елемента І (AND).

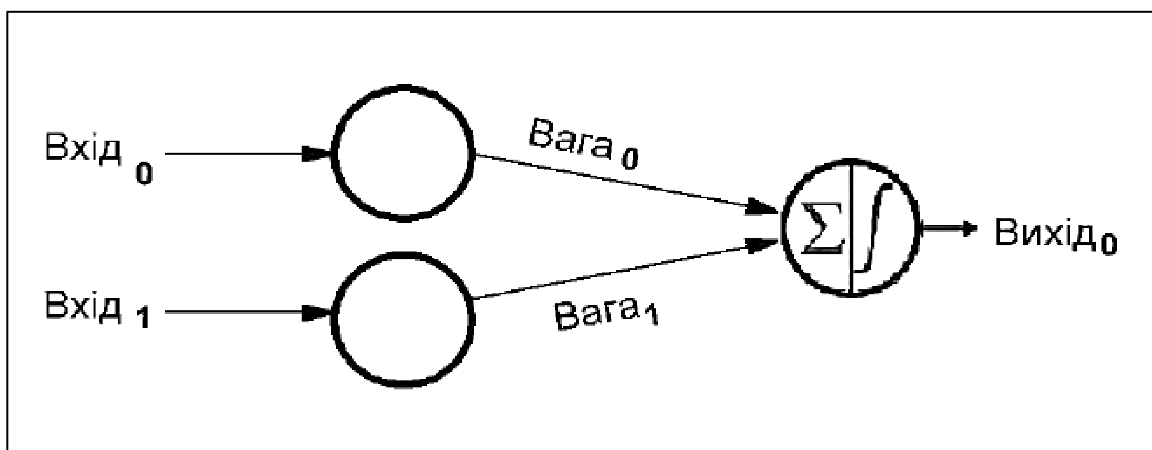


Рис. 1.25. Одношаровий перцептрон

Розмірність вхідних даних цієї нейронної мережі дорівнює 2, тому легко побудувати вхідні вибірки як двовимірний графік (рис. 1.26, *а*). Припустимо, що вхід_0 відповідає горизонтальній осі, а вхід_1 відповідає вертикальній осі. Чотири можливі комбінації вхідних значень будуть розташовані так, як показано на рис. 1.26, *а*.

Оскільки відтворюється операція І, нейронна мережа має змінити свої ваги таким чином, щоб вихідне значення дорівнювало одиниці для вхідного вектора [1,1] і нулю для інших трьох вхідних векторів. Ґрунтуючись на цій інформації, можна поділити простір вхідних даних на секції, що відповідають необхідним класифікаціям вихідних значень.

Лінійно роздільні дані. При реалізації операції І вхідні вектори, побудовані на графіку (рис. 1.26, *б, в*), можна класифікувати, нарисувавши пряму лінію. Усе з одного боку лінії отримує вихідне значення, що дорівнює одиниці, а все на іншому боці – нульове вихідне значення. Таким чином, у разі операції І дані, подані в нейронній мережі, є лінійно нероздільними. Це також стосується й операції АБО.

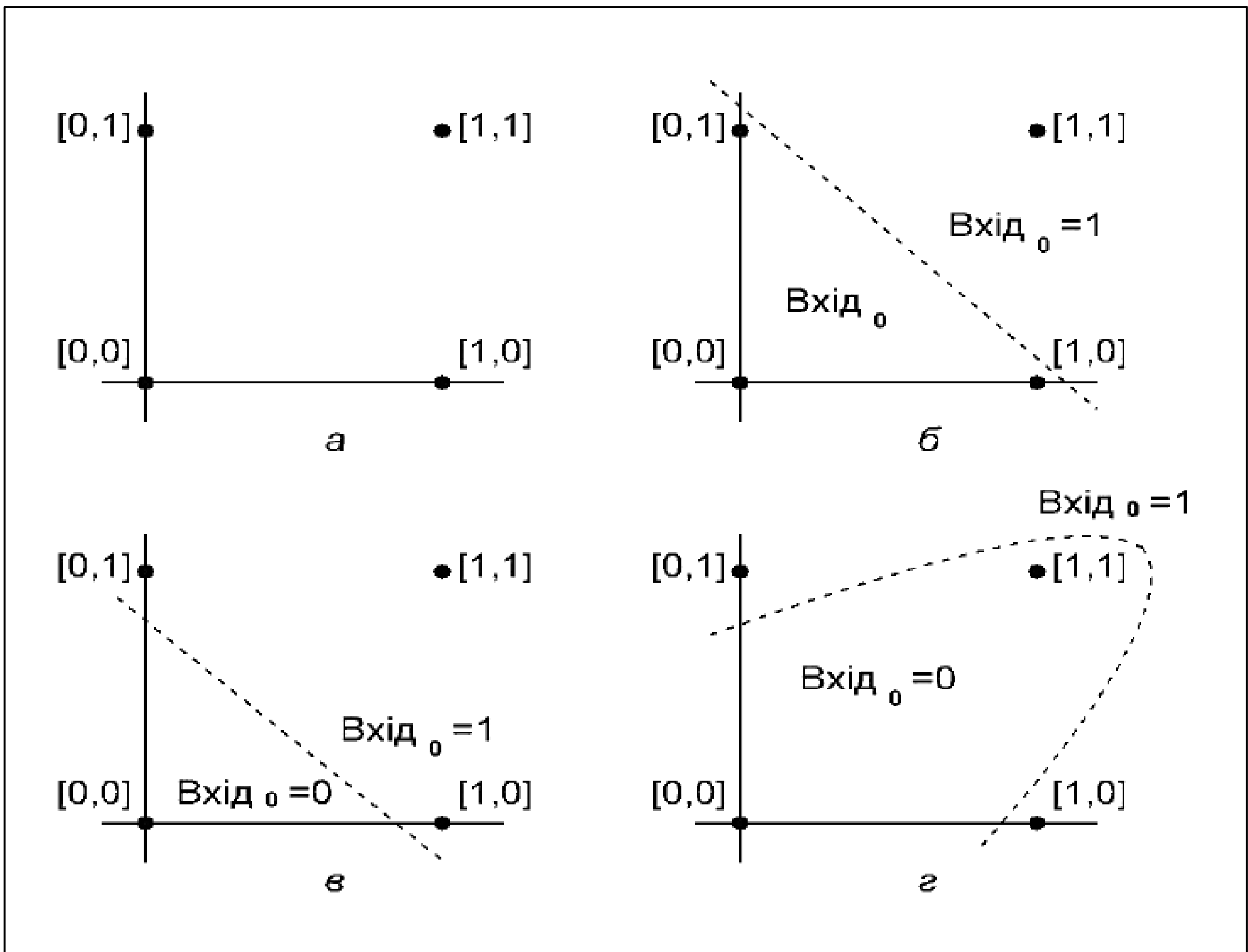


Рис. 1.26. Можливі комбінації вхідних даних (*а*); поділ простору вхідних даних залежно від потрібних вихідних значень (*б*); розділення вхідних даних для операції АБО (*в*); приклад вхідних даних, які не є лінійно нероздільними (*г*)

Зазначимо, що одношаровий перцептрон може вирішити завдання, тільки якщо дані є лінійно нероздільними. Це є правильним незалежно від розмірності вхідних вибірок. Двовимірний випадок легко візуалізувати, тому що можна побудувати точки й розділити їх лінією. Щоб узагальнити концепцію лінійної роздільності, замість «лінії» можна використовувати слово «гіперплощина». Гіперплощина − це геометричний об'єкт, який може розділяти дані в n-вимірному просторі. У двовимірнму середовищі гіперплощина є одновимірним об'єктом (тобто лінією). У тривимірному середовищі гіперплощина − це звичайна двовимірна площина. У n-вимірному середовищі гіперплощина має (n-1) вимірювань.

Вирішення завдань, які не є лінійно нероздільними. Під час процедури навчання одношаровий перцептрон використовує навчальні вибірки, щоб визначити, де повинна знаходитися гіперплощина класифікації. Після того як гіперплощину, що надійно розділяє дані на правильні категорії класифікації, знайдено − він готовий до роботи. Однак перцептрон не знайде цієї гіперплощини, якщо вона не існує. Розглянемо приклад зв'язку «вхід-вихід», який не є лінійно нероздільним.

Такою операцією є «виключає АБО» (або додавання за модулем 2). Не можна розділити дані «виключає АБО» за допомогою прямої лінії. Таким чином, одношаровий перцептрон не може реалізувати функціональність, що надається логічним елементом «виключає АБО». Тому можна стверджувати, що завдання багатьох інших (набагато більш цікавих) додатків не можуть бути вирішені з допомогою одношарового перцептрона.

Однак можна істотно розширити можливості нейронних мереж вирішувати завдання, просто увівши один додатковий шар вузлів. Це перетворює одношаровий перцептрон на багатошаровий (MLP, multilayer perceptron). Такий шар називають «прихованим», оскільки він не має прямого інтерфейсу з зовнішнім середовищем.

Додавання в перцептрон прихованого шару − досить простий спосіб значного поліпшення системи в цілому, але не можна сподіватися, що такий метод не матиме недоліків. Головний недолік − навчання стає більш складним. Цю проблему й розглянемо далі.

1.12. Оновлення вагових коефіцієнтів

Уведемо для багатошарових перцептронів (MLP, multilayer perceptron) основні поняття й означення. Це буде сприяти більш ясному розумінню особливостей процесу оновлення вагових коефіцієнтів і характеру зворотного поширення.

- ***Сигнал преактивації*** (скорочено ***SpreA***). Цей термін означає сигнал, який є вхідним для функції активації вузла (він є числом для однієї навчальної ітерації). Таке число отримують шляхом обчислення скалярного добутку масиву, що містить вагові коефіцієнти, і масиву, що містить значення, які виходять з вузлів у попередньому шарі. Нагадаємо, що скалярний добуток є еквівалентним поелементному множенню двох масивів з подальшим підсумовуванням елементів масиву, отриманого внаслідок цього множення.
- ***Сигнал постактивації*** (скорочено ***SpostA***). Цим терміном позначається сигнал (це також число для окремої ітерації), який виходить з вузла. Він створюється шляхом застосування функції активації до сигналу преактивації. Позначимо функцію активації $f_A()$ і назвемо її логістичною функцією сигмоїдального типу. Таку функцію докладно розглянуто в розд.1.2.
- ***ItoH*** і ***HtoO*** − ідентифікатори вагових матриць, які використовуються для позначення ваг прихованого шару. ***ItoH*** указує ваги, які застосовуються до значень, що передаються з вхідних вузлів у приховані вузли (**ItoH** − **I**nput **to** **H**idden), а **HtoO** визначає ваги, які застосовуються до значень, що передаються з прихованих вузлів у вихідний вузол (**HtoO** − **H**idden **to** **O**utput).
- ***Мета*** − правильне вихідне значення для навчальної вибірки, яке позначається літерою ***T*** (**T**arget).

- ***Сигнал помилки*** ($\boldsymbol{S}_{\text{помилки}}$) − це остання помилка, яка поширюється назад до прихованого шару через функцію активації вихідного вузла.
- ***Кінцева помилка*** (***FE*** − **F**inal **E**rror) − це різниця між сигналом постактивації від вихідного вузла ($\boldsymbol{SpostA, O}$ і метою **T**. Розраховується за формулою ***FE*** **=** $\boldsymbol{SpostA, O}$ **−** ***T***.
- ***Швидкість навчання*** − скорочена назва ***LR*** (**L**earning **R**ate).
- ***Градієнт*** являє собою внесок заданої ваги в сигнал помилки. Ми змінюємо ваги, віднімаючи цей внесок (помножений на швидкість навчання, якщо необхідно).

Сигнал преактивації вузла обчислюється шляхом отримання скалярного добутку − поелементного множення двох масивів (або векторів). Потім ці добутки підсумовуються. Перший масив містить значення постактивації з попереднього шару, а другий масив містить ваги, які з'єднують попередній шар з поточним шаром. Таким чином, якщо масив постактивації попереднього шару позначено як $\boldsymbol{x}$, а вектор ваг − як $\boldsymbol{w}$, то значення преактивації обчислюється так:

$$\boldsymbol{SpreA} = \boldsymbol{wx} = \sum(\boldsymbol{w_1 x_1} + \boldsymbol{w_2 x_2} + \cdots + \boldsymbol{w_n x_n}).$$

Зміщення (позначається як $\boldsymbol{b}$) змінює цю процедуру таким чином:

$$\boldsymbol{SpreA} = (\boldsymbol{wx}) + \boldsymbol{b} = \sum(\boldsymbol{w_1 x_1} + \boldsymbol{w_2 x_2} + \cdots + \boldsymbol{w_n x_n}) + \boldsymbol{b}.$$

Зміщується сигнал, який обробляється функцією активації, що робить мережу більш гнучкою і стійкою. Використання букви $\boldsymbol{b}$ для позначення зміщення нагадує «перетин осі $\boldsymbol{y}$» у стандартному рівнянні для прямої лінії: $\boldsymbol{y} = \boldsymbol{mx} + \boldsymbol{b}$. Це не випадковий збіг. Також можна помітити, що масив ваг є еквівалентним нахилу:

$$\boldsymbol{SpreA} = (\boldsymbol{wx}) + \boldsymbol{b};$$

$$\boldsymbol{y} = \boldsymbol{mx} + \boldsymbol{b}.$$

Формули поновлення вагових коефіцієнтів отримують шляхом взяття похідної від функції помилки (зазвичай використовують середньоквадратичну помилку) відносно ваги, яку необхідно змінити. Для ваг від прихованих вузлів до вихідних вузлів (**HtoO**) використовуються такі показники:

$$\boldsymbol{S}_{\text{помилки}} = \boldsymbol{FE} \times \boldsymbol{f_{A'}}(\boldsymbol{SpreA, O}),$$

$$\text{градієнт}_{\boldsymbol{HtoO}} = \boldsymbol{S}_{\text{помилки}} \times \boldsymbol{SpostA, H}, \text{вага}_{\boldsymbol{HtoO}} =$$

$$= \text{вага}_{\boldsymbol{HtoO}} - (\boldsymbol{LR} \times \text{градієнт}_{\boldsymbol{HtoO}}).$$

Сигнал помилки ($\boldsymbol{S}_{\text{помилки}}$) розраховується шляхом множення кінцевої помилки (***FE***) на значення, що отримується при взятті похідної від функції активації за сигналом преактивації. Він доставляється у вихідний вузол (символ $\boldsymbol{A}'$ означає першу похідну в $\boldsymbol{f_{A'}(SpreA, O)}$. Потім обчислюється градієнт шляхом множення сигналу помилки на сигнал постактивації з прихованого шару. Оновлення ваги проводиться шляхом віднімання цього градієнта від поточного значення ваги. Якщо необхідно змінити розмір кроку, то градієнт множать на швидкість навчання.

Для ваг від вхідних вузлів до прихованих вузлів (**ItoH**) маємо таке:

$$\text{градієнт}_{\boldsymbol{ItoH}} = \boldsymbol{FE} \times \boldsymbol{f_{A'}(SpreA, O)} \times \text{ вага}_{\boldsymbol{HtoO}} \times \boldsymbol{f_{A'}(SpreA, H)} \times \text{вхід} \Longrightarrow$$

$$\Longrightarrow \text{градієнт}_{\boldsymbol{ItoH}} = \boldsymbol{S}_{\text{помилки}} \times \text{вага}_{\boldsymbol{HtoO}} \times \boldsymbol{f_{A'}(SpreA, H)} \times \text{ вхід}\,\boldsymbol{,}\ \text{вага}_{\boldsymbol{ItoH}} =$$

$$\text{вага}_{\boldsymbol{ItoH}} - (\boldsymbol{LR} \times \text{градієнт}_{\boldsymbol{ItoH}}).$$

Числові значення ваг переданих функцій активації вузла під час навчання збільшують або зменшують нахил вхідних даних, а вхідні дані зміщуються по вертикалі. Перехід логістичної функції від $\boldsymbol{f_A(x) = 0}$ до $\boldsymbol{f_A(x) = 1}$ відцентровано при вхідному значенні $\boldsymbol{x = 0}$. Тому, використовуючи зміщення для збільшення або зменшення сигналу преактивації, можна впливати на появу переходу. Так, можна зміщувати графік функції активації вліво або вправо. Ваги ж визначають за правилом − як швидко вхідне значення проходить через $\boldsymbol{x = 0}$. Це впливає на крутизну переходу на графіку функції активації.

Для ваг від вхідних вузлів до прихованих вузлів помилка повинна поширюватися назад через додатковий шар, і це виконується шляхом множення сигналу помилки на вагу між прихованим і вихідним вузлами, з'єднаними з вузлом, який нас цікавить. Таким чином, для поновлення ваги між вхідним і прихованим вузлами, сигнал помилки множиться на вагу, що з'єднує перший прихований вузол з вихідним вузлом. Завершення обчислень відбувається після виконання множення, аналогічного множенню поновлення ваг між прихованими і вихідними вузлами. Для цього береться похідна від функції активації за сигналом преактивації прихованого вузла, а «вхідне» значення розглядається як сигнал постактивації від вхідного вузла.

Зворотне поширення полягає в необхідності оновити ваги між вхідними й прихованими вузлами на основі різниці між згенерованим вихідним сигналом нейронної мережі й цільовими вихідними значеннями, наданими навчальними даними.

Зворотне поширення є способом, з допомогою якого сигнал помилки прямує назад до одного або кількох прихованих вузлів. Масштабування сигналу помилки виробляється з використанням як ваг, що йдуть від прихованого вузла, так і похідної від функції активації прихованого вузла. Під-

сумкова процедура − це спосіб поновлення ваги на основі внеску цієї ваги у вихідну помилку, навіть якщо цей внесок прихований непрямим зв'язком між вагою між вхідним і прихованим вузлами і згенерованим вихідним значенням.

1.13. Навчальні набори даних для нейронних мереж

Далі будемо використовувати для навчання багатошарового перцептрона згенеровані в `Excel` вибірки, а потім подивимося, як нейронна мережа працює з перевірними вибірками. Обговоримо, як використовувати `Excel` для отримання навчальних даних для створюваної нейронної мережі.

Що таке навчальні дані? Зазвичай навчальні вибірки − це дані вимірювань у поєднанні з «рішеннями», які допоможуть нейронній мережі узагальнити всю цю інформацію відповідно до зв'язку «вхід-вихід».

Припустимо, необхідно, щоб нейронна мережа передбачала якість об'єкта на основі кольору, форми й розміру. Якщо ви не впевнені, як саме колір, форма і щільність пов'язані з якістю, і є можливість виміряти колір, форму і розмір, а також визначено критерії якості, то необхідно дослідити властивості безлічі об'єктів, записати їх характеристики, а потім помістити всю цю інформацію в таблицю (рис. 1.27).

Кожен рядок таблиці − це навчальна вибірка, і в ній чотири стовпці: три з них (колір, форма і розмір) є стовпцями вхідних даних, а четвертий − цільовим вихідним значенням.

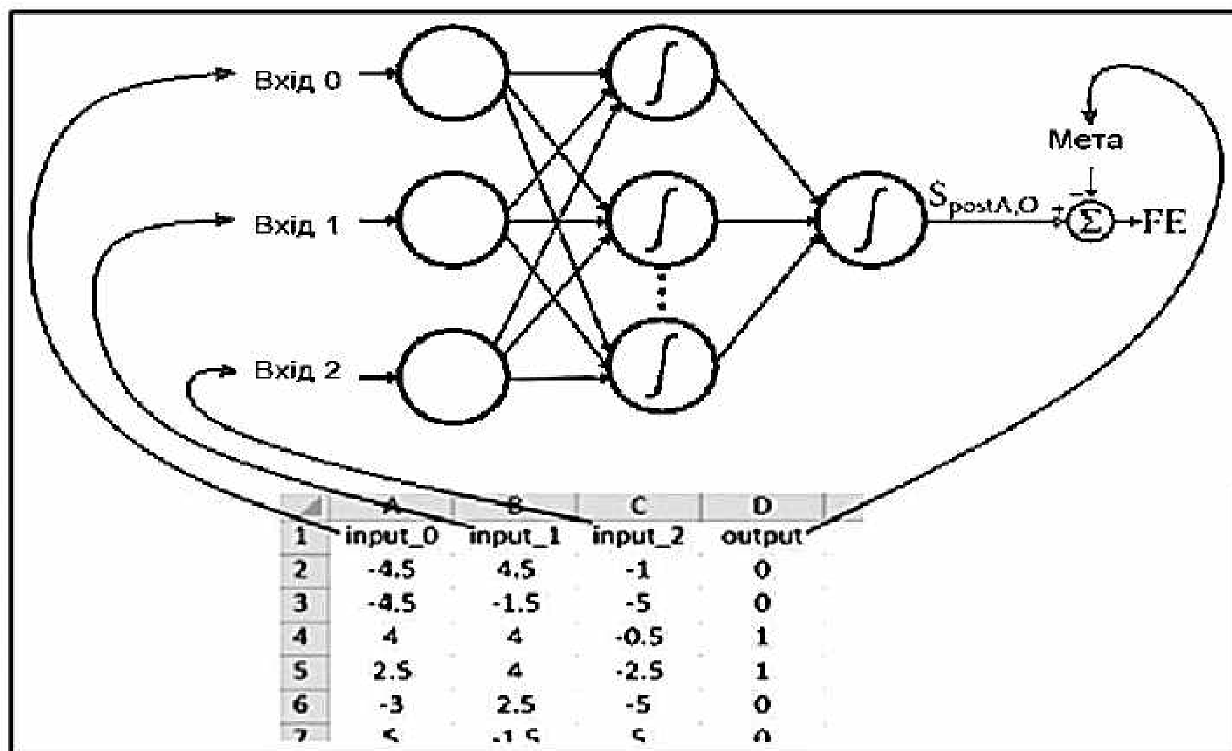


Рис. 1.27. Зв'язок між даними в `Excel` і параметрами нейронної мережі

Під час навчання нейронна мережа знайде зв'язок (якщо такій існує) між трьома вхідними даними і вихідним значенням.

Оцінювання навчальних даних. Ці дані мають оброблятися в числовому вигляді. Необхідно кількісно оцінити результати вимірювань і ваші класифікації.

Наприклад, для кольору можна привласнити кожному об'єкту значення від -1 до +1, де -1 являє собою чорний колір, а +1 означає білий. Якість об'єкта можна оцінювати за п'ятибальною шкалою від «незадовільного» до «відмінного», а потім використовувати унітарний код для зіставлення цих оцінок якості з вихідним вектором з п'яти елементів (рис. 1.28).

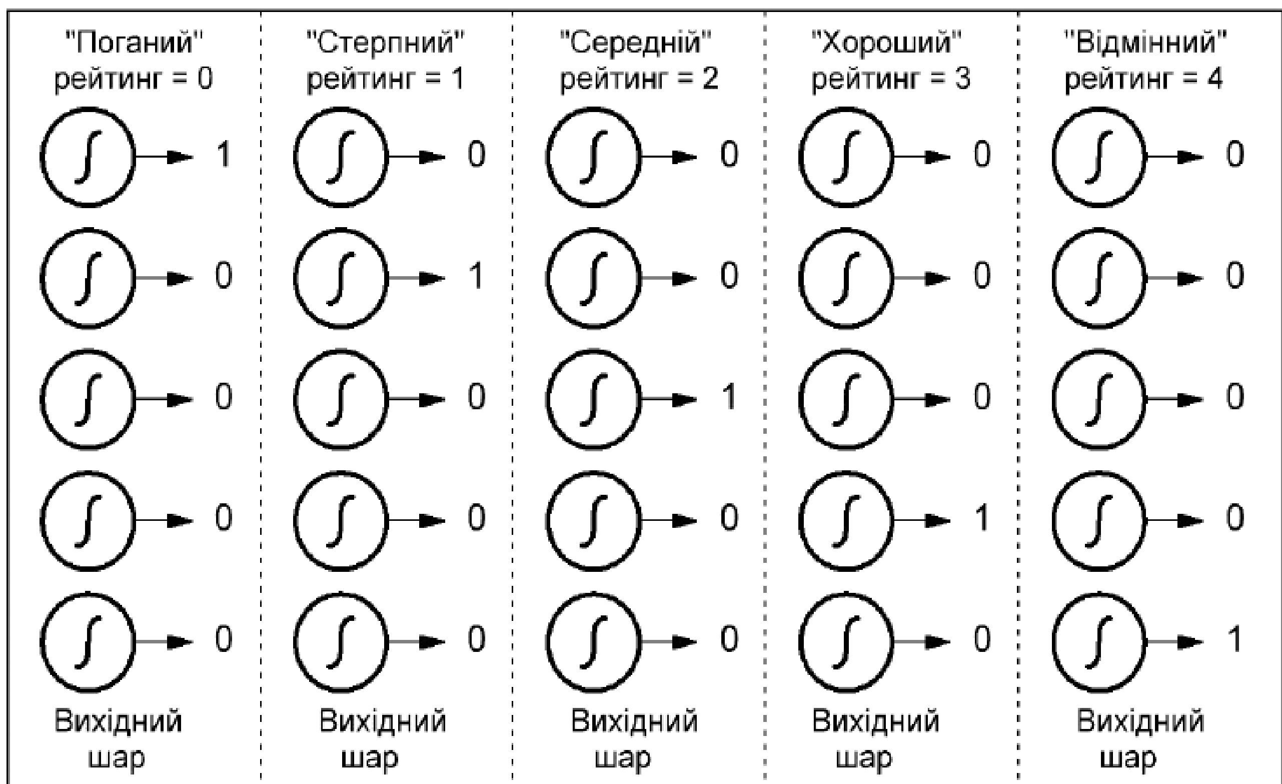


Рис. 1.28. Унітарний код для вихідних значень нейронної мережі

Вихідна схема, яка використовує унітарний код, дає змогу визначити недвійкові класифікації, сумісні з логістичною сігмоїдальною функцією активації. Вихідні дані цієї функції є, по суті, двійковими, оскільки область переходу на графіку є вузькою порівняно з нескінченним діапазоном вхідних значень, для яких вихідне значення є дуже близьким до мінімального або максимального. Це наочно видно на рис. 1.29.

Таким чином, немає сенсу створювати нейронну мережу з одним вихідним вузлом, а потім надавати навчальні вибірки, які мають вихідні значення 0, 1, 2, 3 або 4 (або, якщо ви хочете залишатися в діапазоні від 0 до 1, то це будуть 0, 0,2, 0,4, 0,6 або 0,8). Функція активації вихідного вузла буде стійко дотримуватися мінімального й максимального вихідних значень.

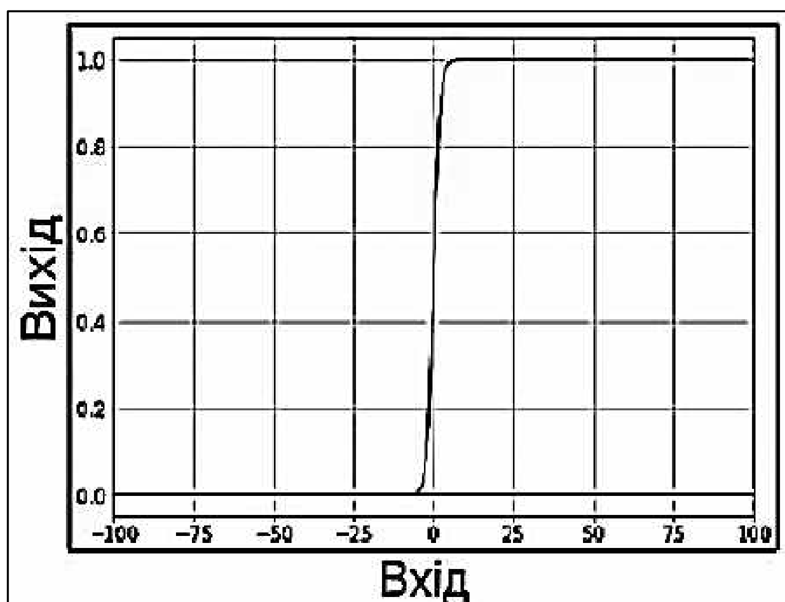


Рис. 1.29. Графік логістичної функції

Створення набору навчальних даних. Нейронна мережа на Python, приклад якої було показано раніше, імпортує навчальні вибірки з файла Excel.

D2 | fx | =IF(AND(A2>0, B2>0, C2<0), 1, 0)

	A	B	C	D	E
1	input_0	input_1	input_2	output	
2	-4.5	4.5	-1	0	
3	-4.5	-1.5	-5	0	
4	4	4	-0.5	1	
5	2.5	4	-2.5	1	
6	-3	2.5	-5	0	
7	5	-1.5	5	0	

Рис. 1.30. Навчальні дані в таблиці Excel

Поточний код для перцептрона обмежений одним вихідним вузлом, тому все, що можна зробити, − це виконати класифікацію «істина/неправда». Вхідні значення − це випадкові числа від -5 до +5, згенеровані за формулою `Excel`, як це показано на рис. 1.30:

```
=RANDBETWEEN(-10, 10)/2
```

Як показано на рис. 1.27, такий результат розраховується так:

```
=IF(AND(A2>0, B2>0, C2<0), 1, 0)
```

Таким чином, вихідне значення одне `true`, тільки якщо `input_0` більше нуля, `input_1` більше нуля, а `input_2` менше нуля. В іншому випадку вихідне значення одне − `false`.

Це математичний зв'язок «вхід-вихід», який перцептрон має витягти з навчальних даних. Завжди можна створити безліч вибірок. Для такого простого завдання, як це, можна досягти дуже високої точності класифікації з 5000 вибірками й однією епохою.

Навчання нейронної мережі. Необхідно встановити вхідні розмірності на три елементи (`I_dim = 3`, якщо використовувати зазначені імена змінних). Налаштуємо нейронну мережу так, щоб у ній було чотири прихованих вузли (`H_dim = 4`), і виберемо швидкість навчання, що дорівнює `0,1` (`LR = 0,1`).

Далі згідно з інструкцією `training_data = pandas.read_excel (...)` уводиться назва таблиці. Після натискання кнопки «`Run`» навчання з 5000 вибірками триватиме всього кілька секунд.

Валідація нейромережі. Нагадаємо, що валідація (validation) − це процес оцінювання кінцевого продукту, тобто необхідно перевірити, чи відповідає програмне забезпечення сподіванням і вимогам клієнта. Це динамічний механізм перевірки та тестування фактичного продукту. Для оцінювання ефективності нейронної мережі створюється друга електронна таблиця і генеруються вхідні й вихідні значення з використанням таких самих формул. Потім ці перевірні дані імпортуються так само, як і навчальні дані:

```
training_data = pandas.read_excel('MLP_Tdata.xlsx')
target_output = training_data.output
training_data = training_data.drop(['output'], axis=1)
training_data = np.asarray(training_data)
training_count = len(training_data[:,0])

validation_data = pandas.read_excel('MLP_Vdata.xlsx')
validation_output = validation_data.output
validation_data = validation_data.drop(['output'], axis=1)
validation_data = np.asarray(validation_data)
validation_count = len(validation_data[:,0])
```

Наступний фрагмент коду показує, як виконати базову валідацію:

```
# перевірка

correct_classification_count = 0
for sample in range(validation_count):
    for node in range(H_dim):
        preActivation_H[node] =
np.dot(validation_data[sample,:], weights_ItoH[:, node])
        postActivation_H[node] = lo-
gistic(preActivation_H[node])

    preActivation_O = np.dot(postActivation_H, weights_HtoO)
    postActivation_O = logistic(preActivation_O)
```

```
    if postActivation_O > 0.5:
        output = 1
    else:
        output = 0

    if output == validation_output[sample]:
        correct_classification_count += 1

print('Percentage of correct classifications:')
print(correct_classification_count*100/validation_count)
```

Використовується стандартна процедура прямого поширення для обчислення сигналу постактивації вихідного вузла, а потім оператор `if/else` для застосування порогового значення, який перетворює значення постактивації на класифікаційне значення `true/false`.

Точність класифікації обчислюється шляхом порівняння значення класифікації з цільовим значенням для поточної вибірки валідації, підрахунку кількості правильних класифікацій і ділення на кількість вибірок валідації.

Пам'ятайте, що якщо закоментувати інструкцію `np.random.seed (1)`, то при кожному запуску програми ваги будуть ініціалізуватися різними випадковими значеннями, і, отже, точність класифікації буде змінюватися від одного запуску до наступного. При виконанні 15 окремих запусків з параметрами, зазначеними вище, отримують 5000 навчальних і 1000 перевірних вибірок. Найнижча точність класифікації становить 88,5 %, найвища − 98,1 %, а середня − 94,4 %.

Таким чином, розглянуто важливу теоретичну інформацію, що належить до навчальних даних нейронної мережі, проведено перший експеримент з навчання і валідації багатошарового перцептрона мовою Python.

1.14. Визначення кількості прихованих шарів і прихованих вузлів у нейронній мережі

Перцептрони, що складаються тільки з вхідних і вихідних вузлів (так звані одношарові перцептрони), є не надто корисними, тому що вони не можуть апроксимувати складні зв'язки «вхід-вихід», які характеризують багато типів реальних явищ. Одношарові перцептрони обмежені лінійно роздільними завданнями. Навіть така базова функція, як логічна функція «виключає АБО», не є лінійно нероздільною.

Додавання прихованого шару між вхідним і вихідним шарами перетворює перцептрон на універсальний апроксиматор, а це, по суті, означає, що він здатний захоплювати і відтворювати надзвичайно складні зв'язки «вхід-вихід».

Наявність прихованого шару робить навчання трохи складнішим, тому що вагові коефіцієнти між вхідним і прихованим шарами непрямо вплива-

ють на кінцеву помилку. Це використовується для позначення різниці між вихідним значенням нейронної мережі й цільовим значенням, заданим навчальними даними.

Методику, яку використовують для навчання багатошарового перцептрона, називають зворотним поширенням кінцевої помилки назад у бік входу нейронної мережі. Це дає змогу ефективно змінювати ваги, які не приєднані безпосередньо до вихідного вузла. Процедура зворотного поширення є розширюваною, тобто процедура дає змогу навчати ваги, пов'язані з довільною кількістю прихованих шарів.

Базову структуру нейронної мережі «багатошаровий перцептрон», або MLP (multilayer perceptron), показано раніше на рис. 1.22.

Скільки прихованих шарів необхідно для мережі? Як і слід було очікувати, на це запитання немає простої відповіді. Однак важливо зрозуміти, що перцептрон з одним прихованим шаром − це потужна обчислювальна система. Якщо не підвищується якість з одним прихованим шаром, то спробуйте спочатку інші вдосконалення − може бути, потрібно оптимізувати швидкість навчання, збільшити кількість епох навчання або розширити набір навчальних даних. Додавання другого прихованого шару збільшує складність коду й час оброблення.

Слід також пам'ятати, що перевантажена нейронна мережа − це не просто марна трата ресурсів процесора і зусиль на написання коду, це може насправді дати «позитивну шкоду», роблячи мережу чутливою до перенавчання (перетренування).

Перцептрон з надлишковою обчислювальною потужністю і недостатніми навчальними даними може використовувати занадто специфічне рішення замість пошуку узагальненого рішення, яке буде більш ефективно класифікувати нові вхідні вибірки.

Скільки потрібно прихованих вузлів? Пошук оптимальної розмірності для прихованого шару завжди пов'язаний зі спробами й помилками. Занадто велика кількість вузлів є небажаною, але їх має бути достатньо, щоб нейронна мережа могла вловити складності зв'язків «вхід-вихід».

Метод спроб і помилок часто дає хороші результати, але потрібна обґрунтована відправна точка. Існує три практичних правила для вибору розмірності прихованого шару:

1. Якщо в нейронній мережі є тільки один вихідний вузол, а необхідний зв'язок «вхід-вихід» є досить простим, то розумно вибрати розмірність прихованого шару, що дорівнює двом третинам вхідної розмірності.

2. Якщо є кілька вихідних вузлів або необхідний зв'язок «вхід-вихід» є складним, то можна зробити розмірність прихованого шару таким, що дорівнює сумі вхідної і вихідної розмірності (але при цьому вона має бути меншою від подвоєної вхідної розмірності).

3. Якщо необхідний зв'язок «вхід-вихід» є вкрай складним, то доцільно встановити розмірність прихованого шару такою, що є на одиницю меншою від подвоєної вхідної розмірності.

1.15. Підвищення точності прихованого шару нейронної мережі

Проведемо деякі експерименти з класифікації та зберемо дані про взаємозв'язок між розмірністю прихованого шару й продуктивністю мережі. А також розглянемо, як змінити прихований шар для підвищення точності нейронної мережі, використовуючи реалізацію на Python і приклади завдань.

Кількість вузлів у прихованому шарі впливає на властивості класифікації та швидкість нейронної мережі «перцептрон». Проведемо експерименти, які допоможуть сформулювати припущення про те, як розмірність прихованого шару дають змогу спроектувати нейронну мережу з заданими властивостями.

Порівняння ефективності на Python. Код нейронної мережі на Python, поданий раніше, уже містить розділ, у якому розраховується точність навченої нейронної мережі при класифікації вибірок з перевірного набору даних. Тому достатньо додати код, який буде повідомляти час виконання для навчання (що містить операції прямого й зворотного поширення). Для реальної роботи з класифікації (що містить тільки операцію прямого поширення) використовуємо функцію `time.perf_counter ()`:

```
# Навчання

t_trainingstart = time.perf_counter()

for epoch in range(epoch_count):
            weights_HtoO[H_node] -= LR * gradient_HtoO

t_trainingstop = time.perf_counter()
```

Час початку й закінчення валідації генерується так само:

```
# Перевірка

t_validationstart = time.perf_counter()

correct_classification_count = 0
        correct_classification_count += 1

t_validationstop = time.perf_counter()

print('Percentage of correct classifications:')
```

Два вимірювання часу оброблення подано таким чином:

```
print('Percentage of correct classifications:')
print(correct_classification_count*100/validation_count)

print('Training time:')
print(t_trainingstop - t_trainingstart)

print('Validation time:')
print(t_validationstop - t_validationstart)
```

Навчальні дані й методика вимірювань. Нейронна мережа буде виконувати класифікацію «істина/неправда» для вхідних вибірок, що складаються з чотирьох числових значень від -20 до +20. Таким чином, у нас є чотири вхідних вузли й один вихідний вузол, а вхідні значення генеруються з допомогою формули в Excel (рис. 1.31).

A2 f_x =RANDBETWEEN(-20, 20)

	A	B	C	D	E
1	input_0	input_1	input_2	input_3	output
2	5	-19	-13	17	0
3	-12	-10	9	6	0
4	19	14	8	5	1
5	-20	6	17	-20	0
6	18	15	-10	1	0
7	6	-10	-8	-9	0
8	3	11	-10	-8	0

Рис. 1.31. Генерування вхідних даних

Набір навчальних даних складається з 40000 вибірок, а перевірний набір − з 5000 вибірок. Швидкість навчання становить 0,1, виконується тільки одна навчальна епоха.

Виконаємо три експерименти, що подають зв'язки «вхід-вихід» різного ступеня складності. Інструкція `np.random.seed (1)` є закоментованою, тому початкові значення ваг будуть різними, а отже, буде різною й точність класифікації.

У кожному експерименті програма буде виконуватися п'ять разів (з одними й тими ж даними навчання й перевірки) для кожної розмірності прихованого шару, а остаточні виміри точності й часу оброблення будуть являти собою середнє арифметичне цих результатів, отриманих при п'яти окремих запусках.

Експеримент 1: завдання низької складності. У цьому експерименті вихідні дані мають значення `true`, тільки якщо перші три вхідних сигнали є більшими від нуля, як показано нижче, на скріншоті Excel (рис. 1.32). Зверніть увагу − четвертий вхід не впливає на вихідне значення.

E2 f_x =IF(AND(A2>0, B2>0, C2>0), 1, 0)

	A	B	C	D	E
1	input_0	input_1	input_2	input_3	output
2	3	13	-3	7	0
3	19	-20	17	-20	0
4	7	-2	20	16	0
5	-4	-8	-13	-15	0
6	-2	18	12	-8	0

Рис. 1.32. Експеримент 1. Залежність між вхідними даними й вихідним значенням

Цей зв'язок «вхід-вихід» можна кваліфікувати як досить простий для багатошарового перцептрона. Тому почнемо з розмірності прихованого шару, що дорівнює двом третинам вхідної розмірності.

Оскільки не може бути прихованого шару з дробовою кількістю вузлів, почнемо з `H_dim = 2`. Результати розрахунків занесено в табл. 1.1.

Таблиця 1.1

Результати експерименту № 1

Кількість прихованих вузлів	Точність класифікації, %	Час навчання, с	Час валідації, с
2	88,83	3,856	0,014
3	93,96	5,386	0,173
4	96,60	7,246	0,190
5	98,23	8,693	0,218

Очевидним є поліпшення класифікації аж до п'яти прихованих вузлів. Але ці значення перебільшують вигоду від збільшення кількості прихованих вузлів від чотирьох до п'яти, тому що точність одного із запусків з чотирма прихованими вузлами становила 88,6 %, і це знизило середнє значення.

Якщо виключити цей запуск з низькою точністю, то середня точність для чотирьох прихованих вузлів буде насправді трохи вищою, ніж середня для п'яти прихованих вузлів. У цьому випадку чотири прихованих вузли забезпечать найкращий баланс між точністю та швидкістю.

Ще одна важлива річ, яку слід зазначити в цих результатах, − відмінність полягає в тому, як розмірність прихованого шару впливає на час навчання та час оброблення. Перехід від двох до чотирьох прихованих вузлів збільшує час валідації в 1,3 раза, але при цьому час навчання збільшується в 1,9 раза.

Навчання є значно складнішим в обчислювальному відношенні, ніж оброблення з прямим поширенням, тому необхідно пам'ятати про те, як конфігурація мережі впливає на здатність навчати нейронну мережу за прийнятний час.

Експеримент 2: завдання помірної складності. На рис. 1.33 зображено зв'язок «вхід-вихід». Тепер на вихідне значення впливають усі чотири входи, і це не так просто, як в експерименті 1. Результати зведено в табл. 1.2.

У цьому випадку п'ять прихованих вузлів дадуть найкраще поєднання точності і швидкості, хоча в черговий раз при запусках для чотирьох при-

хованих вузлів було отримано одне значення точності, яке було значно нижчим, ніж інші. Якщо проігнорувати цей викид, результати для чотирьох, п'яти і шести прихованих вузлів будуть дуже схожими.

Той факт, що запуски з п'ятьма і шістьма прихованими вузлами породили будь-які викиди, приводить до цікавих висновків − можливо, збільшення розмірності прихованого шару робить мережу більш стійкою до умов, які з якоїсь причини роблять навчання особливо складним.

E2 fx =IF(AND(B2>A2, C2<(B2+A2), D2>0), 1, 0)

	A	B	C	D	E	F
1	input_0	input_1	input_2	input_3	output	
2	-17	-14	16	-9	0	
3	11	14	20	-1	0	
4	0	15	20	-11	0	
5	-1	19	-14	3	1	

Рис. 1.33. Експеримент 2. Залежність між вхідними даними й вихідним значенням

Таблиця 1.2

Результати експерименту № 2

Кількість прихованих вузлів	Точність класифікації, %	Час навчання, с	Час валідації, с
3	91,76	5,465	0,171
4	96,64	7,022	0,192
5	98,74	8,621	0,235
6	98,70	10,080	0,252

Експеримент 3: завдання високої складності. На рис. 1.34 показано, що новий зв'язок «вхід-вихід» знову використовує всі чотири вхідних значення, і введено нелінійність (піднесення до квадрата одного з вхідних сигналів і добування квадратного кореня з іншого). Результати зведено в табл. 1.3.

Навіть з сімома прихованими вузлами точність виявилася нижчою, ніж тільки з трьома прихованими вузлами в завданні низької складності. Можна було б поліпшити продуктивність для завдання високої складності, змінивши інші параметри нейронної мережі, наприклад зміщення або «відпал» (annealing) швидкості навчання.

Тим не менш, є сенс зберігати розмірність прихованого рівня на рівні семи, доки не стане ясно, що інші вдосконалення дадуть змогу нейронній мережі підтримувати адекватну продуктивність з меншим прихованим рівнем.

E2 fx =IF(AND((A2*A2)>16, SQRT(ABS(B2))>2, C2<0, D2>0), 1, 0)

	A	B	C	D	E	F	G
1	input_0	input_1	input_2	input_3	output		
2	6	-6	-18	-8	0		
3	16	9	11	11	0		
4	-6	-14	0	2	0		
5	-9	-17	7	18	0		

Рис. 1.34. Експеримент 3. Залежність між вхідними даними й вихідним значенням

Таблиця 1.3

Результати експерименту № 3

Кількість прихованих вузлів	Точність класифікації, %	Час навчання, с	Час валідації, с
4	89,92	6,960	0,191
5	91,06	8,718	0,225
6	91,64	10,086	0,254
7	92,38	11,788	0,274

Висновок. Ми розглянули кілька цікавих вимірювань, які створюють досить чітку картину взаємозв'язку між розмірністю прихованого шару й продуктивністю перцептрона. Можна вважати, що ці експерименти дають тільки базову інформацію, на яку можна спиратися, для експериментів з проектуванням і навчанням нейронних мереж.

2. ЗГОРТКОВІ НЕЙРОННІ МЕРЕЖІ ДЛЯ КЛАСИФІКАЦІЇ ЗОБРАЖЕНЬ

1998 року опубліковано одну з перших робіт, присвячених новій архітектурі нейронних мереж [1]. Ця архітектура, що отримала назву CNN (Convolutional Neural Network), довгий час була незатребуваною. Однак 2012 року, після змагань з комп'ютерного зору, на основі бази даних **ImageNet** [4] було розроблено згорткову нейронну мережу, здатну класифікувати мільйони зображень з тисяч різних категорій з помилкою всього 15,8 % [2]. Ці результати набули широкого визнання серед розробників і користувачів і стали стимулом до впровадження CNN у різних додатках з розпізнавання образів у системах технічного зору.

Нагадаємо, що база даних **ImageNet** − це проект зі створення й супроводження масивної бази даних анотованих зображень, призначений для відпрацювання й тестування методів розпізнавання образів і машинного зору. Станом на 2016 рік у базу даних було записано близько десяти мільйонів URL з зображеннями, які пройшли ручну анотацію для ImageNet. З 2010 року ведеться проект ILSVRC (англ. ImageNet Large Scale Visual Recognition Challenge − кампанія з широкомасштабного розпізнавання образів в **ImageNet**), у межах якого різні програмні продукти щорічно змагаються в класифікації й розпізнаванні об'єктів і сцен у базі даних **ImageNet**.

Мета цього розділу − ознайомити читача з проблематикою формування архітектури згорткових нейронних мереж для класифікації зображень, зі структурою й топологією цих мереж, методикою вибору активаційних функцій і методами навчання нейронної мережі. Навчання − процедура мінімізації функції помилки шляхом коригування вагових коефіцієнтів синаптичних зв'язків між нейронами.

2.1. Методи нейромережних технологій і проектів для класифікації зображень

Машинне навчання. Нейронні мережі, розглянуті раніше, є одним з різновидів алгоритмів машинного навчання, або `machine learning`. Машинне навчання − це один з підрозділів штучного інтелекту. Основною властивістю алгоритмів `machine learning` є їх здатність навчатися в процесі роботи. Наприклад, при побудові алгоритму дерева рішень не має апріорної інформації про характер вхідних даних. Використовується тільки деякий вхідний набір об'єктів і значення ознак для кожного з них разом з міткою класу. У процесі побудови дерева рішень алгоритм сам виявляє приховані закономірності, тобто навчається, і після навчання здатний передбачати клас уже для нових об'єктів, яких він не бачив раніше.

Виокремлюють два основних типи машинного навчання: навчання з учителем і навчання без учителя. У першому випадку мається на увазі, що алгоритму крім самих вихідних даних надається деяка додаткова інформація про них, яку він може в подальшому використовувати для навчання. Найбільш популярними завданнями для навчання з учителем є завдання

класифікації і регресії. Наприклад, завдання класифікації можна сформулювати таким чином: маючи певний набір об'єктів, кожен з яких належить до одного з декількох класів, необхідно визначити, до якого саме з цих класів належить новий об'єкт.

Навчання без учителя характеризується тим, що алгоритму не надається додаткової інформації, крім самого набору вихідних даних. Популярний приклад навчання без учителя − завдання кластеризації. Суть її полягає в тому, що на вхід алгоритму подається деяка кількість об'єктів, що належать різним класам (але до якого класу належить кожен об'єкт, невідомо, може бути невизначеною і кількість класів), і мета роботи алгоритму − розбити цю множину об'єктів на підмножини, що належать до одного класу.

Алгоритми машинного навчання для завдань класифікації можна структурувати таким чином:

- класифікатори, що базуються на правилах (Rule-based Classifiers), − пошук правил віднесення об'єктів до того чи іншого класу на базі статистичної метрики з допомогою оператора «`if - then`»;
- логістична регресія ґрунтується на пошуку лінійної площині, максимально точно розділяє простір на два півпростори так, що об'єкти різних класів належать різним півпросторам. Рівняння цільової площини визначається у вигляді лінійної комбінації вхідних параметрів. Для навчання подібного класифікатора зазвичай застосовують метод градієнтного спуску;
- байєсівський класифікатор − класифікація полягає в пошуку класу з максимальною апостеріорною ймовірністю при умові, що всі параметри мають значення, що відповідають екземпляру, який класифікується.

Однак нейронні мережі на основі алгоритмів машинного навчання мають кілька серйозних недоліків. Одна з найбільш важливих проблем при використанні алгоритмів машинного навчання − вибір правильних ознак для навчання. Це особливо важливо при вирішенні завдань розпізнавання зображень, мови і т. ін. Зазвичай вибір набору ознак для навчання визначає успішність роботи алгоритму. Алгоритми машинного навчання потребують істотних обчислювальних ресурсів і великих витрат часу.

Глибоке навчання (`Deep learning`). З огляду на теорію машинного навчання, `deep learning` є підмножиною так званого `representation learning`. Основна концепція `representation learning` − автоматичний пошук ознак, на основі яких у подальшому буде працювати деякий алгоритм, наприклад класифікації.

Ще одна важлива проблема при використанні машинного навчання − наявність факторів мінливості зображень. Це істотно впливає на вигляд вихідних даних. У завданнях розпізнавання зображень такими факторами можуть бути кут, під яким предмет на зображенні повернуто до спостеріга-

ча, освітлення і т. д. Тому для завдань ідентифікації об'єктів розумно враховувати не конкретні низькорівневі факти, такі як колір певного пікселя, а характеристики вищого рівня абстракції. Однак визначення наявності тільки однієї або декількох можливих ознак, виявлення їх усіх і складання алгоритмів для перевірки зображення на їх наявність − ці заняття є малопродуктивними. У таких ситуаціях можна використовувати переваги підходу `deep learning`.

Глибоке навчання базується на поданні вихідного об'єкта у вигляді ієрархічної структури ознак таким чином, що кожен наступний рівень ознак будується на основі елементів попереднього рівня. Так, якщо йдеться про зображення, то найнижчим рівнем будуть вихідні пікселі зображення, наступним рівнем будуть відрізки, які можна виокремити серед цих пікселів, потім – кути та інші геометричні фігури, у які складаються відрізки. На наступному рівні утворюються вже впізнавані для людини об'єкти, наприклад колеса, і нарешті, останній рівень ієрархії відповідає за конкретні предмети на зображенні, наприклад, за автомобіль.

Згорткові мережі − одна з найбільш популярних останнім часом моделей глибокого навчання, що застосовується, першою чергою, для розпізнавання зображень. Концепція згорткових мереж базується на трьох основних ідеях:

- локальна чутливість (`local receptive fields`) − при розпізнаванні зображень. Розпізнавання того чи іншого елемента на зображенні має передусім впливати на його безпосереднє оточення, у той час як пікселі, що знаходяться в іншій частині зображення, швидше, з цим елементом ніяк не зв'язані і не містять інформації, яка допомогла б правильно його ідентифікувати;
- розділення ваги (`shared weights`) − наявність у моделі розділених ваг фактично означає, що один і той же об'єкт може бути знайдений у будь-якій частині зображення, при цьому для його пошуку в усіх частинах зображення застосовується один і той же набір ваг;
- сабсемплінг (`subsampling`) − концепція, що дає змогу зробити модель більш стійкою до незначних відхилень від шуканого патерна, у тому числі пов'язаних з дрібними деформаціями, зміною освітлення і т. д. Ідея сабсемплінгу полягає в тому, що при зіставленні з патерном ураховується не точне значення для певного пікселя або області пікселів, а його агрегація в деякому околі, наприклад середнє або максимальне значення.

Основа згорткових нейронних мереж − двовимірна просторова згортка. З математичної точки зору, основою згорткових нейронних мереж є операція матричної згортки, яка полягає в поелементному перемножуванні матриці, що являє собою невелику ділянку вихідного зображення (наприклад, 5x5 пікселів), з матрицею того ж розміру, так званим ядром згортки, і подальшому підсумовуванні отриманих значень. При цьому ядро згортки є певним шаблоном, а отримане внаслідок підсумовування число

характеризує ступінь схожості області зображення з цим шаблоном. Кожний шар згорткової мережі складається з певної кількості шаблонів, і завдання навчання мережі полягає в підборі правильних значень у цих шаблонах − так, щоб вони відображали найважливіші характеристики вихідних зображень. Кожен шаблон зіставляється послідовно з усіма частинами зображення − саме в цьому виявляється ідея поділу ваг. Шари такого типу в згортковій мережі називають шарами згортки. Крім шарів згортки в цих мережах є шари сабсемплінгу, які замінюють невеликі області зображення одним числом, тим самим одночасно зменшуючи розмір зразка для роботи наступного шару й роблячи мережу більш стійкою до невеликих змін у даних. В останніх же шарах згорткової мережі зазвичай використовується один або кілька повнозв’язаних шарів, яких навчають для виконання безпосередньо класифікації об'єктів. Останніми роками використання згорткових мереж стало фактично стандартом при класифікації зображень і дає змогу отримати найкращі результати при вирішенні таких завдань.

2.2. Основні етапи розвитку згорткових нейронних мереж для класифікації зображень

Далі розглянемо основні етапи й еволюцію розвитку нейронних мереж на базі глибокого навчання для розпізнавання зображень. Покажемо, яке місце при синтезі нейронних мереж займає deep learning.

Нагадаємо, що **ImageNet** (http://image-net.org/) − це велика візуальна база даних, що містить понад 14 мільйонів зображень з високою роздільною здатністю. Ці зображення були анотовані провідними фахівцями в області оброблення зображень. **ImageNet** містить понад 20 000 категорій. Тому згаданий прорив 2012 року в рішенні ILSVRC 2012 на **AlexNet** часто вважається початком революції глибокого навчання 2010-х років.

Після успішного впровадження **AlexNet** на розгляд ImageNet було подано багато інших архітектур глибокого навчання для досягнення більш високої продуктивності. У табл. 2.1 наведено перелік найбільш значущих робіт зі створення згорткових нейронних мереж для класифікації зображень і стислі коментарі до основних аспектів їх архітектури на базі глибокого навчання. Крім того, тут є посилання на публікації, зроблені авторами у зв'язку з презентацією цих проектів.

Таблиця 2.1

Створення згорткових нейронних мереж

№ п/п	Найменування проекту, рік, публікації	Стислий опис
1	**AlexNet** 2012 р. Публікація: [5]	Проект **AlexNet** був переможцем LSVRC-2012 і являє собою просту, але потужну мережну архітектуру зі згортками. Їх об'єднують рівнями один поверх іншого, за якими йдуть повністю зв'язані рівні вверху. Ця архітектура зазвичай використовується як відправна точка при застосуванні підходу глибокого навчання до завдань комп'ютерного зору

Закінчення табл. 2.1

№ п/п	Найменування проекту, рік, публікації	Стислий опис
2	**VGG-16** and **VGG-19** 2014 р. Публікація: [6]	Модель **VGGNet** запропоновано групою Visual Geometry Group (VGG) Оксфордського університету. Вона посіла друге місце в LSVRC 2014 завдяки використанню тільки фільтрів 3 x 3 по всій мережі замість фільтрів великого розміру (таких як 7 x 7 і 11 x 11). Основний внесок цієї роботи полягає в тому, що вона показує, що глибина мережі є критичним компонентом у досягненні кращого розпізнавання або точності класифікації в згорткових нейронних мережах. Її недоліки полягають у тому, що вона дуже повільно навчається, а вага її мережної архітектури є досить великою (533 МБ для **VGG-16** і 574 МБ для VGG-19). **VGGNet-19** використовує 138 мільйонів параметрів
3	**GoogLeNet/ Inception-V1** 2014 р. Публікація: [7]	Проект **GoogLeNet** (або **Inception-V1**) був переможцем LSVRC-2014, досягнувши 5-го рівня кращих помилок − 6,67 %, що дуже близько до показників на рівні людини. Ця архітектура є навіть глибшою, ніж **VGGNet**. Однак **GoogLeNet** використовує тільки одну десяту кількості параметрів **AlexNet** (від 60 до 4 мільйонів параметрів) через архітектури девяти паралельних модулів, початкового модуля, який базується на декількох дуже маленьких пакетах з метою зменшення кількості параметрів
4	**ResNet-18, -34, -50, -101** and -**152** 2015 р. Публікація: [8]	Залишкові мережі Microsoft (**ResNets**) перемогли LSVRC-2015, і сьогодні є найглибшою мережею зі 153 згортковими шарами, що дає помилку класифікації в топ-5 − 4,9 % (що трохи краще, ніж точність людини). Ця архітектура містить факти, як пропускалися з'єднання, також відомі як вентильні блоки або закриті повторювані блоки, що дає змогу вносити інкрементні зміни в навчання. **ResNet-34** використовує 21,8 мільйона параметрів, **ResNet-50** − 25,6 мільйона, **ResNet-101** − 44,5 мільйона, **ResNet-152** − 60,2 мільйона
5	**Inception-V3** 2015 р. Публікація: [9]	Початкова архітектура була подана як **GoogLeNet** (також відома як **Inception-V1**). Пізніше цю архітектуру було змінено для введення пакетної нормалізації (**Inception-V2**). Архітектура **Inception-V3** містить додаткові ідеї факторизації, мета яких − зменшити кількість з'єднань параметрів без зниження ефективності мережі
6	**Inception-V4** 2016 р. Публікація: [10]	**Inception-V4** походить від **GoogLeNet**. Ця архітектура має більш уніфіковану спрощену архітектуру і більше початкових модулів, ніж **Inception-V3**. **Inception-V4** зміг досягти 80,2 % точності першої статті і 95,2 % топ-5 точності на LSVRC

Далі в табл. 2.2 наведено сучасні проекти з використання алгоритмів глибокого навчання для вирішення різноманітних завдань виявлення об'єктів.

Таблиця 2.2

Сучасні згорткові мережі

№ п/п	Найменування проекту, рік, публікації	Стислий опис
1	**R-CNN** 2014 р. Публікація: [11]	Згорткова мережа на основі регіонів **Region-based Convolutional Network** (**R-CNN**) з перших підходів з використанням згорткових нейронних мереж для об'єкта виявлення, що показує, що така нейронна мережа може привести до більш високої ефективності виявлення об'єктів порівняно з системами на основі більш простих функцій, подібних до **HOG**. Цей алгоритм можна розбити на три етапи: 1. Створення набору регіональних пропозицій. 2. Виконання прямого проходу через модифіковану версію **AlexNet** для кожної пропозиції регіону для отримання векторів ознак. 3. Виявлення потенційних об'єктів з допомогою кількох SVM класифікаторів, а також лінійних регресорів, що змінюють координати обмежувальної рамки
2	**Faster R-CNN** 2015 р. Публікація: [12]	Швидка згорткова мережа на основі регіонів. **The Fast Region-based Convolutional Network** (**Fast R-CNN**) є модифікацією попереднього методу для ефективної класифікації об'єктних додатків. Додатково **Fast R-CNN** використовує кілька інновацій для підвищення швидкості навчання й тестування (одночасно підвищуючи точність виявлення)
3	**Faster R-CNN** 2015 р. Публікація: [13]	**Faster R-CNN** − це модифікація **Fast R-CNN**, яка представляє мережу регіональних програм (**RPN**)
4	**RFCN** 2016 р. Публікація: [15]	**Region-based Fully Convo-lutional Network (RFCN**) − структура тільки зі згортковими шарами, що допускають повне зворотне поширення для навчання і виведення для точного й ефективного виявлення об'єктів
5	**YOLO** 2016 р. Публікація: [16]	**You only look once (YOLO)** − архітектура глибокого навчання, яка прогнозує як обмежувальні рамки, так і ймовірності класів за один крок. Порівняно з іншими детекторами глибокого навчання **YOLO** робить більше помилок у локалізації, але з меншою ймовірністю дає помилкові спрацьовування в фоновому режимі
6	**SSD** 2016 р. Публікація: [17]	**Single Shot MultiBox Detector (SSD)** − єдина глибока нейронна мережа, розроблена для прогнозування як обмежувальних прямокутників, так і ймовірностей класів одночасно з допомогою наскрізної згорткової архітектури нейронної мережі

Закінчення табл. 2.2

№ п/п	Найменування проекту, рік, публікації	Стислий опис
7	**YOLO V2** 2016 р. Публікація: [18]	Автори подали **YOLO9000**, а також **YOLO V2** однією публікацією. **YOLO9000** − виявлення об'єктів в реальному часі. Ця система може виявляти понад 9000 категорій об'єктів, а **YOLO V2** − поліпшена версія **YOLO** спрямована на підвищення точності (залишаючись швидким детектором)
8	**NASNet** 2017 р. Публікація: [19]	Автори подали нейромережний пошук, який використовує ідею рекурентної нейронної мережі для створення архітектур нейронних мереж. **Neural architecture search net (NASNet)** призначено для вивчення архітектури моделі з метою оптимізації кількості шарів, а також підвищення точності
9	**Mask-R-CNN**	**Mask Region-based Convolutional Network (Mask R-CNN)** − ще одне розширення моделі **Faster R-CNN**, яке додає паралельну гілку до виявлення прямокутника з метою прогнозування маски об'єкта. Маска об'єкта − це його сегментація попіксельно в зображенні. Дає змогу сегментувати екземпляр об'єкта

2.3. Аналіз властивостей основних бібліотек алгоритмів глибокого навчання для роботи з нейронними мережами

Для реалізації алгоритмів роботи з нейронними мережами зазвичай використовують одну або декілька наявних бібліотек. Тому доцільно проаналізувати відомі рішення з реалізації алгоритмів deep learning як в теоретичному, так і в практичному плані. Це допоможе виявити переваги й недоліки кожної з бібліотек, що дуже важливо при виборі джерел інформаційної підтримки конкретних нейромережних проектів.

Аналізу піддавалися бібліотеки: **Deeplearning4j**, **Theano**, **Pylearn2**, **Torch**, **Caffe**, **TensorFlow** і **Keras**. Розглянемо більш детально характеристики кожної з них.

Deeplearning4j (www.deeplearning4j.org) − бібліотека з відкритим вихідним кодом для реалізації нейронних мереж та алгоритмів глибокого навчання, написана мовою Java. Підтримується компанією Skymind. Усередині цієї бібліотеки використовується бібліотека для швидкої роботи з n-вимірними масивами **ND4J** розробки тієї ж компанії. **Deeplearning4j** підтримує безліч мереж. Серед цих мереж − багатошаровий персептрон, згорткові мережі, Restricted Bolzmann Machines, Stacked Denoising Autoencoders, Deep Autoencoders, Recursive autoencoders, Deep-belief networks, рекурентні мережі та деякі інші.

Theano (www.github.com/Theano/Theano) − бібліотека мовою Python з відкритим вихідним кодом, яка дає змогу ефективно створювати, обчислювати й оптимізувати математичні вирази з використанням багатовимірних масивів. Для подання багатовимірних масивів і дій над ними при цьому використовується python-бібліотека NumPy. Цю бібліотеку призначено, першою чергою, для наукових досліджень, її було створено групою вчених з університету Монреаля. Можливості **Theano** дуже великі, і робота з нейронними мережами − тільки одна з невеликих її частин. При цьому саме ця бібліотека є найбільш популярною і найчастіше згадується, коли йдеться про роботу з `deep learning`.

Pylearn2 (www.github.com/lisalab/pylearn2) − python-бібліотека з відкритим вихідним кодом, побудована на основі **Theano**, однак вона має більш зручний і простий інтерфейс для дослідників, що надає готовий набір алгоритмів і забезпечує просте конфігурування мереж у форматі YAML-файлів. Розробили й підтримують її вчені з лабораторії LISA університету Монреаля.

Torch (www.torch.ch) − бібліотека для обчислень і реалізації алгоритмів машинного навчання, написана мовою C, але дає змогу дослідникам під час роботи з нею використовувати набагато більш зручну скриптову мову Lua. Ця бібліотека надає власну ефективну реалізацію операцій над матрицями, багатовимірними масивами, підтримує обчислення на **GPU, дає змогу** реалізовувати повнозв'язні і згорткові мережі, має відкритий вихідний код.

Caffe (www.caffe.berkeleyvision.org) − бібліотека, сконцентрована на ефективній реалізації алгоритмів глибокого навчання, що розробляється, першою чергою, Berkley Vision and Learning Center. Як і всі попередні бібліотеки, має відкритий вихідний код. **Caffe** реалізована мовою C, проте надає також зручний інтерфейс для Python. Підтримує повнозв'язні і згорткові мережі, описує мережі у вигляді набору шарів у форматі `.prototxt`, підтримує обчислення на GPU. До переваг бібліотеки належить також наявність великої кількості попередньо навчених моделей і прикладів, що в поєднанні з іншими характеристиками робить бібліотеку найбільш простою для початку роботи.

TensorFlow (https://www.tensorflow.org/install/) − бібліотека, розроблена корпорацією Google для роботи з тензорами. Широко використовується для побудови нейронних мереж. Обчислення на відкритих вихідних кодах підтримуються мовою програмування C ++. На основі цієї бібліотеки будуються високорівневі бібліотеки для роботи з нейронними мережами на рівні цілих шарів. Так, деколи популярна бібліотека **Keras** стала використовувати **Tensorflow** як основний бекенд для обчислень замість аналогічної бібліотеки **Theano**.

Keras (https://keras.io/) − бібліотека для побудови нейронних мереж, що підтримує основні види шарів і структурні елементи. Підтримує як рекурентні, так і згорткові нейромережі, забезпечує реалізацію відомих архі-

тектур нейромереж (наприклад, VGG16). Деякий час тому шари з цієї бібліотеки стали доступними всередині бібліотеки **Tensorflow**. Існують готові функції для роботи з зображеннями і текстом. Ця бібліотека дає змогу на більш високому рівні працювати з нейронними мережами. Як бібліотеки для бекенд можуть використовуватися як **Tensorflow**, так і **Theano**. **Keras** − це бібліотека для Python з відкритим вихідним кодом, що дає змогу легко створювати нейронні мережі. **Keras** сумісна з **TensorFlow**, **Microsoft Cognitive Toolkit**, **Theano** і **MXNet**. **Caffe**, **Keras**, **Tensorflow** і **Theano** − числові платформи мовою Python для розроблення алгоритмів глибокого навчання, що найчастіше використовуються, але є досить складними у використанні.

Оцінювання ефективності використання різних бібліотек. Об'єктивним комплексним показником ефективності використання наявних бібліотек та алгоритмів глибокого навчання є статистичні показники застосування цих бібліотек при реалізації тих чи інших прикладних нейромережних проектів. Доречно аналізувати й зіставляти результати такої статистичної звітності для врахування тенденцій розвитку нейромережних технологій на практиці. На рис. 2.1 показано статистичну звітність за 2018 р. На діаграмі чітко простежується лідерство таких платформ, як **TensorFlow**, **Keras**, **Caffe**, **Theano**. Однак при використанні тієї або іншої платформи необхідно враховувати супутні обставини, наприклад: обмеження на вибір через відсутність швидкодіючих комп'ютерів, труднощі доступу до ліцензійного програмного продукту та ін.

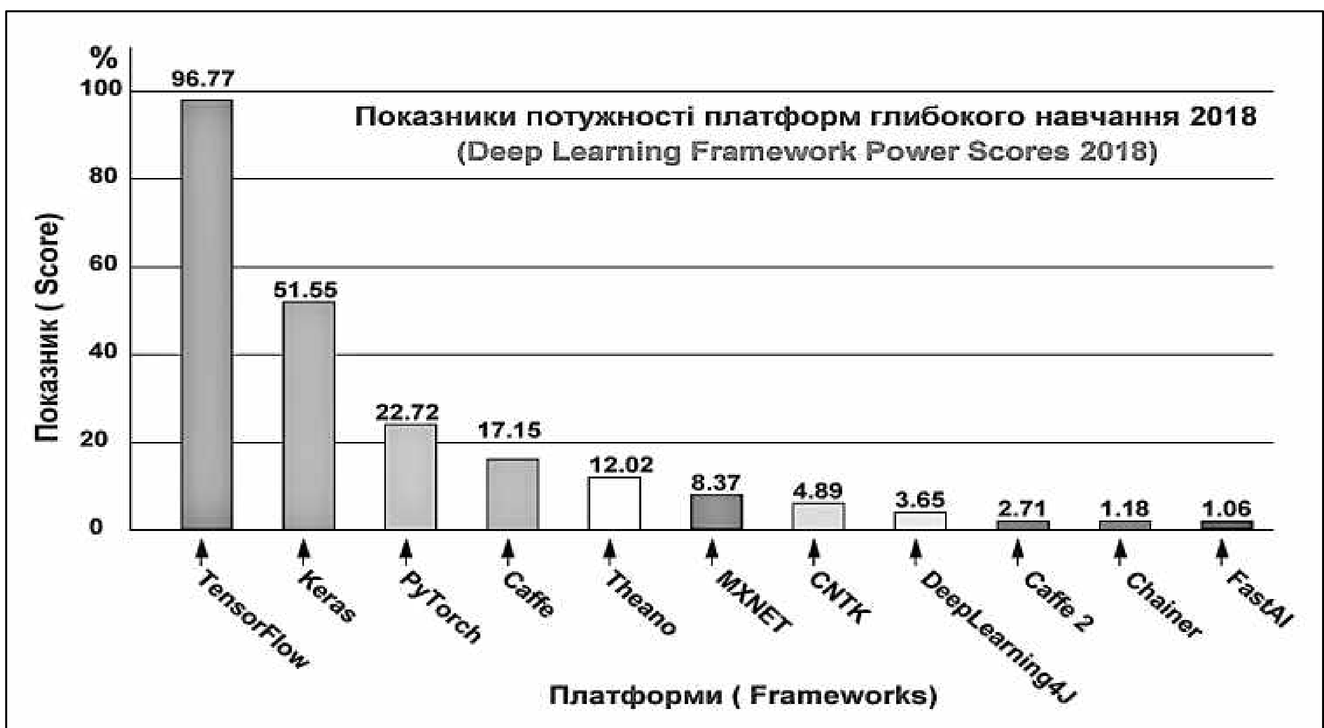


Рис. 2.1. Оцінювання ефективності використання різних бібліотек глибокого навчання

З описаних бібліотек найбільш популярними за багатьма показниками є бібліотеки **Theano** і **Caffe**. **Theano** досить складна в установленні та на-

лаштуванні, але для неї існує велика кількість якісної і добре структурованої документації і прикладів робочого коду. Проте реалізація навіть елементарної нейронної мережі в цій бібліотеці потребує написання великої кількості власного коду. Тому, незважаючи на потенційно набагато ширші можливості цієї бібліотеки порівняно з **Caffe**, доцільно використовувати останню.

Розглянемо детальніше характеристики бібліотеки **Caffe,** яка надає досить простий і зручний для дослідника інтерфейс, даючи змогу легко конфігурувати й навчати нейронні мережі.

Для роботи з бібліотекою потрібно створити опис мережі в форматі `prototxt` (`protocol buffer definition file` − мова опису даних, створена компанією Google). Опис мережі є добре структурованим і зрозумілим для людини і являє собою, по суті, почерговий опис кожного з її шарів. Як з вхідними даними бібліотека може працювати з базою даних (`leveldb` або `lmdb`), in-memory-даними, HDF5-файлами й зображеннями. Також є можливість використовувати з метою розроблення й тестування спеціальний вид даних − DummyData.

Бібліотека підтримує створення шарів таких типів: InnerProduct (повнозв'язний шар), `Splitting` (перетворює дані для передання відразу на кілька вихідних шарів), `Flattening` (перетворює дані з багатовимірної матриці на вектор), `Reshape` (дає змогу змінити розмірність даних), `Concatenation` (перетворює дані з декількох вхідних шарів на один вихідний), `Slicing` тощо. Для згорткових мереж підтримуються також особливі типи шарів − `Convolution` (шар згортки), `Pooling` (шар сабсемплінгу) і `Local Response Normalization` (шар для локальної нормалізації даних). Крім того, підтримуються кілька функцій втрат, які застосовуються при навчанні мережі (`Softmax, Euclidean, Hinge, Sigmoid Cross-Entropy, Infogain` і `Accuracy`), і функцій активації нейронів (`Rectified-Linear, Sigmoid, Hyperbolic Tangent, Absolute Value, Power` і `BNLL`), які також конфігуруються у вигляді окремих шарів мережі.

Для роботи бібліотеки з використанням стандартних скриптів також потрібно створити файл `solver.prototxt`, що описує конфігурацію навчання мережі: кількість ітерацій для навчання, `learning rate`, платформа для обчислень `cpu` або `gpu` тощо.

Навчання моделі може бути реалізоване із застосуванням вбудованих програм (після їх доопрацювання під поточну задачу) або вручну шляхом написання коду з використанням наданого `api` мовою Python або Matlab. При цьому вже існують програми, що дають змогу не тільки виконати навчання мережі, але також, наприклад, створити базу даних на основі наданого списку зображень, при цьому зображення перед додаванням у базу даних будуть зведені до фіксованих розмірів і нормалізовані. Програми, з допомогою яких здійснюється навчання, також інкапсулюють деякі допоміжні дії, наприклад, оцінюють поточну точність моделі через кілька ітерацій і

зберігають поточний стан навченої моделі в файл снапшотів. Використання файлів снапшотів дає змогу в подальшому продовжити навчання моделі замість того, щоб починати спочатку, якщо виникає така необхідність. Після деякої кількості ітерацій можна змінити конфігурацію моделі, наприклад додати новий шар (при цьому ваги вже навчених раніше шарів буде збережено). Це й дає змогу реалізувати описаний раніше механізм покрокового навчання.

2.4. Методи побудови згорткових нейронних мереж

Загальноприйнятий підхід. У класичних методах вирішення завдань аналізу інформації, що міститься в зображеннях, не передбачається використання методів глибокого навчання. Термін «оброблення зображень» стосується широкого класу завдань, вхідними даними для яких є зображення, а вихідними можуть бути не тільки зображення, а й набори пов'язаних з ними характерних ознак. Існує безліч варіантів таких завдань: класифікація, сегментація, анотування, виявлення об'єктів та ін.

Розглянемо класифікацію зображень не тільки тому, що це найпростіше завдання, а й тому, що вона є основою багатьох інших завдань.

Загальний підхід до завдання класифікації зображень складається з таких кроків:

1. Генерація значущих ознак зображення.
2. Класифікація зображення на основі його ознак.

Загальноприйнята послідовність операцій полягає в тому, що поверх створених вручну ознак використовуються такі прості моделі, як багаторівневе сприйняття (MultiLayer Perceptron, MLP), машина векторів підтримки (Support Vector Machine, SVM), метод k найближчих сусідів і логістична регресія. Ознаки генеруються з використанням різних перетворень (наприклад, переведення у відтінки сірого й визначення порогів) і дескрипторів (наприклад, гістограми орієнтованих градієнтів (Histogram of Oriented Gradients, HOG)) або трансформації масштабно-інваріантних ознак (Scale-Invariant Feature Transform, SIFT) тощо.

Основним обмеженням загальноприйнятих методів є участь експерта, що вибирає набір і послідовність кроків для генерації ознак.

Згодом було помічено, що більшість технік генерації ознак можна узагальнити, використовуючи ядра (фільтри) − невеликі матриці (зазвичай розміром 5 × 5), які застосовуються для просторової згортки з вихідними зображеннями. Згортку можна розглядати як послідовний двоступінчастий процес:

1. Пройти одним і тим же фіксованим ядром по всьому вихідному зображенню.
2. На кожному кроці розрахувати скалярний добуток ядра і вихідного зображення в точці поточного розташування ядра.

Результат згортки зображення і ядра називають картою ознак. Це наочно показано на рис. 2.2, 2.3.

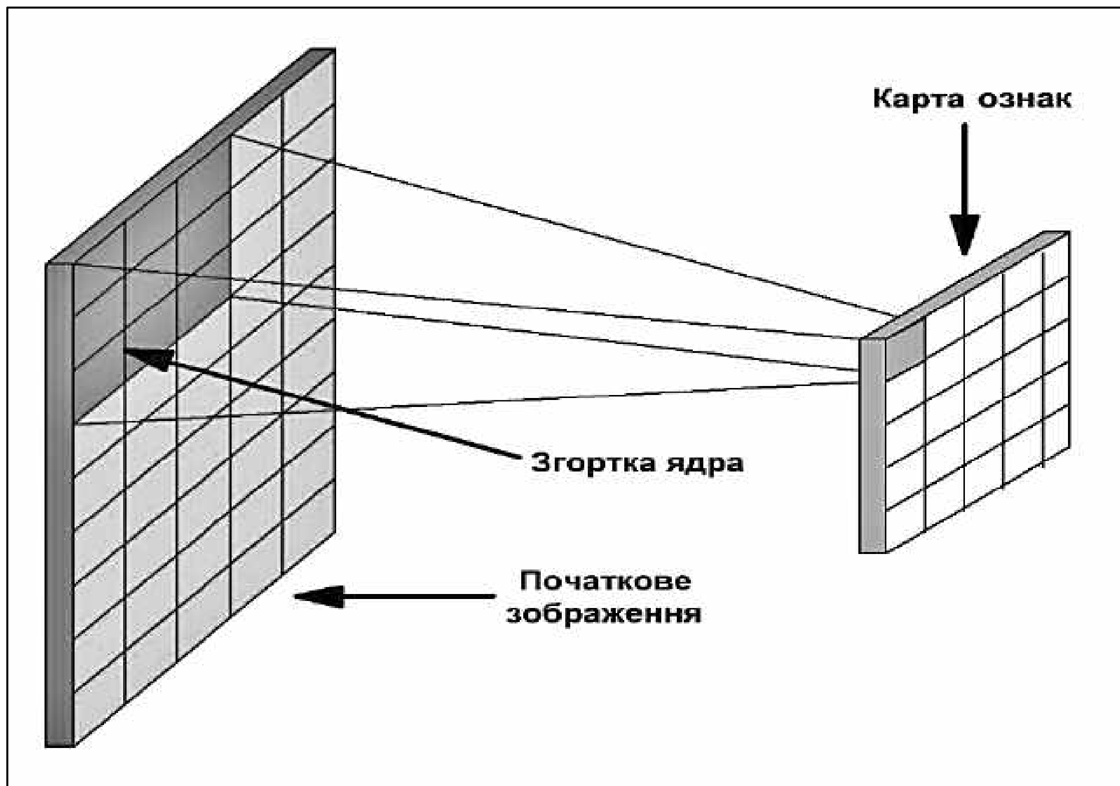


Рис. 2.2. Згортка ядра (темно-зелений) і вихідного зображення (зелений), як наслідок − карта ознак (жовтий)

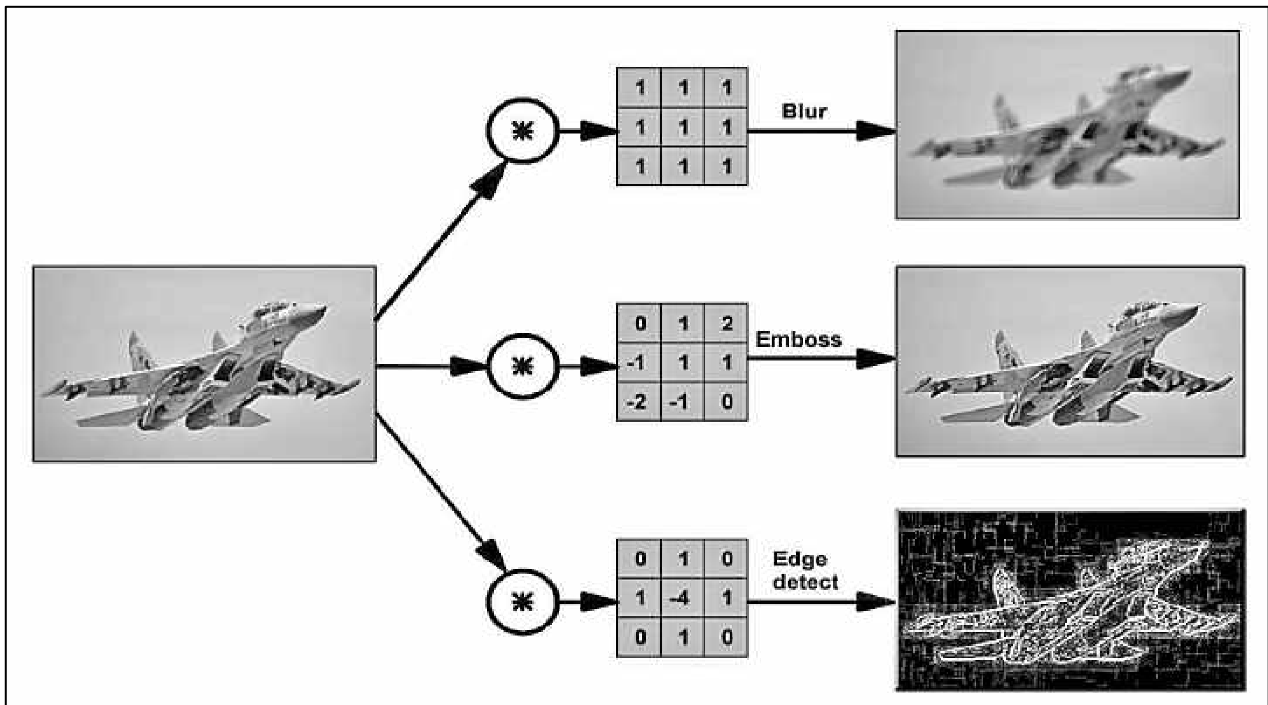


Рис. 2.3. Результат згортки зображення з різними ядрами

Простий приклад трансформації, яку можна зробити з допомогою фільтрів, − це розмиття зображення. Візьмемо фільтр, що складається з

усіх одиниць. Він розраховує середнє значення по околу, що визначається фільтром. Зазвичай, як у цьому випадку, окіл є квадратною ділянкою, але може бути хрестоподібним або яким завгодно ще. Усереднення веде до втрати інформації про точне положення об'єктів, розмиваючи таким чином усі зображення. Подібне інтуїтивне пояснення можна навести для будь-якого фільтра, створеного вручну.

2.5. Згорткові нейронні мережі

Згортковий підхід до класифікації зображень має недоліки:

- багатоступінчастий процес замість наскрізної послідовності;
- фільтри є відмінним інструментом узагальнення, але вони являють собою фіксовані матриці, не завжди ясно, як вибирати ваги у фільтрах.

Однак існують такі зображення, яких навчають фільтри, що працюють за базовим принципом, що є основою CNN. Принцип простий − будемо навчати фільтри, що застосовуються для опису зображень, з метою найкращого виконання ними свого завдання.

Винахід CNN не належить тільки одному автору. Але один з перших випадків їх застосування − це **LeNet-5*** у роботі [1].

CNN не мають описаних вище недоліків: немає необхідності в попередньому визначенні фільтрів, і процес навчання стає наскрізним. Типова архітектура CNN складається з таких частин:

- згорткові шари;
- шари підвибірки;
- щільні (повнозв'язні) шари.

Розглянемо кожну частину докладніше.

Згорткові шари є основним структурним елементом CNN і мають певні характеристики.

Локальна (розріджена) зв'язність. У щільних шарах кожен нейрон з'єднаний з кожним нейроном попереднього шару (тому їх і назвали щільними). У згортковому шарі кожен нейрон з'єднаний лише з невеликою частиною нейронів попереднього шару (рис. 2.4).

Розмір ділянки, з якою з'єднаний нейрон, називають розміром фільтра (довжиною фільтра в разі одновимірних даних, часових серій, або шириною/висотою в разі двовимірних даних або зображень). Розмір фільтра дорівнює 3.

Ваги, з якими здійснюється з'єднання, називають фільтром (вектором для одновимірних даних і матрицею для двовимірних).

Крок − це відстань, на яку фільтр переміщається за даними. Ідея локальної зв'язності є не що інше, як ядро, що переміщається на деякий крок. Кожен нейрон згорткового рівня являє собою і реалізує одне конкретне положення ядра, що ковзає по вихідному зображенню.

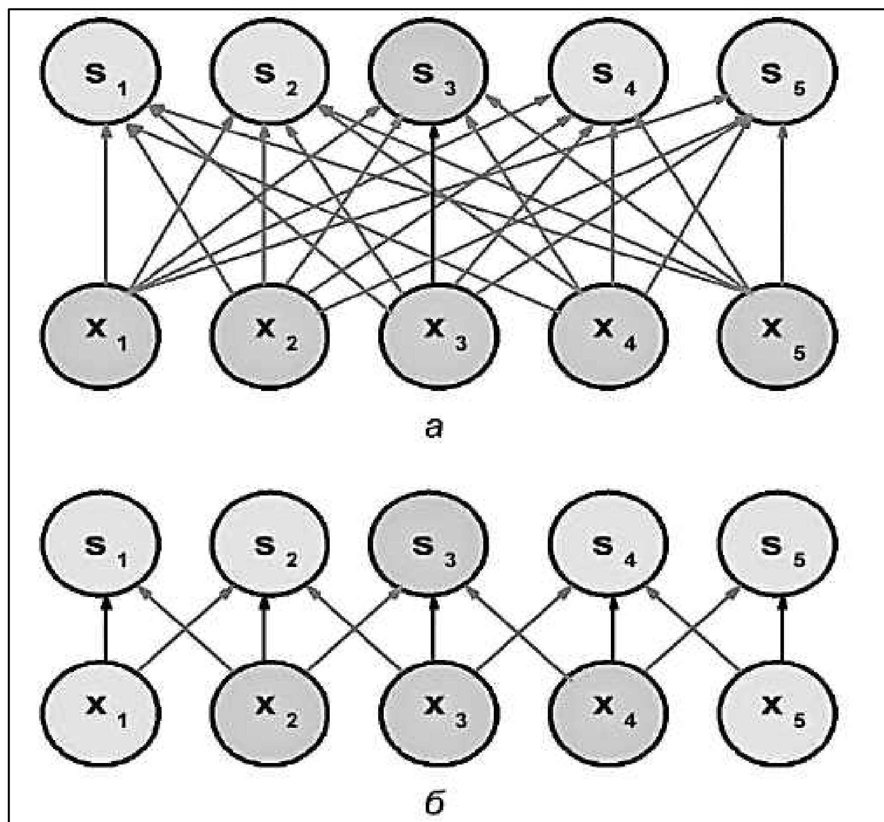


Рис. 2.4. З'єднання нейронів у щільній мережі (*а*); розріджена зв'язність (*б*)

Ще одна важлива властивість − так звана зона сприйнятливості, що відображає кількість позицій вихідного сигналу, які може «бачити» поточний нейрон. Наприклад, зона сприйнятливості першого шару мережі, показаної на рис. 2.5, дорівнює розміру фільтра 3. Тут кожен нейрон з'єднаний тільки з трьома нейронами вихідного сигналу. Однак на другому шарі зона сприйнятливості вже дорівнює 5, оскільки нейрон другого шару агрегує три нейрона першого шару, кожен з яких має зону сприйнятливості 3. Зі збільшенням глибини зона сприйнятливості збільшується лінійно.

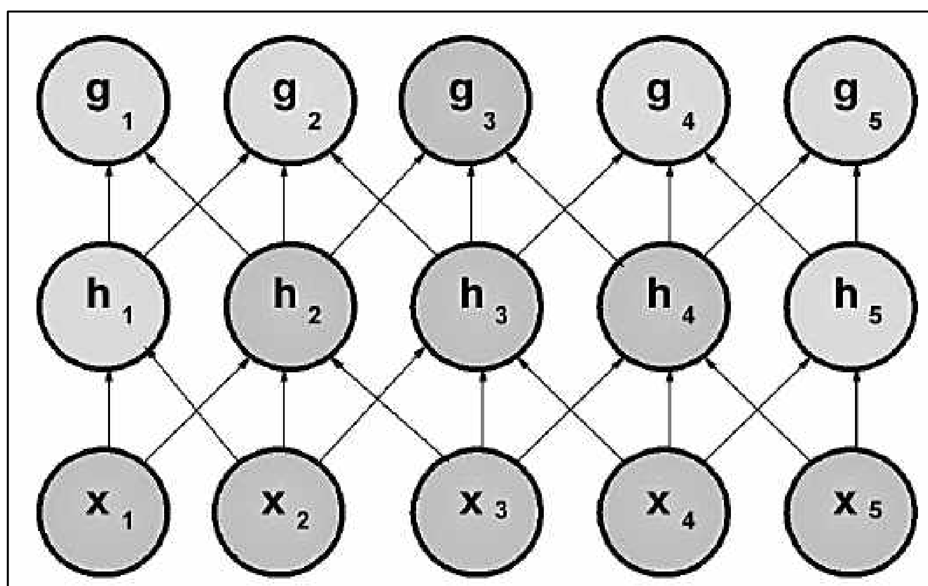


Рис. 2.5. Два сусідніх одновимірних згорткових шари

Спільні параметри. Нагадаємо, що в класичній теорії оброблення зображень одне й те ж ядро ковзає по всьому зображенню. Тут застосовується та сама ідея. Фіксується тільки розмір фільтра вагових коефіцієнтів для одного шару, і ці вагові коефіцієнти застосовуються до всіх нейронів у шарі. Це є рівносильним ковзанню одного й того ж ядра по всьому зображенню. Але може виникнути запитання, як ми можемо чогось навчитися з такою малою кількістю параметрів.

На рис. 2.6 кілька вагових коефіцієнтів вказують на один і той же параметр навчання, темними стрілками позначено однакові вагові коефіцієнти.

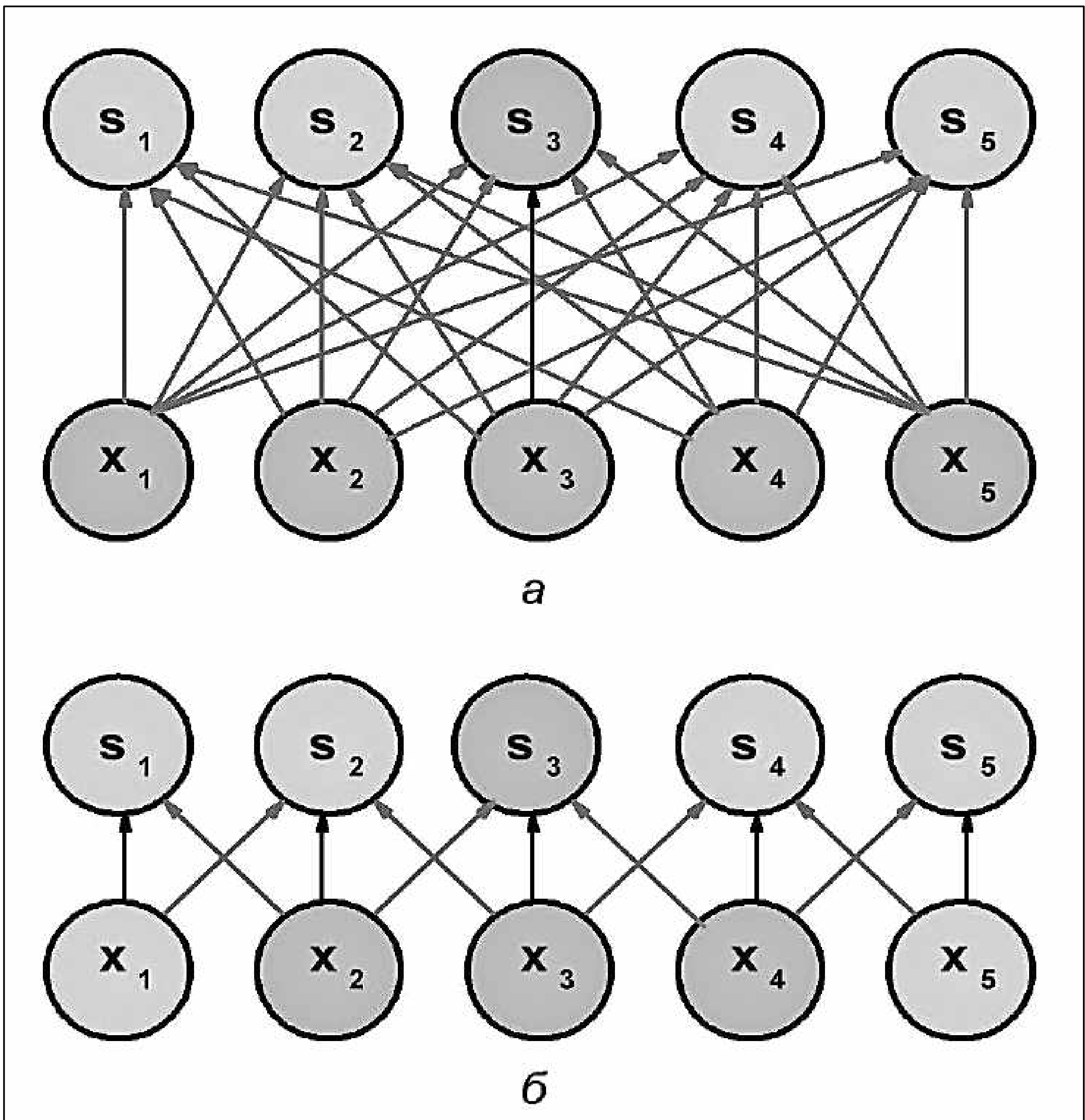


Рис. 2.6. Приклад поділу параметрів

Просторова структура. Відповідь на це запитання проста: будемо навчати кілька фільтрів в одному шарі! Фільтри розміщуються паралельно один до одного, формуючи таким чином новий вимір.

Пояснимо подану ідею на прикладі двовимірного RGB-зображення 227 × 227 (рис. 2.7). Маємо триканальне вхідне зображення. Це означає наявність тривимірних вхідних даних.

Рис. 2.7. Приклад тривимірних вхідних даних

Розглянемо розмірність каналів як глибину зображення (це не те ж саме, що глибина нейронних мереж, яка дорівнює кількості шарів мережі). Як визначити ядро для цього випадку?

Приклад двовимірного ядра, по суті, являє собою тривимірну матрицю з додатковим виміром глибини. Цей фільтр дає згортку з зображенням, тобто ковзає по зображенню в просторі, розраховуючи скалярні добутки (рис. 2.8).

Відповідь проста, хоча все ще не очевидна − зробимо ядро також тривимірним. Перші два виміри (ширина й висота ядра) залишаться колишніми, а третій вимір завжди дорівнює глибині вхідних даних.

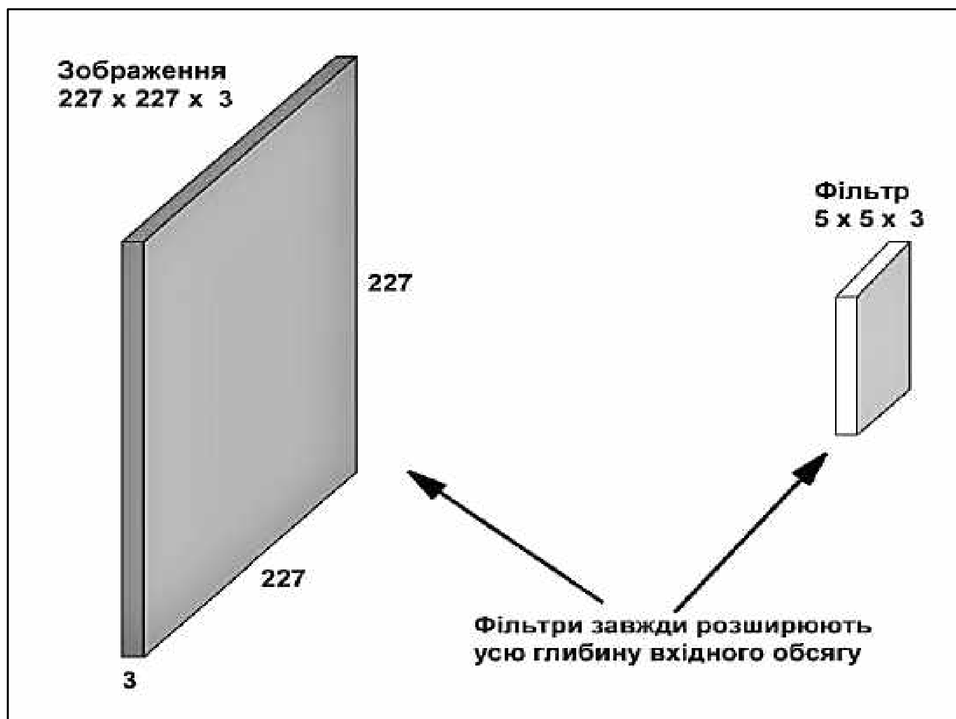


Рис. 2.8. Згортка фільтра із зображенням

Приклад просторового кроку згортки показано на рис. 2.9. Результатом скалярного добутку фільтра й невеликої ділянки зображення 5 × 5 × 3 (тобто 5 × 5 × 5 + 1 = 76 розмірність скалярного добутку + зсув) є одне число.

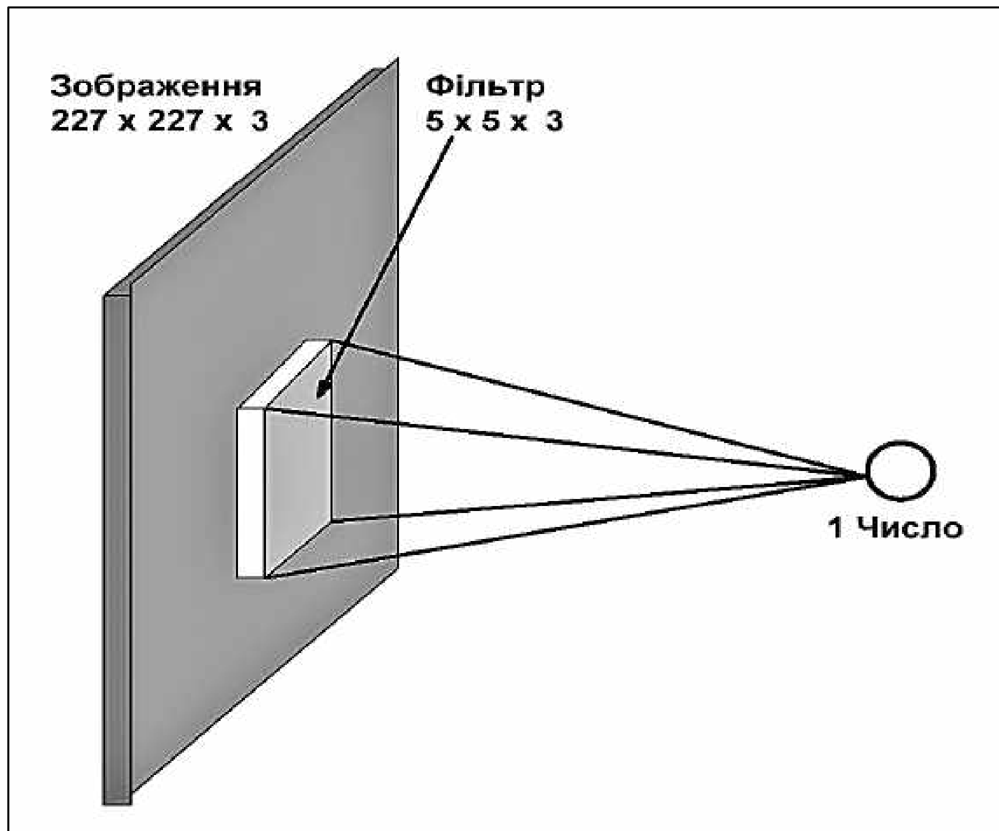


Рис. 2.9. Просторовий крок згортки

У цьому випадку вся ділянка 5 × 5 × 3 вихідного зображення трансформується в одне число, а саме тривимірне зображення буде трансформовано в карту ознак (активаційну карту). Карта являє собою набір нейронів, кожен з яких розраховує свою власну функцію з урахуванням двох основних принципів, розглянутих вище − локальної зв'язності (кожен нейрон зв'язаний лише з малою частиною вхідних даних) і поділу параметрів (усі нейрони використовують один і той же фільтр). В ідеалі ця карта буде такою ж, як у прикладі загальноприйнятої мережі, − вона зберігає результати згортки вхідного зображення і фільтра.

Карту ознак як результат згортки ядра з усіма просторовими положеннями зображено на рис. 2.10. Зазначимо, що глибина цієї карти дорівнює 1, оскільки використовувся тільки один фільтр. Але можна використовувати більше фільтрів, наприклад 6. Усі вони будуть взаємодіяти з одними й тими ж вхідними даними і будуть працювати незалежно один від одного. Подальший крок − поєднання цих карт ознак (рис. 2.11). Їх просторові розміри однакові, оскільки однакові розміри фільтрів. Таким чином, об'єднані карти ознак можна розглядати як нову тривимірну матрицю, розмірність глибини якої подана картами ознак від різних ядер. Тому канали RGB вхідного зображення є не що інше, як три вихідних карти ознак.

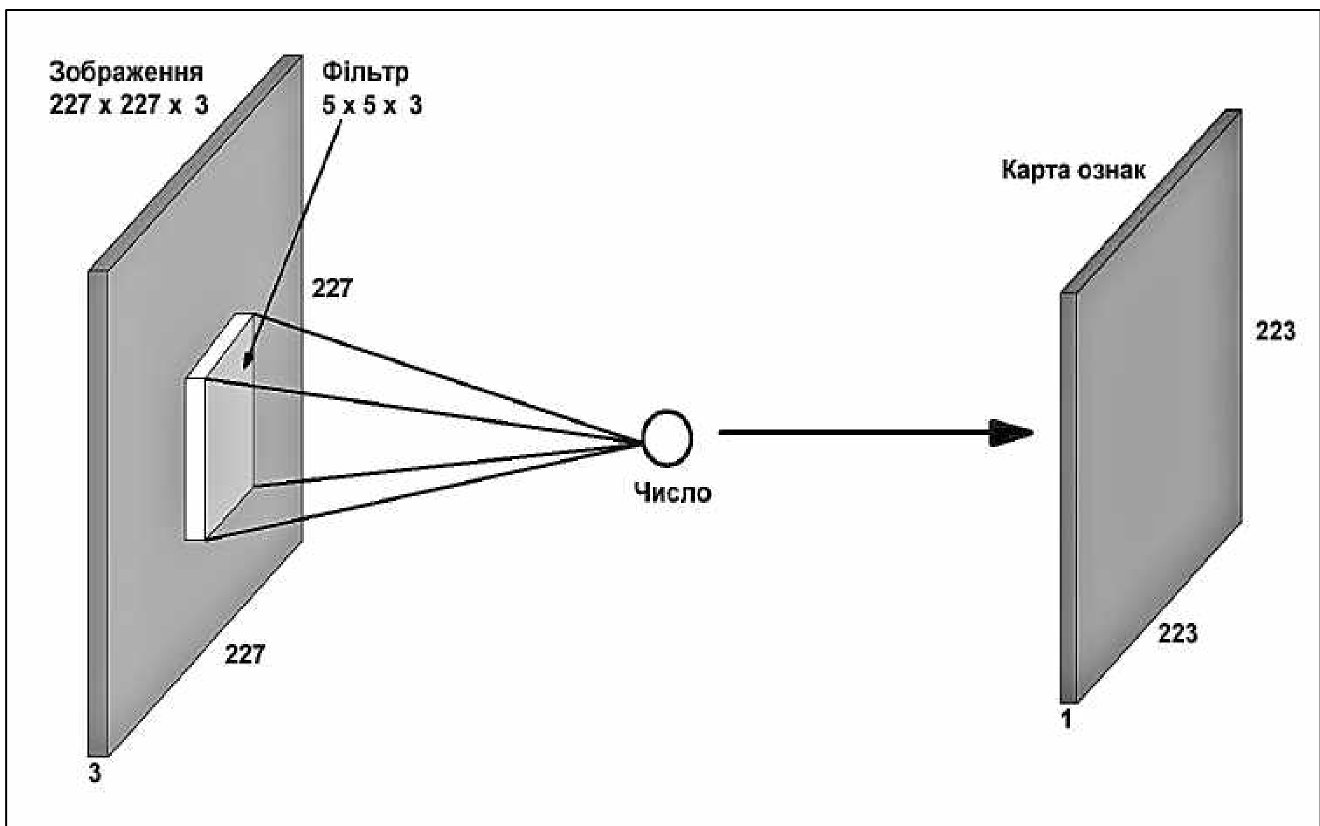


Рис. 2.10. Карта ознак як результат згортки ядра з усіма просторовими положеннями

Таке розуміння карт ознак та їх поєднання є дуже важливим, оскільки, усвідомивши це, можна розширити архітектуру мережі та встановлювати

згорткові шари один поверх іншого, збільшуючи тим самим зону сприйнятливості і збагачуючи свій класифікатор (рис. 2.12).

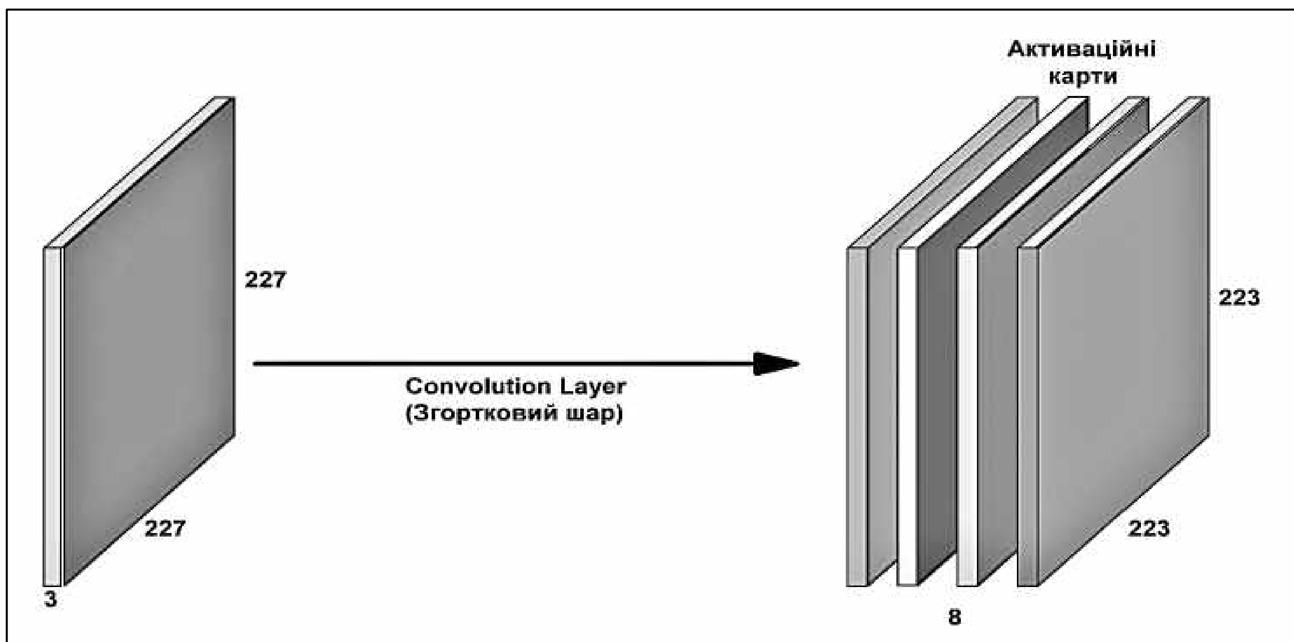


Рис. 2.11. Паралельне застосування декількох фільтрів до вхідного зображення і результівна множина активаційних карт

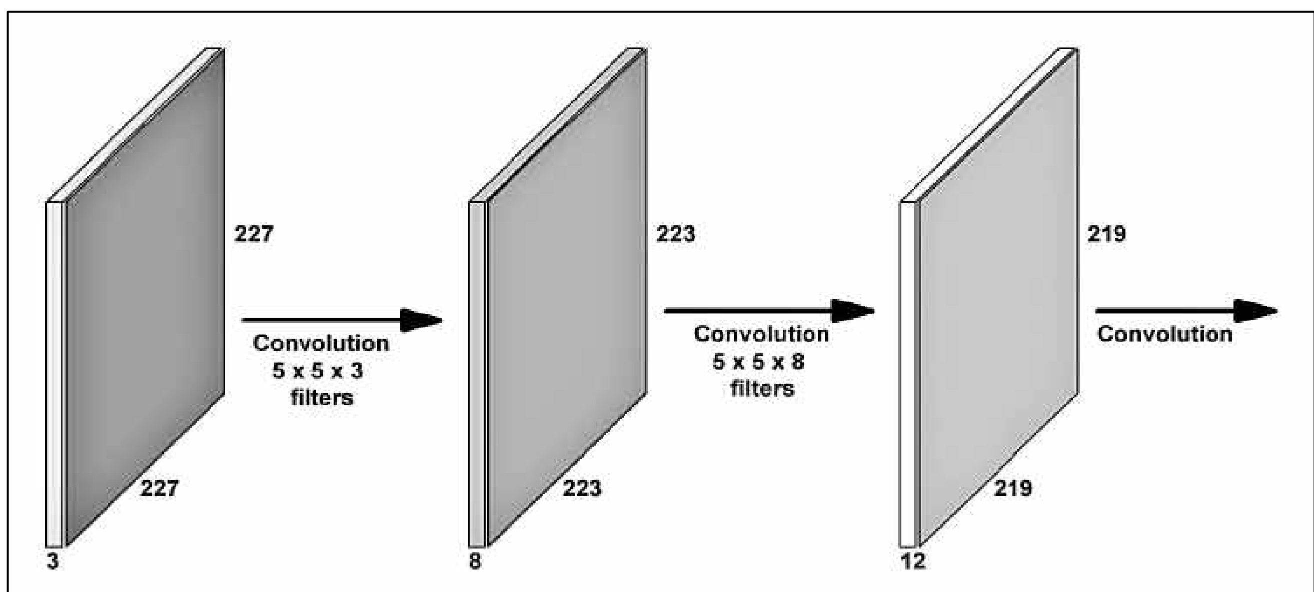


Рис. 2.12. Згорткові шари, установлені поверх один одного, у кожному шарі розміри фільтрів і їх кількість можуть бути різними

Тепер зрозуміло, що таке згорткова мережа. Основна мета цих шарів так само, як і при загальноприйнятому підході, − виявити значущі ознаки зображення. І якщо в першому шарі ці ознаки можуть бути дуже простими (наявність вертикальних/горизонтальних ліній), то зі збільшенням глибини мережі підвищується ступінь їх абстракції.

Шари підвибірки. Згорткові шари є основним структурним елементом CNN. Але існує ще одна важлива частина, що часто використовується, −

це шари підвибірки (рис. 2.13). У загальноприйнятій теорії оброблення зображень немає прямого аналога, але підвибірку можна розглядати як другий тип ядра. Що ж це таке?

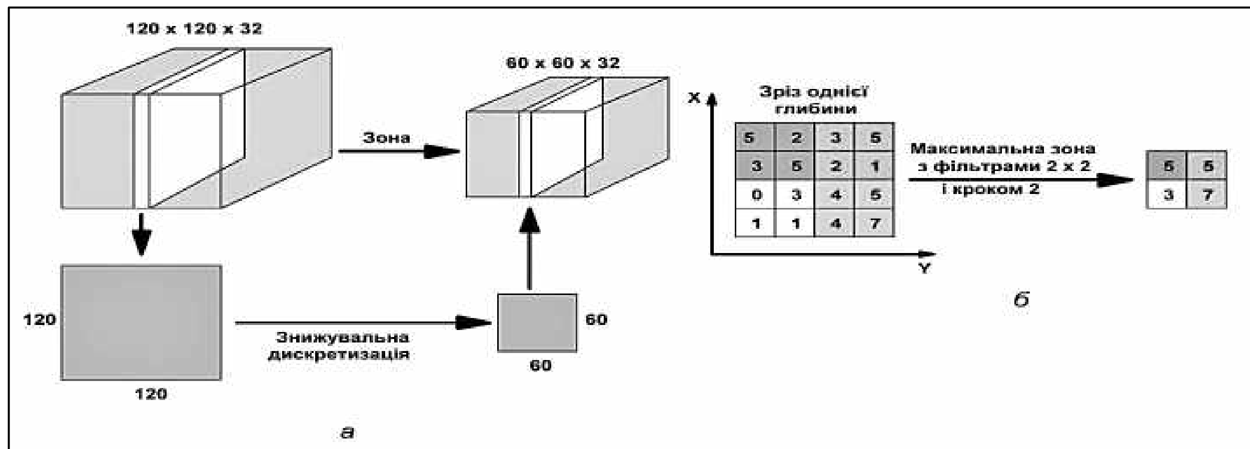


Рис. 2.13. Приклади підвибірки (*а*); підвибірка, що змінює просторові (але не канальні) розміри масивів даних (*б*)

Підвибірка фільтрує ділянку околу кожного пікселя вхідних даних певною агрегаційною функцією, наприклад, максимум, середнє і т. ін. Підвибірка, по суті, − те саме, що й згортка, але можливість спільного використання пікселів не обмежується скалярним добутком. Ще одна важлива відмінність − підвибірка працює тільки в просторовому вимірі. Характерною рисою шару підвибірки є те, що крок зазвичай дорівнює розміру фільтра (типовим значенням є 2).

Підвибірка має три основні мети:

- зменшення просторової розмірності, або субдискретизація, для зменшення кількості параметрів;
- збільшення зони сприйнятливості, оскільки завдяки нейронам підвибірки в наступних шарах акумулюється більше кроків вхідного сигналу;
- трансляційна інваріантність до невеликих неоднорідностей у положенні шаблонів у вхідному сигналі, оскільки під час розрахунку агрегаційних статистик невеликих околів вхідного сигналу підвибірка може ігнорувати невеликі просторові переміщення в ньому.

Щільні шари. Згорткові шари й шари підвибірки мають одну мету − генерація ознак зображення. Завершальним кроком є класифікація вхідного зображення на основі виявлених ознак. У CNN це роблять щільні шари на вершині мережі. Цю частину мережі називають класифікаційною. Вона може містити кілька шарів поверх один одного з повною зв’язністю, але зазвичай закінчується шаром класу `softmax`, активованим багатозмінною логістичною активаційною функцією, у якому кількість блоків дорівнює кількості класів. На виході цього шару знаходиться розподіл імовірностей по класах для вхідного об'єкта. Тепер зображення можна класифікувати, вибравши найбільш імовірний клас.

3. РЕСУРСИ ГЛИБОКОГО НАВЧАННЯ В БІБЛІОТЕЦІ OPENCV

Як уже зазначалося, глибоке навчання істотно перевершує традиційні підходи до додатків машинного навчання. Це причина, з якої архітектури глибокого навчання застосовуються в багатьох областях, включаючи комп'ютерний зір. Загальні додатки глибокого навчання поширюються на завдання автоматичного розпізнавання мови, розпізнавання зображень, біоінформатики тощо. У зв'язку зі зростаючими потребами в застосуванні згорткових нейромережних технологій прикладаються значні зусилля і для розвитку відповідних сервісних ресурсів. Прикладом цього може бути розширення функціональних можливостей відомої бібліотеки OpenCV, яка застосовується для розширення можливостей мови програмування Python.

Ідею ієрархічного узагальнення та керування наявними ресурсами підтримки нейронних мереж глибокого навчання було реалізовано в бібліотеці OpenCV шляхом створення пакета функцій та алгоритмів для керування параметрами мереж глибокого навчання. Для цього, починаючи з версії OpenCV 3.1, у бібліотеку було введено модуль глибоких нейронних мереж (DNN), який реалізує прямий зв'язок з глибокими мережами, попередньо навчений з використанням деяких популярних фреймворків глибокого навчання. Це, наприклад, **Caffe** (http://caffe.berkeleyvision.org/), **TensorFlow** (https://www.tensorflow.org/), **Torch**/**Pytorch** (https://www.tensorflow.org/). (http://torch.ch/), **Darknet** (https://pjreddie.com/darknet/) і моделі в форматі **ONNX** (https://onnx.ai/).

В OpenCV 3.3 статус модуля `Deep learning` було підвищено з репозиторію `opencv_contrib` до основного сховища (https://github.com/opencv /opencv/tree/master/modules/dnn). У цій версії було значно збільшено і швидкодію модуля DNN.

3.1. Загальні характеристики глибокого навчання

Глибоке навчання має суттєві відмінності від традиційних підходів до машинного навчання. Методи глибокого навчання перевершують машинне навчання в багатьох задачах комп'ютерного зору, але слід брати до уваги деякі міркування для вибору − коли і який метод застосовувати для виконання певної обчислювальної задачі. Ці міркування потрібно формулювати так:

1. Алгоритми глибокого навчання повинні мати високопродуктивну інфраструктуру для правильного навчання, на відміну від методів машинного навчання, які можуть працювати на машинах низького рівня.
2. При відсутності адекватної моделі процесів, що відбуваються в предметній області, методи глибокого навчання перевершують інші методи, оскільки потребують менше зусиль під час розроблення функцій. Проектування ознак можна визначити як процес застосування

знань предметної області для зниження складності даних. Методи глибокого навчання намагаються витягти високорівневі функції з даних, що робить глибоке навчання набагато більш просунутим, ніж традиційні підходи до машинного навчання. У глибокому навчанні завдання пошуку релевантних функцій є частиною алгоритму, і вона автоматизована за рахунок скорочення завдань самоаналізу й розроблення функцій для кожної проблеми.

3. І машинне, і глибоке навчання здатні обробляти масиви даних величезних розмірів. Однак методи машинного навчання мають набагато більший сенс при роботі з невеликими наборами даних. У цьому сенсі ключова відмінність між обома підходами полягає в їх продуктивності при збільшенні масштабу даних. Наприклад, при роботі з невеликими наборами даних алгоритмам глибокого навчання важко знайти закономірності в даних, і вони не працюють належним чином, оскільки їм потрібен великий обсяг даних для коригування своїх внутрішніх параметрів. Практичне правило − ураховувати, що глибоке навчання перевершує інші методи, якщо розмір даних є великим, і традиційні алгоритми машинного навчання є кращими, коли набір даних − невеликий.

На рис. 3.1 добре видно, наскільки продуктивнішим є підхід до вирішення завдання виявлення об'єкта методами глибокого навчання порівняно з класичним методом машинного навчання.

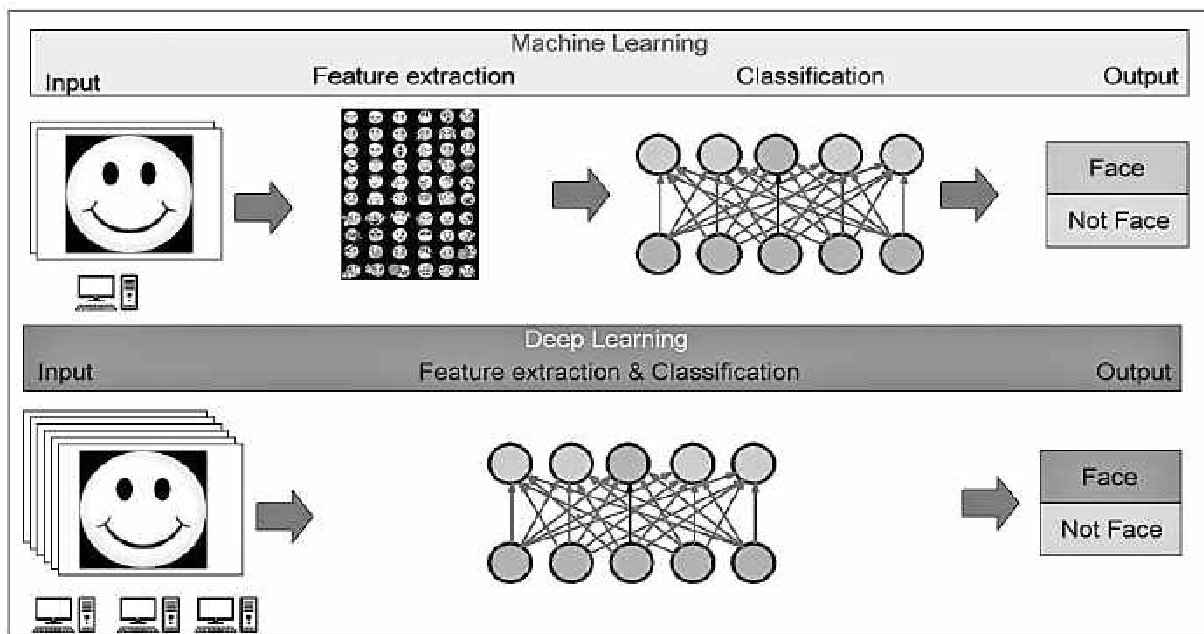


Рис. 3.1. Порівняння методів машинного і глибокого навчання

Зверніть увагу на такі відмінності між двома методами навчання − машинного й глибокого:

- обчислювальні ресурси (при глибокому навчанні використовуються високопродуктивні комп'ютери, а при машинному − малопродуктивні);

- розроблення ознак (при глибокому навчанні здійснюються вилучення й класифікація ознак об'єктів на одному етапі, а при машинному – на різних етапах);
- розміри наборів даних (при глибокому навчанні – великі/дуже великі набори даних порівняно з машинним навчанням, де використовуються середні/великі набори).

3.2. Функції блока DNN в OpenCV та їх основні властивості

Розглянемо, як застосувати деякі з цих архітектур як для виявлення об'єктів, так і для класифікації зображень, але спочатку варто розглянути деякі функції, які бібліотека OpenCV надає в модулі DNN.

Призначення функції `cv2.dnn.blobFromIma`. Ця функція використовується для виявлення, відстеження й розпізнавання облич з обчисленнями глибокого навчання. У скрипті `face_detection_opencv_dnn.py` детектор облич на основі глибокого навчання [26] застосовується для виявлення облич на зображеннях.

Перший крок – це завантаження попередньо навчених моделей. Це робиться таким чином:

```
net =
cv2.dnn.readNetFromCaffe("deploy.prototxt","res10_300x300_ssd_i
ter_140000_fp16.ca
```

Нагадуємо, що файл `deploy.prototxt` визначає архітектуру моделі, а файл `res10_300x300_ssd_iter_140000_fp16.caffemodel` містить ваги для фактичних шарів. Щоб виконати прямий прохід усієї мережі для обчислення вихідних даних, входом у мережу має бути великий двійковий об'єкт. Великий двійковий об'єкт можна розглядати як набір зображень, які було належним чином попередньо оброблено, щоб попросити мережу.

Це попереднє оброблення складається з декількох операцій – змінення розміру, обрізання, віднімання середніх значень, масштабування й заміна синього і червоного каналів.

Наприклад, у прикладі виявлення облич ми виконали таку команду:

```
# Load image:
image = cv2.imread("test_face_detection.jpg")
# Create 4-dimensional blob from image:
blob=cv2.dnn.blobFromImage(image,1.0,(300,300),[104.,117.,123.],
False,False)
```

У цьому випадку це означає, що ми хочемо запустити модель на зображеннях BGR з розміром 300 x 300, застосовуючи середнє віднімання (104, 117, 123) значень для синього, зеленого і червоного каналів відповідно. Це можна звести в таку таблицю:

Model	Scale	Size WxH	Mean subtraction	Channels order
OpenCV face detector	1.0	300 x 300	104, 177,123	BGR

На цьому етапі можна встановити BLOB-об'єкт як вхідні дані й отримати виявлення таким чином:

```
# Set the blob as input and obtain the detections:
net.setInput(blob)
detections = net.forward()
```

Розглянемо функції `cv2.dnn.blobFromImage()` і `cv2.dnn.blobFromImages()` більш детально. Для цього визначимо сигнатури обох функцій, а також скрипти `blob_from_image.py` і `blob_from_images.py`. Вони можуть бути корисні для розуміння цих функцій. У цих програмах також можна використовувати функцію OpenCV `cv2.dnn.imagesFromBlob ()`.

Запис `cv2.dnn.blobFromImage()` має такий вигляд:

```
Retval =
cv2.dnn.blobFromImage(image[,scalefactor[,size[,mean[,swapRB[,
crop[,ddepth]
```

Ця функція створює чотиривимірний об'єкт із зображення. Він додатково змінює розмір зображення до розміру й обрізає вхідне зображення від центра, віднімає середні значення, масштабує значення за коефіцієнтом масштабування і міняє місцями синій і червоний канали:

- image − вхідне зображення для попереднього оброблення;
- `scalefactor` − множник для значень зображення. Його можна використовувати для масштабування зображень. Значення за замовчуванням − 1.0, якщо масштабування не виконується;
- `size` − просторовий розмір вихідного зображення;
- `mean` − скаляр з середніми значеннями, узятими із зображення. Якщо виконується віднімання середнього, то значення мають бути (середнє `R`, середнє `G`, середнє `B`) при використанні `swapRB = True`;
- `swapRB` − цей прапор можна використовувати для перемикання каналів `R` і `B` у зображенні, установивши для нього значення `True`;
- `crop` − це прапор, який указує, буде картинка обрізуватися після змінення розміру чи ні;
- `depth` − глибина вихідного великого двійкового об'єкта. Можна вибрати `CV_32F` або `CV_8U`. Якщо `crop = False`, то змінення розміру зображення виконується без обрізання. В іншому випадку, якщо (`crop = True`), спочатку застосовується змінення розміру, а потім зображення обрізається від центра;

- значення за замовчуванням − scalefactor = 1.0, size = Size (), mean = Scalar (), swapRB = false, crop = false і ddepth = CV_32F

Запис cv.dnn.blobFromImages() має такий вигляд:

```
Retval =
cv.dnn.blobFromImages(images[,scalefactor[,size[,mean[,swapRB[,
crop[,ddepth]
```

Ця функція створює чотиривимірний об'єкт з різних фотографій. Він може виконати прямий прохід усієї мережі, щоб обчислити результат відразу декількох зображень. Як використовувати цю функцію належним чином, показано в такому коді:

```
# Create a list of images:
images = [image, image2]
# Call cv2.dnn.blobFromImages():
blob_images =
cv2.dnn.blobFromImages(images,1.0,(300,300),[104.,117.,123.],Fa
lse, False)
# Set the blob as input and obtain the detections:
net.setInput(blob_images)
detections = net.forward()
```

Далі показано функції cv2.dnn.blobFromImage() і cv2.dnn.blobFromImages(). Розглянемо коди програм blob_from_image.py і blob_from_images.py.

У програмі blob_from_image.py спочатку завантажується зображення BGR і створюється чотиривимірний великий двійковий об'єкт з використанням функції cv2.dnn.blobFromImage(). Можна перевірити, який вигляд має форма створеного великого двійкового об'єкта (1,3,300,300). Потім уводиться функція get_image_from_blob (), яку можна використовувати для виконання зворотних перетворень попереднього оброблення, щоб знову отримати вхідне зображення.

Код функції get_image_from_blob має такий вигляд:

```
def
get_image_from_blob(blob_img,scalefactor,dim,mean,swap_rb,mean_
added):
    """Returns image from blob assuming that the blob is from
only one image""
    images_from_blob = cv2.dnn.imagesFromBlob(blob_img)
    image_from_blob = np.reshape(images_from_blob[0],dim) /
scalefactor
    image_from_blob_mean = np.uint8(image_from_blob)
    image_from_blob = image_from_blob_mean + np.uint8(mean)

    if mean_added is True:
```

```
        if swap_rb:
            image_from_blob = image_from_blob[:, :, ::-1]
        return image_from_blob
    else:
        if swap_rb:
            image_from_blob_mean = image_from_blob_mean[:, :,
::-1]
        return image_from_blob_mean
```

У цьому сценарії використовується функція для отримання різних зображень з великого двійкового об'єкта, як показано в такому фрагменті коду:

```
Load image:
image = cv2.imread("face_test.jpg")

# Call cv2.dnn.blobFromImage():
blob_image =
cv2.dnn.blobFromImage(image,1.0,(300,300),[104.,117.,123.],Fals
e, False

# The shape of the blob_image will be (1,3,300,300):
print(blob_image.shape)
# Get different images from the blob:
img_from_blob =
get_image_from_blob(blob_image,1.0,(300,300,3),[104.,117.,123.]
,False, True
img_from_blob_swap =
get_image_from_blob(blob_image,1.0,(300,300,3),[104.,117.,123.
img_from_blob_mean =
get_image_from_blob(blob_image,1.0,(300,300,3),[104.,117.,123.
img_from_blob_mean_swap =
get_image_from_blob(blob_image,1.0,(300,300,3),104.,117
```

Створені зображення опишемо так:

- зображення `img_from_blo`b відповідає вихідному зображенню `BGR` з розміром, зміненим до (`300, 300`);
- зображення `img_from_blob_swap` відповідає вихідному зображенню `BGR` з розміром, зміненим до (`300, 300`), а синій і червоний канали помінялися місцями;
- зображення `img_from_blob_mean` відповідає вихідному зображенню `BGR` з розміром, зміненим до (`300, 300`), де скаляр із середніми значеннями не додають до вихідного зображення;
- зображення `img_from_blob_mean_swap` відповідає вихідному зображенню `BGR розміром`, зміненим до (`300,300`), де скаляр із середніми значеннями не додано до зображення, а синій і червоний канали помінялися місцями.

Результат роботи цього коду можна побачити на рис. 3.2.

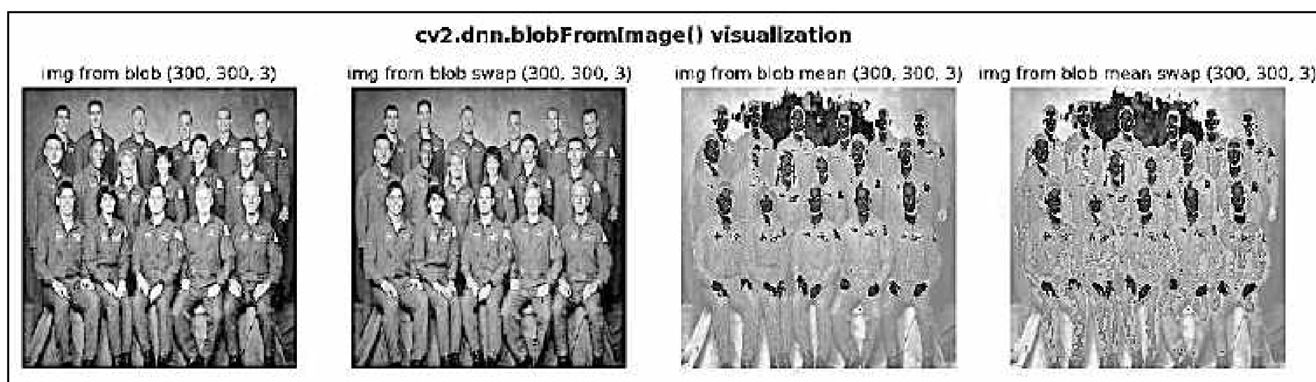


Рис. 3.2. Результати перетворення вихідного зображення

На цьому скріншоті бачимо чотири отриманих зображення: `img_from_blob`, `img_from_blob_swap`, `img_from_blob_mean` і `img_from_blob_mean_swap`.

У програмі `blob_from_images.py` спочатку завантажуються два зображення `BGR`, створюється чотиривимірний великий двійковий об'єкт з використанням функції `cv2.dnn.blobFromImages()`. Можна перевірити форму створеного великого двійкового об'єкта (`2, 3, 300, 300`). Потім викликається функція `get_images_from_blob()`, яку можна використовувати для виконання зворотних перетворень попереднього оброблення, щоб знову отримати вхідні зображення.

Код функції `get_images_from_blob` є таким:

```
def
get_images_from_blob(blob_imgs,scalefactor,dim,mean,swap_rb,mean
_added):
    """Returns images from blob"""

    images_from_blob = cv2.dnn.imagesFromBlob(blob_imgs)
    imgs = []
    for image_blob in images_from_blob:
    image_from_blob = np.reshape(image_blob, dim) / scalefactor
    image_from_blob_mean = np.uint8(image_from_blob)
    image_from_blob = image_from_blob_mean + np.uint8(mean)
    if mean_added is True:
        if swap_rb:
            image_from_blob = image_from_blob[:, :, ::-1]
        imgs.append(image_from_blob)
    else:
        if swap_rb:
            image_from_blob_mean = image_from_blob_mean[:, :,
::-1]
        imgs.append(image_from_blob_mean)
return imgs
```

Як було показано раніше, функція `get_images_from_blob()` повертає зображення з великого двійкового об'єкта, використовуючи функцію OpenCV `cv2.dnn.imagesFromBlob ()`. У поточному сценарії ця функція використовується для отримання різних зображень з великого двійкового об'єкта:

```
# Load images and get the list of images:
image = cv2.imread("face_test.jpg")
image2 = cv2.imread("face_test_2.jpg")
images = [image, image2]

# Call cv2.dnn.blobFromImages():
blob_images =
cv2.dnn.blobFromImages(images,1.0,(300,300),[104.,117.,123.],Fa
lse, False
# The shape of the blob_image will be (2,3,300,300):
print(blob_images.shape)

# Get different images from the blob:
imgs_from_blob =
get_images_from_blob(blob_images,1.0,(300,300,3),[104.,117.,123
.
imgs_from_blob_swap =
get_images_from_blob(blob_images,1.0,(300,300,3),[104.,117.,
imgs_from_blob_mean =
get_images_from_blob(blob_images,1.0,(300,300,3),[104.,117.,
imgs_from_blob_mean_swap =
get_images_from_blob(blob_images,1.0,(300,300,3),[104.,
```

У попередньому коді використовувалася функція `get_images_from_blob()` для отримання різних зображень з великого двійкового об'єкта. Створені зображення (рис. 3.3) мають такі властивості:

- зображення `imgs_from_blob` відповідають вихідним зображенням `BGR` розміром (`300,300`);
- зображення `imgs_from_blob_swap` відповідають вихідним зображенням `BGR`, розмір яких змінений до (`300,300`), а синій і червоний канали помінялися місцями;
- зображення `imgs_from_blob_mean` відповідають вихідним зображенням `BGR` з розміром, зміненим до (300, 300), де скаляр із середніми значеннями не додано до зображення;
- зображення `imgs_from_blob_mean_swap` відповідають вихідним зображенням `BGR` з розміром, зміненим до (`300,300`), де скаляр із середніми значеннями не додано до зображення, а синій і червоний канали помінялися місцями.

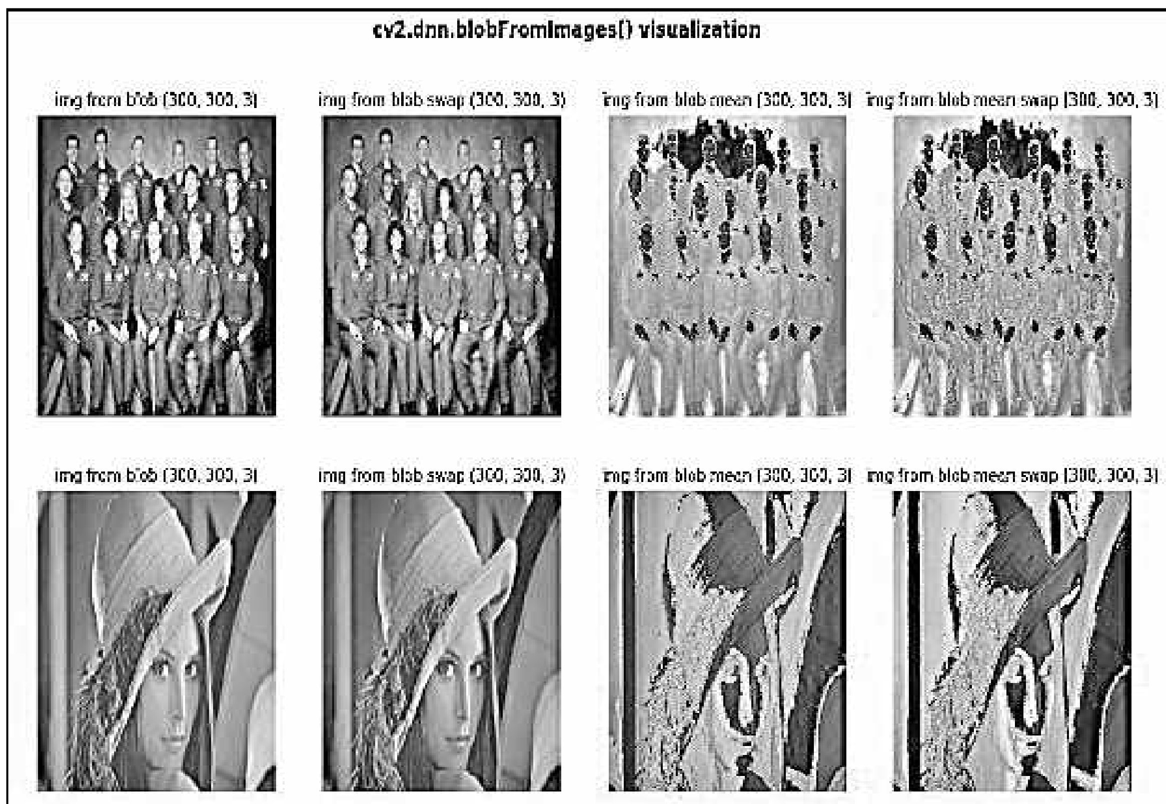


Рис. 3.3. Результат дії функції get_images_from_blob

Існує ще одна характерна особливість функцій `cv2.dnn.blobFromImage()` і `cv2.dnn.blobFromImages()`, яка полягає в тому, що параметр кадрування вказує, обрізано зображення чи ні. У разі кадрування зображення обрізається від центра, як показано на рис. 3.4.

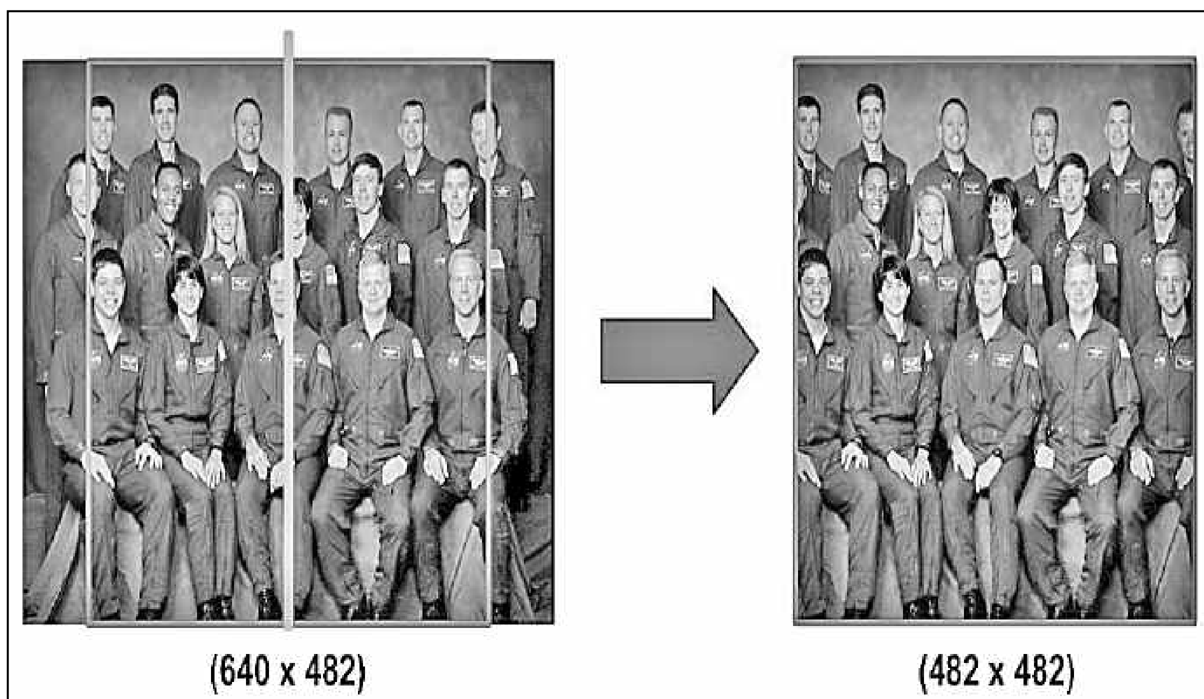


Рис. 3.4. Зображення, обрізане від центра

Очевидно, що кадрування виконується від центра зображення, позначеного жовтою лінією. Щоб відтворити кадрування, яке OpenCV виконує всередині функцій `cv2.dnn.blobFromImage()` і `cv2.dnn.blobFromImages()`, закодуємо функцію `get_cropped_img()` таким чином:

```
def get_cropped_img(img):
    """Returns the cropped image"""

    # calculate size of resulting image:
    size = min(img.shape[1], img.shape[0])

    # calculate x1, and y1
    x1 = int(0.5 * (img.shape[1] - size))
    y1 = int(0.5 * (img.shape[0] - size))

    # crop and return the image
    return img[y1:(y1 + size), x1:(x1 + size)]
```

Очевидно, що розмір обрізаного зображення залежить від мінімального розміру початкового зображення. Отже, у попередньому прикладі обрізане зображення матиме розмір 482 x 482 (див. рис. 3.4).

У програмі `blob_from_images_cropping.py` помітний ефект обрізання. Реплікація процедури обрізання міститься в функції `get_cropped_img()`:

```
# Load images and get the list of images:
image = cv2.imread("face_test.jpg")
image2 = cv2.imread("face_test_2.jpg")
images = [image, image2]

# To see how cropping works, we are going to perform the crop-
ping formulation that
# both blobFromImage() and blobFromImages() perform applying it
to one of the input images:
cropped_img = get_cropped_img(image)
# cv2.imwrite("cropped_img.jpg", cropped_img)

# Call cv2.dnn.blobFromImages():
blob_images =
cv2.dnn.blobFromImages(images,1.0,(300,300),[104.,117.,123.],Fa
lse,False
blob_blob_images_cropped =
cv2.dnn.blobFromImages(images,1.0,(300,300),[104.,117.,

# Get different images from the blob:
imgs_from_blob =
get_images_from_blob(blob_images,1.0,(300,300,3),[104.,117.,
123.
imgs_from_blob_cropped =
get_images_from_blob(blob_blob_images_cropped,1.0,(300, 300,
```

Результат роботи скрипта `blob_from_images_cropping.py` можна побачити на рис. 3.5.

Ефект кадрування видно в двох завантажених зображеннях. Також можна помітити (а це дуже важливо), що співвідношення сторін зберігається.

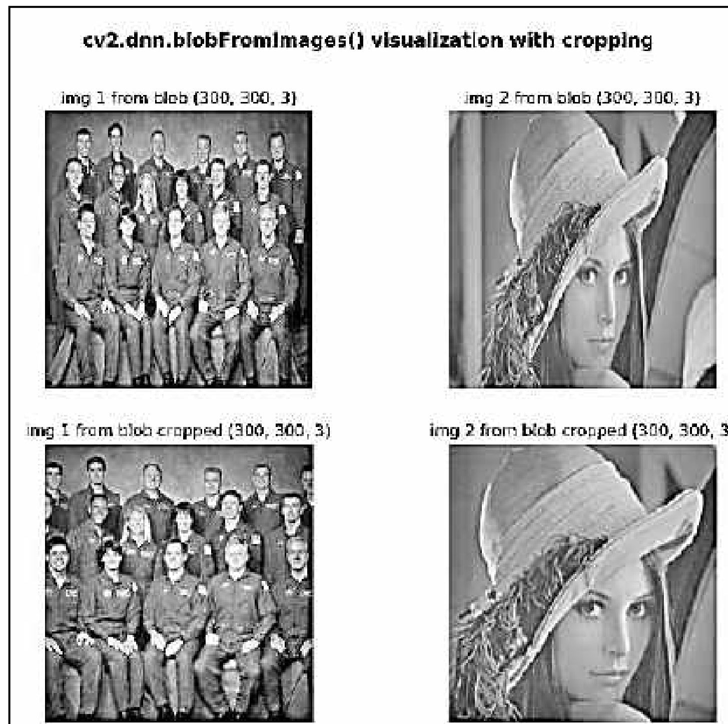


Рис. 3.5. Результакт обрізання зображення

3.3. Класифікація з використанням моделей, попередньо навчених в OpenCV

У цьому розділі розглянемо кілька прикладів того, як виконувати класифікацію зображень з використанням різних попередньо навчених моделей. Зверніть увагу, що можна отримати час виведення, використовуючи метод `net.getPerfProfile()` таким чином:

```
# Feed the input blob to the network, perform inference and get
the output:
net.setInput(blob)
preds = net.forward()
# Get inference time:
t, _ = net.getPerfProfile()
print('Inference time:
%.2fms'%(t*1000.0/cv2.getTickFrequency()))
```

Як бачимо, метод `net.getPerfProfile()` викликається після виконання рішення. Він повертає загальний час виведення і таймінги для шарів. Таким чином, можна порівняти час виведення, використовуючи різні архітектури глибокого навчання.

Розглянемо основні архітектури класифікації глибокого навчання, починаючи з архітектури **AlexNet**.

Архітектура AlexNet. Класифікація зображень з використанням модуля OpenCV **DNN** на базі попередньо навчених моделей **AlexNet** і **Caffe** виконується в програмі `image_classification_ opencv_alexnet_ caffe.py`:

- перший крок − завантажити імена класів;
- другий крок − завантажити серіалізовані моделі Caffe з диска;
- третій крок − завантажити вхідне зображення для класифікації;
- четвертий крок − створити великий двійковий об'єкт розміром (227, 2327) з середніми значеннями обчислення (104, 117, 123);
- п'ятий крок − передати вхідний BLOB-об'єкт у мережу, виконати логічний висновок та отримати вихідні дані;
- шостий крок − отримати 10 індексів з найбільшою ймовірністю (у порядку убування).

Таким чином, індекс з найбільшою ймовірністю (верхній прогноз) буде першим. Нарешті, нарисуємо клас і ймовірність, пов'язані з верхнім прогнозом на зображенні. Результат цього сценарію можна побачити на скріншоті, показаному на рис. 3.6.

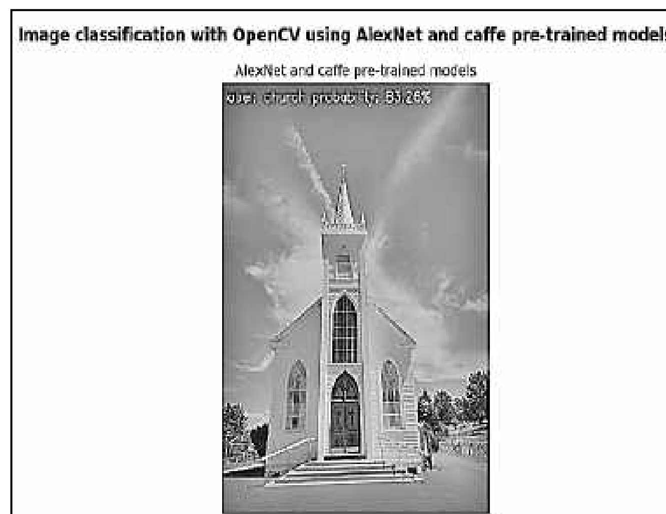


Рис. 3.6. Класифікація зображень з використанням архітектури AlexNet

Найбільш імовірний прогноз відповідає зображенню церкви з імовірністю 0,8325679898.

Ось десять кращих прогнозів:

1. Мітка: церква, імовірність 0,8325679898.
2. Мітка: монастир, імовірність 0,043678388.
3. Мітка: мечеть, імовірність 0,03827961534.
4. Етикетка: дзвіночок, імовірність 0,02479489893.
5. Мітка: маяк, імовірність 0,01249620412.
6. Мітка: купол, імовірність 0,01223050058.
7. Мітка: ракета, імовірність 0,006323920097.

8. Мітка: снаряд, імовірність 0,005275635514.
9. Мітка: палац, імовірність 0,004289720673.
10. Мітка: замок, імовірність 0,003241452388.

Також слід зазначити, що ми виконуємо таке під час рисування класу та імовірності:

```
text = "label: {} probabil-
ity:{:.2f}%".format(classes[indexes[0]],preds[0][indexes[0]] *
print(text)
y0, dy = 30, 30
for i, line in enumerate(text.split('\n')):
    y = y0 + i * dy
   cv2.putText(image,line,(5,y),cv2.FONT_HERSHEY_SIMPLEX,0.8,(0,
255,255),2)
```

Таким чином, текст можна розділити й нарисувати на різних лініях зображення. Наприклад, якщо виконаємо наступний код, то текст буде нарисований у два рядки:

```
    text = "la-bel: {} \ nprobability: {:. 2f}%". format
(classes [indexes [0]], preds [0] [indexes [0]
```

Слід зазначити, що файл bvlc_alexnet.caffemodel не включено в репозиторій, оскільки він перевищує обмеження на розмір файлу GitHub у 100,00 МБ. Завантажити його можна з [27]. Отже, перед запуском сценарію необхідно завантажити файл `bvlc_alexnet.caffemodel`.

Архітектура GoogLeNet. Як і в попередньому сценарії, класифікація зображень з використанням попередньо навчених моделей **GoogLeNet** і **Caffe** виконується програмою `image_classification_ opencv_googlenet _caffe.py`. Результат цієї роботи показано на рис. 3.7.

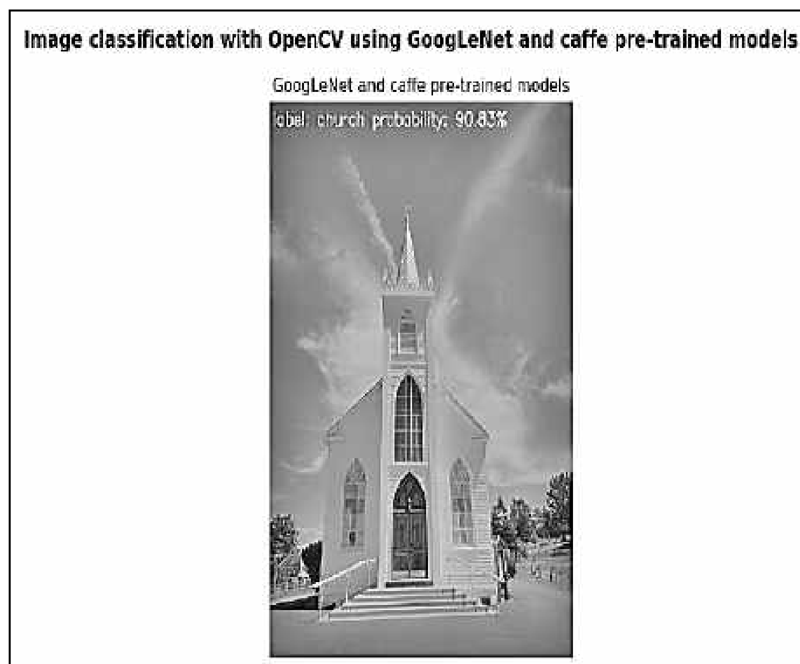


Рис. 3.7. Класифікація зображень з використанням архітектури **GoogLeNet**

Видно, що при архітектурі GoogLeNet імовірність правильного прогнозу збільшилася з 0,8325679898 до 0,9082632661 порівняно з архітектурою **AlexNet**.

Імовірності десяти кращих прогнозів:

1. Мітка: церква, імовірність 0,9082632661.
2. Етикетка: дзвіночок, імовірність 0,06350905448.
3. Мітка: монастир, імовірність 0,02046923898.
4. Мітка: купол, імовірність 0,002624791814.
5. Мітка: мечеть, імовірність 0,001077500987.
6. Етикетка: фонтан, імовірність 0,001011475339.
7. Мітка: палац, імовірність 0,0007750992081.
8. Мітка: замок, імовірність 0,0002349214483.
9. Етикетка: п'єдестал, ймовірність 0,0002306570677.
10. Етикетка: аналоговий годинник, імовірність 0,0002107089822.

Архітектура ResNet-50. У цьому випадку для класифікації зображень використано програму (`image_classification_opencv_restnet_50_caffe.py`) з попередньо навченими моделями **Caffe**. Результат показано на рис. 3.8.

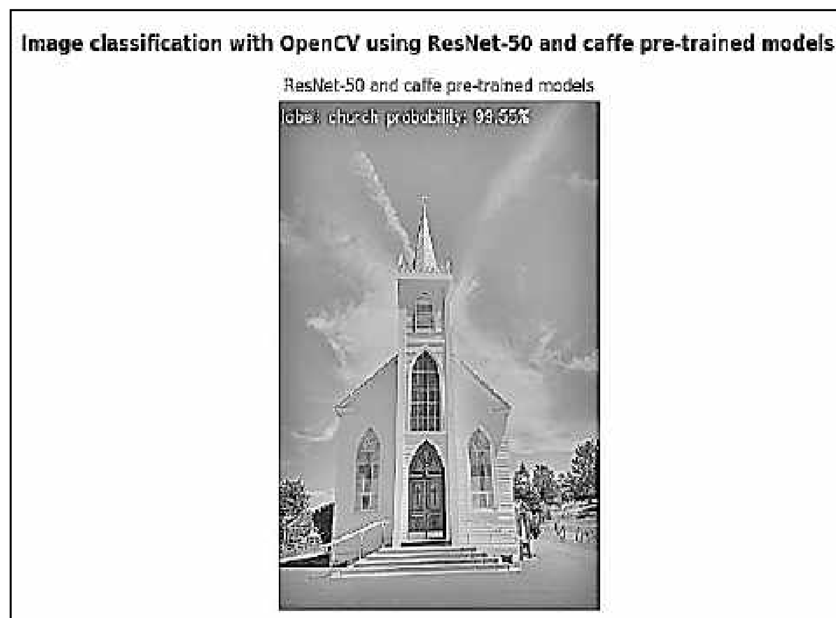


Рис. 3.8. Класифікація зображень з використанням архітектури **ResNet-50**

При архітектурі **ResNet-50** імовірність правильного прогнозу ще збільшилася порівняно з розглянутими архітектурами й становить 0,9955400825.

Ось ймовірності десяти кращих прогнозів:

1. Мітка: церква, імовірність 0,9955400825.
2. Мітка: купол, імовірність 0,002429900225.
3. Етикетка: дзвіночок, імовірність 0,0007424423238.
4. Мітка: монастир, імовірність 0,0003768313909.
5. Етикетка: штахетник, імовірність 0,0003282549733.

6. Мітка: мечеть, імовірність 0,000258318265.
7. Мітка: будинок на колесах, імовірність 0,0001083607058.
8. Мітка: стілець, імовірність 2,96174203e-05.
9. Мітка: палац, імовірність 2,621001659e-05.
10. Мітка: маяк, імовірність 2,02897063e-05.

3.4. Приклади виявлення об'єктів з використанням моделей, попередньо навчених в OpenCV

Розглянемо кілька прикладів того, як виявляти об'єкти з використанням різних попередньо навчених моделей. Поставлено завдання − виявити екземпляри семантичних об'єктів певних класів (наприклад, кішок, автомобілів і людей) у зображеннях або відео.

***Архітектура MobileNet з фреймворком SSD*.** **MobileNets-SSD** можна розглядати як ефективні згорткові нейронні мережі для додатків мобільного зору.

MobileNet-SSD було навчено на наборі даних COCO і точно налаштовано на PASCAL VOC, при цьому правильне виявлення становило 72,27 % (рис. 3.9). При точному налаштуванні на PASCAL VOC можна виявити 20 класів об'єктів:

- людина − людина;
- тварина − птах, кішка, корова, собака, кінь і вівця;
- транспортний засіб − літак, велосипед, човен, автобус, автомобіль, мотоцикл і поїзд;
- у приміщенні − пляшка, стілець, обідній стіл, рослина в горщику, диван і телевізор/монітор.

У програмі `object_detection_opencv_mobilenet_caffe.py` визначатиме об'єкти з допомогою модуля OpenCV **DNN** з використанням попередньо навчених моделей **MobileNet-SSD** і **Caffe**.

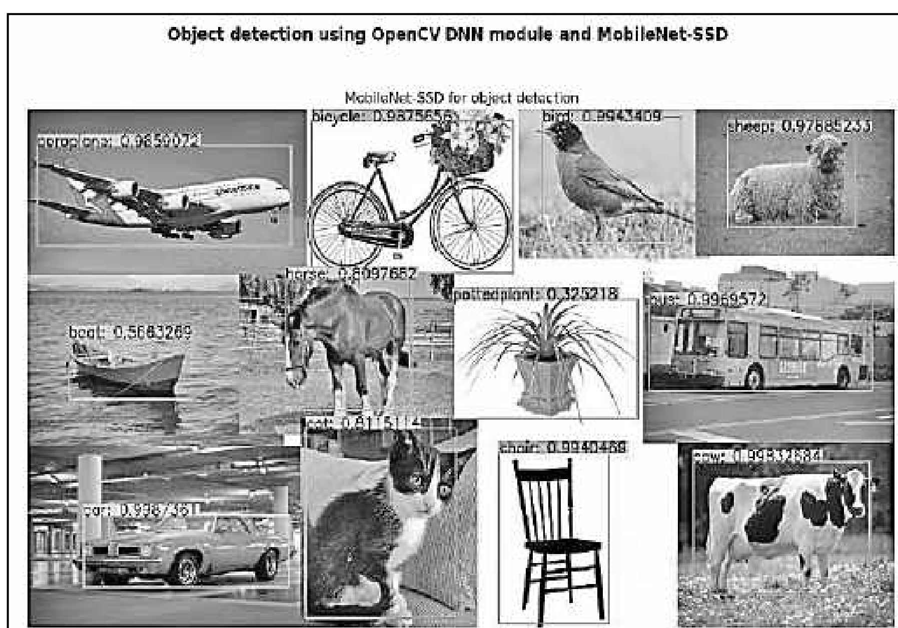


Рис. 3.9. Виявлення об'єктів з використанням архітектури **MobileNet-SSD**

Результат роботи цієї програми показано на рис. 3.9. Видно, що всі об'єкти були правильно виявлені з високою точністю.

Архітектура YOLO. У програмі `object_detection_opencv_yolo_darknet.py` виявлення об'єктів здійснюється з допомогою **YOLO v3**. Така архітектура використовує кілька прийомів для поліпшення навчання й підвищення продуктивності, включаючи багатомасштабні прогнози і поліпшений класифікатор магістралі. Результат роботи цієї програми показано на рис. 3.10.

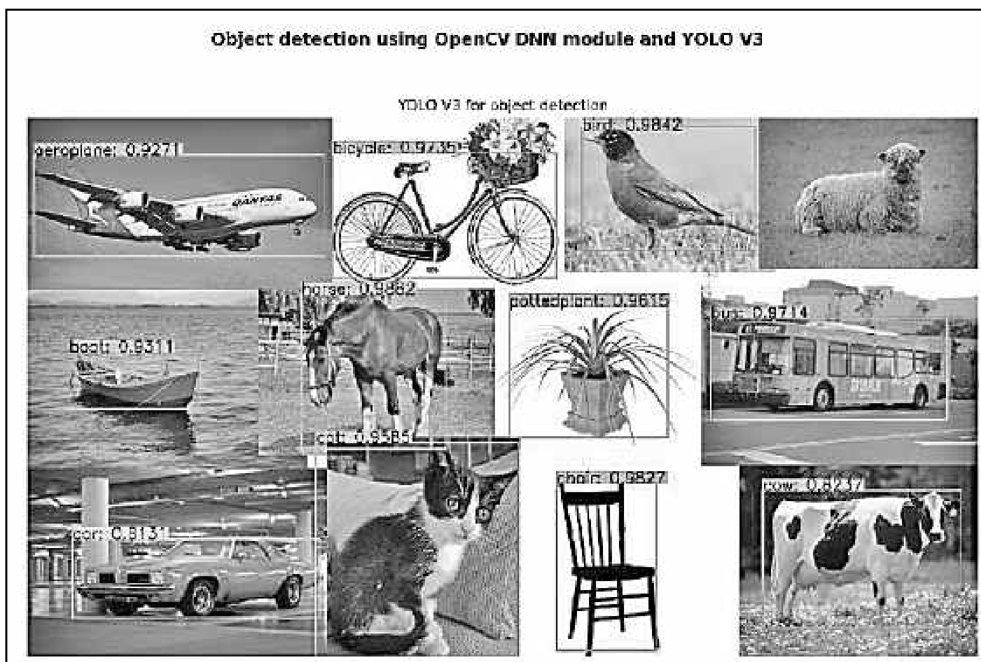


Рис. 3.10. Виявлення об'єктів з використанням архітектури **YOLO v3**

Видно, що всі об'єкти, крім одного (вівці), були правильно виявлені з високою точністю. Однак імовірність правильного виявлення об'єктів є дещо меншою порівняно з використанням архітектури **MobileNet-SSD**.

Перед запуском скрипта необхідно завантажити файл `yolov3.weights`. Цей файл не включено в репозиторій, оскільки він перевищує обмеження GitHub на розмір файлу в 100,00 МБ. Його можна завантажити з [28].

3.5. Бібліотека TensorFlow

TensorFlow − це програмна платформа з відкритим вихідним кодом для машинного й глибокого навчання, розроблена командою Google Brain для внутрішнього використання. Пізніше **TensorFlow** було випущено під ліцензією Apache 2015 року. У цьому підрозділі розглянемо кілька прикладів, щоб ознайомитися з бібліотекою **TensorFlow**.

Вступні приклади використання бібліотеки TensorFlow. Бібліотека **TensorFlow** подає обчислення, які необхідно виконати, зв'язуючи операції в граф обчислень. Після створення цього графа обчислень можна відкрити сеанс **TensorFlow** і виконати граф обчислень, щоб отримати ре-

зультати. Цю процедуру можна побачити в програмі `tensorflow_basic_op.py`, яка виконує операцію множення, визначену всередині графа обчислень таким чином:

```
# path to the folder that we want to save the logs for
Tensorboard
logs_path = "./logs"

# Define placeholders:
X_1 = tf.placeholder(tf.int16, name="X_1")
X_2 = tf.placeholder(tf.int16, name="X_2")

# Define a multiplication operation:
multiply = tf.multiply(X_1, X_2, name="my_multiplication")
```

Значення для наповнювачів надаються при запуску графіка в сеансі, як це показано в такому фрагменті коду:

```
# Start the session and run the operation with different in-
puts:
with tf.Session() as sess:
   summary_writer = tf.summary.FileWriter(logs_path,
sess.graph)

print("2 x 3 =
{}".format(sess.run(multiply,feed_dict={X_1:2,X_2: 3})))
print("[2,3]x[3,4] =
{}".format(sess.run(multiply,feed_dict={X_1:[2,3], X_2: 3})))
```

Обчислювальний граф параметризовано для доступу до зовнішніх входів − заповнювачів. В одному сеансі виконуємо два множення з різними входами. Оскільки обчислювальний граф є ключовим моментом у **TensorFlow**, візуалізація графа може допомогти як зрозуміти, так і налаштувати їх з допомогою **TensorBoard** і програмного забезпечення для візуалізації, яке поставляється з будь-якої стандартної установки **TensorFlow**. Щоб візуалізувати граф обчислень з допомогою **TensorBoard**, необхідно записати файли журналу програми з допомогою `tf.summary.FileWriter()`, як показано раніше. Якщо виконати цей сценарій, то каталог журналів буде створений у тому ж місці, де виконувався цей сценарій. Щоб запустити `TensorBoard`, необхідно виконати такий код:

```
$ Tensorboard --logdir = "./ logs"
```

Це згенерує посилання (http://localhost:6006/), яке можна ввести в свій браузер і побачити сторінку **TensorBoard** (рис. 3.11).

Крім того, оскільки графи **TensorFlow** можуть мати багато тисяч вузлів, для спрощення візуалізації можна створювати області, а **TensorBoard** використовує цю інформацію для визначення ієрархії вузлів у графі. Цю ідею показано в програмі `tensorflow_basic_ops_scope.py`, де визначаються дві операції (додавання й множення) усередині області дії:

```
with tf.name_scope('Operations'):
    addition = tf.add(X_1, X_2, name="my_addition")
    multiply = tf.multiply(X_1, X_2, name="my_multiplication")
```

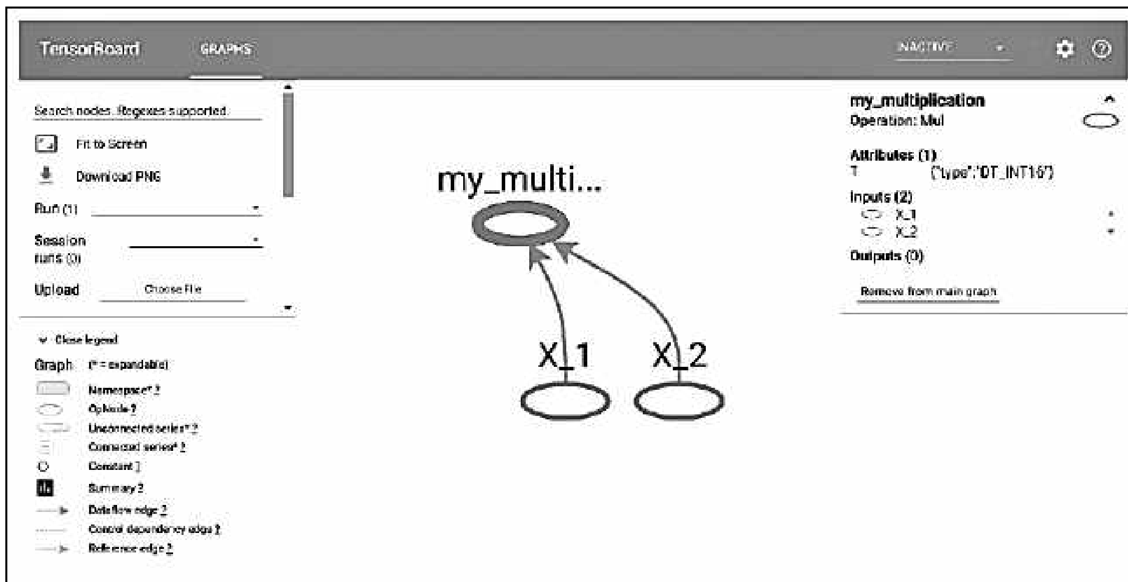


Рис. 3.11. Сторінка **TensorBoard**

Виконавши сценарій і повторивши попередні кроки, можна побачити обчислювальний граф, показаний у **TensorBoard** (рис. 3.12).

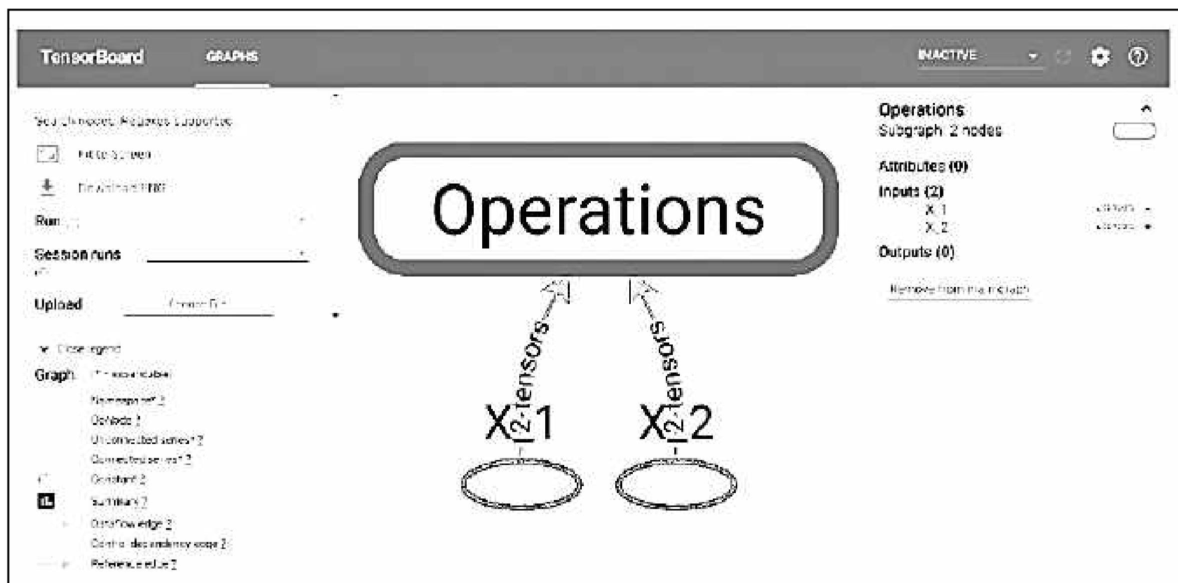


Рис. 3.12. Обчислювальний граф Operation

Зверніть увагу, що в програмах можна використовувати константи (`tf.Constant`) і змінні (`tf.Variable`). Відмінність між `tf.Variable` і `tf.placeholder` полягає в часі, коли значення передаються. Як видно з попередніх прикладів, з `tf.placeholder` не потрібно вказувати початкове значення, а ці значення вказуються під час виконання з аргументом `feed_dict` усередині сеансу. З іншого боку, при використанні змінної `tf.Variable` потрібно вказати початкове значення при її оголошенні.

Заповнювач − це просто змінна, якій згодом будуть присвоєні дані. При навчанні/тестуванні алгоритму заповнювачі зазвичай використовуються для подання системи адаптації тестування в граф обчислень.

3.6. Лінійна регресія в TensorFlow

Як приклад розрахуємо лінійну регресію, використовуючи **TensorFlow**. Це допоможе зрозуміти додаткові концепції, які знадобляться при навчанні й тестуванні алгоритмів глибокого навчання.

Розглянемо три сценарії. У кожному сценарії необхідно торкнутися таких тем:

- `tensorflow_linear_regression_training.py` – программа, що генерує модель лінійної регресії;
- `tensorflow_linear_regression_testing.py` − програма, яка завантажує створену модель і використовує її для створення нових прогнозів;
- `tensorflow_save_and_load_using_model_builder.py` – програма, що завантажує створену модель та експортує її для виведення з допомогою `SavedModelBuilder()`. Цей сценарій також завантажує остаточну модель, щоб зробити нові прогнози.

Лінійна регресія − дуже поширений статистичний метод, що дає змогу моделювати взаємозв'язок із заданого двовимірного набору точок. Модельна функція має такий вигляд:

$$\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}.$$

Це рівняння описує лінію з нахилом $\boldsymbol{W}$ і точкою перетину з $\boldsymbol{y}$ до $\boldsymbol{b}$. Мета полягає в тому, щоб знайти значення для параметрів $\boldsymbol{W}$ і $\boldsymbol{b}$, які забезпечують найкращу відповідність (наприклад, мінімум середньоквадратичної помилки) для точок двовимірної вибірки.

При навчанні моделі лінійної регресії (див. `Tensorflow_linear_ regression_training.py`) першим кроком є створення деяких даних, що будуть використовуватися для навчання алгоритму таким чином:

```
x = np.linspace(0, N, N)
y = 3 * np.linspace(0, N, N) + np.random.uniform(-10, 10, N)
```

Наступним кроком є визначення заповнювачів, щоб передати навчальні дані в оптимізатор під час навчання:

```
X = tf.placeholder("float", name='X')
Y = tf.placeholder("float", name='Y')
```

На цьому етапі оголошуються дві змінні (початкові випадковим чином) для ваг і зсуву:

```
W = tf.Variable(np.random.randn(), name="W")
b = tf.Variable(np.random.randn(), name="b")
```

Наступним кроком буде побудова лінійної моделі:

```
y_model = tf.add(tf.multiply(X, W), b, name="y_model")
```

Визначимо функцію вартості. У цьому випадку будемо використовувати функцію середньоквадратичної вартості помилки, як показано в такому фрагменті коду:

```
cost = tf.reduce_sum(tf.pow(y_model - Y, 2))/(2 * N)
```

Тепер створюємо оптимізатор градієнтного спуску, який мінімізує функцію вартості, змінюючи значення змінних. Далі створюємо оптимізатор градієнтного спуску, який мінімізує функцію вартості, змінюючи значення змінних $\boldsymbol{W}$ і $\boldsymbol{b}$.

Традиційний оптимізатор називають градієнтним спуском − алгоритмом ітеративної оптимізації з метою знаходження мінімуму функції:

```
optimizer =
tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

Параметр швидкості навчання контролює, наскільки змінюються коефіцієнти при кожному оновленні алгоритму градієнтного спуску. Як зазначалося раніше, градієнтний спуск є ітеративним алгоритмом оптимізації, отже, на кожній ітерації параметри змінюються відповідно до параметра швидкості навчання.

Останній крок при створенні моделі − ініціалізація змінних:

```
init = tf.global_variables_initializer()
```

На цьому етапі можна почати процес навчання всередині сеансу, як це показано в такому фрагменті коду:

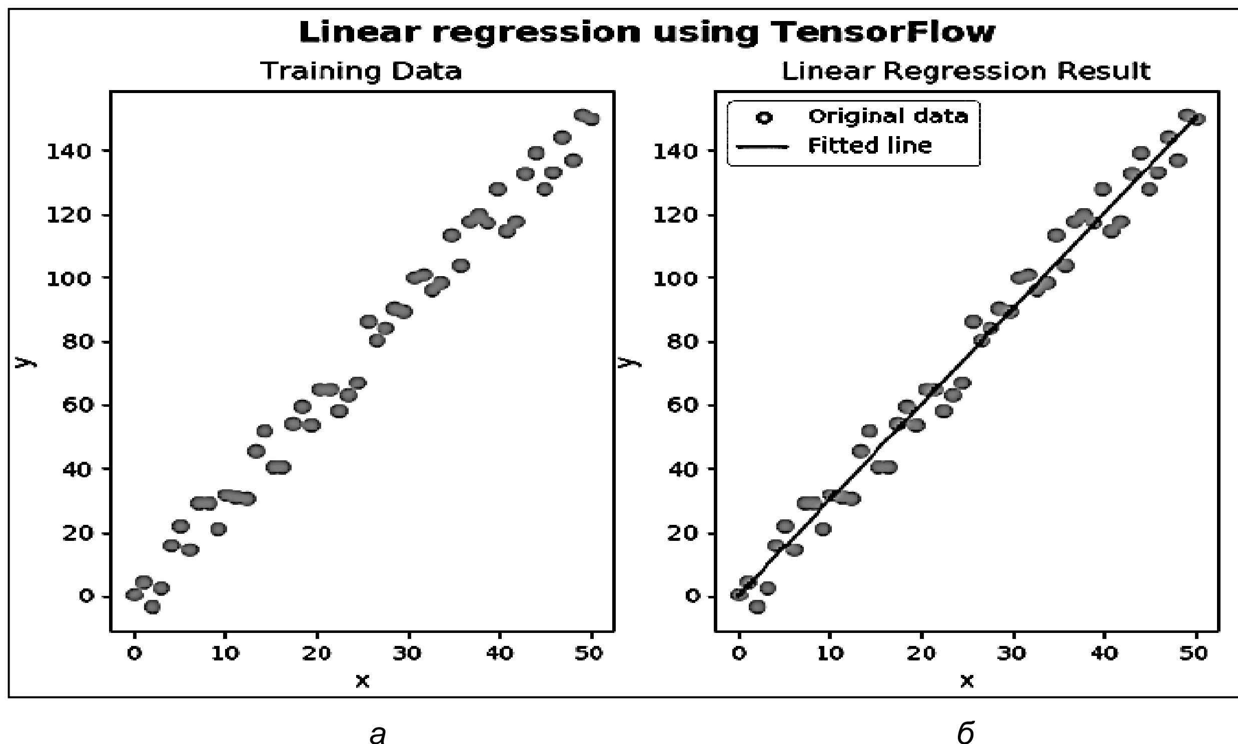
```
# Start the training procedure inside a TensorFlow Session:
with tf.Session() as sess:
    # Run the initializer:
    sess.run(init)
    # Uncomment if you want to see the created graph
    # summary_writer = tf.summary.FileWriter(logs_path,
sess.graph)

    # Iterate over all defined epochs:
    for epoch in range(training_epochs):

        # Feed each training data point into the optimizer:
        for (_x, _y) in zip(x, y):
            sess.run(optimizer, feed_dict={X: _x, Y: _y})

        # Display the results every 'display_step' epochs:
        if (epoch + 1) % disp_step == 0:
            # Calculate the actual cost, W and b:
            c = sess.run(cost, feed_dict={X: x, Y: y})
            w_est = sess.run(W)
            b_est = sess.run(b)
```

```
            print("Epoch",(epoch+1),": cost =",c,"W =",
w_est,"b=",b_est)

    # Save the final model
    saver.save(sess, './linear_regression')

    # Storing necessary values to be used outside the session
    training_cost = sess.run(cost, feed_dict={X: x, Y: y})
    weight = sess.run(W)
    bias = sess.run(b)

print("Training finished!")
```

Як показано в попередньому фрагменті коду, після запуску сеансу запускаємо ініціалізатор, а потім перебираємо всі визначені епохи для навчання моделі лінійної регресії. Крім того, друкуємо результати для кожної епохи `display_step`. Нарешті, коли навчання закінчено, зберігаємо остаточну модель (рис. 3.13, *а*). Лінію, що відповідає моделі лінійної регресії, зображено на рис 3.13, *б*.

При збереженні остаточної моделі з допомогою функції (`saver.save (sessions, './Linear_regression')`) створюються чотири файли:

- `.meta file` містить графік TensorFlow;
- `.data file` містить значення ваг, зсувів, градієнтів і всіх інших збережених змінних;
- `.index file` визначає контрольно-пропускний пункт;
- `checkpoint file` зберігає запис останніх збережених файлів контрольних точок.



а *б*

Рис. 3.13. Результати розрахунку параметрів лінійної регресії

На цьому етапі можна завантажити попередньо навчену модель і використовувати її для прогнозування. Це виконується в програмі `tensorflow_linear_ regression_testing.py`.

Перший крок − при завантаженні моделі потрібно завантажити графік з файлу `.meta`:

```
tf.reset_default_graph ()
imported_meta =
tf.train.import_meta_graph("linear_regression.meta")
```

Другий крок − завантажити значення змінних (зверніть увагу, що значення існують тільки всередині сеансу). Також запускаємо модель, щоб отримати значення $\boldsymbol{W}$, $\boldsymbol{b}$ і нові значення прогнозу таким чином:

```
with tf.Session() as sess:
    imported_meta.restore(sess, './linear_regression')
    # Run the model to get the values of the variables W, b and
new prediction values:
    W_estimated = sess.run('W:0')
    b_estimated = sess.run('b:0')
    new_predictions = sess.run(['y_model:0'], {'X:0': new_x})
```

На цьому етапі можна показати дані навчання, лінію регресії і нові значення прогнозу (сині точки на рис. 3.14).

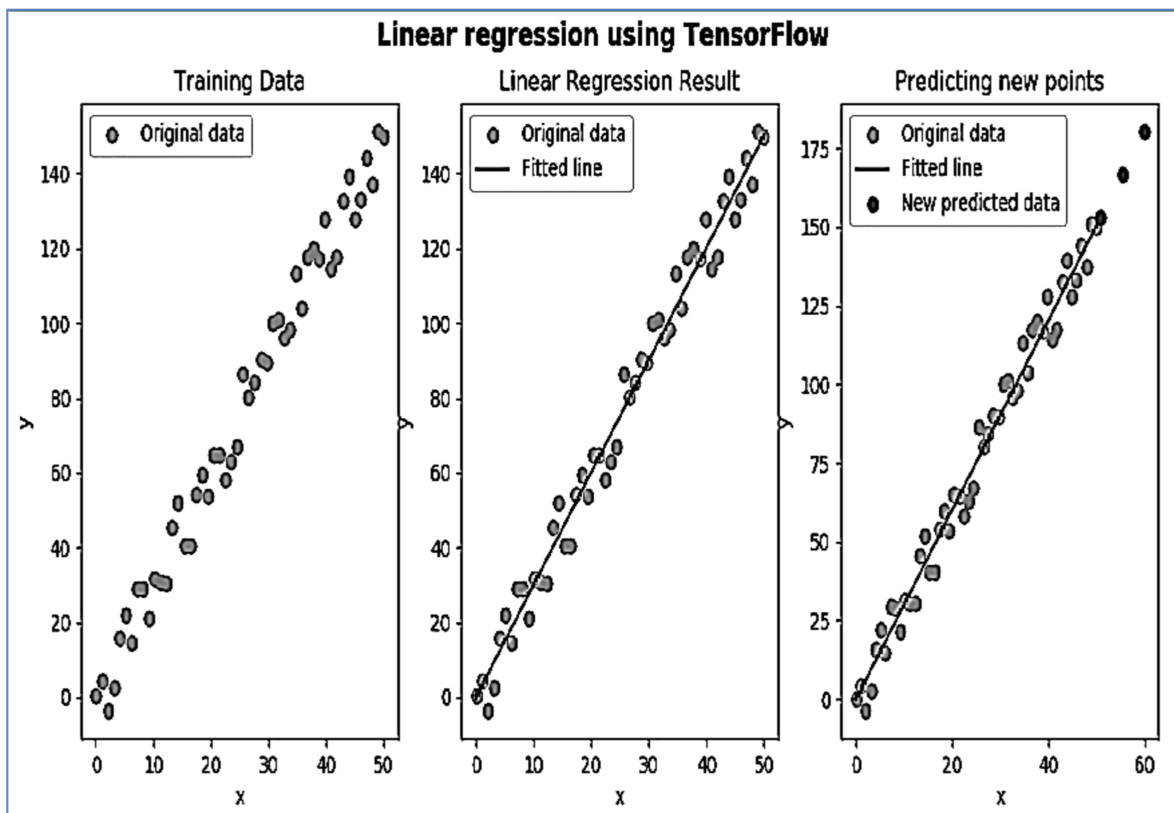


Рис. 3.14. Результати розрахунку параметрів лінійної регресії з прогнозованими даними

При використанні моделі бажано, щоб модель та її ваги були упаковані в один файл для полегшення зберігання, керування версіями й онов-

лення. Результатом буде двійковий файл з розширенням .pb, що містить топологію й ваги навченої мережі. Процес створення цього двійкового файлу й методи його використання виконуються в програмі tensorflow_save_and_load_using_model_builder.py.

У цій програмі закодовано функцію export_model() для експорту навченої моделі з допомогою SaveModel [29] таким чином:

```
def export_model():
    """Exports the model"""
    trained_checkpoint_prefix = 'linear_regression'

    loaded_graph = tf.Graph()
    with tf.Session(graph=loaded_graph) as sess:
        sess.run(tf.global_variables_initializer())

        # Restore from checkpoint
        loader =
tf.train.import_meta_graph(trained_checkpoint_prefix + '.meta')
        loader.restore(sess, trained_checkpoint_prefix)

        # Add signature:
        ...
        signature_map = {signa-
ture_constants.DEFAULT_SERVING_SIGNATURE_ DEF_KEY: signature}

        # Export model:
        builder =
tf.saved_model.builder.SavedModelBuilder('./my_model')
        builder.add_meta_graph_and_variables(sess,
signture_def_map = signature_map,
tags=[tf.saved_model.tag_constants.SERVING])
        builder.save()
```

Це створить файл saved_model.pb усередині папки my_model. На цьому етапі, щоб перевірити, чи правильно згенеровано експортовану модель, можна як імпортувати, так і використовувати її для створення нових прогнозів таким чином:

```
with tf.Session(graph=tf.Graph()) as sess:
    tf.saved_model.loader.load(sess,[tf.saved_ mod-
el.tag_constants.SERVING], './my_model'
    graph = tf.get_default_graph()
    x = graph.get_tensor_by_name('X:0')
    model = graph.get_tensor_by_name('y_model:0')
    print(sess.run(model, {x: new_x}))
```

Після виклику функції завантаження графік буде завантажений за замовчуванням. Крім того, також завантажуються змінні, так що можна почати виведення будь-яких нових даних. Буде виведений масив [153.04472

`166.54755 180.05037]`, який відповідає прогнозованим значенням, створеним цією моделлю.

3.7. Розпізнавання рукописних цифр з використанням бібліотеки TensorFlow

У цьому прикладі будемо класифікувати зображення з допомогою **TensorFlow**. Зокрема, створюємо просту модель (модель регресії `softmax`) для навчання й прогнозування рукописних цифр на зображеннях з використанням набору даних **MNIST**.

Регресія **Softmax** − це узагальнення логістичної регресії, яке можна використовувати для класифікації об'єктів, що належать до декількох класів. Набір даних **MNIST** [30] містить безліч рукописних цифрових зображень (рис. 3.15).

Рис. 3.15. Приклад цифрових зображень з набору даних MNIST

Програма `mnist_tensorflow_save_model.py` створює модель для вивчення й прогнозування рукописних цифр на зображеннях. Основні етапи роботи описано далі.

Для автоматичного імпорту цього набору даних можна використовувати такий код:

```
from tensorflow.examples.tutorials.mnist import input_data
data = input_data.read_data_sets("MNIST/",one_hot=True)
```

Завантажений набір даних складається з трьох частин − 55 000 рядків навчальних даних `mnist.train`, 10000 рядків тестових даних `mnist.test` і 5000 рядків даних перевірки `mnist.validation`. Крім того, частини навчання, тестування й перевірки містять відповідну мітку для кожної

цифри. Наприклад, навчальні дані складаються з `mnist.train.images` (зображення навчального набору даних) і `mnist.train.labels` (ярлики навчального набору даних). Кожне зображення складається з 28 х 28 пікселів, унаслідок чого виходить масив з 784 елементів. Параметр `one_hot = True` означає, що мітки будуть подані таким чином, що тільки один біт буде дорівнювати 1 для певної цифри. Наприклад, для 9 відповідною позначкою буде `[0 0 0 0 0 0 0 0 0 1]`.

Цей метод називають гарячим кодуванням. Це означає, що мітки були перетворені з одного числа на вектор, довжина якого дорівнює кількості можливих класів. Таким чином, усі елементи вектора будуть установлені в нуль, крім елемента $\boldsymbol{i}$, значення якого буде дорівнювати 1, що відповідає $\boldsymbol{i}$-му класу.

При визначенні заповнювачів необхідно порівняти їх форми й типи, щоб передати дані в такs змінні:

```
x = tf.placeholder(tf.float32, shape=[None, 784],
name='myInput')
y = tf.placeholder(tf.float32, shape=[None, 10], name='Y')
```

При визначенні `None` заповнювача в нього можна додати стільки прикладів, скільки необхідно. У цьому випадку заповнювач `x` може містити будь-який 784-вимірний вектор. Отже, форма цього тензора має вигляд `[None, 784]`. Крім того, можна також створити заповнювач `y` для подання істинної мітки. У цьому випадку форма цього тензора матиме вигляд `[None, 10]`.

На цьому етапі можна почати побудову графа обчислень. Перший крок − створити змінні $\boldsymbol{W}$ і $\boldsymbol{b}$:

```
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
```

Створено змінні $\boldsymbol{W}$ і $\boldsymbol{b}$, які будуть ініційовані нулями, оскільки **TensorFlow** оптимізує ці значення при навчанні. Розмірність $\boldsymbol{W}$ становить `[784, 10]`, помножимо її на 784-вимірний масив, що відповідає поданню певного зображення, щоб отримати 10-вимірний вихідний вектор. Тепер можна реалізувати модель таким чином:

```
output_logits = tf.matmul(x, W) + b
y_pred = tf.nn.softmax(output_logits, name='myOutput')
```

Тут `tf.matmul()` використовується для множення матриць, а `tf.nn.softmax()` − для застосування функції `softmax` до вхідного тензора, а це означає, що вихідні дані нормалізовано, їх можна інтерпретувати як імовірності. На цьому етапі можна визначити використовувану функцію втрат, оптимізатор (у цьому випадку `AdamOptimizer` [31]).

Оцінювання точності моделі:

```
# Define the loss function, optimizer, and accuracy
```

```
loss =
tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(label
s=y, logits=output_logits)
optimizer =
tf.train.AdamOptimizer(learning_rate=learning_rate,name='Adam-
op').minimize(loss)
correct_prediction =
tf.equal(tf.argmax(output_logits,1),tf.argmax(y,1),
name='correct_pred'
accuracy                                                       =
tf.reduce_mean(tf.cast(correct_prediction,tf.float32),
name='accuracy')
```

Нарешті, можна навчити модель, перевірити її з допомогою даних перевірки `mnist.validation`, а також зберегти модель:

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for i in range(num_steps):
        # Get a batch of training examples and their corre-
sponding labels.
        x_batch, y_true_batch = da-
ta.train.next_batch(batch_size)

        # Put the batch into a dict to be fed into the place-
holders
        feed_dict_train = {x: x_batch, y: y_true_batch}
        sess.run(optimizer, feed_dict=feed_dict_train)

    # Validation:
    feed_dict_validation = {x:data.validation.images,y: da-
ta.validation.labels}
    loss_test, acc_test = sess.run([loss, accuracy],
feed_dict=feed_dict_validation)
    print("Validation loss: {}, Validation accuracy:
{}".format(loss_test, acc_test))
    # Save model:
    saved_path_model = sav-
er.save(sess,'./softmax_regression_model_mnist')
    print('Model has been saved in {}'.format
(saved_path_model))
```

Після збереження моделі її можна використовувати для розпізнавання рукописних цифр на зображеннях. У програмі `mnist_save_and_load_model_builder.py` потрібно створити модель `saved_model.pb` усередині папки `my_model` і використовувати її для створення нових прогнозів завантаження зображень з допомогою OpenCV. Щоб зберегти модель, використовуємо функцію `export_model()`, яку було

подано в попередньому підрозділі. Щоб робити нові прогнози, використовуємо такий код:

```
# Load some test images:
test_digit_0 = load_digit("digit_0.png")
test_digit_1 = load_digit("digit_1.png")
test_digit_2 = load_digit("digit_2.png")
test_digit_3 = load_digit("digit_3.png")

with tf.Session(graph=tf.Graph()) as sess:
    tf.saved_model.loader.load(sess,
[tf.saved_model.tag_constants.SERVING], './my_model'
    graph = tf.get_default_graph()
    x = graph.get_tensor_by_name('myInput:0')
    model = graph.get_tensor_by_name('myOutput:0')
    output = sess.run(model, {x: [test_digit_0, test_digit_1,
test_digit_2, test_digit_3]})
    print("predicted labels: {}".format(np.argmax(output, ax-
is=1)))
```

Тут `test_digit_0`, `test_digit_1`, `test_digit_2` і `test_digit_3` – це чотири завантажених зображення, кожне з яких містить по одній цифрі. Щоб завантажити кожне зображення, використовуємо функцію `load_digit()`:

```
def load_digit(image_name):
    """Loads a digit and pre-process in order to have the prop-
er format"""

    gray = cv2.imread(image_name, cv2.IMREAD_GRAYSCALE)
    gray = cv2.resize(gray, (28, 28))
    flatten = gray.flatten() / 255.0
    return flatten
```

Потрібно попередньо обробити кожне зображення, щоб мати правильний формат, що відповідає формату зображень бази даних MNIST. Якщо виконаємо цей скрипт, то отримаємо передбачений клас для кожного зображення:

```
predicted labels: [0 1 2 3]
```

3.8. Бібліотека Keras

Keras [32] − це API (Application Programming Interface) нейронної мережі високого рівня з відкритим вихідним кодом, написаним мовою Python (сумісною з Python 2.7−3.6). Він може працювати поверх **TensorFlow**, **Microsoft Cognitive Toolkit**, **Theano** або **PlaidML,** його розроблено для можливості швидкого експериментування. У цьому підрозділі наведемо два приклади. У першому прикладі − лінійної регресії − використовуємо ті ж вхідні дані, що і в прикладі **TensorFlow** у попередньому підрозділі. У

другому прикладі класифікуємо деякі рукописні цифри з використанням набору даних **MNIST** так само, як використовували в попередньому підрозділі з **TensorFlow**. Таким чином, можна легко побачити відмінності між двома бібліотеками при вирішенні однотипних завдань.

Оцінювання параметрів лінійної регресії з допомогою бібліотеки Keras. Набір даних `linear_regression_keras_training.py` виконує навчання моделі лінійної регресії.

Перший крок − створення даних, які будуть використовуватися для навчання/тестування алгоритму:

```
# Generate random data composed by 50 (N = 50) points:
x = np.linspace(0, N, N)
y = 3 * np.linspace(0, N, N) + np.random.uniform(-10, 10, N)
```

Другий крок − створення моделі, для чого створюється функція `create_model()`, як показано в такому фрагменті коду:

```
def create_model():
    """Create the model using Sequencial model"""

    # Create a sequential model:
    model = Sequential()
    # All we need is a single connection so we use a Dense lay-
er with linear activation:
    model.add(Dense(input_dim=1, units=1, activation="linear",
kernel_initializer="uniform"
    # Compile the model defining mean squared error(mse) as the
loss
    model.compile(optimizer=Adam(lr=0.1), loss='mse')

    # Return the created model
    return model
```

При використанні **Keras** найпростішим типом моделі є послідовна модель, яку можна розглядати як лінійний стек шарів, і її використовуємо в цьому прикладі для створення моделі. Крім того, для більш складних архітектур можна використовувати функціональний API Keras, який дає змогу будувати довільні графи шарів. Отже, використовуючи послідовну модель, будуємо модель шляхом накладення шарів з допомогою методу `model.add()`. Тут використовується один щільний або повністю зв'язаний шар з лінійною функцією активації. Потім можна скомпілювати (або налаштувати) модель, що визначає середньоквадратичну помилку (**MSE**) як функцію втрат. У цьому випадку використовується оптимізатор `Adam` і встановлюється швидкість навчання 0,1.

На цьому етапі можна навчити модель, що завантажує дані, з допомогою методу `model.fit()`:

```
linear_reg_model.fit(x,  y,  epochs=100,  validation_split=0.2,
verbose=1)
```

Після навчання можна отримати значення як W, так і b (вивчені параметри), які будуть використовуватися для розрахунку прогнозів таким чином:

```
w_final, b_final = get_weights(linear_reg_model)
```

Функція `get_weights()` повертає значення цих параметрів:

```
def get_weights(model):
    """Get weights of w and b"""

    w = model.get_weights()[0][0][0]
    b = model.get_weights()[1][0]
    return w, b
```

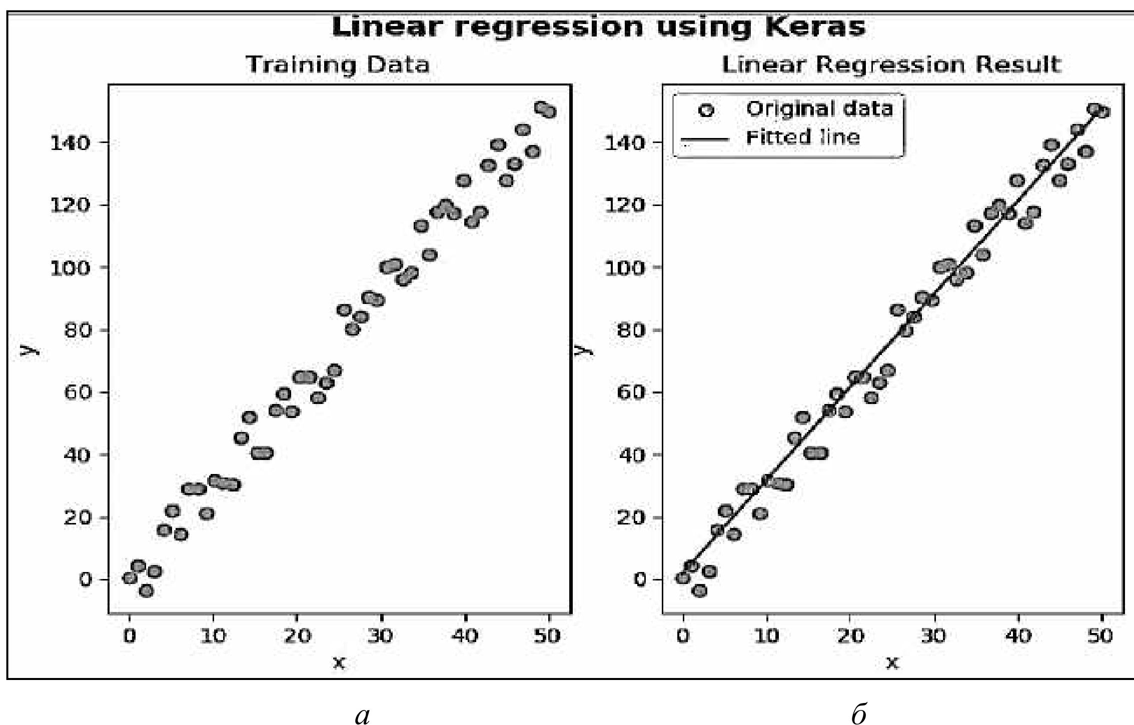
На цей момент можна побудувати такі прогнози:

```
# Calculate the predictions:
predictions = w_final * x + b_final
```

Також можна зберегти модель:

```
linear_reg_model.save_weights("my_model.h5")
```

Результат роботи цієї програми показано на рис. 3.16.



а *б*

Рис. 3.16. Результати розрахунку параметрів лінійної регресії

Тут зображено як дані навчання (рис. 3.16, *а*), так і підігнану лінію, що відповідає моделі лінійної регресії (рис. 3.16, *б*).

Попередньо навчену модель можна завантажити, щоб робити прогнози. Цей приклад можна побачити в програмі `linear_regression_keras_testing.py`.

Перший крок − завантажити ваги:

```
linear_reg_model.load_weights('my_model.h5')
```

Другий крок – увести функцію `get_weights()`, що дає змогу візуалізувати вивчені параметри:

```
m_final, b_final = get_weights(linear_reg_model)
```

На цьому етапі отримуємо прогнози навчальних даних, а також нові прогнози:

```
predictions = linear_reg_model.predict(x)
new_predictions = linear_reg_model.predict(new_x)
```

Третій крок − показати отримані результати (рис. 3.17). Попередньо навчену модель використано для створення нових прогнозів (сині точки).

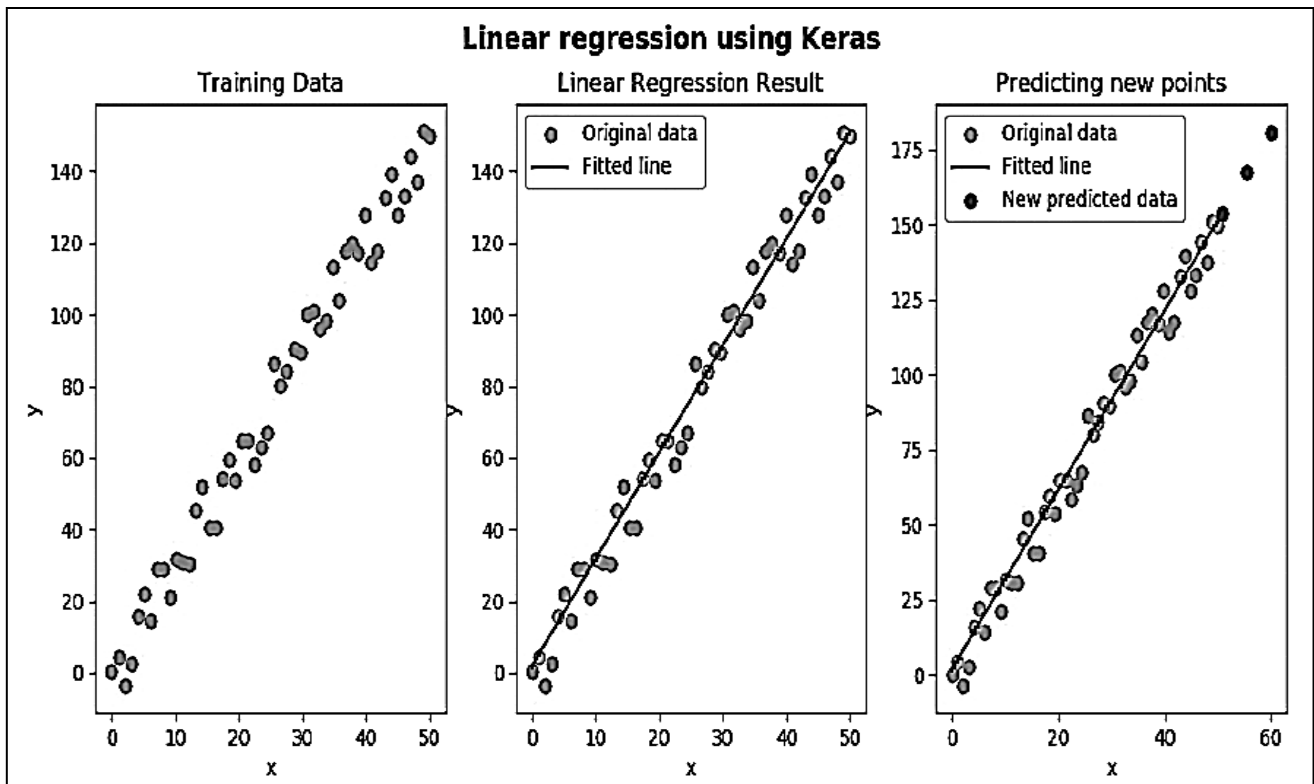


Рис. 3.17. Результати розрахунку параметрів лінійної регресії з прогнозованими даними

Розпізнавання рукописних цифр з використанням Keras. У цьому прикладі будемо розпізнавати рукописні цифри з набору даних **MNIST** з допомогою **Keras** і з використанням Google Colaboratoty.

Google Colaboratoty (скорочено **Colab**) дає змогу писати й виконувати довільний код мовою Python 3 через браузер і дуже добре підходить для машинного навчання, аналізу даних і навчання. **Colab** працює з більшістю основних браузерів, його найбільш ретельно протестовано з останніми версіями **Chrome**, **Firefox** і **Safari**. З технічної точки зору, **Colab** − це розміщена на хості служба **Jupyter** для ноутбуків, яка не потребує налаштування для використання, але при цьому надає безкоштовний доступ до обчислювальних ресурсів, включаючи графічні процесори.

Блокноти **Colab** зберігаються на **Google Drive** або можуть бути завантажені з **GitHub**. Код ноутбука мовою Python, створений на платформі **Google Colaboratory**, виконується на віртуальній машині з Linux у хмарі Google під обліковим записом користувача. Віртуальні машини видаляються, якщо вони не діють протягом деякого часу, і їх максимальний термін експлуатації встановлюється службою **Colab**. Максимальний термін експлуатації запущеного ноутбука може становити 12 годин. Ноутбуки також відмикаються від віртуальних машин, якщо вони занадто довго не використовуються. Максимальний час життя віртуальної машини й поведінка тайм-ауту просто можуть змінюватися з часом або залежно від використання.

Типи графічних процесорів (**GPU**), доступних у **Colab**, з часом змінюються. Це необхідно для того, щоб служба **Colab** могла безкоштовно надавати доступ до цих ресурсів. Графічні процесори, доступні в **Colab**, часто містять такі відеокарти: **Nvidia K80s**, **T4s**, **P4s** і **P100s**.

Для створення нового ноутбука необхідно перейти за посиланням [33]. Після реєстрації з'явиться робоче вікно ноутбука **Google Colaboratory** (рис. 3.18).

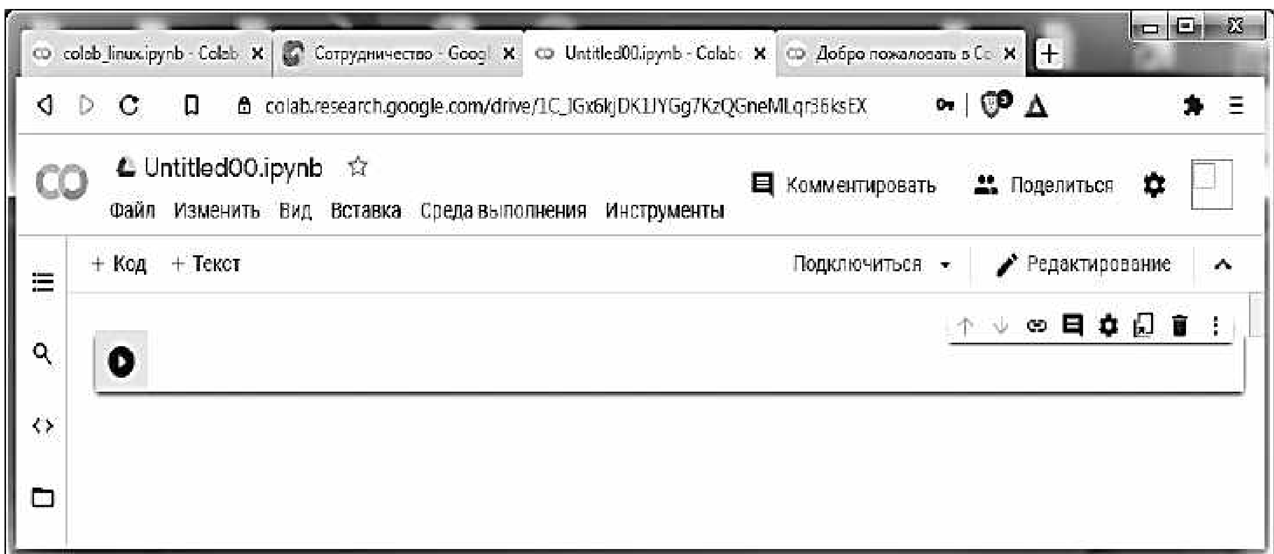


Рис. 3.18. Вікно ноутбука Google Colaboratory після його створення

Для прискорення навчання нейромережі можна використовувати відеокарту (**GPU**). Для цього в меню **Colab** необхідно вибрати **«Середовище виконання»** → **«Змінити середовище виконання»**, у діалозі в опції «Апаратний прискорювач» установити **«GPU»**, далі − **«Зберегти»** (рис. 3.19).

Початкову інформацію щодо роботи з **Google Colaboratory** викладено в [34].

У прикладі розглядається програма, яка реалізує навчання повнонейронної мережі з двох шарів (прихований шар складається з 800 нейронів, вихідний шар − з 10 нейронів) для розпізнавання рукописних цифр з набору даних **MNIST** з використанням **Keras**.

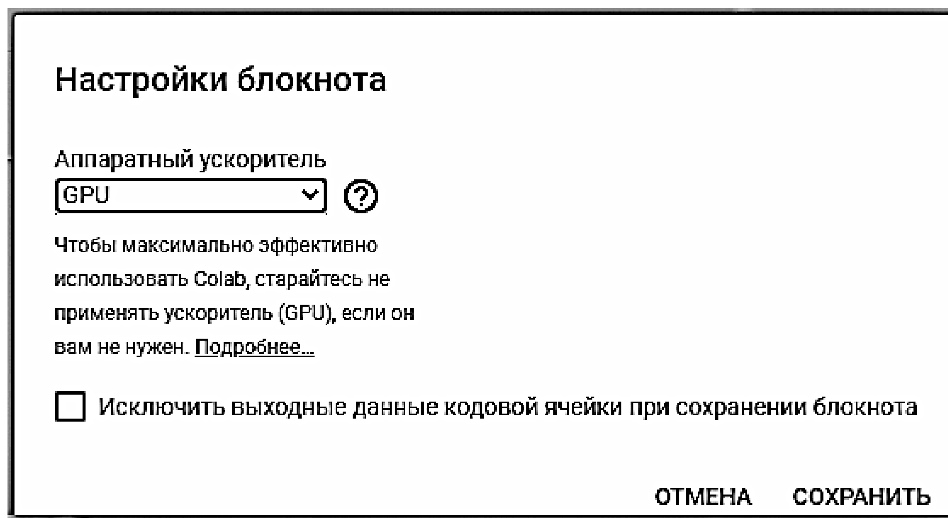


Рис. 3.19. Вибір **GPU** для навчання нейромережі

Стисло опишемо роботу програми. Для цього виділимо найбільш значущі фрагменти програмного коду для реалізації поставленого завдання і вставимо в програмний код короткі коментарі:

```
# Імпорт бібліотек, необхідних для роботи програми
import numpy
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.utils import np_utils
import matplotlib.pyplot as plt
# Установлення seed-генератора випадкових чисел для повторюваності результатів
numpy.random.seed(42)
# Завантаження з бази даних MNIST. Це дані про рукописні цифри, які будуть використовуватися для навчання нейромережі та її тестування
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# Формування списку з назвами класів для чисел
classes = ['0','1','2','3','4','5','6','7','8','9']
# Виконання перегляду двадцяти зображень рукописних цифр
plt.figure(figsize=(15,15)) # розмір зображення, що виводиться
for i in range(100, 120): # виведення зображень з 100 по 120
    plt.subplot(4,5,i-100+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False) # виведення без сітки на зображенні
    plt.imshow(X_train[i], cmap=plt.cm.binary)
    plt.xlabel(classes[y_train[i]])
```

На рис. 3.20, як і на рис. 3.15, показано ще один приклад зображень рукописних цифр з бази даних **MNIST.** У цьому прикладі цифри розташовані у випадковому порядку.

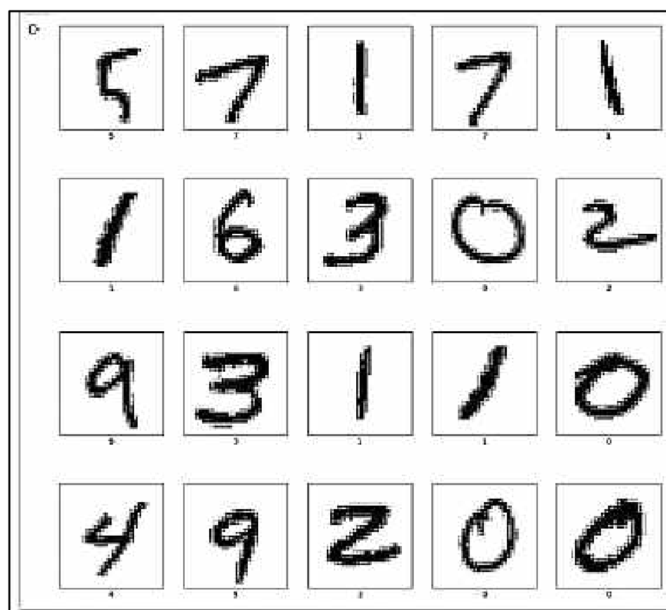


Рис. 3.20. Приклади зображень рукописних цифр з MNIST

```
# Виконання перетворення розмірності зображень
# 60000 даних для тренування нейромережі та валідації
X_train = X_train.reshape(60000, 784)
# 10000 даних для тестування нейромережі
X_test = X_test.reshape(10000, 784) )
# Нормалізація даних
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
# Розподіл даних між 0 і 1
X_train /= 255
X_test /= 255
# Перетворення міток у категорії
Y_train = np_utils.to_categorical(y_train, 10)
Y_test = np_utils.to_categorical(y_test, 10)
# Функція для створення послідовної моделі нейронної мережі
# hidden_size – кількість нейронів у прихованому шарі
def model_1(hidden_size):
# MNIST зображення розміром 28x28 і в шкалі сірого кольору
    height, width, depth = 28, 28, 1
# 10 класів на виході (1 клас на цифру)
    num_classes = 10
# Створення послідовної моделі мережі
# Вхід – це 1D-вектор розміром 28*28=784
    inp = Input(shape=(height * width,))
# Прихований шар мережі, функція активації 'relu'
    hidden_1 = Dense(hidden_size, activation =
'relu',kernel_initializer="normal")(inp)
# Вихідний шар мережі, функція активації 'softmax'
    out = Dense(num_classes, activation =
'softmax',kernel_initialize="normal")(hidden_1)
```

```
# Визначення моделі із заданням вхідного й вихідного шарів
    model = Model(input=inp, output=out)
# Компіляція моделі мережі з використанням оптимізатора 'adam'
    model.compile(loss='categorical_crossentropy', optimiz-
er='adam', metrics=['accuracy'])
    return model
# Функція model_1 повертає відкомпільовану модель мережі
hidden_size = 800 # – кількість нейронів у прихованому шарі
model = model_1(hidden_size)
# Виведення на друк параметрів використовуваної моделі
print(model.summary())
```

На рис. 3.21 зображено результати друку параметрів створеної моделі мережі. Вхідний шар мережі має 784 входи, прихований шар має 800 нейронів і 628 тисяч учнів-параметрів, вихідний шар має 10 нейронів і 8010 учнів-параметрів, усього мережа в двох шарах має 636010 учнів-параметрів.

```
Model: " model_1 "
_________________________________________________________________
Layer (type)                 Output Shape         Param #
=================================================================
input_1 (InputLayer)         [(None, 784)]             0
_________________________________________________________________
dense_1  (Dense)             (None, 800)          628000
_________________________________________________________________
dense_2  (Dense)             (None, 10)           8010
=================================================================
Total params: 636,010
Trainable params: 636,010
Non-trainable params: 0
_________________________________________________________________
```

Рис. 3.21. Результат друку параметрів моделі мережі model_1

```
# Навчання моделі: epochs = 10 – кількість епох навчання, batch_size = 200 – розмір батча, validation_split = 0.2 – 20% тренувальних даних використовуються для валідації, verbose=2 – режим виведення
```

```
model.fit(X_train, Y_train, batch_size=200, epochs=10, valida-
tion_split=0.2, verbose=2)
# Оцінювання якості навчання мережі на тестових даних
scores = model.evaluate(X_test, Y_test, verbose=0)
print("Точність роботи на тестових даних: %.2f%%" %
(scores[1]*100))
```

На рис 3.22 показано результати процесу навчання з використанням GPU і тестування мережі. Одна епоха навчання виконується 1 с, `loss` і `accuracy` – втрати і частка правильних відповідей на навчальній вибірці, `val_loss` і `val_accuracy` – втрати і частка правильних відповідей на валідаційній вибірці. На тестових даних з 10000 даних частка правильних відповідей становила 98,12 %.

```
CO  Untitled3.ipynb ☆                                  Комментировать   Поделиться
    Файл  Изменить  Вид  Вставка  Среда выполнения  Инструме
+ Код  + Текст                                 ✓ ОЗУ Диск          Редактирова

# Обучаем сеть: epochs=10 - число эпох, batch_size=200 - размер батча,
# validation_split=0.2 - 20% тренировочных данных используются для валидации
model.fit(X_train, Y_train, batch_size=200, epochs=10, validation_split=0.2, verbose=2)

# Оцениваем качество обучения сети на тестовых данных
scores = model.evaluate(X_test, Y_test, verbose=0)
print("Точность работы на тестовых данных: %.2f%%" % (scores[1]*100))

Epoch 1/10
240/240 - 1s - loss: 0.3077 - accuracy: 0.9132 - val_loss: 0.1608 - val_accuracy: 0.9553
Epoch 2/10
240/240 - 1s - loss: 0.1258 - accuracy: 0.9639 - val_loss: 0.1102 - val_accuracy: 0.9675
Epoch 3/10
240/240 - 1s - loss: 0.0805 - accuracy: 0.9772 - val_loss: 0.0876 - val_accuracy: 0.9741
Epoch 4/10
240/240 - 1s - loss: 0.0568 - accuracy: 0.9842 - val_loss: 0.0886 - val_accuracy: 0.9730
Epoch 5/10
240/240 - 1s - loss: 0.0417 - accuracy: 0.9890 - val_loss: 0.0807 - val_accuracy: 0.9758
Epoch 6/10
240/240 - 1s - loss: 0.0295 - accuracy: 0.9923 - val_loss: 0.0772 - val_accuracy: 0.9758
Epoch 7/10
240/240 - 1s - loss: 0.0218 - accuracy: 0.9949 - val_loss: 0.0722 - val_accuracy: 0.9786
Epoch 8/10
240/240 - 1s - loss: 0.0163 - accuracy: 0.9962 - val_loss: 0.0731 - val_accuracy: 0.9783
Epoch 9/10
240/240 - 1s - loss: 0.0118 - accuracy: 0.9977 - val_loss: 0.0772 - val_accuracy: 0.9787
Epoch 10/10
240/240 - 1s - loss: 0.0095 - accuracy: 0.9984 - val_loss: 0.0721 - val_accuracy: 0.9794
Точность работы на тестовых данных: 98.12%
```

Рис. 3.22. Результат навчання й тестування мережі

```
# Перегляд інформації про використовувану відеокарту
!cat/proc/driver/nvidia/gpus/0000:00:04.0/information
```

Google Collaboratory як **GPU** для навчання мережі надав відеокарту **Tesla K80** (рис. 3.23).

```
!cat /proc/driver/nvidia/gpus/0000:00:04.0/information

Model:           Tesla K80
IRQ:             35
GPU UUID:        GPU-e47eed53-13d0-bcd7-559c-a52a0221f034
Video BIOS:      80.21.25.00.02
Bus Type:        PCI
DMA Size:        40 bits
DMA Mask:        0xffffffffff
Bus Location:    0000:00:04.0
Device Minor:    0
Blacklisted:     No
```

Рис. 3.23. Відеокарта **Tesla K80**

```
# Навчання мережі зі збереженням історії навчання в об'єкті
history
history = model.fit(X_train, Y_train, batch_size=200,
epochs=10, validation_split=0.2, verbose=2)
# Оцінювання якості навчання мережі на тестових даних
scores = model.evaluate(X_test, Y_test, verbose=0)
print("Точність роботи на тестових даних: %.2f%%" %
(scores[1]*100))
# Виведення на друк умісту об'єкта history
print(history.history.keys())
```

На рис. 3.24 показано результати процесу навчання з використанням GPU і тестування мережі. На 10000 тестових даних частка правильних відповідей становила 98,30 %. Виконано підготовку даних для візуалізації процесу навчання.

Untitled3.ipynb ☆

Файл Изменить Вид Вставка Среда выполнения Инструменты Спра Комментировать Поделиться

+ Код + Текст ОЗУ Диск Редактирова

```
# Обучаем сеть: история обучения сохраняется в объект history
history = model.fit(X_train, Y_train, batch_size=200, epochs=10, validation_split=0.2, verbose=2)

# Оцениваем качество обучения сети на тестовых данных
scores = model.evaluate(X_test, Y_test, verbose=0)
print("Точность работы на тестовых данных: %.2f%%" % (scores[1]*100))

# исследуем объект history
print(history.history.keys())
```

```
Epoch 1/10
240/240 - 1s - loss: 0.0068 - accuracy: 0.9993 - val_loss: 0.0778 - val_accuracy: 0.9787
Epoch 2/10
240/240 - 1s - loss: 0.0054 - accuracy: 0.9995 - val_loss: 0.0803 - val_accuracy: 0.9766
Epoch 3/10
240/240 - 1s - loss: 0.0046 - accuracy: 0.9995 - val_loss: 0.0784 - val_accuracy: 0.9791
Epoch 4/10
240/240 - 1s - loss: 0.0036 - accuracy: 0.9996 - val_loss: 0.0790 - val_accuracy: 0.9793
Epoch 5/10
240/240 - 1s - loss: 0.0022 - accuracy: 1.0000 - val_loss: 0.0754 - val_accuracy: 0.9807
Epoch 6/10
240/240 - 1s - loss: 0.0013 - accuracy: 1.0000 - val_loss: 0.0761 - val_accuracy: 0.9806
Epoch 7/10
240/240 - 1s - loss: 0.0010 - accuracy: 1.0000 - val_loss: 0.0783 - val_accuracy: 0.9800
Epoch 8/10
240/240 - 1s - loss: 8.5689e-04 - accuracy: 1.0000 - val_loss: 0.0790 - val_accuracy: 0.9808
Epoch 9/10
240/240 - 1s - loss: 7.2086e-04 - accuracy: 1.0000 - val_loss: 0.0813 - val_accuracy: 0.9802
Epoch 10/10
240/240 - 1s - loss: 6.1591e-04 - accuracy: 1.0000 - val_loss: 0.0798 - val_accuracy: 0.9810
Точность работы на тестовых данных: 98.30%
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

Рис. 3.24. Результат навчання мережі

```
# Візуалізація історії навчання мережі за параметрами accuracy (частка правильних відповідей на навчальному наборі) і val_accuracy (частка правильних відповідей на валідаційному наборі)
plt.plot(history.history['accuracy'], label='Accuracy на навчальному наборі ')
plt.plot(history.history['val_accuracy'], label='Val_accuracy на валідаційному наборі')
plt.xlabel('Епоха навчання')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

На рис. 3.25 зображено історію навчання мережі за параметрами accuracy і val_accuracy (параметри навчання мережі − кількість епох epochs = 10, розмір batch_size = 200 − розмір батча).

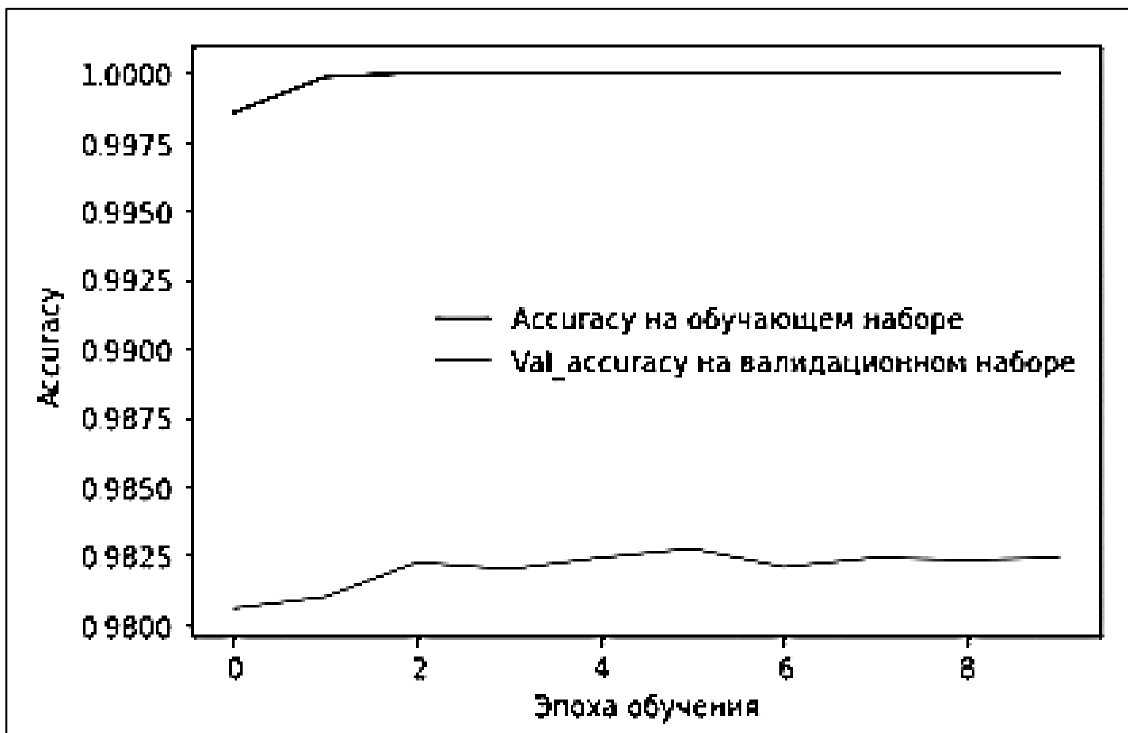


Рис. 3.25. Результат візуалізації історії навчання мереж за параметрами accuracy і val_accuracy

```
# Візуалізація історії навчання мережі за параметрами loss (втрати на навчальному наборі) і val_loss (втрати на валідаційному наборі)
plt.plot(history.history['loss'], label='loss на навчальному наборі')
plt.plot(history.history['val_loss'], label='val_loss на валідаційному наборі')
plt.xlabel('Епоха навчання')
plt.ylabel('loss')
plt.legend()
plt.show()
```

На рис. 3.26 зображено історію навчання мережі за параметрами `loss` (помилка на навчальному наборі) і `val_loss` (помилка на валідаційному наборі) (параметри навчання мережі − кількість епох `epochs = 10`, розмір `batch_size = 200` − розмір батча).

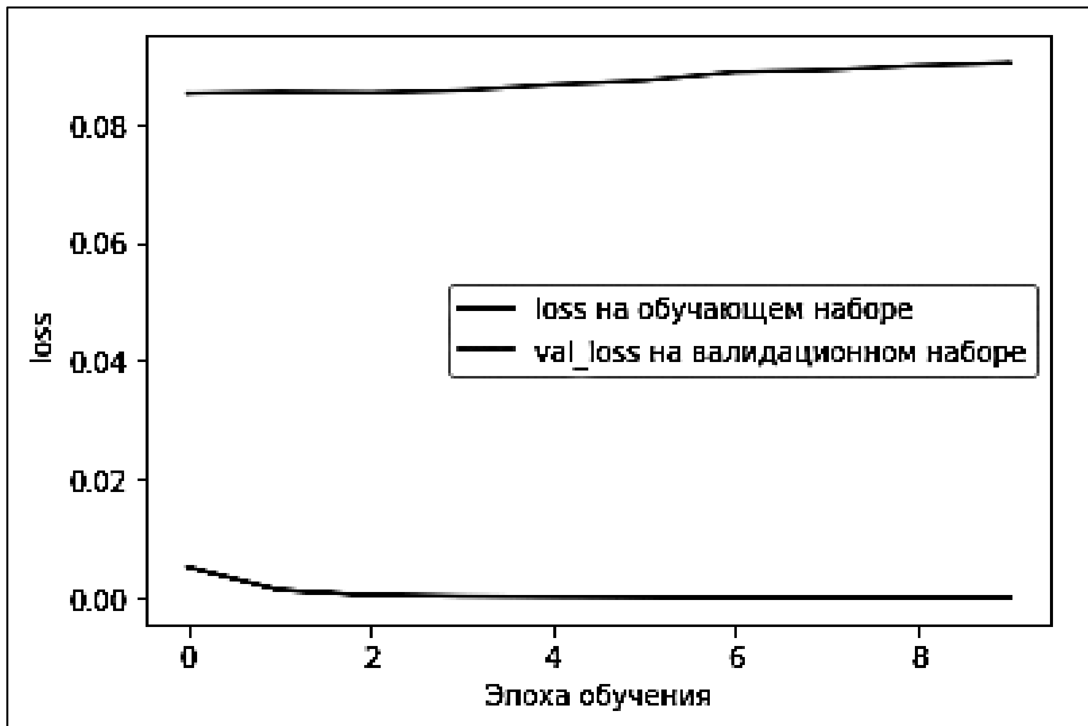


Рис. 3.26. Результат візуалізації історії навчання мережі за параметрами `loss` і `val_loss`

При навчанні можна використовувати техніку зупинення навчання при перенавчанні, для чого застосовується об'єкт `Early Stopping Callback`:

```
# Імпорт Callback EarlyStopping
From tensorflow.keras.callbacks import EarlyStopping
# Створюємо EarlyStopping Callback
early_stopping_callback =
EarlyStopping(monitor='val_accuracy',patience=5)
```

Параметр `monitor = 'val_accuracy'` указує, за якою метрикою якості спостерігатимемо. Це метрика `'val_ accuracy'` (частка правильних відповідей на валідаційному наборі даних). Параметр `patience = 5` (параметр "терпіння") визначає, протягом скількох епох не повинно відбуватися поліпшення потрібної метрики `'val_ accuracy'`, після чого відбудеться раннє зупинення. Тобто якщо значення метрики `'val_ accuracy'` не збільшується протягом п'яти епох, то навчання зупиняється:

```
History = model.fit(X_train, Y_train, batch_size=200,
epochs=25, validation_split=0.2, verbose=2,
callbacks=[early_stopping_callback])
print("Навчання зупинено на епосі", ear-
ly_stopping_callback.stopped_ epoch)
```

На рис. 3.27 показано результат використання техніки зупинення навчання мережі при її перенавчанні − необхідно було навчати мережу протягом 25 епох, але навчання було зупинено на 6-й епосі:

```
# Оцінювання якості навчання мережі на тестових завданнях
scores = model.evaluate(X_test, Y_test, verbose=0)
print("Точність роботи на тестових даних: %.2f%%" % (scores[1]*100))
```

```
Epoch 1/25
240/240 - 1s - loss: 1.5087e-08 - accuracy: 1.0000 - val_loss: 0.1573 - val_accuracy: 0.9823
Epoch 2/25
240/240 - 1s - loss: 1.4683e-08 - accuracy: 1.0000 - val_loss: 0.1573 - val_accuracy: 0.9824
Epoch 3/25
240/240 - 1s - loss: 1.4365e-08 - accuracy: 1.0000 - val_loss: 0.1575 - val_accuracy: 0.9823
Epoch 4/25
240/240 - 1s - loss: 1.4044e-08 - accuracy: 1.0000 - val_loss: 0.1578 - val_accuracy: 0.9823
Epoch 5/25
240/240 - 1s - loss: 1.3746e-08 - accuracy: 1.0000 - val_loss: 0.1579 - val_accuracy: 0.9823
Epoch 6/25
240/240 - 1s - loss: 1.3374e-08 - accuracy: 1.0000 - val_loss: 0.1582 - val_accuracy: 0.9821
Epoch 7/25
240/240 - 1s - loss: 1.3113e-08 - accuracy: 1.0000 - val_loss: 0.1581 - val_accuracy: 0.9824
Обучение остановлено на эпохе 6
```

Рис. 3.27. Результат використання техніки зупинення навчання при перенавчанні

На рис. 3.28 показано результати тестування мережі після процесу навчання з використанням техніки зупинення навчання при перенавчанні. За шість епох навчання на 10000 тестових даних частка правильних відповідей становила 98,32 %. Виконано підготовку даних для візуалізації процесу навчання:

```
# Прогнозування рукописних цифр навченою мережею
Predictions = model.predict(X_test)
n = 13 # Номер зображення
# Дані на виході з мережі в форматі one-hot-encoding для зображення з номером n
print(predictions[n])
# (28, 28) – розмір зображення рукописної цифри
plt.imshow(X_test[n].reshape(28, 28), cmap = plt.cm.binary)
plt.show()
# Визначення номера класу зображення, передбаченого мережею
print("Номер класу прогнозованого зображення: " ,
np.argmax(predictions[n]))
```

```
# Оцениваем качество обучения сети на тестовых данных
scores = model.evaluate(X_test, Y_test, verbose=0)
print("Точность работы на тестовых данных: %.2f%%" % (scores[1]*100))

Точность работы на тестовых данных: 98.32%
```

Рис. 3.28. Результат тестування мережі − правильних відповідей 98,32 %

На рис. 3.29 показано результат прогнозування рукописної цифри навченою мережею − виконується прогнозування зображення n = 13 з набору `X_test` (це цифра 0), `[1.0000000e +00 1.7093709e-37 3.0309103e-24 4.9832887e-31 3.7200619e-29 5.4203912e-27 1.1472053 e-23 1.6242751e-15 5.7805069e-27 6.2236556e-23]` − імовірності на виходах десяти нейронів вихідного шару мережі, наведено зображення прогнозованої цифри і результат прогнозування − клас зображення 0.

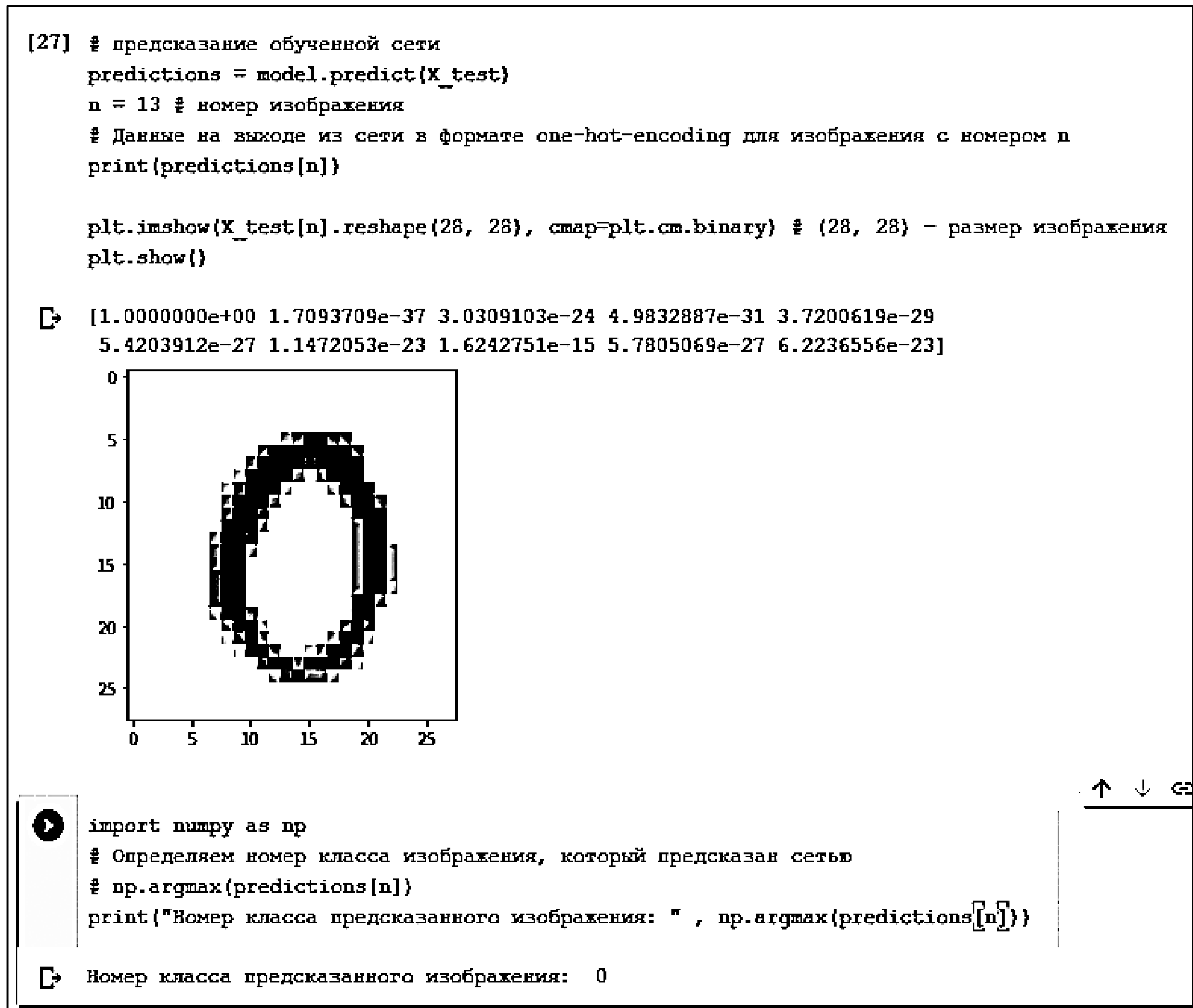


Рис. 3.29. Результат прогнозування мережею рукописної цифри

Отже, проста нейронна мережа з двох шарів (прихований шар складається з 800 нейронів, вихідний шар − з 10 нейронів) забезпечила частку правильних відповідей на тестових даних 98,32 %.

Таким чином, ми ознайомили вас з глибоким навчанням з використанням деяких популярних бібліотек: **OpenCV**, **TensorFlow** і **Keras**. Спочатку провели огляд сучасних архітектур глибокого навчання як для класифікації зображень, так і для виявлення об'єктів. Далі розглянули модулі глибокого навчання в **OpenCV**. Вони представляють бібліотеку **DNN** на базі реалізації прямого проходу (логічного висновку) з глибокими мережами, попередньо навченими з використанням різних платформ глибокого навчання.

З цього випливає головний висновок − починаючи з **OpenCV 3.3**, попередньо навчені мережі можна використовувати для прогнозування в різних нейромережних додатках. Це дає змогу істотно знизити трудовитрати й заощадити багато часу при реалізації робочих проектів.

4. ПРАКТИЧНЕ ВИКОРИСТАННЯ МЕТОДІВ І ЗАСОБІВ НЕЙРОННИХ МЕРЕЖ

Продемонструємо приклади практичного використання технологій нейронних мереж для ефективного вирішення завдань технічного зору. Сьогодні спостерігається експоненціальне зростання актуальних завдань у цій області − оброблення біометричної інформації, організація високоякісного інтелектуального відеоспостереження, забезпечення безпеки і face-control у сегменті масових громадських заходів і багато інших корисних додатків. Одним з найбільш актуальних і важливих завдань СТЗ є виявлення людських облич (face detection) на фотографіях і в кадрах відеопотоків і подальше надійне їх розпізнавання (face recognition) та ідентифікація за цими обличчями їх власників. При цьому до якості виконання завдання виявлення облич ставляться досить жорсткі вимоги. Головні з них − імовірність правильного виявлення обличчя має становити не менше 95...98 %, а оброблення відеоданих має проводитися в реальному масштабі часу.

4.1. Вимоги до компонентів проекту та критерії ефективності його роботи

Опишемо основні особливості вирішення завдання виявлення облич з використанням нейронних мереж. Ще на етапі ескізного проектування необхідно чітко сформулювати вимоги до головних компонентів цього технічного проекту. Для цього необхідно:

- обґрунтувати склад використовуваних програмних та апаратних засобів;
- сформулювати критерії якості роботи проектованої системи;
- визначити один або кілька методів оброблення інформації та алгоритми, що їм відповідають;
- вибрати зручну мову програмування й обґрунтувати використання конкретних додаткових інформаційних ресурсів (бібліотек, що підключаються, і баз даних);
- визначити структуру й основні компоненти архітектури нейронної мережі;
- вибрати методи навчання нейронної мережі, використовуючи наявні ресурси, або провести процес навчання самостійно;
- створити програму для реалізації алгоритмів детектування облич і провести її тестування.

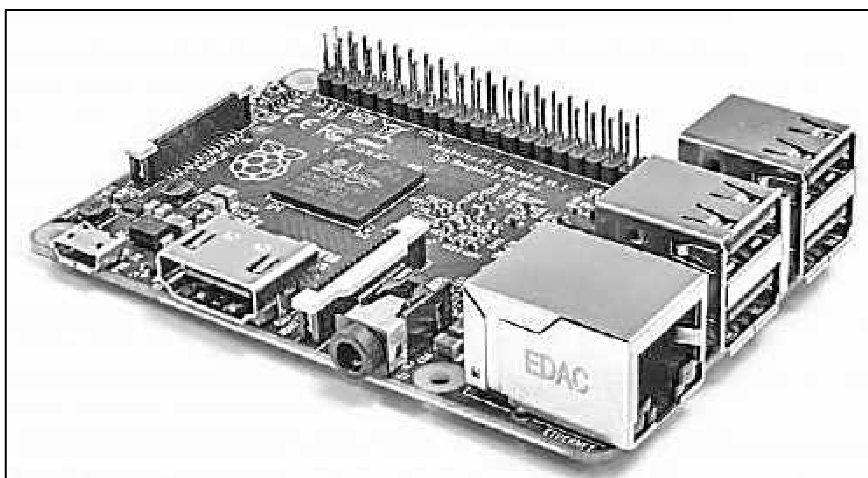
Розглянемо більш докладно методи вирішення поставленого завдання та використовувані засоби.

4.1.1. Програмні й апаратні ресурси

Мови програмування. При реалізації проекту використано мову програмування Python і ресурси бібліотеки OpenCV. Такий вибір обумовлений відкритим доступом до програмних продуктів і їх сумісністю з операційними системами Windows, Linux і Android. Особливість написання кодів мовою Python − формування необхідних можливостей програмування шляхом підключення зовнішніх бібліотек. Імпортовані ресурси бібліотеки OpenCV для вирішення цього завдання є порівняно невеликими.

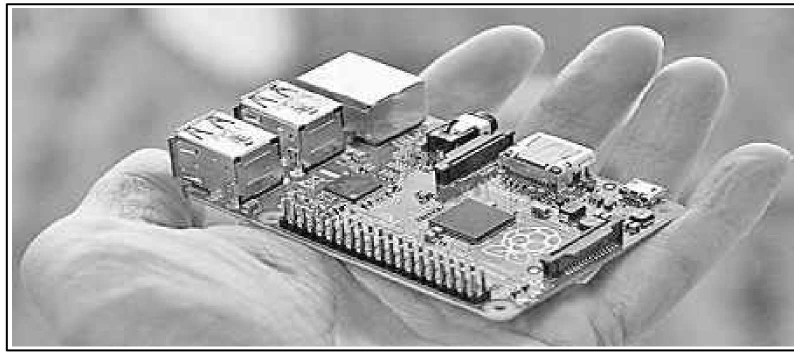
Апаратні платформи. При такому підході робота алгоритмів легко реалізується не тільки на стандартному ПК, але й на одній з моделей лінійки відомих мікрокомп'ютерів (Lego, Intel Galileo, Android IOIO OTG, Arduino та ін.). Тут впевнено лідирує платформа Raspberry Pi, представлена лінійкою моделей з різною апаратною реалізацією за доступними цінами. Кращі з них за швидкодією і місткістю пам'яті не поступаються настільним ПК. Так, Raspberry Pi 4 Model B, зовнішній вигляд і специфікацію якого показано на рис. 4.1, має 4 ГБ оперативної пам'яті, швидкий 4-ядерний процесор (1,5 ГГц), підтримку двох дисплеїв з роздільною здатністю до 4K, Gigabit Ethernet, USB 3.0, Wi-Fi, Bluetooth 5.0 і живленням через блок USB-C (5В/3А). Дуже важливо, що Raspberry Pi 4 має два порти USB 3.0, які в 10 разів є швидшими від USB 2.0 і добре підходять для під'єднання швидких периферійних блоків (флеш-накопичувачів, web-камер та ін.).

Зручність роботи з мовою програмування Python також полягає в тому, що є можливість роботи з операційною системою Windows (на персональному комп'ютері) і на мікрокомп'ютері Raspberry Pi з операційною системою Raspbian. Однак необхідно пильнувати за тим, щоб версії використовуваних бібліотек у межах цього проекту повністю збігалися.

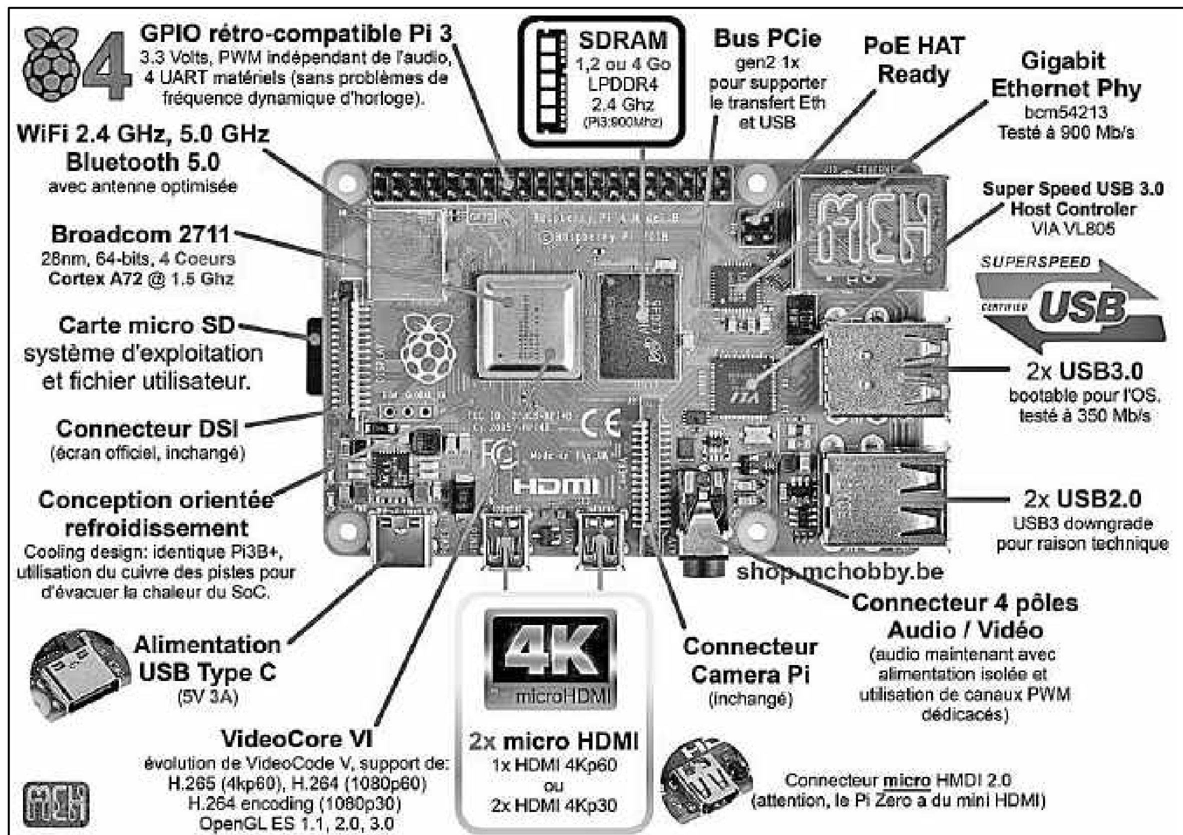


а

Рис. 4.1. Одноплатний комп'ютер Raspberry Pi (*а*, *б*) та його специфікація (*в*)



б



в

Рис. 4.1. Закінчення

Відеореєстратори. Зазвичай мобільні СТЗ комплектуються недорогими некаліброваними web-камерами з малою роздільною здатністю і можливістю їх під'єднння до Raspberry Pi через USB-порти. Більш високу якість порівняно з web-камерами мають спеціалізовані pi-камери для плат Raspberry Pi, що використовуються в монокулярних системах відеоспостереження. Наприклад, Raspberry Pi Camera Module v2 − високоякісний датчик зображення Sony IMX219 з фіксованим фокусом. Його під'єднують до плати мікрокомп'ютера з використанням спеціального CSi-інтерфейсу. Розміри датчика – (25 x 23 x 9) мм, вага − 3 г. Для під'єднання до Raspberry Pi використовують короткий плоский кабель. Схему під'єднання відеореєстратора до плати комп'ютера зображено на рис. 4.2.

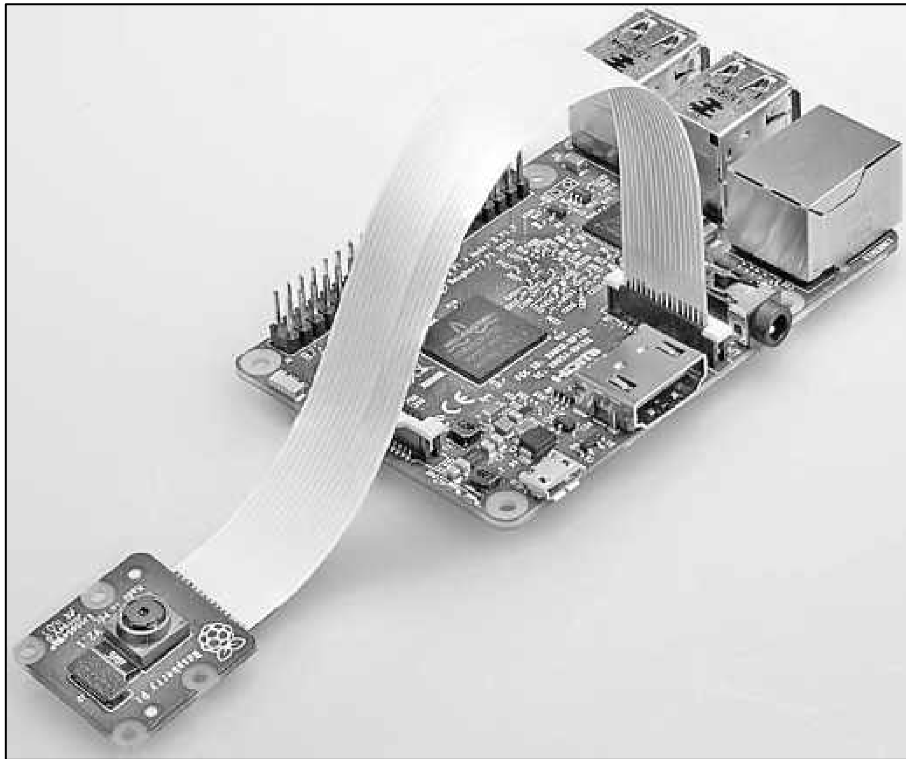


Рис. 4.2. Під’єднання Pi-камери до комп'ютера Raspberry Pi

Основні можливості цього датчика є такими:

- восьмимегапіксельна камера здатна робити фотографії з розрізненням 3280 x 2464;
- запис відео з роздільною здатністю 1080p, 30FPS; 720p, 60FPS; 640 × 480 p, 60 / 90FPS;
- програмне забезпечення підтримується в останній версії Raspbian Operating System.

Тут і далі будемо використовувати скорочення: p − pixel, FPS − frame per second.

4.1.2. Критерії ефективності системи детектування облич

Для створення об'єктивних критеріїв якості роботи системи детектування облич СТЗ уведемо такі показники та їх абревіатури:

- FPS (Frame per second). Ця величина характеризує швидкість змінення/читання кадрів реєстрованого відеопотоку. Значення FPS обмежується паспортними даними використовуваної відеокамери.
- FR (Frame Resolution) описує роздільну здатність кадру відео в пікселях (320 x 240, 640 x 480, 1280 x 960 тощо).
- FMB (Frame medium brightness). Це показник середнього рівня яскравості кадру, що об'єктивно відображає ступінь освітленості сцени в поточний момент часу.
- PCFD (Probability of correct face detection). Це показник імовірності правильного виявлення облич у кадрі відеопослідовності.

Ці найважливіші показники загалом і характеризують показники якості роботи системи детектування облич.

Традиційні методи захоплення відеоданих істотно обмежують швидкість оброблення відеоданих, що найчастіше серйозно обмежує можливість роботи в реальному масштабі часу. Однак, використовуючи багатопоточність, можна помітно зменшити вплив затримки введення/виведення, залишаючи основний інформаційний потік без блокування. Тобто існує можливість підвищення FPS практично до технологічної межі використання відеокамери.

Зрозуміло, що при вирішенні завдань візуалізації відеоданих завжди виправданим є прагнення до підвищення якості шляхом збільшення роздільної здатності кадру (FR). Однак слід пам'ятати, що значне збільшення кількості пікселів у кадрі неминуче призведе до погіршення (зменшення) показника FPS. Компроміс можна знайти експериментально, використовуючи нові ефективні алгоритми захоплення і введення відеоданих, запропоновані далі.

Зазначимо, що знання показника середньої яскравості кадру відеопотоку (FMB) в умовах великої мінливості освітлення сцени є дуже корисним. Це дає змогу шляхом виконання порогових процедур установлювати режим увімкнення/вимкнення контрастування (еквалізації) кадрів відеопотоку. Крім того, при низькому рівні яскравості кадру (шляхом порівняння FMB із заздалегідь установленим порогом) у системах тривалого відеоспостереження зручно вмикати/вимикати підсвічування відеокамери.

Спираючись на взяті показники якості, можна сформулювати рекомендації з побудови алгоритмів високоякісного введення відеоданих для сучасних стаціонарних і мобільних систем детектування облич.

Для узагальненого оцінювання ефективності роботи алгоритмів детектування облич за показником PCFD можна застосовувати два методи − якісний (візуальний) і кількісний. У першому випадку проводиться візуальне спостереження за різнокольоровими прямокутниками, що обмежують виявлене обличчя та його елементи (очі, ніс і рот). Якщо процес їх відтворення сприймається як безперервний, то якість детектування можна вважати прийнятною (~ 100 %). Однак інерційність зору спостерігача при високій частоті змінення кадрів (30 $с^{-1}$) не дає змоги візуально сприйняти пропуски виявлення в окремих кадрах. Це може істотно спотворити результати тестування. Тому для кількісного оцінювання ефективності в програмі для всіх кадрів відеопослідовності необхідно формувати одновимірний масив (1 − за фактом виявлення обличчя в кадрі, 0 − у разі пропуску обличчя), за яким будуються ймовірнісні характеристики якості роботи алгоритму. Однак слід пам'ятати, що для цього необхідна достовірно анотована тестова відеопослідовність. Найпростіший варіант такої послідовності − відеоряд, у кожному кадрі якого є обличчя для виявлення, дотримано умови освітлення (фронтальне джерело світла) і підтримується сприятливий геометричний фактор (кут нахилу голови менше 30°). Зрозуміло, мож-

на використовувати і більш складний тестовий відеоряд, у якому в певні інтервали часу обличчя в кадрі немає. Тоді можна оцінити не тільки ймовірності правильного виявлення облич, але й імовірності помилкового виявлення (помилкового виявлення обличчя при його відсутності в кадрі).

4.2. Розширений опис основних програмних рішень

4.2.1. Ресурси для програмування

Як уже зазначалося раніше, під час аналізу різних методів уведення відеоданих використовувалися мова програмування Python і всі доступні внутрішні й зовнішні ресурси. Головна особливість написання програмних кодів мовою Python − формування необхідних можливостей проекту шляхом під'єднання власних пакетів (наприклад, numpy, pip, pi-camera) і бібліотек (Pillow, Matplotlib та ін.). Істотно збільшує програмні ресурси і під'єднання зовнішніх бібліотек. У нашому випадку це бібліотека OpenCV, імпортовані ресурси якої для вирішення поставлених завдань є порівняно невеликими:

```
# import the necessary packages
from imutils.video import VideoStream
from imutils.video import FPS
import numpy as np
import argparse
import imutils
import datetime
import time
import cv2
```

Використання цих пакетів (а в деяких випадках й інших пакетів і модулів) істотно не збільшує навантаження на процесор і створює хороші передумови для оброблення даних у реальному масштабі часу.

Зверніть увагу на використання пакета `imutils` і модуля `argparse`, що порівняно рідко застосовуються.

`Imutils` − набір зручних функцій для спрощення основних операцій оброблення зображень, таких як зсув, повороти, змінення розмірів, відображення їх в Matplotlib, з допомогою OpenCV і Python. Пакет imutils використовують і при створенні багатопоточного класу.

Мова Python забезпечує доступ до вбудованої web-камери комп'ютера (зовнішньої камери з USB-під'єднанням або pi-камери) з допомогою функцій захоплення відеоданих OpenCV. Це дає змогу значно збільшити FPS шляхом створення нового потоку, який тільки опитує камеру на зчитування нових кадрів, а основний потік обробляє поточні кадри.

`Argparse` − модуль Python для оброблення опцій та аргументів командного рядка, у якому викликається програма. Модуль `argparse` раціонально використовувати для перетворення відеофайлів на послідовність звичайних зображень та їх збереження з допомогою бібліотеки OpenCV.

4.2.2. Методи введення відеоданих

Класичний метод введення й читання відеоданих з web-камери в системах монокулярного зору здійснюється з допомогою функції OpenCV для відеозахоплення `cv2.VideoCapture(0)`. Наведемо фрагмент програмного коду мовою Python для практичної реалізації цього методу:

```
cv2.VideoCapture(0)
while True:
    (grabbed, frame) = stream.read()
    ...
    ...
    cv2.imshow("Frame",frame)
```

Однак функція відеозахоплення `VideoCapture` і метод читання даних `read ()` блокують основний потік програмного коду з оброблення відеоданих доти, доки кадр не буде зчитано з пристрою камери і не повернуто в основну програму. На жаль, цей метод, що характеризується простотою, часто стає головною перешкодою − він обмежує можливість оброблення відеопотоку в реальному масштабі часу.

Метод уведення апаратними засобами Raspberry Pi здійснюється з допомогою функції `PiCamera()`. Однак при цьому виникає обмеження величини FSP через дію програмного навантаження. Цей метод не дає змоги використовувати дві і більше камер. Зазначимо також, що необхідною є тимчасова затримка для оброблення відображення кадрів. Програмний код цього методу має вигляд:

```
camera = PiCamera()
camera.framerate = 32.
Stream = camera.capture_continuous( rawCapture,format =
"bgr",use_video_port = True)
time.sleep(2.0)
for (i,f) in enumerate(stream):
frame = f.array
    ...
    ...
cv2.imshow("Frame", frame)
```

Метод багатопотокового введення полягає у створенні класу VideoStream () для перенесення читання кадрів web-камери або usb-пристроїв в інший потік, абсолютно відокремлений від основної програми мовою Python. Це дає змогу безперервно зчитувати кадри з потоку введення/виведення, поки основний потік обробляє поточний кадр. Як тільки основний потік завершує оброблення чергового кадру, йому просто потрібно витягнути черговий кадр з потоку введення/виведення. Це відбувається без очікування блокування операцій введення/виведення. Програмний код класу багатопотокового введення VideoStream наведено нижче:

```
fvs=VideoStream(usePiCamera = args["picamera"] > 0).start()
time.sleep(2.0)
while True:
    frame = vs.read()
    cv2.imshow("Frame",frame)
fvs.stop()
```

Створення класу FPS. Важливе доповнення до функціональності багатопотокового відеовведення − створення класу FPS, який використовується для візуалізації та оцінювання швидкості введення даних і дає кількісні докази того, що багатопотоковість дійсно збільшує FPS:

```
fps = FPS().start()
while fvs.more():
     ...
     ...
     fps.update()
fps.stop()
printfps()
```

Результати обчислення поточних значень FPS відображаються на екрані монітора Raspberry Pi.

Створення класу FR. `Frame Resolution` є дуже важливим параметром кадру, від вибору якого безпосередньо залежить FPS відеопотоку. Наприклад, найбільш можлива роздільна здатність pi-камери становить FR = 1280 x 960. При цьому FPS = 30 $с^{-1}$.

Для вибору й контролю розмірів кадру створено клас `Resolution`, що має максимальну роздільну здатність кадру й пропорції, при яких фрейм не деформується. Наведемо фрагмент коду для реалізації цієї процедури:

```
fvs=VideoStream(usePiCamera(0)] >.start()
while fvs.more():
     resolution.frame(fvs)
     frame1=resolution.init(1,1)
     frame2=resolution.init(2,1)
     cv2.imshow("Frame1",frame1)
     cv2.imshow("Frame2",frame2)
     ...
fvs.stop()
```

4.2.3. Стабілізація контрастності відеоданих

Реєстрація відеоданих зазвичай відбувається при різного роду перешкодах і постійному динамічному зміненні фону спостережуваної сцени. При цьому одним з головних негативних факторів є мінливість освітленості. Зазначимо, що це можуть бути як умови освітлення, що швидко змінюються, так і такі, що повільно змінюються, наприклад, через настання сутінків. Усе це призводить до погано контрольованих варіацій контрастності кадрів

і, отже, до погіршення якості оброблення. Нагадаємо, що зазвичай змінення значень яскравості для відеоданих визначається діапазоном `0...255`. Це відповідає формату даних `uint8`. Для подолання цих труднощів (погано контрольованих змінень контрастності кадрів залежно від освітленості) запропоновано вихідну відеопослідовність з колірного простору RGB перетворити на простір YUV з допомогою функції

```
img_yuv = cv2.cvtColor(img, cv2.COLOR_BGR2YUV)
```

Діапазон значень RGB становить `[0 ÷ 255]` для кожної компоненти, а для колірного простору YUV використовуються діапазони:

- `Y → [0...255];`
- `U → [-112...112];`
- `V → [-157...157].`

Характерною особливістю колірного простору YUV є те, що в ньому використовується явний поділ інформації про яскравість і колір. Колір подається у вигляді трьох компонент − яскравісної (Y) і двох кольорорізницевих (U і V). Після перекладання кадру RGB відеопослідовності в колірний простір YUV у ньому здійснюється процедура еквалізації (підвищення контрастності) тільки компоненти Y з допомогою функції

```
img_yuv[:,:,0] = cv2.equalizeHist(img_yuv[:,:,0]),
```

а потім видбувається зворотне перетворення кадру з формату YUV на формат RGB:

```
img_output = cv2.cvtColor(img_yuv,cv2.COLOR_YUV2BGR)
```

При цьому баланс кольору зберігається без змін, оскільки кольорорізницеві компоненти U і V перетворень не зазнавали.

Зазначимо, що перехід з колірного простору RGB у простір YUV дає змогу просто оцінити середній рівень яскравості кадру за компонентою Y. Найбільш об'єктивним і стійким показником, на наш погляд, є FMB (Frame medium brightness), який розраховується як

$$\boldsymbol{FMB} = \frac{\mathbf{1}}{\boldsymbol{MN}} \sum_{\boldsymbol{i}=\mathbf{0}}^{\boldsymbol{N}} \sum_{\boldsymbol{j}=\mathbf{0}}^{\boldsymbol{M}} \boldsymbol{Y}(\boldsymbol{i},\boldsymbol{j}),$$

де $\boldsymbol{Y}(\boldsymbol{i},\boldsymbol{j})$ − двовимірний масив чисел, що визначають яскравість пікселів зображення кадру розміром $\boldsymbol{M}\text{x}\boldsymbol{N}$. Цей показник тим більш є корисним, оскільки при малих рівнях яскравості кадру в системах відеоспостереження для розпізнавання облич зручно використовувати процедуру його порівняння з попередньо встановленим порогом для автоматичного ввімкнення/вимкнення системи підсвічування сцени.

Таким чином, використання процедури еквалізації забезпечує вирівнювання гістограми розподілу яскравостей і зводить показник середньої яскравості зображення до значення `MB = 127` незалежно від того, яким цей показник був для вихідного зображення. Це дає можливість стабілізувати рівень середньої яскравості кадрів, а отже, усунути фактори нестабільності освітленості сцени, які негативно впливають на якість їх подальшого оброблення.

4.3. Експериментальні дослідження алгоритмів уведення відеоданих та їх результати

Для оцінювання ефективності пропонованих методів підвищення якості відеоданих проведено експериментальні дослідження з допомогою відеореєстраторів (рис. 4.2) і спеціалізованої програми Video Stream Quality Control, написаної мовою Python з використанням відповідних ресурсів бібліотеки OpenCV. Ця програма базується на використанні описаних алгоритмів уведення і попереднього оброблення вхідних відеоданих, уведених з допомогою web- або pi-камер у мікрокомп'ютер Raspberry Pi. Повний лістинг програмного коду винесено в дод. 1 цієї роботи.

Стартове вікно інтерфейсу програми показано на рис. 4.3.

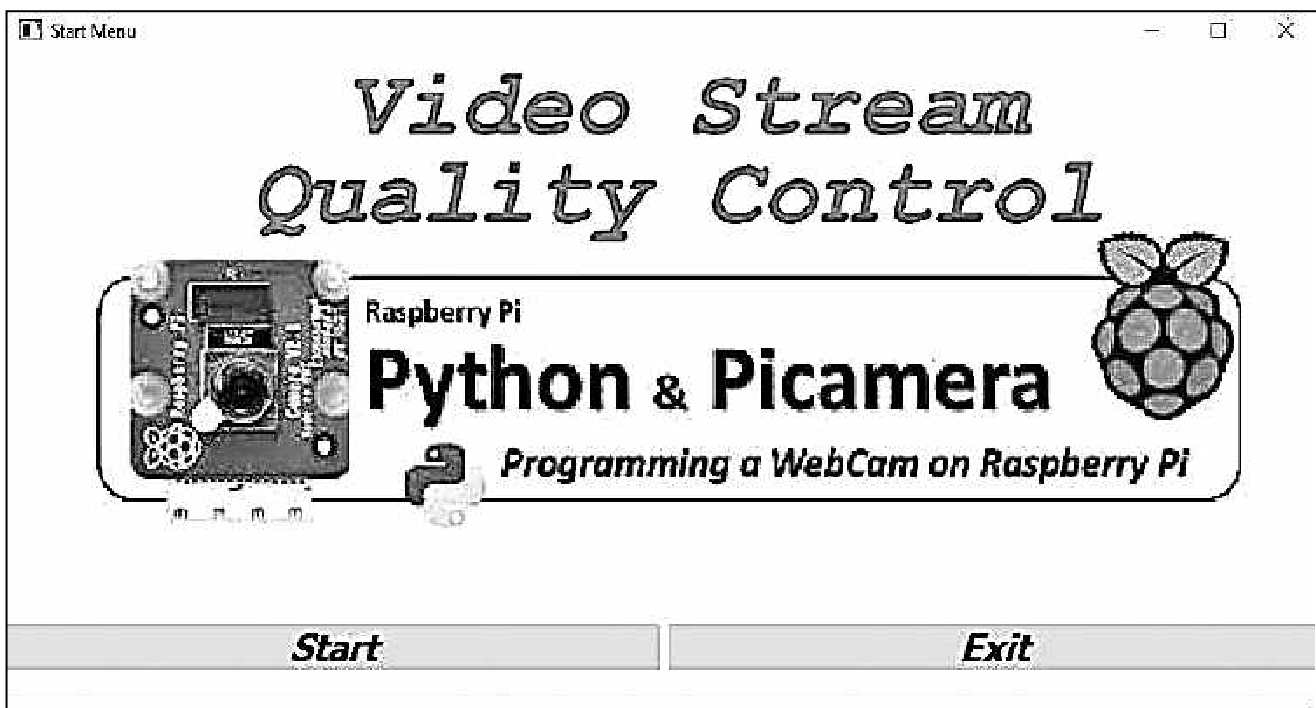
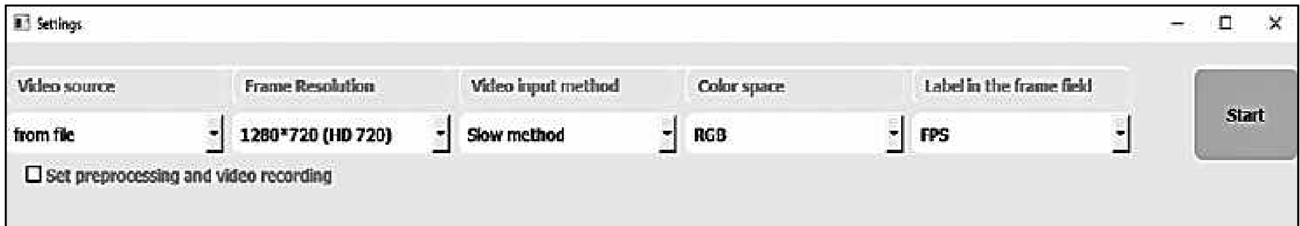


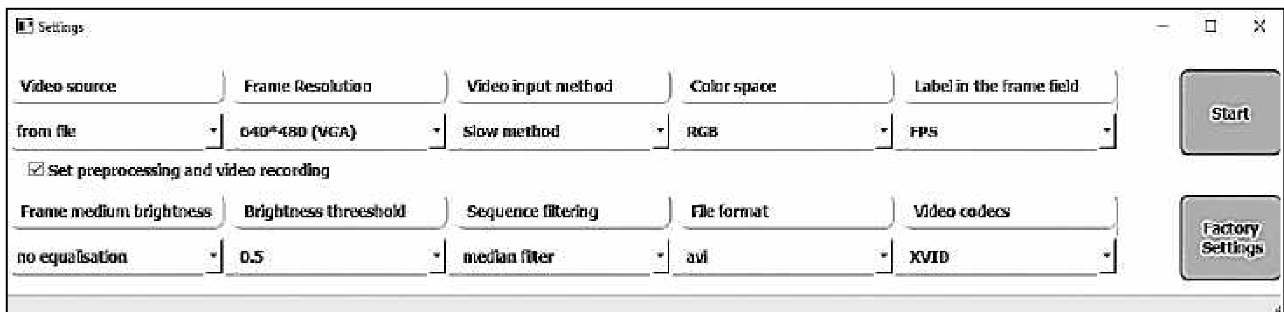
Рис. 4.3. Стартове вікно програми для проведення експериментів

На рис. 4.4 показано два вікна установлення робочих режимів програми. У вікно налаштувань (рис. 4.4, *а*) користувач переходить зі стартового вікна програми після натискання кнопки «Start». У ньому налаштовується джерело інформації (from web-camera, from pi-camera або from file), з допомогою випадних меню встановлюється роздільна здатність кадру (Frame

resolution), визначається колірний простір для відеоданих (Color space) і найменування параметрів відображаються в полі кадру (Label in the frame field).



а



б

Рис. 4.4. Вигляд вікон установлення режимів роботи програми Video Stream Quality Control

Натиснувши кнопку «`Start`» у початковому вікні налаштувань (див. рис. 4.4, *а*), можна перейти в розширене вікно налаштувань (див. рис. 4.4, *б*). У цьому вікні з допомогою випадних меню можна легко налаштувати параметри попереднього оброблення й подальшого запису відеоданих (`Set preprocessing and video recording`).

Узагальнену UML-діаграму роботи програми зображено на рис. 4.5.

З урахуванням комплексного характеру проведених досліджень у програму було закладено можливості організації різних варіантів алгоритму відеозахоплення і введення відеоданих:

- класичний варіант − з функцією `cv2.VideoCapture(0)`;
- прискорений варіант з допомогою апаратних засобів комп'ютера Raspberry Pi;
- найшвидший варіант − багатопотокове введення з допомогою спеціального класу `VideoStream`).

Крім цього в програмі можна попередньо встановити необхідну роздільну здатність екрана, а також в умовах недостатньої освітленості сцени − режим стабілізації яскравості кадру.

Програма `Video Stream Quality Control` надає багато сервісних можливостей. Насамперед це можливість відображення на екрані відеопотоку, що вводиться з допомогою функції `cv2.imshow(frame)`. Функція `transformation_frame(frame)` дає змогу змінювати формат

кадрів та їх розміри, а в разі необхідності встановлювати й режим стабілізації яскравості кадру. Крім того, передбачено спеціальні написи в полі кадру щодо інформації про поточне значення FPS відеопотоку, розміри кадру і стабілізацію яскравості зображення сцени.

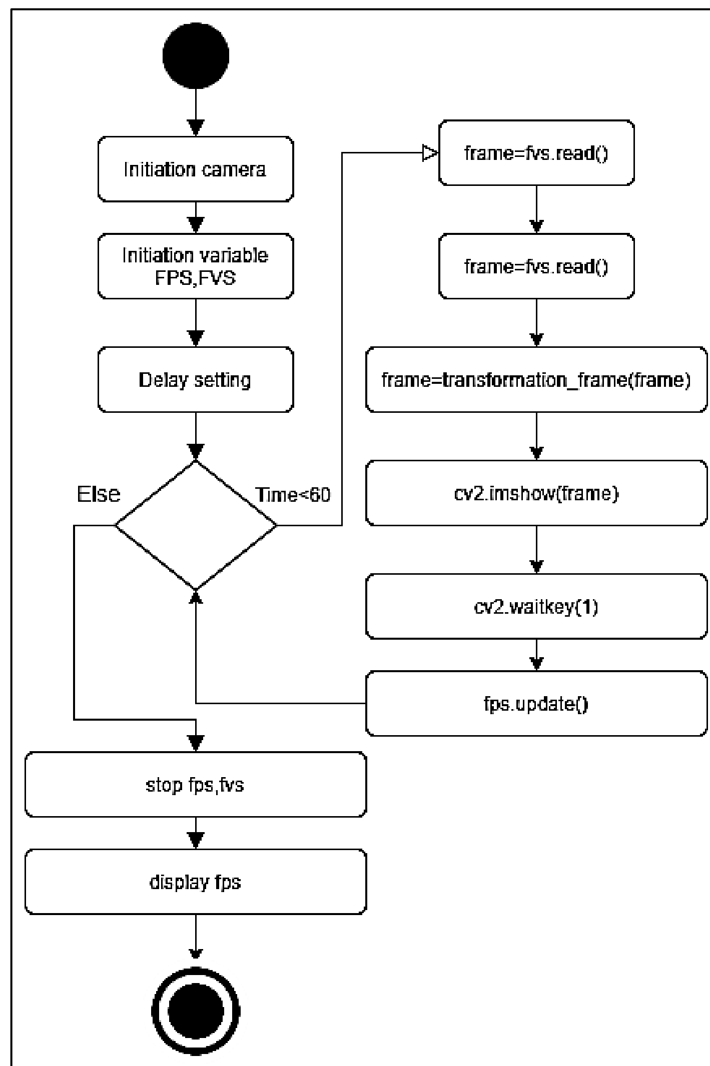


Рис. 4.5. UML-діаграма активностей спеціалізованої програми Video Stream Quality Control

Зазначимо, що для встановлення робочого режиму класу `VideoStream` (захоплення необхідної кількості кадрів вхідного відеопотоку) вибрано інтервал часу 1 с, а для отримання стійких і достовірних оцінок FPS інтервал усереднення 10 с.

При плануванні експерименту ставилися завдання виявлення і вивчення взаємозв'язку між характером змінень показника швидкодії FPS і роздільною здатністю FR-кадрів при введенні відеоданих у мікрокомп'ютер Raspberry Pi, а також їх попереднього оброблення. Крім того, досліджувався ступінь впливу на швидкість процедури введення додаткового навантаження на процесор комп'ютера внаслідок дії алгоритму стабілізації яскравості кадрів відеопотоку. На основі цих даних сформуль-

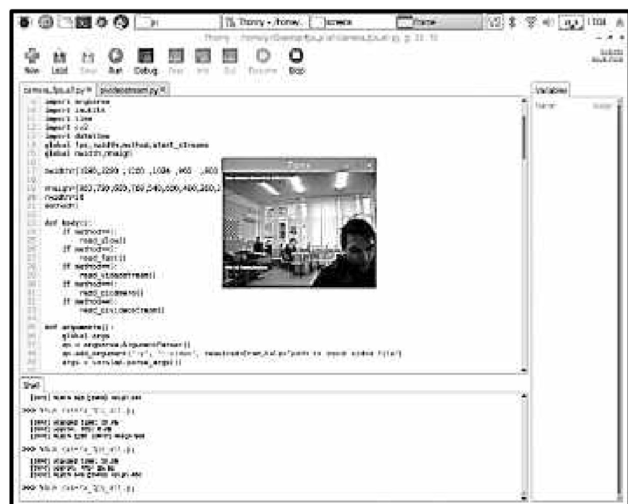
овано рекомендації щодо оптимальної конфігурації програмних засобів уведення і попереднього оброблення прийнятих відеоданих. Не менш важливим є і завдання зручної і наочної візуалізації отриманих експериментальних даних.

На рис. 4.6 показано два варіанти формування кадрів відеопотоку з різною роздільною здатністю і різним співвідношенням розмірів сторін кадру. У першому випадку при роздільній здатності 1280 x 960 (рис. 4.6, *а*) кадр практично повністю накриває площу робочого вікна. Напис у лівому нижньому кутку кадру відображає поточне значення FPS. При перегляді відеопотоку значення FPS змінюється в реальному масштабі часу. Напис у лівому верхньому кутку показує, який метод уведення відеоданих використовується в цьому випадку (`Slow method` − повільний метод, а `Fast method` − швидкий).

На рис. 4.6, *б* на екрані комп'ютера відображено кадр відеопотоку з роздільною здатністю 320 x 240. Така роздільна здатність зазвичай використовується в робототехніці. Через малу площу цього кадру можна легко побачити екранний фрагмент коду програми, а в лівому нижньому кутку командного вікна − поточні розрахункові значення FPS. Наведемо фрагмент таких даних:

```
[INFO] elasped time: 10.67
[INFO] approx. FPS: 8.53
[INFO] Width 1280
[INFO] Heigh 960.
```

а



б

Рис. 4.6. Кадри відеопотоку, сформовані з різним розрізненням: *а* −1280 x 960; *б* − 320x240

Порівнявши показники FPS у прикладах, показаних на рис. 4.6, бачимо, що перехід від малого розрізнення кадру до великого призводить до істотного зниження швидкості введення − FPS зменшується від 23 до 8 $с^{-1}$. Потенційно можливий (паспортний) показник FPS дорівнює 30 $с^{-1}$.

На рис. 4.7 показано два відеокадри з роздільною здатністю 640 x 480.

Їх уведено в комп'ютер Raspberry Pi з допомогою багатопотокового методу `VideoStream`.

а *б*

Рис. 4.7. Стабілізація яскравості кадрів відеопотоку

Через недостатнє освітлення сцени (рис. 4.7, *а*) до попереднього оброблення відеопотоку введено процедуру контрастування (еквалізації) кадрів (рис. 4.7, *б*). Раніше згадувалося про те, що це потребує перекладання кадру `RGB` відеопослідовності в колірний простір `YUV` з подальшою еквалізацією яскравісної компоненти `Y` і повернення в колірний простір `RGB`. Зрозуміло, що це створює додаткове навантаження на процесор і негативно впливає на швидкість уведення − FPS зменшується з 24 до 19 $с^{-1}$. Однак якість зображення кадру підвищується.

На рис. 4.8 показано залежність показника швидкості введення відеоданих FPS з допомогою web-камери від вибраної роздільної здатності кадру. Аналіз проводився для форматів з порівняно невисокою роздільною здатністю (від QVGA − 320 × 240 до Full HD – 1920 x 1080). Порівняльний аналіз ефективності двох методів уведення (класичного й багатопотокового) показав безсумнівну перевагу останнього. Особливо це позначається при високій роздільній здатності кадрів. Так, наприклад, при роздільній здатності HD 720 з FPS швидкого методу показник становить 17,60, що перебільшує такий показник повільного методу приблизно вдвічі (FPS = 9,76). Такі дані є корисними для практичного застосування.

Як і в попередньому дослідженні, вивчалися дані про характеристики швидкості введення відеопотоку з допомогою різних методів (повільного й швидкого). Але при цьому для відеоспостереження в комплекті з мікрокомп'ютером Raspberry Pi використовувалася спеціалізована pi-камера. Залежність значень FPS від роздільної здатності відеокадрів показано на рис. 4.9. Очевидно, що використання методу багатопотокового введення (`Fast method`) у цьому випадку істотно перевершує можливості повільного методу (`Slow method`).

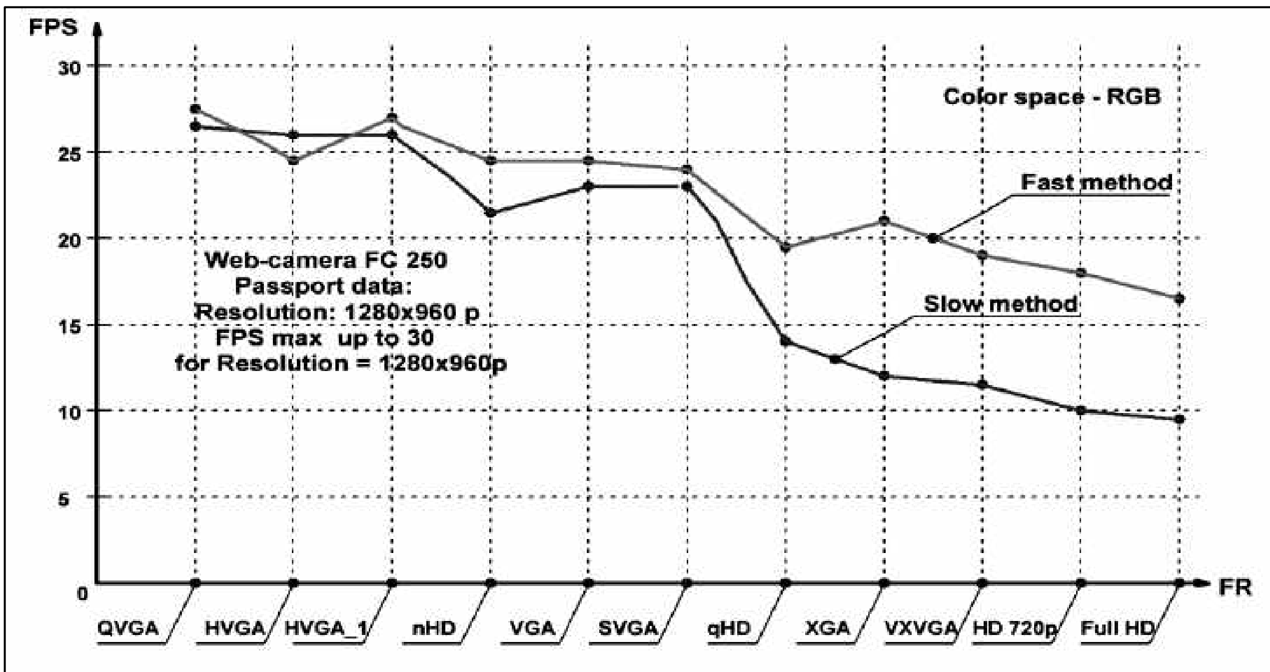


Рис. 4.8. Показники швидкості введення з web-камер відеоданих для різної роздільної здатності кадрів відеопотоку

FPS
90
80
70
60
50
40
30
20
10
0
Color space - RGB
Pi-camera Omni Vision OV5647
Passport data:
Max resolution: 2592x1944 p;
FPS max up to 30
for resolution = 1080p;
FPS max up to 60
for resolution = 720p;
FPS max up to 60/90
for resolution = 640x480.
Fast method
Slow method
FR
QVGA
HVGA
HVGA_1
nHD
VGA
SVGA
qHD
XGA
VXVGA
HD 720p
Full HD

Рис. 4.9. Показники швидкості введення відеоданих з рі-камер

Особливо це помітно при малих значеннях роздільної здатності кадру. Наприклад, при роздільній здатності QVGA (320 x 240) швидкість уведення даних збільшується в 3,5 раза і наближається до потенційних можливостей pi-камери за швидкодією. На основі цих даних можна зробити висновок − у практично важливих додатках є доцільним для цього завдання використовувати pi-камери і `Fast method`, що ґрунтується на застосуванні багатопотокового класу `VideoStream`.

Наведемо результати аналізу впливу алгоритмів попереднього оброблення відеоданих (`preliminary processing of video data`) на швидкість їх уведення (FPS) при фіксованому значенні роздільної здатності екрану (FR).

Дослідження проводилися з використанням двох методів уведення: `Slow method` − повільний, `Fast method` − швидкий, що базується на багатопотоковому введенні.

Аналізувалися такі варіанти попереднього оброблення відеоданих:

- без попереднього оброблення відеопотоку;
- з використанням еквалізації;
- з НЧ-фільтрацією (low pass filter);
- з медіанною фільтрацією (medianblur);
- з гаусівською фільтрацією (gaussianblur);
- комбінація еквалізації і гаусівської фільтрації.

Результати аналізу цього експерименту наочно показано на рис. 4.10. Слід зазначити, що при читанні відеоданих з файлу помітно підвищується рівень FPS завдяки відсутності технологічних обмежень, властивих як web-, так і pi-камерам. Максимальна FPS становить ~ 175 $с^{-1}$. На швидкість уведення даних найбільш істотно негативно впливає застосування медіанної фільтрації (FPS зменшується приблизно в 4 рази). І нарешті, очевидно, що застосування багатопотокового введення відео (`Fast method`) практично не дає виграшу за швидкодією порівняно з повільним методом `Slow method`.

Результати другої серії експериментів показано на рис. 4.11. У цьому випадку як джерело відео використовувалася web-камера. Усі інші параметри експерименту повністю збігаються з попередніми дослідами. Аналіз показав, що при використанні web-камери швидкість уведення (FPS) при різних варіантах попереднього оброблення відеоданих змінюється незначно. Однак, метод уведення `Fast method` є приблизно вдвічі швидшим від повільного методу `Slow method` і впритул наближається до потенційно можливого FPS (30 $с^{-1}$).

Наведені приклади, звичайно, не охоплюють усіх особливостей цього завдання. Можна досліджувати вплив на швидкість уведення і якість кадрів інших факторів (наприклад, швидкості введення відео при використанні pi-камер). Однак матеріали таких досліджень виходять за межі цієї публікації.

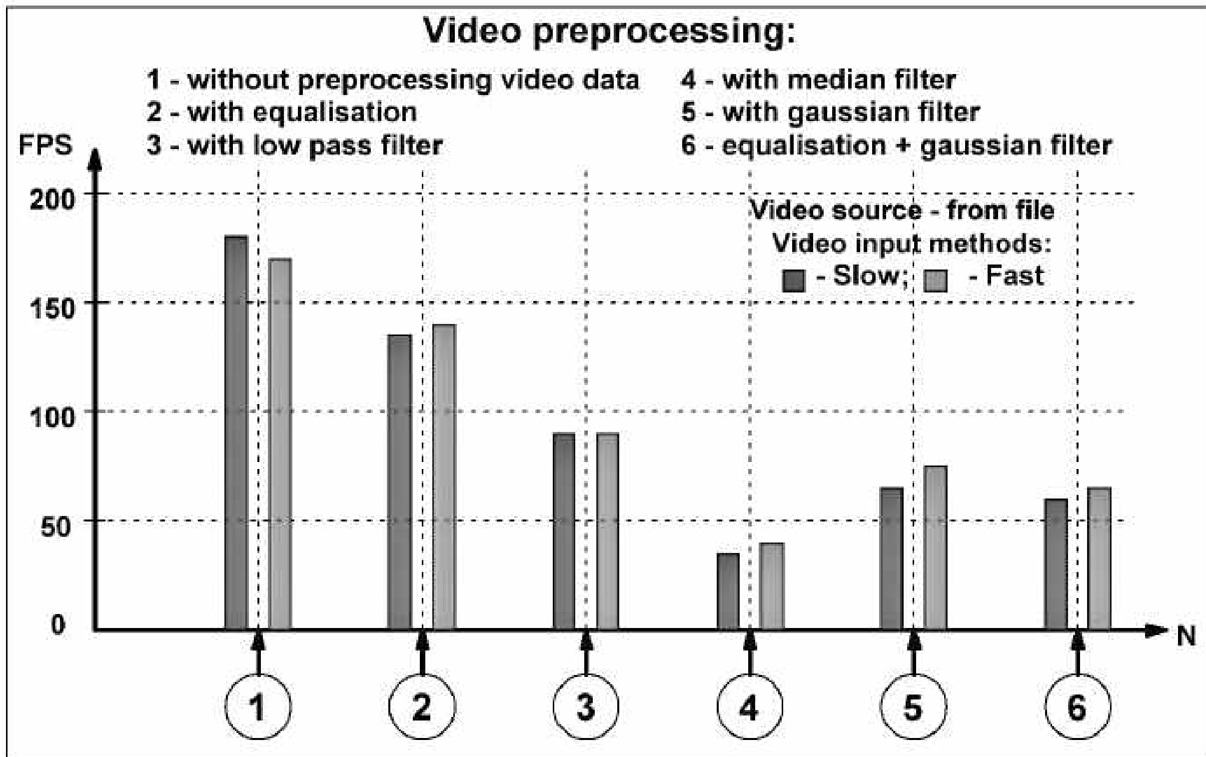


Рис. 4.10. Залежність швидкості введення відеоданих (FPS) при читанні файлу відео від характеру попереднього оброблення відеоданих

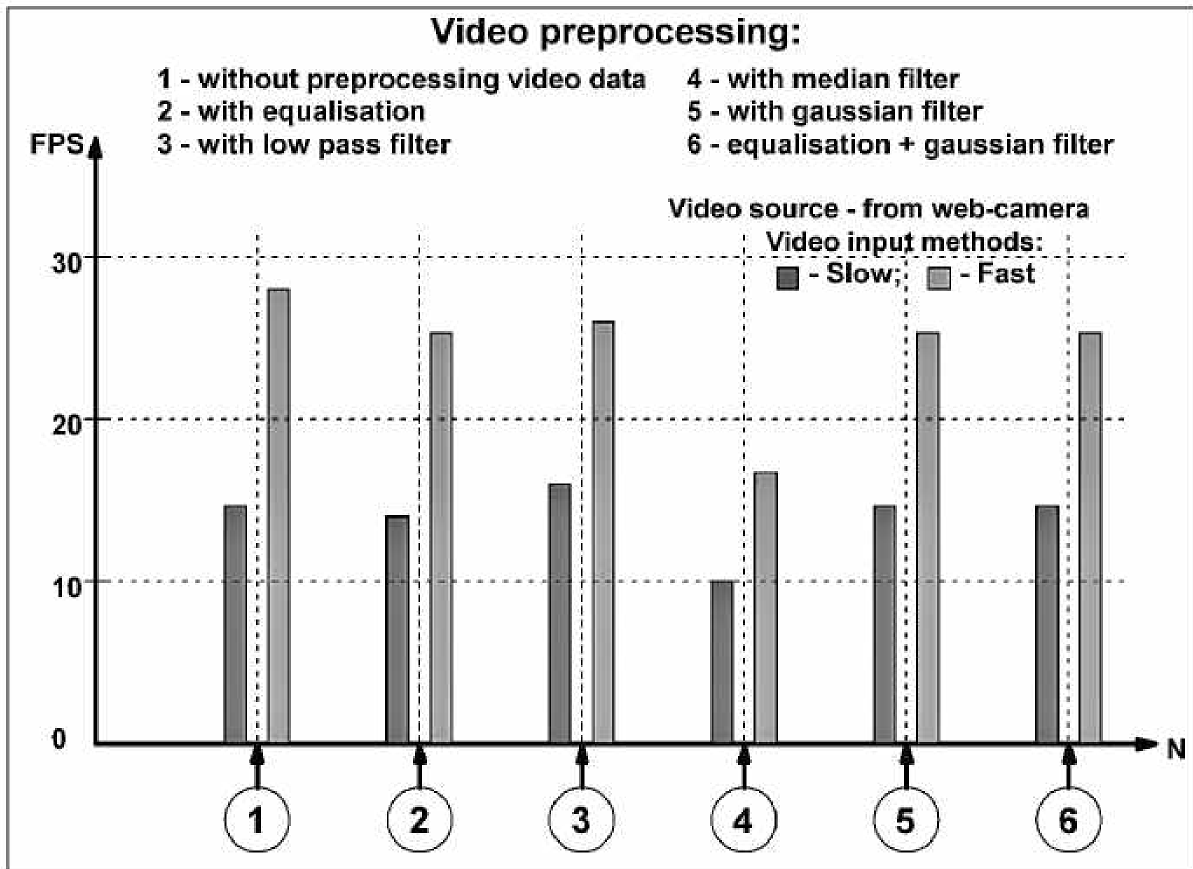


Рис. 4.11. Залежність швидкості введення відеоданих (FPS) при читанні відео з web-камери від характеру попереднього оброблення відеоданих

4.4. Алгоритм виявлення облич Віоли − Джонса

У 2001 року вперше опубліковано матеріали з побудови алгоритму виявлення облич (face detection) Віоли − Джонса [20, 21]. Цей метод, що містить усі основні елементи технології нейронних мереж, дуже швидко набув популярності і став справжнім проривом в області розпізнавання облич. Тепер його по праву можна називати класичним методом. Уважаємо, що читачам буде корисно ознайомитися як з теоретичною суттю методу, так і з варіантом практичної реалізації алгоритму Віоли − Джонса.

У запропонованому методі використано технологію змінного вікна. Для цього формується рамка розміром, меншим, ніж вихідне зображення. Її переміщають з деяким кроком по зображенню і з допомогою каскаду слабких класифікаторів визначають, чи є в цьому вікні обличчя. Метод змінного вікна є дуже популярним при вирішенні різних завдань комп'ютерного зору.

Суть цього методу полягає в інтегральному поданні зображення, побудові класифікаторів на базі алгоритму адаптивного бустингу (AdaBoost) і комбінуванні класифікаторів у каскадну структуру. У методі Віоли − Джонса вперше використано каскади вейвлетів (примітивів) Хаара, що являють собою розбивку заданої прямокутної області на набори різнотипних прямокутних підобластей. Усе це дало змогу побудувати детектор облич, що працює в режимі реального часу з досить високою якістю і можливістю виявлення більше одного обличчя на зображенні.

Однак існує велика кількість супутніх факторів, які обмежують ефективність роботи такого алгоритму. Головні з них:

- низька швидкість роботи алгоритмів навчання (для цього потребується велика кількість тестових даних і великий час навчання, який зазвичай вимірюється днями);
- невизначеність просторового положення обличчя на вже згадуваному зображенні (наприклад, кути нахилу голови більше 30°);
- проблеми з якістю освітлення сцени і багато інших обмежень.

Розглянемо окремі елементи алгоритму Віоли − Джонса більш докладно.

Узагальнену схему виявлення облич для алгоритму Віоли − Джонса зображено на рис. 4.12. Схема працює таким чином: перед початком виявлення алгоритм навчання на основі тестових зображень навчає базу даних, що складається з ознак, їх паритету і меж. Далі цей алгоритм шукає об'єкти на різних масштабах зображення, використовуючи створену базу даних. Алгоритм Віоли − Джонса на виході дає всі обличчя знайдених необ'єднаних об'єктів на різних масштабах. Наступне завдання полягає в прийнятті рішення про те, які зі знайдених об'єктів є в кадрі, а яких немає.

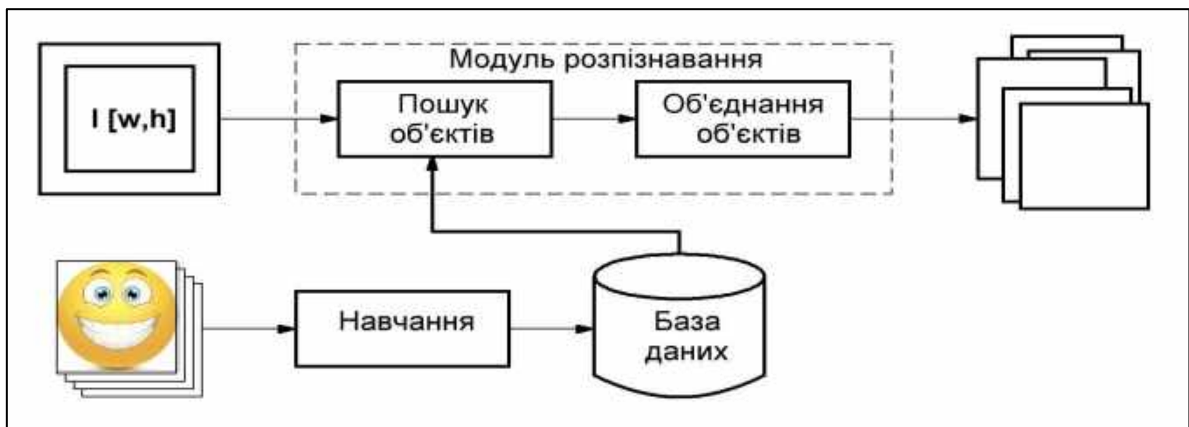


Рис. 4.12. Узагальнена схема детектування облич

Ознаки класу у вигляді примітивів Хаара були подані П. Віолою і М. Джонсом, і їх назвали так через подібність до вейвлетів Хаара. Вейвлет Хаара є згорткою одновимірного сигналу у вигляді різниці двох найближчих відліків (рис. 4.13).

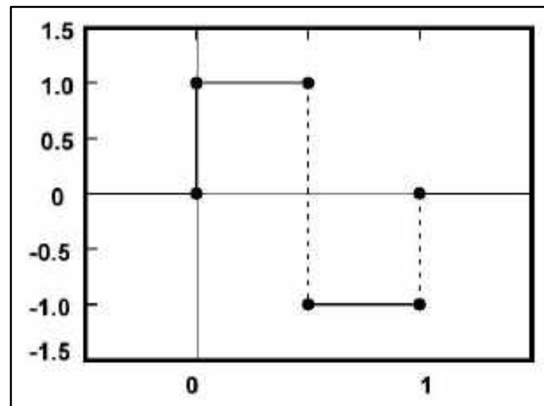


Рис. 4.13. Вейвлет Хаара

Ознака Хаара являє собою прямокутну область, розділену на дві суміжні області. Ознаки накладаються на зображення на різних позиціях і в різних масштабах (рис. 4.14). Ознака визначається як різниця між сумою пікселів з різних областей.

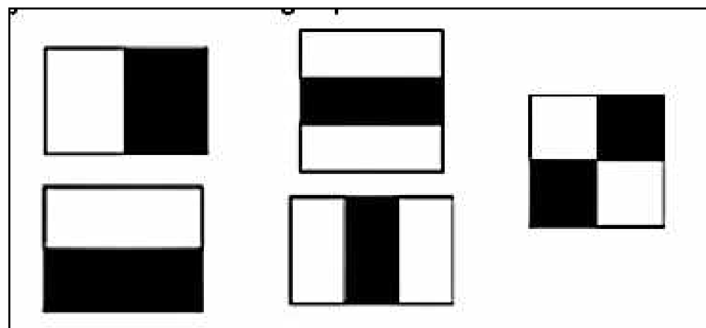
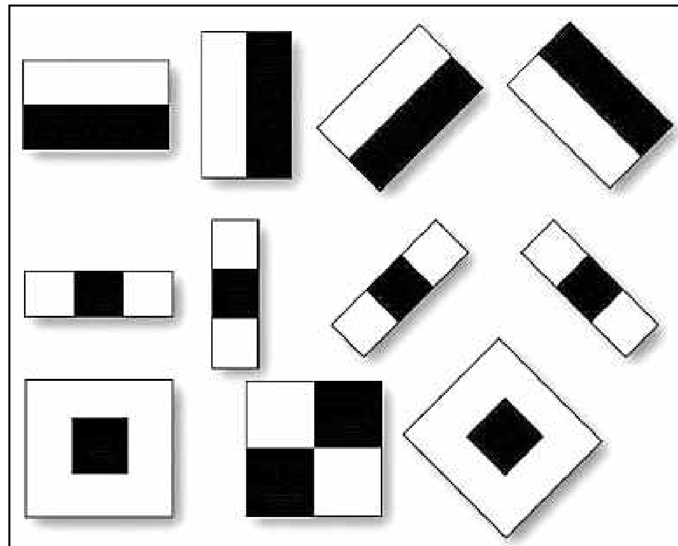


Рис. 4.14. Приклади ознак Хаара, що використовуються для розпізнавання облич

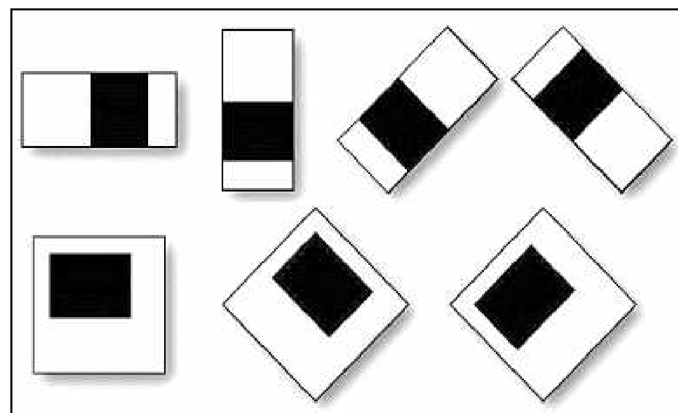
Кожна ознака може показати наявність (або відсутність) будь-якої конкретної характеристики зображення, наприклад межі або зміни текстур.

Наприклад, ознака з двох суміжних прямокутних областей може показати, де знаходиться межа між темним і світлим регіонами. Похилі ознаки вдало розширюють простір ознак, наприклад, дають змогу визначити наявність краю під кутом 45°.

У стандартному методі Віоли − Джонса використовуються прямокутні примітиви Хаара (рис. 4.15, *а*), у розширеному методі Віоли − Джонса в бібліотеці OpenCV − додаткові (рис. 4.15, *б*). На рис. 4.16 показано можливу їх комбінацію для пошуку облич.



а



б

Рис. 4.15. Основні (*а*) і додаткові (*б*) примітиви Хаара

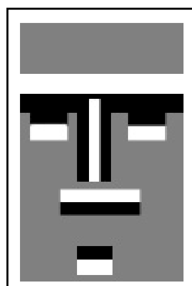


Рис. 4.16. Приклад комбінації примітивів Хаара для пошуку облич

Ознаки Хаара дають точкове значення перепаду яскравості по осях X і Y. Тому загальна ознака Хаара для розпізнавання облич являє собою набір двох суміжних прямокутників, які знаходяться вище очей і на щоках. Значення ознаки обчислюється за формулою

$$F = X - Y,$$

де X − сума значень яскравості точок, що закриваються світлою частиною ознаки, а Y − сума значень яскравості точок, що закриваються темною частиною ознаки.

Розрахунок суми значень інтенсивностей для кожної ознаки потребує значних обчислювальних ресурсів. П. Віола і М. Джонс запропонували використовувати інтегральне подання зображення (докладніше буде далі).

Схема навчання. Є тестова вибірка зображень. Розмір тестової вибірки − приблизно 10 000 зображень. На рис. 4.17 показано приклад навчальних зображень облич. Алгоритм навчання працює з зображеннями у відтінках сірого.

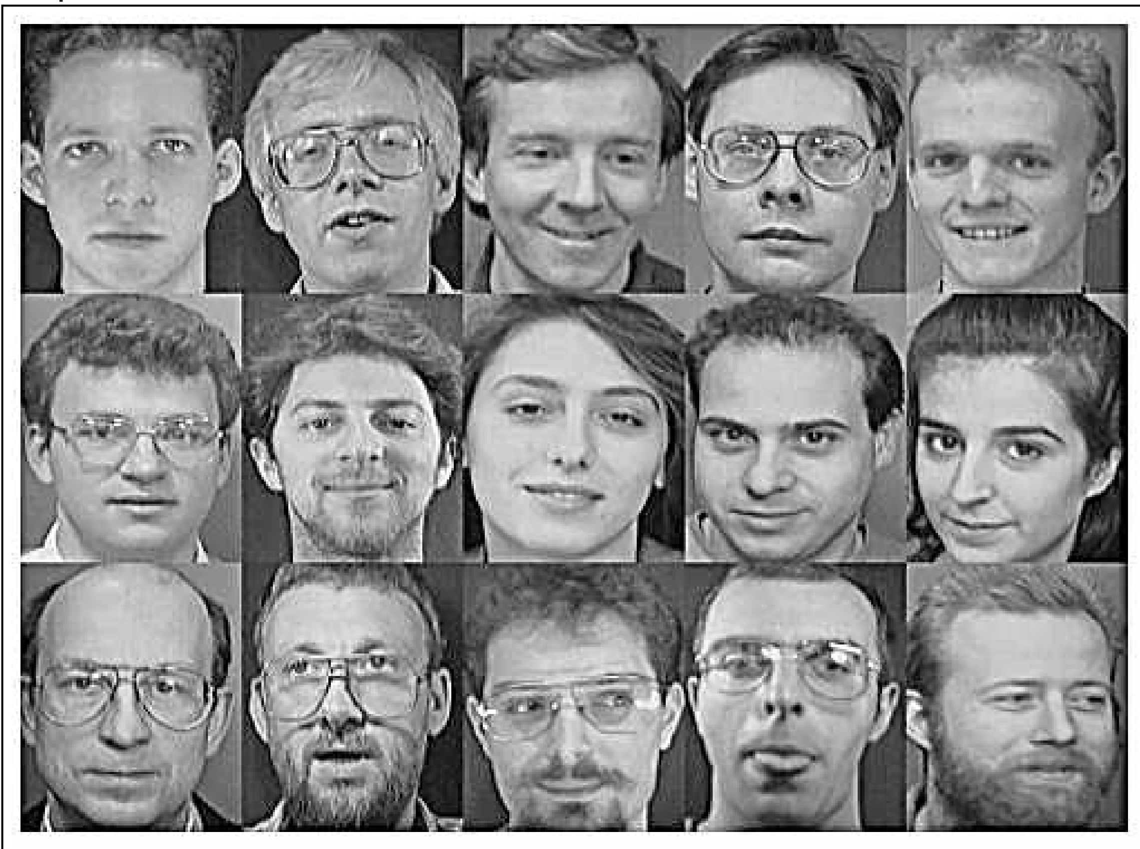


Рис. 4.17. Навчальні зображення облич

При розмірі тестового зображення 24 x 24 пікселі кількість конфігурацій однієї ознаки дорівнює близько 40 000 і залежить від мінімального розміру маски. У сучасних алгоритмах використовується близько 20 масок. Для кожної маски, кожної конфігурації тренується такий слабкий класифікатор, який дає найменшу помилку на всій тренувальній базі. Він додається в базу даних.

Таким чином, алгоритм навчається. І на виході алгоритму має місце база даних з **T** слабких класифікаторів. Узагальнену схему алгоритму навчання зображено на рис. 4.18.

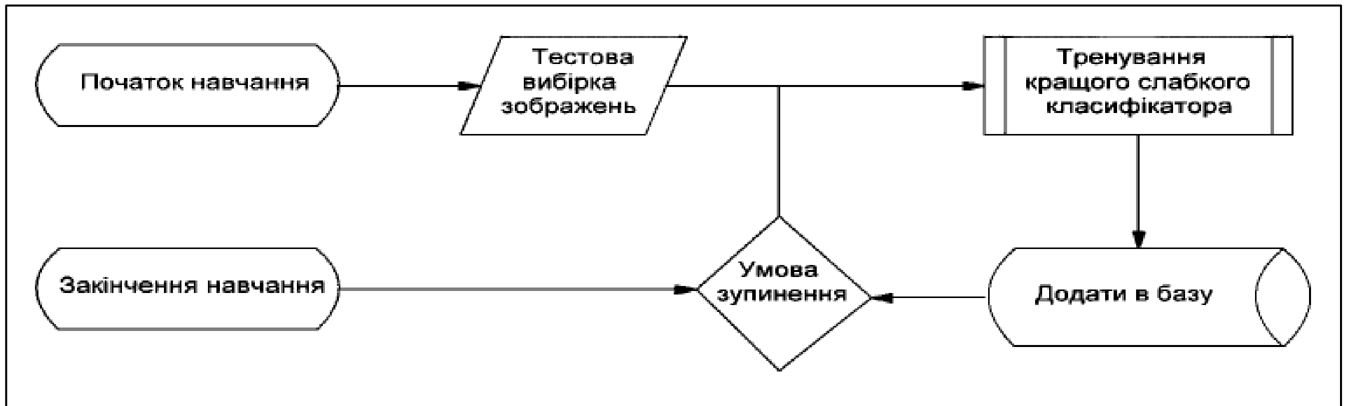


Рис. 4.18. Структура алгоритму навчання

Навчання алгоритму Віоли − Джонса − це машинне навчання алгоритму з учителем, причому тут можливою є така проблема, як перенавчання. У нашому випадку умовою зупинення навчання є досягнення заздалегідь заданої кількості слабких класифікаторів у базі.

Для алгоритму необхідно заздалегідь підготувати тестову вибірку з $\boldsymbol{l}$ зображень, що містять шуканий об'єкт, і $\boldsymbol{n}$ зображень, що не містять цього об'єкта. Тоді кількість усіх тестових зображень буде такою:

$$\boldsymbol{n} = \boldsymbol{l} + \boldsymbol{m}.$$

Множина

$$\boldsymbol{X} = \{\boldsymbol{x_1}, \boldsymbol{x_2}, \dots, \boldsymbol{x_n}\} -$$

це множина всіх тестових зображень, де для кожного зображення заздалегідь відомо, чи присутній шуканий об'єкт, чи ні, і чи відображено його у множині $\boldsymbol{Y}$:

$$\boldsymbol{Y} = \{\boldsymbol{y_1}, \boldsymbol{y_2}, \dots, \boldsymbol{y_n}\},$$

де $\mathbf{y_i} = \begin{cases} 1, & \text{якщо об'єкт є на зображенні } \mathrm{x_i}; \\ 0, & \text{якщо інакше.} \end{cases}$

Під ознакою $\boldsymbol{j}$ будемо розуміти структуру вигляду

$$\boldsymbol{j} = \{\text{маска}, \text{положенння}, \text{розмір}\}.$$

Тоді відгуком ознаки буде функція $\boldsymbol{f_j}(\boldsymbol{x})$, яка обчислюється як різниця інтенсивностей пікселів у світлій і темній областях. Слабкий класифікатор має вигляд

$$\boldsymbol{h_j}(\boldsymbol{x}) = \begin{cases} \mathbf{1}, \text{якщо } \boldsymbol{p_j f_j}(\boldsymbol{x}) < \boldsymbol{p_j \theta_j}; \\ \mathbf{0}, \text{якщо інакше.} \end{cases}$$

Завдання слабкого класифікатора − угадувати наявність об'єкта у більш ніж 50 % випадків.

Потім створюється дуже сильний класифікатор з використанням процедури навчання AdaBoost, що складається з $\boldsymbol{T}$ слабких класифікаторів і має вигляд

$$\boldsymbol{H}(\boldsymbol{x}) = \begin{cases} \mathbf{1}, \text{якщо} \sum_{t=1}^{T} \boldsymbol{a}_t \boldsymbol{h}_{j(t)}(\boldsymbol{x}) \geq \frac{\mathbf{1}}{\mathbf{2}} \sum_{t=1}^{T} \boldsymbol{a}_t \ ; \\ \mathbf{0}, \text{якщо інакше.} \end{cases}$$

Алгоритм AdaBoost є ефективним засобом навчання класифікації. Особливістю підходу, що використовується в AdaBoost, є принцип відбору й об'єднання набору простих і малоефективних властивостей в одне вирішальне правило, що має високу класифікувальну здатність. Отримання такого вирішального правила є основним завданням адаптації розпізнавання образів (у нашому випадку облич). Алгоритм навчання AdaBoost будує вирішальне правило у вигляді лінійної комбінації вихідних значень слабких класифікаторів, при цьому відбувається експоненціальне зменшення помилки зі збільшенням кількості властивостей на навчальному наборі.

Цільова функція навчання має такий вигляд:

$$\boldsymbol{T}, \boldsymbol{h}_{j\,(1)}, \boldsymbol{h}_{j(2)}, \ldots, \boldsymbol{a}_1, \ldots, \boldsymbol{a}_T = \boldsymbol{argmin} \sum_{i=1}^{n} \left| \boldsymbol{H}(\boldsymbol{x}_i, \boldsymbol{T}, \boldsymbol{h}_{j(1)}, \ldots, \boldsymbol{h}_{j(T)}, \boldsymbol{a}_1, \ldots, \boldsymbol{a}_T) \right|.$$

Інтегральне подання зображень має вигляд матриці, розміри якої збігаються з розмірами вихідного зображення $\boldsymbol{I}$, а кожний її елемент розраховується за формулою

$$\boldsymbol{II}(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=0, j=0}^{i \leq x, j \leq y} \boldsymbol{I}(\boldsymbol{r}, \boldsymbol{c}),$$

де $\boldsymbol{I}(\boldsymbol{r}, \boldsymbol{c})$ − яскравість пікселя вихідного зображення.

Кожен елемент матриці $\boldsymbol{II}(\boldsymbol{x}, \boldsymbol{y})$ є сумою пікселів у прямокутнику від (0, 0) до (x, y). Розрахунок такої матриці займає лінійний час. Для того щоб обчислити суму прямокутної області в інтегральному поданні зображення, необхідно виконати всього чотири операції звернення до масиву і три арифметичні операції. Це дає змогу швидко розраховувати ознаки Хаара для зображення в навчанні й розпізнаванні. Наприклад, розглянемо прямокутник ABCD на рис. 4.19. Суму всередині прямокутника ABCD можна виразити через суми й різниці суміжних прямокутників за формулою

$$\sum \boldsymbol{ABCD} = \boldsymbol{II}(\boldsymbol{A}) + \boldsymbol{II}(\boldsymbol{C}) - \boldsymbol{II}(\boldsymbol{B}) - \boldsymbol{II}(\boldsymbol{D}).$$

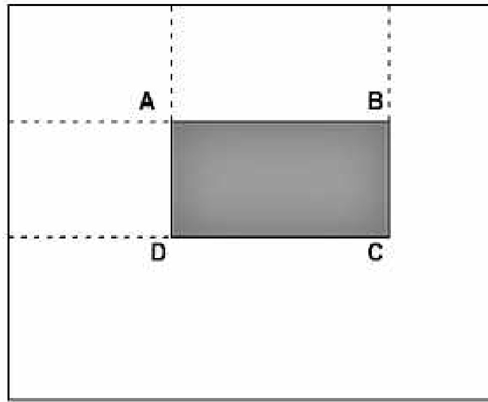


Рис. 4.19. Прямокутник як сума елементів матриці інтегрального подання зображення

Одна з головних переваг алгоритму Віоли − Джонса − використання інтегрального подання зображень, що дає змогу істотно підвищити швидкодію методу.

4.5. Програмна реалізація алгоритму виявлення облич Віоли − Джонса

Використовувані ресурси. При створенні алгоритмів автори використовували мову програмування Python і ресурси бібліотеки OpenCV. Такий вибір обумовлений відкритим доступом до програмних продуктів та їх сумісністю з операційними системами Windows, Linux і Android. При цьому підході робота алгоритмів легко реалізується на одноплатному комп'ютері Raspberry Pi. Особливість написання кодів мовою Python − формування необхідних можливостей програмування шляхом під’єднання зовнішніх бібліотек. Імпортовані ресурси бібліотеки OpenCV для вирішення нашого завдання є порівняно невеликими:

```
import cv2,
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches.
```

Використовуваний набір засобів програмування не дає значного навантаження на процесор і створює хороші передумови для оброблення даних в реальному масштабі часу. Для істотного збільшення швидкодії при обробленні відеоданих усі основні процедури реалізуються з допомогою стандартних функцій OpenCV, оптимізованих розробниками бібліотеки ще на етапі їх створення.

Стабілізація контрастності відеоданих. Виявлення і реєстрація облич з допомогою різних відеосенсорів зазвичай відбувається при різного роду перешкодах і постійному динамічному зміненні фону спостережуваної сцени. При цьому одним з головних негативних факторів є мінливість освітленості. Зазначимо, що це можуть бути як умови освітлення, що швидко змінюються, так і повільні його змінення, наприклад, через настання сутінків. Усе це призводить до погано контрольованих варіацій контрастності

кадрів і, отже, до погіршення якості оброблення.

Вирішення завдання стабілізації контрастності відеоданих докладно розглядалося в підрозд. 4.2.3 цієї роботи. Тому не станемо повторюватися, а лише зазначимо, що використання процедури еквалізації забезпечує вирівнювання гістограми розподілу яркостей і зводить показник середньої яскравості зображення до значення MB = 127 незалежно від того, яким цей показник був для вихідного зображення. Це дає можливість стабілізувати рівень середньої яскравості кадрів, а отже, спростити налаштування параметрів пристроїв виявлення й детектування облич та їх основних елементів (очей, носа, рота). Таке оброблення робить алгоритм більш стійким до впливу зовнішніх факторів. Приклад використання процедури еквалізації в роботі алгоритму виявлення обличчя та очей показано на рис. 4.20.

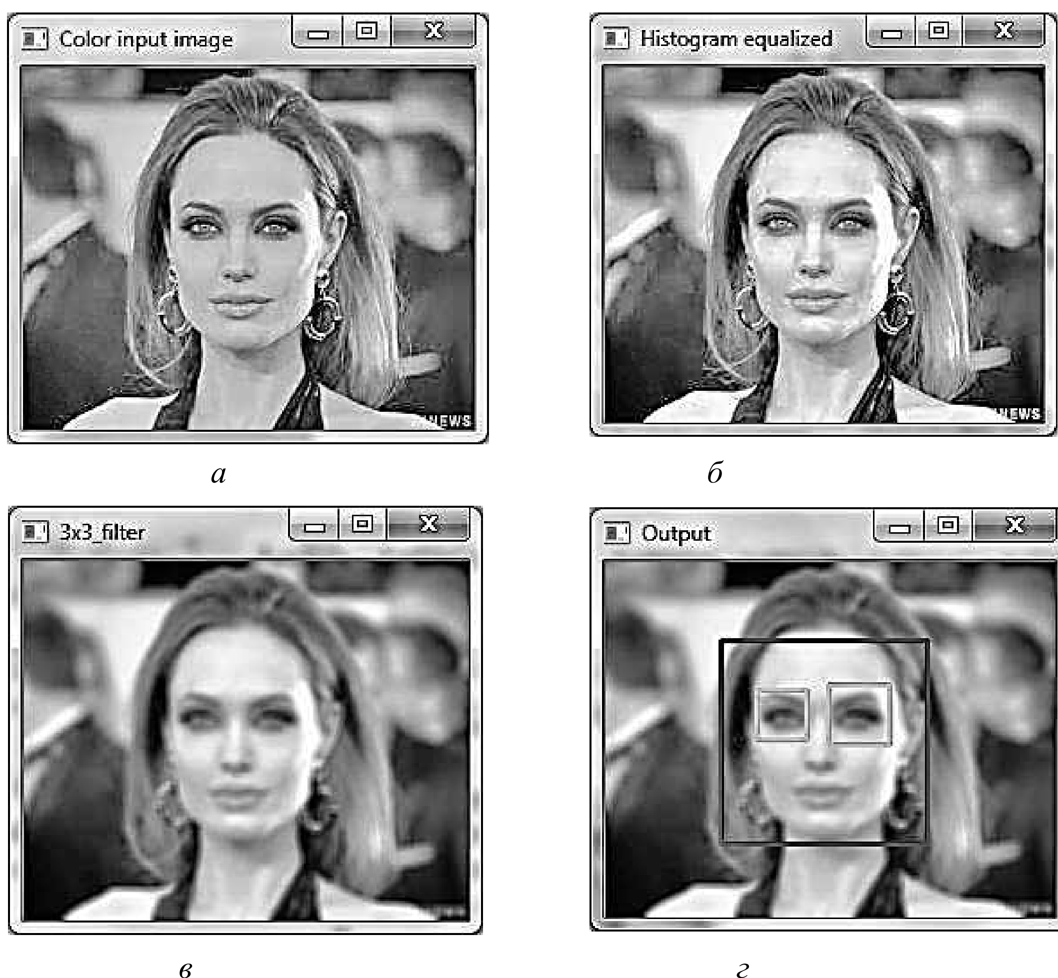


Рис. 4.20. Процедура еквалізації: Input image (*а*); histogram equalized (*б*); image after filtration (*в*); image after detection (*г*)

Виявлення й детектування облич на базі методу Віоли − Джонса. Як зазначалося раніше, сучасні системи детектування облич часто орієнтовані на використання методу Віоли − Джонса, який базується на інтегральному поданні зображення, побудові класифікаторів на базі алгоритму адаптивного бустингу і створенні класифікаторів у вигляді каскадної струк-

тури. Найбільш трудомістким є процес навчання каскадів Хаара з допомогою алгоритму машинного навчання AdaBoost. Однак для каскадних класифікаторів Хаара існує велика кількість уже навчених каскадів, у тому числі в стандартному постачанні бібліотеки OpenCV. Її установчий пакет містить набір готових навчених класифікаторів, збережених у вигляді файлів з розширенням «*.xml», у якому є класифікатори для пошуку як обличчя, так і його окремих частин (очей, рота, носа).

При синтезі алгоритмів виявлення найбільш продуктивним є використання стандартних (заздалегідь навчених) класифікаторів. У зв'язку з цим уважаємо за доречне навести для довідки список основних попередньо навчених класифікаторів:

- `haarcascade_eye_tree_eyeglasses.xml;`
- `haarcascade_fullbody.xml;`
- `haarcascade_mcs_lefteye.xml;`
- `haarcascade_profileface.xml;`
- `haarcascade_eye.xml;`
- `haarcascade_lefteye_2splits.xml;`
- `haarcascade_mcs_mouth.xml;`
- `haarcascade_righteye_2splits.xml;`
- `haarcascade_frontalface_alt2.xml;`
- `haarcascade_lowerbody.xml;`
- `haarcascade_mcs_nose.xml;`
- `haarcascade_smile.xml;`
- `haarcascade_frontalface_alt_tree.xml;`
- `haarcascade_mcs_eyepair_big.xml;`
- `haarcascade_mcs_rightear.xml;`
- `haarcascade_upperbody.xml;`
- `haarcascade_frontalface_alt.xml;`
- `haarcascade_mcs_eyepair_small.xml;`
- `haarcascade_mcs_righteye.xml;`
- `haarcascade_frontalface_default.xml;`
- `haarcascade_mcs_leftear.xml;`
- `haarcascade_mcs_upperbody.xml.`

Не зупиняючись детально на призначенні і властивостях окремих класифікаторів, зазначимо, що особливий інтерес серед них становлять ті, які дають змогу визначити положення очей. Це дуже корисна процедура, оскільки дає можливість оцінити відстань між зіницями, завдяки чому в подальшому аналізі можна усувати фактори, пов'язані з нахилом голови. Класифікатори, навчені пошуку очей, є дуже чутливими до наявності окулярів. Здебільшого дають збій, особливо для окулярів, що повністю приховують очі. У таких випадках слід використовувати більш ефективний класифікатор для виявлення очей `haarcascade_eye_tree_eyeglasses.xml`, навче-

ний пошуку на зображенні очей під окулярами. Його можна використовувати як запасний варіант під час збою в роботі звичайного класифікатора.

Найбільш ефективними є методи пошуку основних елементів у вже виділеній області обличчя, оскільки це локалізує регіон пошуку, скорочує час аналізу і значно знижує ймовірність помилкових спрацьовувань. Вибір областей інтересу для виявлення очей, носа й рота потребує оптимізації положення й розмірів області інтересу для кожного елемента. Приклад такого розбиття виділеної області на зони пошуку окремих елементів обличчя в нашому алгоритмі показано на рис. 4.21. Розміри областей пошуку нашого алгоритму визначалися експериментально, але, якщо необхідно, їх можна оптимізувати додатково. Зверніть увагу, що існує дві можливості виявлення очей − виділення загальної для двох очей області інтересу (або області пошуку) і пошук і виявлення лівого й правого очей окремо. В OpenCV для цього передбачено відповідні класифікатори. Який з цих методів є кращим, визначимо експериментальним шляхом.

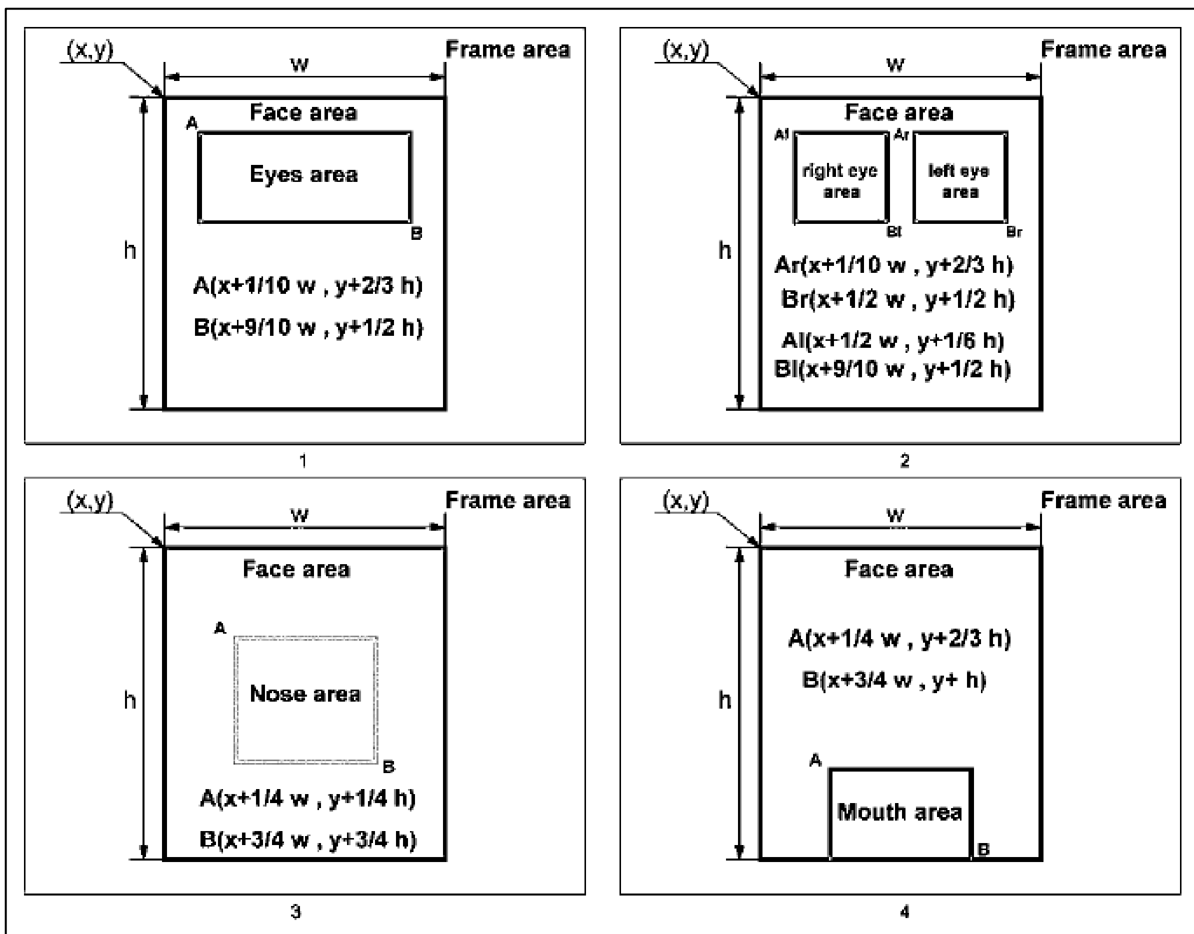


Рис. 4.21. Зони пошуку облич та їх елементів у кадрі (координати окремих зон пошуку)

Показники якості роботи алгоритмів і методи їх використання. Об'єктивним критерієм ефективності роботи будь-якого алгоритму детектування облич та їх елементів у кадрах відеопослідовності є показник імовірності правильного виявлення обличчя за умови його наявності в кадрі. Процедура виявлення обличчя в поточному кадрі із застосуванням

відповідного каскадного класифікатора в разі успіху завершується побудовою прямокутника з допомогою функції OpenCV

```
cv2.rectangle(frame,(x,y)(x+w,y+h),255,0,0),2)
```

Успішне детектування обличчя в програмному коді алгоритму має завершуватися появою логічної 1, в іншому випадку — появою логічного 0. У нашій роботі для створення стійкого критерію якості правильного виявлення облич розраховувалася ймовірність $\boldsymbol{P_n}$, отримана за результатами підрахунку у вікні розміром 100 кадрів кількості успішних виявлень і нормована на розмір вікна. При стандартній швидкості змінення кадрів відеокамери 30 $с^{-1}$ вікно має постійний час змінення $\boldsymbol{P_n}$ приблизно 3,3 с. Якість роботи детектора можна вважати задовільним при $\boldsymbol{P_n} \geq 0,9$. За показником імовірності правильного виявлення облич у кадрі легко аналізувати вплив різних факторів (у тому числі змінення освітленості, геометричних факторів та ін.) на якість роботи алгоритмів виявлення й детектування облич.

Аналогічним чином розраховуються й імовірності правильного виявлення очей, носа, рота у відповідних областях. Ці показники обчислюються як умовна ймовірність такої події за умови правильного виявлення всього обличчя. З імовірністю, прийнятною для практичного використання, події виявлення окремих елементів обличчя можна вважати незалежними, а отже, імовірність правильного виявлення всіх елементів оцінювати як добуток імовірностей виявлення цих елементів.

Для адаптації пропонованих алгоритмів детектування облич до умов змінення зовнішнього освітлення сцени запропоновано організувати два канали оброблення відеоданих (рис. 4.22). В одному з каналів вихідний відеопотік використовується безпосередньо для виявлення облич, а в іншому − проводиться його попередня еквалізація для зведення яскравості кадру до середнього значення. На виходах обох каналів оброблення в ковзних вікнах розміром 100 кадрів обчислюються ймовірності правильного виявлення облич та їх елементів, а потім їх зіставляють у блоці порівняння. Залежно від знака різниці ймовірностей $\Delta\boldsymbol{P_n}$ з допомогою петлі зворотного зв'язку на вихід системи передаються результати виявлення облич з того каналу, у якому умови освітлення є більш комфортними для роботи алгоритму виявлення облич.

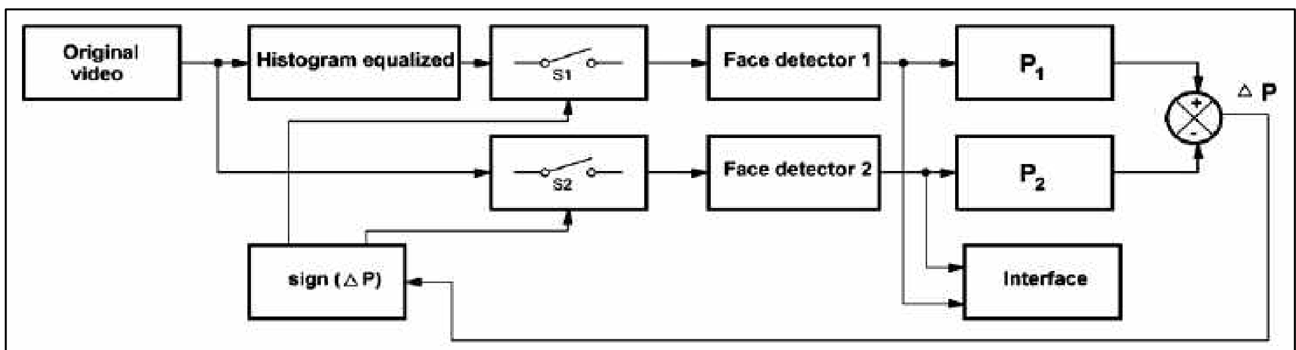


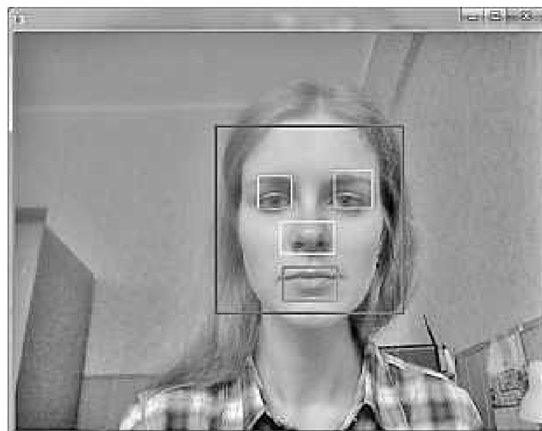
Рис. 4.22. Узагальнена структура адаптивного алгоритму детектування облич

Практична реалізація алгоритмів. Робочі версії алгоритмів виявлення облич створювалися з урахуванням описаних вище підходів і ресурсів. Було поставлено завдання виявлення облич, очей, носа і рота. При цьому виявлення очей здійснювалося двома способами − з використанням класифікатора для детектування очей у загальній зоні пошуку і класифікаторів роздільного виявлення (правого й лівого очей окремо). Далі порівняємо ці два способи за показником імовірності виявлення очей.

Для економії ресурсів і простоти реалізації алгоритмів у нашому проекті використовувався набір попередньо навчених каскадних класифікаторів Хаара для відповідних елементів обличчя, імпортований з бібліотеки OpenCV. Перелік цих класифікаторів:

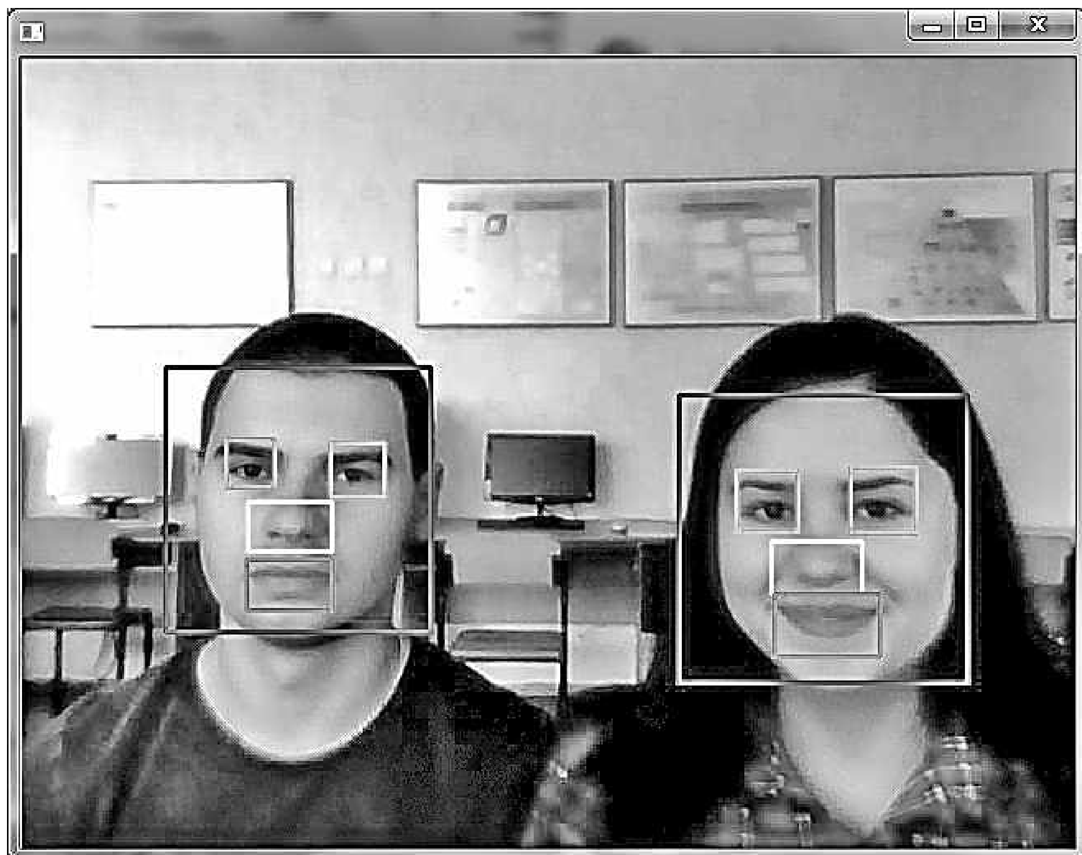
- `face_cascade = cv2.CascadeClassifier('haarcascade_frontalface_default.xml');`
- `eye_cascade = cv2.CascadeClassifier('haarcascade_eye.xml')` події виявлення окремих елементів обличчя;
- `right_eye_cascade = cv2.CascadeClassifier('haarcascade_righteye_2splits.xml');`
- `left_eye_cascade = cv2.CascadeClassifier('haarcascade_lefteye_2splits.xml');`
- `nose_cascade = cv2.CascadeClassifier('haarcascade_nose.xml');`
- `mouth_cascade = cv2.CascadeClassifier('haarcascade_mouth.xml').`

Результати виявлення одного або декількох облич з допомогою запропонованих алгоритмів показано на рис. 4.23. Слід зазначити, що найбільш стійкі результати виявлення обличчя та його елементів спостерігаються за умови наявності джерела освітлення сцени, фронтально розташованого відносно обличчя. При сприятливому геометричному факторі процедури виявлення передбачається нахил голови на кут не більше 30°.



а

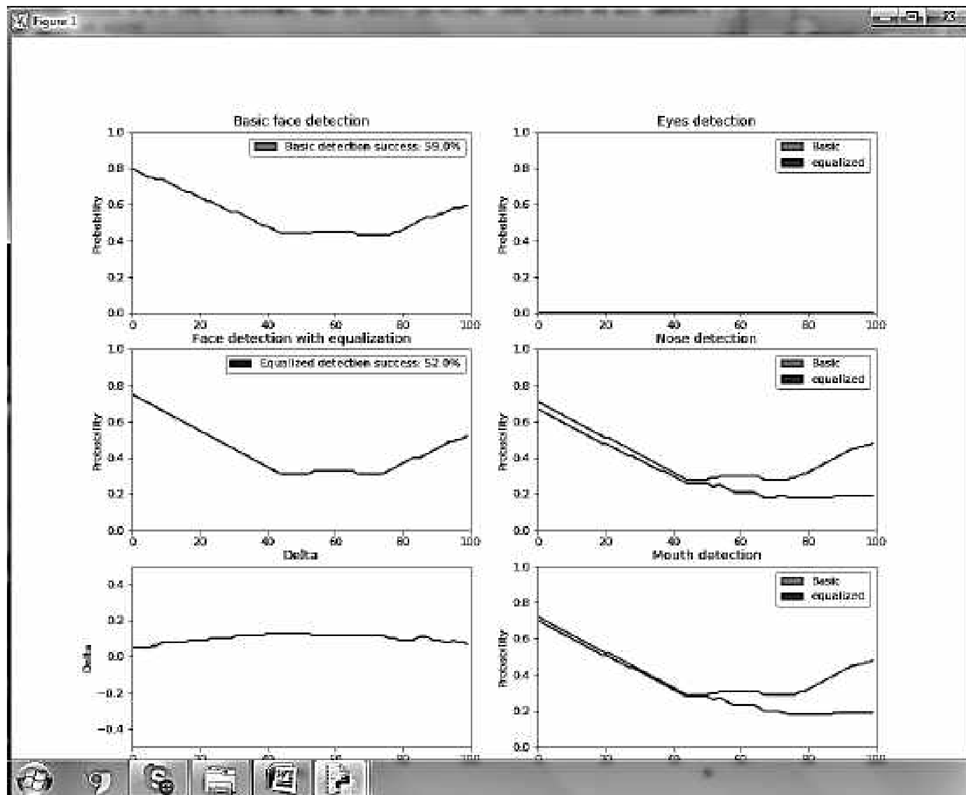
Рис. 4.23. Результати виявлення облич та їх елементів у кадрі



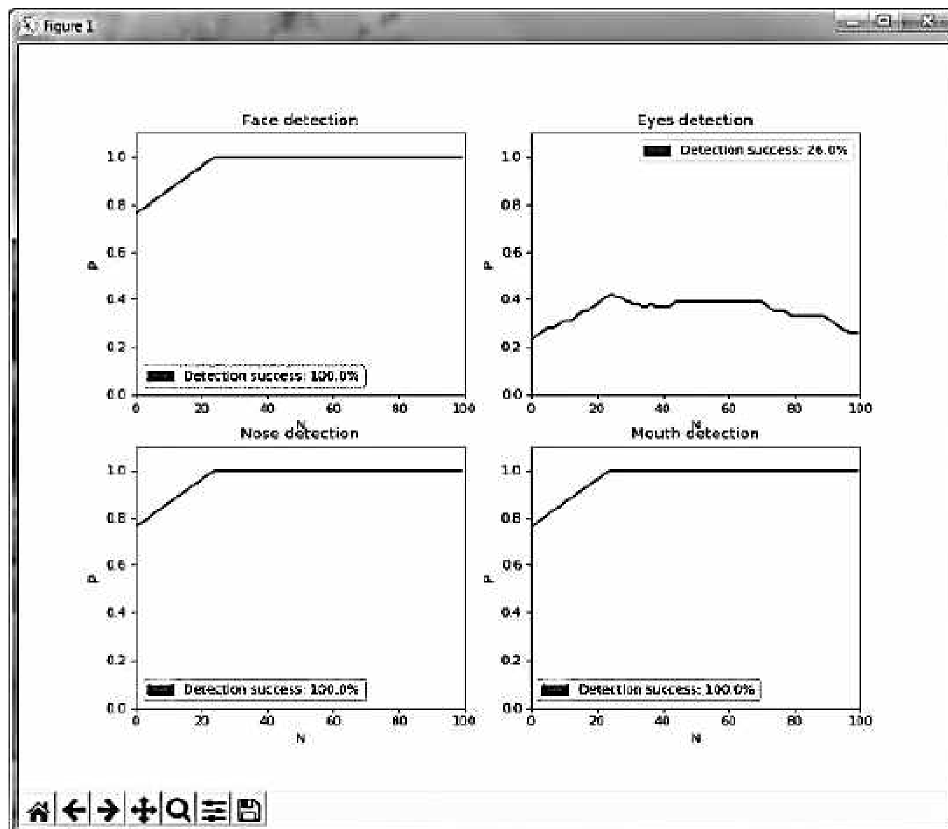
б

Рис. 4.23. Закінчення

Результати експериментальних досліджень. Специфічні властивості алгоритмів виявлення облич, що базуються на використанні навчених каскадних класифікаторів, визначають імовірнісний характер їх функціонування. Тому основним показником якості таких алгоритмів у нашій роботі вважається ймовірність правильного виявлення обличчя та його елементів. Для наочності отриманих результатів у першому варіанті побудови алгоритму виявлення синтезовано спеціальну графічну форму, усередині якої в окремих вікнах відстежуються в динаміці ймовірності виявлення осіб при роботі без застосування процедури еквалізації (червоні криві) і з її застосуванням (сині криві) для обличчя та його елементів (очей, носа і рота). Крім того, у лівому нижньому вікні цієї форми відображається поточна різниця ймовірностей правильного виявлення (при еквалізації кадру відео і без неї). Вигляд цієї узагальненої звітної форми зображено на рис. 4.24, *а*. Аналогічним чином було побудовано графічну форму для поточного оцінювання ефективності виявлення і для другого алгоритму. Форма містить чотири вікна, у яких відображаються поточні значення ймовірностей правильного виявлення обличчя, очей, носа і рота (рис. 4.23, *б*). Ці засоби візуалізації поточних процесів виявлення є досить ефективними, але не дають відповіді на основне запитання, якою є ефективність роботи алгоритму при тривалому аналізі відеопослідовності.



а



б

Рис. 4.24. Вікна поточного контролю ймовірностей правильного виявлення облич та їх елементів

Для узагальненого оцінювання ефективності роботи алгоритмів застосовуються два методи − якісний (візуальний) і кількісний. У першому випадку проводиться візуальне спостереження за різнокольоровими прямокутниками, що обмежують виявлене обличчя та його елементи (очі, ніс і рот). Якщо процес їх відтворення сприймається як безперервний, то якість детектування можна вважати прийнятною (~ 100 %). Однак інерційність зору спостерігача при високій частоті змінення кадрів (30 $с^{-1}$) не дає змоги візуально сприймати пропуски виявлення окремих кадрів. Це може істотно спотворити результати тестування. Тому для кількісного оцінювання ефективності в програмі для всіх кадрів відеопослідовності формується одновимірний масив (1 − за фактом виявлення обличчя в кадрі, 0 − у разі пропуску обличчя), за яким будуються ймовірнісні характеристики якості роботи алгоритму. Однак, слід мати на увазі, що для цього необхідною є достовірно анотована тестова відеопослідовність. Найпростіший варіант такої послідовності − відеоряд, у кожному кадрі якого є обличчя для виявлення, дотримуються умови освітлення (фронтальне джерело світла) і підтримується геометричний фактор (нахил голови менше 30°). Зрозуміло, що можна використовувати й більш складний тестовий відеоряд, у якому в певні інтервали часу облич у кадрі немає. Тоді можна оцінити не тільки ймовірності правильного виявлення обличчя, але й імовірності помилкового виявлення (виявлення обличчя, коли його в кадрі немає).

На рис. 4.25 зображено спеціальну форму узагальненого оцінювання ефективності роботи алгоритму виявлення облич та їх головних елементів для відеоряду тривалістю 1000 кадрів. Тривалість такого запису становить ~ 33 с. Для процедур виявлення всіх елементів побудовано періодограми, де для кожного з N кадрів у відповідність ставиться 1 (якщо виявлено) і 0 (при невиявленні) відповідного елемента обличчя. Крім того, для кожної періодограми обчислюється ймовірність правильного виявлення на вибірці обсягом 1000 кадрів. Така форма не тільки дає змогу отримати узагальнені оцінки ймовірностей виявлення, а й наочно показує, на яких кадрах виявлення не було виконано.

Для оцінювання ефективності зроблено кілька тестових відеозаписів, що відповідають заданим вимогам. Записи виконувалися у форматі * avi з розміром кадрів 480 x 640 пікселів з нерухомо встановленої камери. Істотні артефакти, обумовлені рухом людини в кадрі, було виключено. Відстань від обличчя до об'єктива відеокамери становило ~ 1 м. Для контролю рівня освітленості сцени при реєстрації тестових відеозаписів використовувався люксометр Ю-16 з фотоелементом Ф-102, який дає змогу фіксувати рівень освітленості з точністю ±10 %.

Оскільки запропоновані алгоритми мають забезпечувати автоматичну стабілізацію яскравості кадрів аналізованої відеопослідовності незалежно від рівня освітленості, насамперед зіставлялися результати виявлення облич у записах з різними рівнями освітленості сцени. Результати цього експерименту було зведено в таблицю.

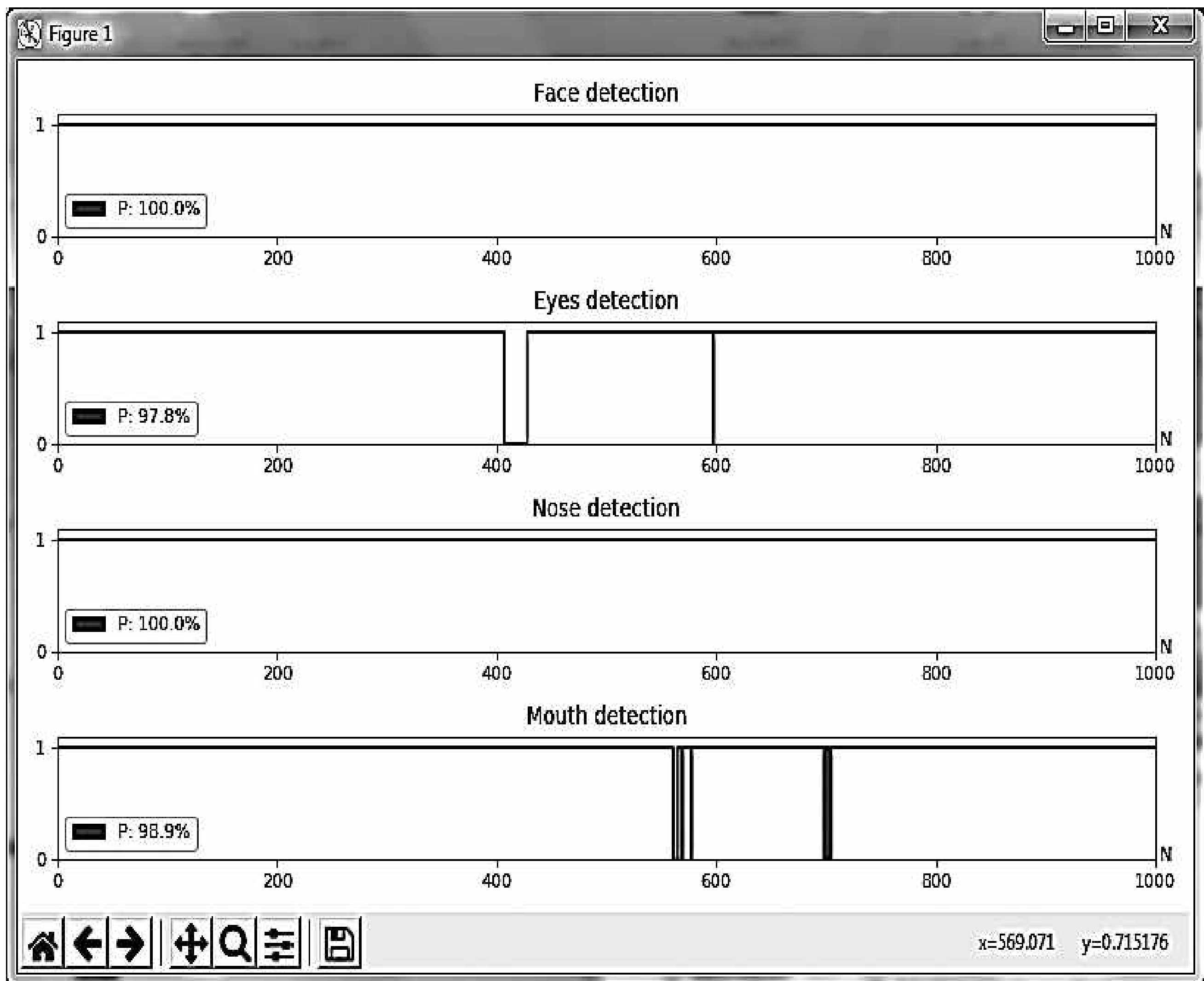


Рис. 4.25. Вікно оцінювання ефективності виявлення обличчя та його головних елементів за відеоданими

Аналіз табличних даних показав, що ймовірність правильного виявлення окремих елементів обличчя підвищується з підвищенням рівня освітленості при умові, що оброблення даних проводиться без автоматичного регулювання середньої яскравості кадру. У випадках же, коли використовується процедура еквалізації для регулювання яскравості, імовірність виявлення залишається стабільною. Іншими словами, при низькому рівні освітленості ефективність алгоритмів з автоматичним регулюванням яскравостіє є вищою, а при яскравому освітленні сцени якість роботи обох типів алгоритмів є приблизно однаковою. Необхідно відзначити високу якість роботи запропонованих алгоритмів, оскільки схожі сучасні технічні рішення забезпечують імовірність правильного виявлення на рівні 95−97 %.

Показники ефективності пропонованих алгоритмів наведено в табл. 4.1. Крім того, у роботі було проведено дослідження ефективності роботи алгоритмів залежно від відстані між обличчям і відеореєстратором. Початкова відстань дорівнювала 0,7 м, а кінцева − 2,5 м. Решта умов зйомки були аналогічними попереднім дослідженням.

Таблиця 4.1

Показники ефективності пропонованих алгоритмів

Without equalization				
Scene illumination, Lx	Face detection, P %	Eyes detection, P %	Nose detection, P %	Mouth detection, P %
100	100	96,5	98,1	97,2
150	100	97,2	98,5	97,5
200	100	97,8	98,7	97,7
With equalization				
Scene illumination, Lx	Face detection, P %	Eyes detection, P %	Nose detection, P %	Mouth detection, P %
100	100	97,1	98,6	97,4
150	100	97,2	98,6	97,45
200	100	97,1	98,62	97,4

Зазначимо, що процедура виявлення обличчя є нечутливою до збільшення відстані між обличчям і відеокамерою. Але збільшення цієї відстані більш ніж на 1,5 м призводить до переривання процедури виявлення елементів обличчя (очей, носа і рота). Це обмеження пов'язане з відносним зменшенням розмірів обличчя в кадрі й може бути усунуто шляхом адаптивного керування його розмірами.

Розглянутий приклад показує, що використання алгоритму Віоли − Джонса з набором готових і заздалегідь навчених класифікаторів у вигляді файлів з розширенням «* .xml» з бібліотеки OpenCV дає змогу вирішити завдання детектування облич з якістю, цілком прийнятною для практичного застосування.

Повний лістинг програми `Face_detection_Viola_Jones` наведено в дод. 2.

4.6. Виявлення й детектування облич з використанням бібліотеки Caffe

Наведемо альтернативний варіант вирішення завдання детектування облич з використанням фреймворку Caffe (ця бібліотека сконцентрована на ефективній реалізації алгоритмів глибокого навчання) для порівняння з методом машинного навчання (алгоритм Віоли − Джонса). Це завдання будемо вирішувати із застосуванням функції DNN і керувальних ресурсів OpenCV. Основні керувальні функції для роботи із зображеннями (наприклад, функція `cv2.dnn.blobFromIma`) досить докладно описано в підрозд. 3.2. Тому не будемо повторюватися й докладно обговорювати їх властивості, а наведемо повний лістинг програми `face_detection.py`, яка виконує виявлення облич у відеопотоці як за допомогою алгоритму Віоли − Джонса, так і з використанням алгоритму на базі фреймворку Caffe. Для завершеності процесу дослідження в цій програмі також вирішено завдання анонімізації виявлених облич (див. підрозд. 4.8).

Лістинг програми «face_detection.py» має такий вигляд:

```
# імпорт бібліотеки imutils для методів зчитування даних
from imutils.video import FPS
from imutils.video import VideoStream
from imutils.video.pivideostream import PiVideoStream
from imutils.video import FileVideoStream

# імпорт бібліотек оброблення зображень
from picamera.array import PiRGBArray
from picamera import PiCamera
import numpy as np
import argparse
import imutils
import time
import cv2
import datetime

# імпорт файлів детектування й анонімізації
from face_blurring import anonymize_face_pixelate
from face_blurring import anonymize_face_simple
from imutils import face_utils

# ініціалізація констант, які є параметрами виконуваних функцій
global fps,rwidth,method,start_streams
global nwidth,nheigh
nwidth=[1280,1280,1200,1024,960,800,640,640,480,640,320]
nheigh=[960,720,600,768,540,600,480,360,320,240,240]
rwidth=6
method=5
```

```
anonymaze=0
neiro=1
video_input=False
minNeigbors=5
const_confidence=0.5
const_factor=3.0
const_block=20
name_file=0
# ініціалізація каскадів Хаара і функцій Caffe з бібліотеки OpenCV
face_cascade=cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
eye_cascade=cv2.CascadeClassifier('haarcascade_eye.xml')
net =
cv2.dnn.readNetFromCaffe("deploy.prototxt.txt", "res10_300x300_ssd_iter_140000.caffemodel")

# функція детектування облич з допомогою алгоритму Віоли – Джонса
def viola_face(current_frame):
    face_detected = False
    gray = cv2.cvtColor(current_frame, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray, 1.3,
minNeigbors)
    for (x, y, w, h) in faces:
        face_detected = True

# додавання рамки навколо виявленого обличчя
        If anonymaze==0:

cv2.rectangle(current_frame,(x,y),(x+w,y+h),(0,0,255),2)

# якщо параметр є більшим від нуля, то викликається метод анонімізації
        else:
            (startX, startY, endX, endY) = (x,y,x+w,y+h)
            Face = current_frame[startY:endY, startX:endX]
            If anonymaze==1:
                Face = anonymize_face_simple(face, fac-
tor=const_factor)
            If anonymaze==2:
                Face =
anonymize_face_pixelate(face,blocks=const_block)
            current_frame[startY:endY, startX:endX] = face
    return current_frame,face_detected
# Функція детектування облич з допомогою алгоритму Caffe
Def detecting_caffe(frame):
    (h, w) = frame.shape[:2]
```

```
    Blob =
cv2.dnn.blobFromImage(cv2.resize(frame,(300,300)),1.0,(300,300),
(104.0,177.0,123.0))
    net.setInput(blob)
    detections = net.forward()
    detect=False
    for i in range(0, detections.shape[2]):
        confidence = detections[0, 0, i, 2]
        if confidence < const_confidence:
            continue
        box = detections[0, 0, i, 3:7] * np.array([w, h, w, h])
        (startX, startY, endX, endY) = box.astype("int")
        detect=True

# додавання рамки навколо виявленого обличчя
        If anonymaze==0:
            Y = startY - 10 if startY - 10 > 10 else startY +
10
            cv2.rectangle(frame,(startX, startY),(endX,
endY),(0,0,255),2)
        else:
# якщо параметр анонімізації є більшим від нуля, то
викликається метод анонімізації
            face = frame[startY:endY, startX:endX]
            if anonymaze==1:
                face = anonymize_face_simple(face, fac-
tor=const_factor)
            if anonymaze==2:
                face =
anonymize_face_pixelate(face,blocks=const_block)
            frame[startY:endY, startX:endX] = face
    return frame,detect

# Функція основного методу, що викликає режим уведення
відеоданих щодо початкового параметра
def body():
    if method==1:
        read_slow()
    if method==2:
        read_fast()
    if method==3:
        read_videostream()
    if method==4:
        read_picamera()
    if method==5:
        read_pivideostream()

# функція для можливості запускати додаток через консоль
```

```
def arguments():
    global args
    ap = argparse.ArgumentParser()
    ap.add_argument("-v","--video",required=True,help="path to
input video file")
    args = vars(ap.parse_args())

# Функція відображення додаткової інформації про сесію
Def print_information():
    print("[INFO] Name files" +str(name_file))
    print("[INFO] Number method"+str(method)+"number method
neiro"+str(neiro))
    print("[INFO] elasped time: {:.2f}".format(fps.elapsed()))
    print("[INFO] approx. FPS: {:.2f}".format(fps.fps()))
    print("[INFO] approx. Frame: "+str(int(fps._numFrames)))
    print("[INFO] Width",nwidth[rwidth],"[INFO]
Heigh",nheigh[rwidth])
    print("[INFO] frame detecting %",str(((int(fps._numFrames)-
fps._frame_detecting)*100)/fps._numFrames))

# Функція попереднього оброблення зображення
# також цей блок забезпечує додавання тексту на зображенні

def transform_frame(frame):
    if (frame is None):
        return
    # frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    # frame = np.dstack([frame, frame, frame])
    # frame = cv2.cvtColor(frame, cv2.COLOR_BGR2YUV)
    # frame[:,:,0] = cv2.equalizeHist(frame[:,:,0])
    # frame = cv2.cvtColor(frame, cv2.COLOR_YUV2BGR)

    if (method)<4: frame =
imutils.resize(frame,width=nwidth[rwidth],height=nheigh[rwidth]
)

    if(neiro!=0):
        if(neiro==1):
            frame,fps._detected=viola_face(frame)
        else:
            frame,fps._detected=detecting_kofi(frame)


name_meth=["Slow Method","Fast Method","Videostream","Picamera"
, "PiVideostream"]
    Text=name_meth[method-1]+"Resolution"
+str(int(nwidth[rwidth]))+"x"+str(int(nheigh[rwidth]))
    Text_FPS="frame"+str(int(fps._numFrames))+" "+"
"+str(fps_1)+"second"
```

```
v2.putText(frame,Text_FPS,(10,400),cv2.FONT_HERSHEY_SIMPLEX,0.5
, (0,255,0),2)
    frame =
imutils.resize(frame,width=nwidth[rwidth],height=nheigh[rwidth]
)
    return frame

# Функція для закінчення сесії відповідно до таймера
def out_read(start_stream):
    return ((datetime.datetime.now()-
start_stream).total_seconds()>20)

# метод зчитування відеоданих для web-камери без запису у стрім
def read_slow():
    if(video_input): stream = cv2.VideoCapture(0)
    else: stream = cv2.VideoCapture(name_file)
    start_stream = datetime.datetime.now()
    while True:
        (grabbed, frame) = stream.read()
        If not grabbed:
            break
        frame=transform_frame(frame)
        cv2.imshow("Frame", frame)
        cv2.waitKey(1)
        if (out_read(start_stream)):
            break
        fps.update()
    stream.release()

# метод зчитування відеоданих для web-камери із записом у стрім
def read_videostream():
    if(video_input): fvs = VideoStream(0).start()
    else: fvs = VideoStream(name_file).start()
    time.sleep(2.0)
    start_stream = datetime.datetime.now()
    while True:
        if (out_read(start_stream)):
            break
        frame = fvs.read()
        frame=transform_frame(frame)
        cv2.imshow("Frame", frame)
        cv2.waitKey(1)
        fps.update()
    fvs.stop()

#метод зчитування відеоданих для web-камери через додатковий
канал
def read_fast():
```

```
    if(video_input): fvs = FileVideoStream(0).start()
    else: fvs = FileVideoStream(name_file).start()
    time.sleep(1.0)
    start_stream=datetime.datetime.now()
    while fvs.more():
        frame = fvs.read()
        frame=transform_frame(frame)
        cv2.imshow("Frame", frame)
        cv2.waitKey(1)
        if (out_read(start_stream)):
            break
        fps.update()
    fvs.stop()

# метод зчитування відеоданих для Pi-камери без запису в стрім
Def read_picamera():
    camera = PiCamera()
    camera.resolution = (nwidth[rwidth],nheigh[rwidth])
    camera.framerate = 32.
    rawCapture = PiRGBArray(camera, size=(nwidth[rwidth],
nheigh[rwidth]))
    stream = camera.capture_continuous(rawCapture,format =
"bgr", use_video_port = True)
    time.sleep(2.0)
    start_stream=datetime.datetime.now()

    for (i, f) in enumerate(stream):
        frame=f.array
        frame=transform_frame(frame)
        cv2.imshow("Frame", frame)
        key = cv2.waitKey(1) & 0xFF
        if (out_read(start_stream)):
            break
        rawCapture.truncate(0)
        fps.update()
    fps.stop()
    stream.close()
    rawCapture.close()
    camera.close()

# метод зчитування відеоданих для Pi-камери із записом у стрім
Def read_pivideostream():

    fvs = PiVideoStream().start()
    time.sleep(2.0)

    start_stream = datetime.datetime.now()
    while True:
        frame = fvs.read()
        frame=transform_frame(frame)
```

```
        cv2.imshow("Frame", frame)
        key = cv2.waitKey(1) & 0xFF
        if (out_read(start_stream)):

            break
        fps.update()
    fvs.stop()

# ініціація початкових даних, початок запису даних, виклик методу body(), зупинення запису і відображення інформації про сесію
start_streams=datetime.datetime.now()
fps = FPS().start()
fps.frame_detecting = 0
fps.detected = False
body()
fps.stop()
print_information()
cv2.destroyAllWindows()
```

4.7. Анонімізація й розмиття облич

Відомо, що завдання розпізнавання облич (face recognition) складається з двох етапів: попередній етап − виявлення й детектування (face detection), наступний етап − безпосередньо розпізнавання його зображення на наборі даних (біометричні бази даних та ін.). Однак часто виникає необхідність у розв'язанні, по суті, оберненої задачі − анонімізації виявлених у кадрі облич та їх розмиття для забезпечення конфіденційності.

У різних ситуаціях виникає досить багато причин для практичного застосування розмиття обличчя та його анонімізації. Основні з них:

- конфіденційність і захист особистості в громадських/приватних областях;
- захист дітей в інтернеті (наприклад, розмиття облич неповнолітніх на завантажених фотографіях);
- фотожурналістика й новинні репортажі (наприклад, розмиття облич людей, які не підписали форму відмови);
- заняття й поширення набору даних (наприклад, анонімізація людей у наборі даних медичного характеру).

Стисло опишемо суть цієї технології. Для цього розглянемо, що таке розмиття облич і як можна використовувати ресурси бібліотеки OpenCV для анонімізації облич на зображеннях і відеопотоках.

Зазвичай у комп'ютерному зорі використовується два методи для розмиття облич:

- використання розмиття за Гауссом;
- застосування ефекту піксельного розмиття.

Приклад розмиття обличчя за допомогою ефекту піксельного розмиття й анонімізації показано на рис. 4.26. Зверніть увагу, що обличчя розмито, а особистість людини неможливо розпізнати. Це виконано для захисту особистості від негативних впливів.

Рис. 4.26. Приклад розмиття і анонімізації обличчя

Розглянемо процедуру розмиття й анонімізації облич більш детально у вигляді послідовності кроків (дій). Можна виокремити чотири основні кроки розмиття й анонімізації обличчя, як це показано на рис. 4.27.

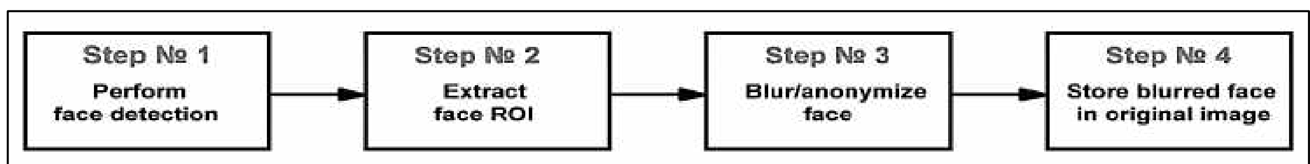


Рис. 4.27. Послідовність операцій з анонімізації облич

Покрокова інструкція в цьому випадку має такий вигляд:

- крок № 1 − виконати розпізнавання облич;
- крок № 2 − витягти інформацію з області інтересу (ROI − Region Of Interest);
- крок № 3 − фактичне розмиття/анонімізація обличчя;
- крок № 4 − збереження розмитого обличчя назад в область інтересу вихідного зображення.

На першому кроці можна використовувати будь-який детектор облич за умови, що він може визначати координати обмежувальної рамки обличчя на зображенні або відеопотоці. Зазвичай для цього використовують детектори на базі алгоритмів Віоли − Джонса або детектори облич на основі глибокого навчання.

Після виявлення обличчя на другому кроці необхідно виділити область інтересу (Face ROI). Після цього детектор облич визначить початкові

і кінцеві координати обмежувальної рамки (x, y) обличчя на зображенні, як показано на рис. 4.28.

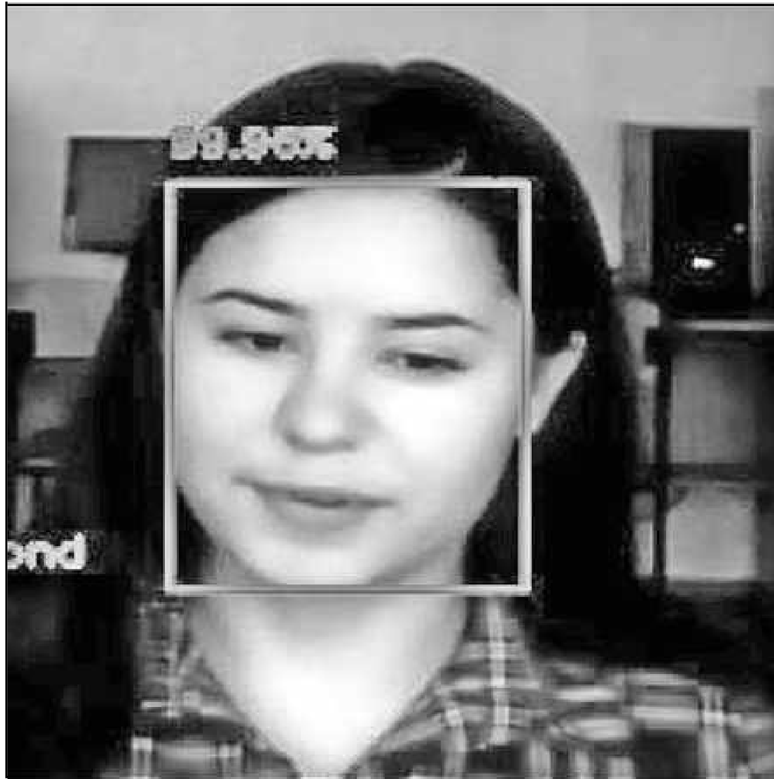


Рис. 4.28. Локалізація виявленого обличчя з допомогою обмежувальної рамки

На третьому кроці проводиться розмиття обличчя (у нашому прикладі з допомогою гаусівського фільтра).

Рис. 4.29. Результати розмиття обличчя гаусівським фільтром

Потім фрагмент розмитого зображення обличчя поміщається у виділену область вихідного зображення, і процедуру ананімізації обличчя можна вважати виконаною.

Стисло опишемо основний алгоритм програми анонімізації облич, для чого створюється цикл перегляду шарів вихідного зображення. Усередині циклу визначається відсоткове відношення кількості фрагментів розпізнаного зображення до заданого граничного значення (параметр confidence). При низькому значенні порогу (confidence) зображення буде фільтруватися навіть при частково закритому фрагменті обличчя, а високі значення порога можуть перешкоджати виконанню процедури фільтрації. Крім того, визначається область дії (рамка), а також тип і властивості фільтра (пікселізація розмиття за Гауссом), що визначають ступінь розмиття й маскування облич. Ця дія закриває вихідне зображення отриманою відфільтрованою ділянкою.

Наведемо код, що виконує анонімізацію зображення:

```
for i in range(0, detections.shape[2]):

    confidence = detections[0, 0, i, 2]

    if confidence > 0.5:
        box = detections[0, 0, i, 3:7] * np.array([w, h, w, h])
        (startX, startY, endX, endY) = box.astype("int")
        face = frame[startY:endY, startX:endX]
        if method== "simple":
            face = anonymize_face_simple(face,factor=3.0)
        else:
            face = anonymize_face_pixelate(face,blocks=100)
        frame[startY:endY, startX:endX] = face
```

З цього фрагмента коду видно, що програма пропонує два методи розмиття облич:

```
face = anonymize_face_simple(face,factor=3.0),
```

або

```
face = anonymize_face_pixelate(face,blocks=100).
```

Структура алгоритму формування функції анонімізації з розмиттям за Гауссом є такою:

```
def anonymize_face_simple(image,factor=3.0):
    (h, w) = image.shape[:2]
    kW = int(w / factor)
    kH = int(h / factor)

    if kW % 2 == 0:
        kW -= 1
    if kH % 2 == 0:
        kH -= 1
```

```
    if( kW<0 or kH<0):
        return image
    image2=cv2.GaussianBlur(image, (kW, kH), 0)
    if (image2 is None):
        image2=image
    return image2
```

Методика анонімізації з допомогою укрупнення пік селів (рис. 4.30) полягає в звичайному усередненні точок у певній області. Алгоритм такої функції анонімізації має вигляд:

```
def anonymize_face_pixelate(image, blocks=3):
    (h, w) = image.shape[:2]
    xSteps = np.linspace(0, w, blocks + 1, dtype="int")
    ySteps = np.linspace(0, h, blocks + 1, dtype="int")
    for i in range(1, len(ySteps)):
        for j in range(1, len(xSteps)):
            startX = xSteps[j - 1]
            startY = ySteps[i - 1]
            endX = xSteps[j]
            endY = ySteps[i]
            roi = image[startY:endY, startX:endX]
            (B, G, R) = [int(x) for x in cv2.mean(roi)[:3]]

cv2.rectangle(image,(startX,startY),(endX,endY),(B,G,R),-1)
return image
```

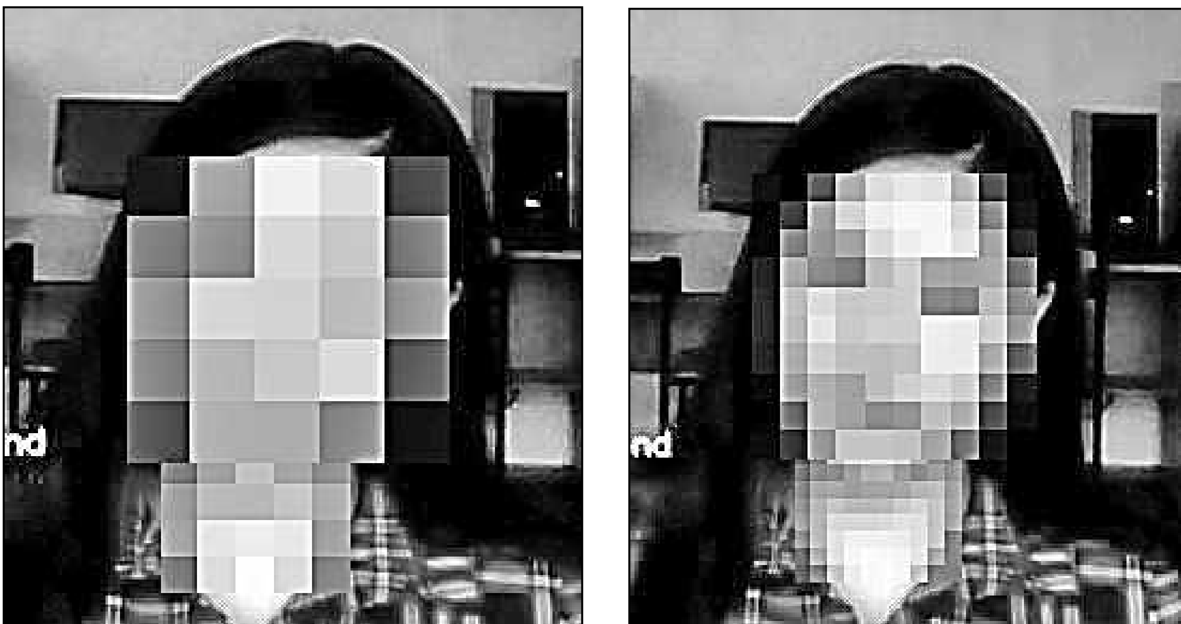


Рис. 4.30. Анонімізація з допомогою укрупнення пікселів

Додаток 1

ЛІСТИНГ ПРОГРАМИ Video_Stream_Quality_Control.py

```
# import the necessary packages
from imutils.video import FPS
from imutils.video import VideoStream
from imutils.video.pivideostream import PiVideoStream
from imutils.video import FileVideoStream

from picamera.array import PiRGBArray
from picamera import PiCamera
import numpy as np
import argparse
import imutils
import time
import cv2
import datetime
global fps,rwidth,method,start_streams
global nwidth,nheigh

nwidth=[1280,1280 ,1200 ,1024 ,960 ,800 ,640 ,640 ,480 ,640
,320]

nheigh=[960,720,600,768,540,600,480,360,320,240,240]
rwidth=6
method=2

def body():
    if method==1:
        read_slow()
    if method==2:
        read_fast()
    if method==3:
        read_videostream()
    if method==4:
        read_picamera()
    if method==5:
        read_pivideostream()

def arguments():
    global args
    ap = argparse.ArgumentParser()
    ap.add_argument("-v", "--video", required=True,help="path
to input video file")
    args = vars(ap.parse_args())

def print_information():
    print("[INFO] elasped time: {:.2f}".format(fps.elapsed()))
    print("[INFO] approx. FPS: {:.2f}".format(fps.fps()))
    print("[INFO] Width",nwidth[rwidth],"[INFO]
```

```
Heigh",nheigh[rwidth])

def transform_frame(frame):
    if (frame is None) :
        return
    #frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    #frame = np.dstack([frame, frame, frame])
    frame = cv2.cvtColor(frame, cv2.COLOR_BGR2YUV)
    frame[:,:,0] = cv2.equalizeHist(frame[:,:,0])
    frame = cv2.cvtColor(frame, cv2.COLOR_YUV2BGR)
    name_meth=["Slow Method","Fast Meth-
od","Videostream","Picamera","PiVideostream"]
    Text=name_meth[method-1]+" Resolution "
+str(int(nwidth[rwidth]))+"x"+str(int(nheigh[rwidth]))
    Text=Text+"  With equalization"
    fps_1=(datetime.datetime.now()-
start_streams).total_seconds()
    Text_FPS="FPS "+str(int(fps._numFrames/fps_1))+"
frame/second"
    cv2.putText(frame,Text, (10, 30),cv2.FONT_HERSHEY_SIMPLEX,
0.5, (0, 255, 0), 2)
    cv2.putText(frame,Text_FPS, (10,
400),cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)

    if (method)<4: frame = imutils.resize(frame,
width=nwidth[rwidth],height=nheigh[rwidth])

    return frame
def out_read(start_stream):
    return ((datetime.datetime.now()-
start_stream).total_seconds()>10)

def read_slow():

    stream = cv2.VideoCapture(0)
    start_stream=datetime.datetime.now()
    while True:
        (grabbed, frame) = stream.read()
        if not grabbed:
            break
        frame=transform_frame(frame)
        cv2.imshow("Frame", frame)
        cv2.waitKey(1)
        if (out_read(start_stream)):
            break
        fps.update()
    stream.release()
def read_videostream():

    fvs = VideoStream(0).start()
```

```
    time.sleep(2.0)
    start_stream=datetime.datetime.now()
    while True:
        frame = fvs.read()
        frame=transform_frame(frame)
        cv2.imshow("Frame", frame)
        cv2.waitKey(1)
        fps.update()

        if (out_read(start_stream)):
            break
    fvs.stop()

def read_fast():
    fvs = FileVideoStream(0).start()
    time.sleep(1.0)
    start_stream=datetime.datetime.now()
    while fvs.more():
        frame = fvs.read()
        frame=transform_frame(frame)
        cv2.imshow("Frame", frame)
        cv2.waitKey(1)
        if (out_read(start_stream)):
            break
        fps.update()
    fvs.stop()
def read_picamera():
    camera = PiCamera()
    camera.resolution = (nwidth[rwidth],nheigh[rwidth])
    camera.framerate = 32.
    rawCapture = PiRGBArray(camera, size=(nwidth[rwidth],
nheigh[rwidth]))
    stream = camera.capture_continuous(rawCapture, for-
mat="bgr",use_video_port=True)
    time.sleep(2.0)
    start_stream=datetime.datetime.now()
    for (i, f) in enumerate(stream):
        frame=f.array
        frame=transform_frame(frame)

        cv2.imshow("Frame", frame)
        key = cv2.waitKey(1) & 0xFF
        if (out_read(start_stream)):
            break
        rawCapture.truncate(0)
        fps.update()
    fps.stop()
    stream.close()
    rawCapture.close()
```

```
    camera.close()
def read_pivideostream():

    fvs =
PiVideoStream(resolution=(nwidth[rwidth],nheigh[rwidth])).start
()
    time.sleep(2.0)

    start_stream=datetime.datetime.now()

    while True:
        frame = fvs.read()
        frame=transform_frame(frame)

        cv2.imshow("Frame", frame)
        key = cv2.waitKey(1) & 0xFF
        if (out_read(start_stream)):

            break
        fps.update()

    fvs.stop()

'''

while (rwidth<13):
    start_streams=datetime.datetime.now()
    fps = FPS().start()
    body()
    fps.stop()
    print_information()

    cv2.destroyAllWindows()
    rwidth+=1
    start_streams=0
'''
start_streams=datetime.datetime.now()
fps = FPS().start()
body()
fps.stop()
print_information()
cv2.destroyAllWindows()
```

Додаток 2

ЛІСТИНГ ПРОГРАМИ Face_detection_Viola_Jones.py

```
# import the necessary packages
import os.path
import cv2
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import matplotlib.gridspec as gridspec

# Define cascades
face_cascade =
cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
right_eye_cascade =
cv2.CascadeClassifier('haarcascade_righteye_2splits.xml')
left_eye_cascade =
cv2.CascadeClassifier('haarcascade_lefteye_2splits.xml')
nose_cascade = cv2.CascadeClassifier('haarcascade_nose.xml')
mouth_cascade = cv2.CascadeClassifier('haarcascade_mouth.xml')

# Define video if exists
video_file = "Video_3.avi"

# Define colors
blue = (255, 0, 0)
green = (0, 255, 0)
red = (0, 0, 255)
yellow = (0, 255, 255)

# Equalization function
def equalize(current_frame):
    # Convert into YUV color space
    frame_in_yuv         =         cv2.cvtColor(current_frame,
cv2.COLOR_BGR2YUV)
    # Equalize by Y component
    frame_in_yuv[:, :, 0] = cv2.equalizeHist(frame_in_yuv[:, :,
0])
    # Convert back to RGB
    result = cv2.cvtColor(frame_in_yuv, cv2.COLOR_YUV2BGR)
    return result

# Filtering function
def apply_filter(current_frame):
    # Define area for filtering
    kernel_3x3 = np.ones((3, 3), np.float32) / 9.0
    # Filter frame according to matrix
    result = cv2.filter2D(current_frame, -1, kernel_3x3)
    return result
```

```
# Actual detecting
def detect_face(current_frame):
    face_detected = 0
    eyes_detected = 0
    nose_detected = 0
    mouth_detected = 0
    # Convert into white/black
    gray = cv2.cvtColor(current_frame, cv2.COLOR_BGR2GRAY)
    # Detect faces
    faces = face_cascade.detectMultiScale(gray, 1.3, 5)
    for (x, y, w, h) in faces:
        face_detected = 1
        # Draw rect around found face
        cv2.rectangle(current_frame, (x, y), (x + w, y + h),
blue, 2)

        # Detect right eye
        right_eye_detected = de-
tect_face_part(current_frame,gray,right_eye_cascade, green,
                                              y + 2 * h /9, y +
2 * h /3,
                                              x + w / 9, x + 4
* w /9)
        # Detect left eye
        left_eye_detected = de-
tect_face_part(current_frame,gray,left_eye_cascade, green,
                                              y + 2 * h / 9, y +
2 * h / 3,
                                              x + 5 * w / 9, x +
8 * w / 9)
        eyes_detected = 1 if right_eye_detected and
left_eye_detected else 0
        # Detect nose
        nose_detected = detect_face_part(current_frame,gray,
nose_cascade, yellow,
                                          y + h / 4, y + 3 * h /
4,
                                          x + w / 4, x + 3 * w /
4)
        # Detect mouth
        mouth_detected = de-
tect_face_part(current_frame,gray,mouth_cascade, red,
                                            y + 2 * h / 3, y + h,
                                            x + w / 4, x + 3 * w
/ 4)

    # Draw results
    return face_detected, eyes_detected, nose_detected,
mouth_detected, current_frame
```

```
# Detecting part of face with according cascade and zone of in-
terest
def  detect_face_part(current_frame,  gray,  cascade,  color,
y_from, y_to, x_from, x_to):
    # Select gray zone
    roi_gray_zone        =        gray[int(y_from):int(y_to),
int(x_from):int(x_to)]
    # Select colored zone
    roi_color_zone = cur-
rent_frame[int(y_from):int(y_to),int(x_from):int(x_to)]

    # Detect part of face using appropriate cascade and zone
    parts = cascade.detectMultiScale(roi_gray_zone)
    for (x, y, w, h) in parts:
        # Assuming we have only one part of a face, after de-
tecting draw area and exit loop
        cv2.rectangle(roi_color_zone, (x, y), (x + w, y + h),
color, 2)
        return 1
    return 0

def process_rsults(shifting_window, y, result):
    shifting_window[:-1] = shifting_window[1:]
    y[:-1] = y[1:]
    shifting_window[-1] = result
    y[-1] = sum(shifting_window) / 100.0
    return shifting_window, y

# Define video capturing
cap = cv2.VideoCapture(video_file) if
os.path.isfile(video_file) else cv2.VideoCapture(0)
# Plot setup starts here
fig   =   plt.figure(num=None,   figsize=(10,   8),   dpi=80,
facecolor='w', edgecolor='k')

gs = gridspec.GridSpec(2, 2)

ax = plt.subplot(gs[0, 0])
ax.set_title('Face detection')

ax_eyes = plt.subplot(gs[0, 1])
ax_eyes.set_title('Eyes detection')

ax_nose = plt.subplot(gs[1, 0])
ax_nose.set_title('Nose detection')

ax_mouth = plt.subplot(gs[1, 1])
ax_mouth.set_title('Mouth detection')

# Set x range
x = np.arange(100)
```

```
# Init shifting windows
basic_shifting_window = np.zeros((100,), dtype=int)
# Init empty y axes data
y_basic = np.zeros((100,), dtype=float)

y_eyes_basic = np.zeros((100,), dtype=float)
eyes_basic_shifting_window = np.zeros((100,), dtype=int)

y_nose_basic = np.zeros((100,), dtype=float)
nose_basic_shifting_window = np.zeros((100,), dtype=int)

y_mouth_basic = np.zeros((100,), dtype=float)
mouth_basic_shifting_window = np.zeros((100,), dtype=int)

originalPlot, = ax.plot(x, y_basic, c='b')
eyes_basic_plot, = ax_eyes.plot(x, y_eyes_basic, c='b')
nose_basic_plot, = ax_nose.plot(x, y_nose_basic, c='b')
mouth_basic_plot, = ax_mouth.plot(x, y_mouth_basic, c='b')

# Add axes labels
ax.set_xlabel("N")
ax_eyes.set_xlabel("N")
ax_nose.set_xlabel("N")
ax_mouth.set_xlabel("N")

ax.set_ylabel("P")
ax_eyes.set_ylabel("P")
ax_nose.set_ylabel("P")
ax_mouth.set_ylabel("P")

ax.set_xlim(0, 100)
ax_eyes.set_xlim(0, 100)
ax_nose.set_xlim(0, 100)
ax_mouth.set_xlim(0, 100)

ax.set_ylim(-0.1, 1.1)
ax_eyes.set_ylim(-0.1, 1.1)
ax_nose.set_ylim(-0.1, 1.1)
ax_mouth.set_ylim(-0.1, 1.1)

plt.ion()
plt.show()
# Finish plot setup

while True:
    ret, frame = cap.read()
    # Try to detect face
    basic_face_detected, \
    basic_eyes_detected, \
    basic_nose_detected, \
    basic_mouth_detected, \
```

```
    basic_frame = detect_face(frame)

    if basic_face_detected != 0:
        cv2.imshow('', basic_frame)
    else:
        # Equalize frame
        equalized = equalize(frame)
        # Filter equalized frame
        filtered = apply_filter(equalized)
        basic_face_detected, \
        basic_eyes_detected, \
        basic_nose_detected, \
        basic_mouth_detected, \
        equalized_frame = detect_face(frame)
        cv2.imshow('', equalized_frame)

    # Shift data windows
    basic_shifting_window, y_basic = pro-
cess_rsults(basic_shifting_window, y_basic,
basic_face_detected)
    eyes_basic_shifting_window, y_eyes_basic = pro-
cess_rsults(eyes_basic_shifting_window, y_eyes_basic,

basic_eyes_detected)
    nose_basic_shifting_window, y_nose_basic = pro-
cess_rsults(nose_basic_shifting_window, y_nose_basic,

basic_nose_detected)
    mouth_basic_shifting_window, y_mouth_basic = pro-
cess_rsults(mouth_basic_shifting_window, y_mouth_basic,

basic_mouth_detected)

    # Count statistic
    current_status_basic = "Detection success:
{0}%".format(y_basic[-1] * 100)

    # Reflect statistic on legend
    y1_patch = mpatches.Patch(color='blue', la-
bel=current_status_basic)
    ax.legend(handles=[y1_patch])

    y_eyes = mpatches.Patch(color='blue', label="Detection suc-
cess: {0}%".format(y_eyes_basic[-1] * 100))
    ax_eyes.legend(handles=[y_eyes])

    y_nose = mpatches.Patch(color='blue', label="Detection suc-
cess: {0}%".format(y_nose_basic[-1] * 100))
    ax_nose.legend(handles=[y_nose])
```

```
    y_mouth = mpatches.Patch(color='blue',label = "Detection
success: {0}%".format(y_mouth_basic[-1] * 100))
    ax_mouth.legend(handles=[y_mouth])
    # Update data for plot
    originalPlot.set_ydata(y_basic)
    eyes_basic_plot.set_ydata(y_eyes_basic)
    nose_basic_plot.set_ydata(y_nose_basic)
    mouth_basic_plot.set_ydata(y_mouth_basic)

    # Redraw plot to reflect changes
    fig.canvas.draw()

    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

# Print final results
print("Basic detection success: {0}%".format(y_basic[-1]))
# Cleanup
cap.release()
cv2.destroyAllWindows()
```

БІБЛІОГРАФІЧНИЙ СПИСОК

1. Gradient-Based Learning Applied to Document Recognition / Y. Lecun, et al. // Proceedings of the IEEE. – 1998. − Vol. 86, № 11. − P. 2278–2324.
2. ImageNet Classification with Deep Convolutional Neural Networks / A. Krizhevsky, et al. // Communications of the ACM. – 2017. – Vol. 60, № 6. – P. 84−90.
3. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification/ K. He, et al. // IEEE International Conference on Computer Vision (ICCV). – 2015.
4. https://ru.wikipedia.org/wiki/ImageNet
5. Krizhevsky, A. ImageNet classification with deep convolutional neural networks / A. Krizhevsky, I. Sutskever, G. E. Hinton // Advances in neural information processing systems. − 2012. − P. 1097–1105.
6. Simonian, K. Very deep convolutional networks for large-scale image recognition/ K. Simonyan, A. Zisserman // ArXiv preprint arXiv. – 2014. – P. 1409−1556.
7. Going deeper with convolutions / C. Szegedy, et al. // Proceedings of the IEEE conference on computer vision and pattern recognition. – 2015. − P. 1−9.
8. Deep residual learning for image recognition / K. He, et al. // In Proceedings of the IEEE conference on computer vision and pattern recognition. − 2016. – P. 770−778.
9. Rethinking the inception architecture for computer vision / C. Szegedy, et al. // In Proceedings of the IEEE conference on computer vision and pattern recognition. – 2016. − P. 2818−2826.
10. Inception-V4, inception-resnet and the impact of residual connections on learning / C. Szegedy, et al. // In AAAI. − 2017. – Vol. 4. – P. 12.
11. Rich feature hierarchies for accurate object detection and semantic segmentation / R. Girshick, et al. // In Proceedings of the IEEE conference on computer vision and pattern recognition. – 2014. – P. 580−587.
12. Fast r-cnn / R. Girshick // In Proceedings of the IEEE international conference on computer vision and pattern recognition. – 2015.− P. 1440−1448.
13. Faster RCNN – Towards real-time object detection with region proposal networks / S. Ren, et al. // In Advances in neural information processing systems. – 2015. – P. 91−99.
14. Object Detection via Region-based Fully Convolutional Networks / J.Dai, et al. // In Advances in neural information processing systems. – 2016. – P. 379−387.

15. You only look once: Unified, Real-Time Object Detection / J. Redmon, et al. // In Proceedings of the IEEE conference on computer vision and pattern recognition. – 2016. – P. 779−788.
16. SSD: Single Shot Multibox Detector / W. Liu, et al. // In European conference on Computer Vision. − 2016. – P. 21−37.
17. Redmon, J. YOLO9000: Better, Faster, Stronger / J. Redmon, A. Farhadi // arXiv preprint. −2017.
18. Zoph, B. Neural Architecture Search with Reinforcement Learning / B. Zoph, Q. Le // arXiv preprint arXiv: 1611.01578. – 2016.
19. Mask r-cnn. In Computer Vision (ICCV) / K. He, et al. // 2017 IEEE International Conference on Computer Vision. – 2017. – P. 2980−2988.
20. Viola, P. Rapid Object Detection using a Boosted Cascade of Simple Features / P. Viola, M.J. Jones // proceedings IEEE Conf. on Computer Vision and Pattern Recognition (CVPR 2001). – 2001.
21. Viola, P. Robust real-time face detection / P. Viola and M. J. Jones // International Journal of Computer Vision. − 2004. – Vol. 57, № 2. – P. 137–154.
22. Гонсалес, Р. Цифровая обработка изображений / Р. Гонсалес, Р. Вудс. – М. : Техносфера. – 2005. – 1072 с.
23. Местецкий, Л. М. Математические методы распознавания образов / Л. М. Местецкий // МГУ, ВмиК. – М., 2002–2004. С. 42–44.
24. Sochman, J. AdaBoost / J. Sochman, J. Matas // Center for Machine Perception, Czech Technical University. − Prague. – 2010.
25. Freund, Y. A Short Introduction to Boosting / Y. Freund, R. Schapire // Shannon Laboratory, USA. – 1999. – P. 771−780.
26. Howse, J. Learning OpenCV 3 Computer Vision with Python - Second Edition / J. Howse, J. Minichino // Packt Publishing. – 2015.
27. Kapur, S. Computer Vision with Python 3 / S. Kapur // Packt Publishing. – 2017.
28. Joshi, P. OpenCV with Python By Example / P. Joshi // Packt Publishing. – 2015.
29. Библиотека компьютерного зрения OpenCV [Электронный ресурс]. – Режим доступа: http://docs.opencv.org/trunk/doc/py_tututoria/py_objdetect/py_face_detection/py_face_detection.html.
30. Villian, A. Mastering OpenCV 4 with Python / A. Villian, // Packt Publishing Ltd. Livery Place 35, Livery Street, Birmingham, B3 2PB, UK.
31. https://github.com/opencv/opencv/tree/master/samples/dnn/face_detector
32. http://dl.caffe.berkeleyvision.org/bvlc_alexnet.caffemodel

33. https://pjreddie.com/media/files/yolov3.weights
34. https://www.tensorflow.org/guide/saved_model
35. http://yann.lecun.com/exdb/mnist/
36. https://www.tensorflow.org/api_docs/python/tf/train/AdamOptimizer)
37. https://keras.io/
38. https://colab.research.google.com/notebooks/intro.ipynb

ЗМІСТ

ПЕРЕДМОВА 3
1. ОСНОВИ ПОБУДОВИ Й НАВЧАННЯ НЕЙРОННИХ МЕРЕЖ 4
1.1. Загальна класифікація нейронних мереж 4
1.2. Штучний нейрон 6
1.3. Перцептрони (вузли) 11
1.4. Узагальнені кількісні характеристики нейронних мереж 15
1.5. Використання вузлів зміщення 16
1.6. Приклад нейронної мережі «перцептрон» для класифікації даних 17
1.7. Навчання простої нейронної мережі «перцептрон» 20
1.8. Підготовка навчальних даних для перцептрона 24
1.9. Вступ до теорії навчання нейронних мереж 27
1.10. Швидкість навчання в нейронних мережах 30
1.11. Машинне навчання з багатошаровим перцептроном 31
1.12. Оновлення вагових коефіцієнтів 34
1.13. Навчальні набори даних для нейронних мереж 37
1.14. Визначення кількості прихованих шарів і прихованих вузлів у нейронній мережі 41
1.15. Підвищення точності прихованого шару нейронної мережі 43
2. ЗГОРТКОВІ НЕЙРОННІ МЕРЕЖІ ДЛЯ КЛАСИФІКАЦІЇ ЗОБРАЖЕНЬ 48
2.1. Методи нейромережних технологій і проектів для класифікації зображень 48
2.2. Основні етапи розвитку згорткових нейронних мереж для класифікації зображень 51
2.3. Аналіз властивостей основних бібліотек алгоритмів глибокого навчання для роботи з нейронними мережами 54
2.4. Методи побудови згорткових нейронних мереж 58
2.5. Згорткові нейронні мережі 60
3. РЕСУРСИ ГЛИБОКОГО НАВЧАННЯ В БІБЛІОТЕЦІ OPENCV 68
3.1. Загальні характеристики глибокого навчання 68
3.2. Функції блока DNN в OpenCV та їх основні властивості 70

3.3. Класифікація з використанням моделей, попередньо навчених в OpenCV 78
3.4. Приклади виявлення об'єктів з використанням моделей, попередньо навчених в OpenCV 82
3.5. Бібліотека TensorFlow 83
3.6. Лінійна регресія в TensorFlow 86
3.7. Розпізнавання рукописних цифр з використанням бібліотеки TensorFlow 91
3.8. Бібліотека Keras 94
4. ПРАКТИЧНЕ ВИКОРИСТАННЯ МЕТОДІВ І ЗАСОБІВ НЕЙРОННИХ МЕРЕЖ 108
4.1. Вимоги до компонентів проекту та критерії ефективності його роботи 108
4.1.1. Програмні й апаратні ресурси 109
4.1.2. Критерії ефективності системи детектування облич 111
4.2. Розширений опис основних програмних рішень 113
4.2.1. Ресурси для програмування 113
4.2.2. Методи введення відеоданих 114
4.2.3. Стабілізація контрастності відеоданих 115
4.3. Експериментальні дослідження алгоритмів уведення відеоданих та їх результати 117
4.4. Алгоритм виявлення облич Віоли − Джонса 125
4.5. Програмна реалізація алгоритму виявлення облич Віоли − Джонса 131
4.6. Виявлення й детектування облич з використанням бібліотеки Caffe 142
4.7. Анонімізація й розмиття облич 148
Додаток 1. Лістинг програми Video_Stream_Quolity_Control.py 153
Додаток 2. Лістинг програми Face_detection_Viola_Jones.py 157
БІБЛІОГРАФІЧНИЙ СПИСОК 163

Навчальне видання

Дергачов Костянтин Юрійович
Краснов Леонід Олександрович
Шостак Анатолій Васильович

ОБ’ЄКТНО-ОРІЄНТОВАНЕ ПРОЕКТУВАННЯ ТЕХНІЧНИХ СИСТЕМ

Частина 1

ОСНОВИ ПОБУДОВИ Й ВИКОРИСТАННЯ НЕЙРОННИХ МЕРЕЖ

Редактор Т. О. Іващенко

Зв. план, 2021
Підписано до друку 26.05.2021
Формат 60×84 1/16. Папір офс. Офс. друк
Ум. друк. арк. 9,3. Обл.-вид. арк. 10,5. Наклад 50 пр.
Замовлення 126. Ціна вільна

Видавець і виготовлювач
Національний аерокосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»
61070, Харків-70, вул. Чкалова, 17
http://www.khai.edu
Видавничий центр «ХАІ»
61070, Харків-70, вул. Чкалова, 17
izdat@khai.edu

Свідоцтво про внесення суб`єкта видавничої справи
до Державного реєстру видавців, виготовлювачів і розповсюджувачів
видавничої продукції сер. ДК № 391 від 30.03.2001