МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, комп'ютерної та програмної

інженерії Кафедра комп'ютерних інформаційних технологій

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри
Савченко А.С.

“ ”_____2020 р.

ДИПЛОМНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИЦІ ОСВІТНЬОГО СТУПЕНЯ “МАГІСТРА”

ЗА СПЕЦІАЛІЗАЦІЄЮ “ІНФОРМАЦІЙНІ УПРАВЛЯЮЧІ СИСТЕМИ ТА ТЕХНОЛОГІЇ (ЗА ГАЛУЗЯМИ)”

Тема: “Породжуюча модель побудови образів”

Виконавиця: Бовт Марія Олегівна

Керівник: доцент, к.т.н. Моденов Юрій Борисович

Нормоконтролер: _____ доцент, к.т.н. Райчев І.Е.

Київ 2020

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, комп'ютерної та програмної інженерії

Кафедра Комп'ютерних інформаційних технологій

Галузь знань, спеціальність, спеціалізація: 12 "Інформаційні технології", 122 "Комп'ютерні науки", "Інформаційні управляючі системи та технології (за галузями)" .

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Савченко А.С.
" " 2020 р.

З А В Д А Н Н Я

на виконання дипломної роботи студентки

Бовт Марії Олегівни . **1. Тема роботи:** "Породжуюча модель побудови образів" . затверджена наказом ректора № 1891/ст від 02.10.2020 р.

2. Термін виконання роботи: з 05.10.2020 до 22.12.2020 р.

3. Вихідні данні до роботи: відеокарта NVIDIA з підтримкою технології CUDA, мінімальний об'єм оперативної пам'яті – 8 GB. **4. Зміст пояснювальної записки:** вступ, теоретичний огляд і вибір засобів і методів, за допомогою яких вирішуватиметься поставлена задача, структура нейронної інформаційної мережі та її реалізація.

5. Перелік обов'язкового ілюстративного матеріалу: база даних фотографій Cat Daraset (Kaggle).

6. Календарний план-графік

№ з/п	***Завдання***	***Термін виконання***	***Підпис керівника***
1.	Аналіз літератури та джерел за темою дипломного проекту.	05.10.2020р – 15.05.2020р.	

2.	Розроблення та затвердження плану дипломного проекту.	16.10.2020р. – 20.10.2020р.	
3.	Проведення консультації з науковим керівником щодо створення першого розділу.	21.10.2020р.	
4.	Розробка розділу 1: Нейронна інформаційна мережа.	22.10.2020. – 28.10.2020р.	
5.	Розробка розділу 2: Структура нейронної мережі.	29.10.2020р. – 23.11.2020р.	
6.	Розробка розділу 3: Реалізація нейронної мережі.	23.11.2020р. – 3.12.2020р.	
7.	Продовження розробки розділу 3; тестування нейронної мережі.	4.12.2020р. – 11.12.2020р.	
8.	Висновки та оформлення пояснювальної записки дипломного проекту.	12.12.2020р. – 14.12.2020р.	
9.	Підписання необхідних документів у встановленому порядку.	14.12.2020р. – 17.12.2020р.	
10.	Підготовка до захисту та попередній захист дипломного проекту на випусковій кафедрі дипломного проекту.	18.12.2020р. – 21.12.2020р.	

7. Консультація з окремого(мих) розділу(ів) роботи:

Назва розділу	Консультант (посада, П.І.Б.)	Дата, підпис	
		Завдання видав	Завдання прийняв

8. Дата видачі завдання: 06.10.2020 р.

Керівник дипломної роботи Моденов Ю.Б.

Завдання прийняла до виконання Бовт М.О. ~~(підпис випускниці)~~ (П.І.Б.)

РЕФЕРАТ

Пояснювальна записка до дипломної роботи "Породжуюча модель побудови образів" містить 75 сторінок, 12 рисунків, 8 наукових джерел.

Мета роботи: моделювання нейронної мережі VGG-16 з використанням фреймворку Keras.

Об'єкт дослідження: глибока нейронна мережа.

Методи дослідження: аугментація даних, бібліотеки фреймворку Keras та Tensorflow, градієнтний спуск, генеративна змагальна мережа. **Предмет дослідження:** нейронна інформаційна мережа, що самостійно створює зображення, попередньо обробивши базу даних зображень із однаковими об’єктами.

Результати магістерської роботи: розроблено нейронну інформаційну

мережу, яка може сама створювати (породжувати) нові зображення. **Результати дипломної роботи можуть бути використані** для передачі художнього стилю, синтезу музики і голосу, ретуші зображень, створення зображень із надвисокою роздільною здатністю, симуляції можливих результатів подій у майбутньому та інше.

Ключові слова: НЕЙРОННА МЕРЕЖА, ЗГОРТКОВА НЕЙРОННА МЕРЕЖА, ГЛИБОКЕ НАВЧАННЯ, РОЗПІЗНАВАННЯ ОБРАЗІВ, ТОНКЕ НАЛАШТУВАННЯ, ПОРОДЖУЮЧА МОДЕЛЬ, ПОРОДЖУЮЧА НЕЙРОННА МЕРЕЖА, ГЕНЕРАТИВНА ЗМАГАЛЬНА МЕРЕЖА, КОМП'ЮТЕРНИЙ ЗІР, ШАР, ВАГИ, НАБІР ДАНИХ.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

ШІ – штучний інтелект

ЗМІ – засоби масової інформації

НМ – нейронна мережа

ЗМІ – засоби масової інформації

ПЗ – програмне забезпечення

CNN – Convolutional neural network (згорткова нейронна мережа) GAN – Generative adversarial networks (генеративна змагальна мережа) DCGAN – Deep Convolutional Generative Adversarial Network (Глибока згорткова генеративна змагальна мережа)

GPU – Graphics Proccesing Unit (Графічний процесор)

ЗМІСТ

ВСТУП

«Чого не можу відтворити, того не розумію»

Річард Фейнман

Неймовірно, як сильно продвинулась промисловість ШІ за останні десять років у багатьох галузях - від безпілотних автомобілів до розпізнавання та синтезу

мови. Завдяки цим досягненням у багатьох компаніях та організаціях заговорили про ШІ не як про щось фантастичне, а як про індустрію, що впливає на наше життя вже сьогодні.

Засоби масової інформації говорять про ШІ чи не кожного дня, а технологічні гіганти один перед іншим розповідають про своїх багатообіцяючі плани на ці технології. Поки деякі інвестори та топ-менеджери намагаються зрозуміти, як створювати цінності в цьому новому світі, більшість відчайдушно намагається розробити, що взагалі це все означає. У той же час, державні управління шукають способи зменшити вплив повальної автоматизації. Розуміючи, що ШІ повпливає на всю світову економіку, учасники цих дискусій представляють собою всі багатообразні сторони, рівні розуміння та ступені компетентності, на яких лиш і можна розробляти та використовувати системи зі ШІ. Тому необхідний весь цей діалог - включаючи питання, відповіді та рекомендації – вести на основі даних та об'єктивної реальності, а не уявлень. Екстрапольовувати уявні результати з наукових публікацій, статей у профільних ЗМІ, спекуляціях та інших мисленнєвих експериментах на наше майбутнє - це занадто просто (і іноді занадто захоплююче).

Актуальність теми дипломної роботи полягає в тому, що розвиток теорії породжуючих нейронних мереж зробити неоціненний вклад у багатьох галузях життєдіяльності людини.

Об'єктом дослідження є глибокі нейронні мережі.

Предметом дослідження є нейронна інформаційна мережа, що спочатку навчилася на великих об’ємах даних (зображень) і тому може сама генерувати інші образи.

Мета дипломної роботи: моделювання нейронної мережі VGG-16 на базі фреймворку Keras.

Для досягнення мети дипломної роботи поставлено такі завдання: - проаналізувати способи реалізацій породжуючих нейронних мереж; - в залежності від висновків аналізу, вибрати спосіб моделювання мережі; - обрати необхідне програмне забезпечення;

- описати структуру нейронної мережі;

- оцінити результати роботи мережі.

Практичною основою для дослідження стала зростаюча популярність у вивченні та імплементуванні методів машинного навчання у сучасних приладах, програмному забезпеченні, що розкриває нескінчений потенціал у використанні даних у виробництві чи професійній діяльності.

РОЗДІЛ 1

НЕЙРОННА ІНФОРМАЦІЙНА МЕРЕЖА

Нейронні мережі (англ. artificial neural networks, ANN) — це обчислювальні системи, які надихалися біологічними нейронними мережами, що складають мозок тварин. Ці системи навчаються задач, поставлених перед ними (поступально покращують свою продуктивність на них), розглядаючи приклади, зазвичай без спеціального програмування під конкретну задачу. Наприклад, у розпізнаванні зображень вони можуть навчатися ідентифікувати зображення, які містять дороги, аналізуючи приклади зображень, мічені як «дорога» та «узбіччя», і використовуючи результати для ідентифікування доріг в інших зображеннях. Вони роблять це без жодного базового знання про дороги: що вони мають розмітку, побудовані з асфальту чи, наприклад, бетону та для чого використовуються. Натомість, вони розвивають свій власний набір доречних характеристик з навчального матеріалу, який вони вивчають та обробляють.

Нейронна мережа ґрунтується на сукупності з'єднаних вузлів, що називають штучними нейронами (так само як біологічні нейрони у головному мозку тварин). Кожне з'єднання (можна провести аналогію з синапсом) може передавати сигнал від одного до іншого штучного нейрона. Штучний нейрон, який отримує сигнал, може обробляти його, й потім сигналізувати штучним нейронам, приєднаним до нього.

В поширених реалізаціях нейронних мереж сигнал на з'єднанні між штучними нейронами є дійсним числом, а вихід кожного штучного нейрону обчислюється нелінійною функцією суми його входів. Штучні нейрони та

побудови образів з'єднання зазвичай мають вагу, яка підлаштовується в перебігу навчання. Вага збільшує або зменшує силу сигналу на з'єднанні. Штучні нейрони можуть мати

такий поріг, що сигнал надсилається лише якщо сукупний сигнал перетинає цей поріг. Штучні нейрони зазвичай організовано в шари. Різні шари виконують різні види перетворень своїх входів. Сигнали проходять від першого (входового) до останнього (вихідного) шару, можливо, після проходження шарами декілька разів.

Первинною метою підходу НМ було розв'язання задач таким же способом, як це робив би людський мозок. З часом увага зосередилася на відповідності певним розумовим здібностям, ведучи до відхилень від біології. НМ використовували в ряді різноманітних задач, включно з комп'ютерним баченням, розпізнаванням мовлення, машинним перекладом, соціально-мережевим фільтруванням, грою в настільні та відеоігри, та медичним діагностуванням [1] .

1.1 Розпізнавання образів

Отже, однією із задач, яка вирішується за допомогою нейронних мереж, є розпізнавання образів. Виокремлюється окрема дисципліна, що займається обробкою та вирішенням задач такого типу.

Комп'ютерний зір — теорія та технологія створення машин, які можуть проводити стеження, виявлення та класифікацію об'єктів.

Як наукова дисципліна, комп'ютерний зір належить до теорії та технології створення штучних систем, які отримують інформацію у вигляді зображень. Відеодані можуть бути представлені у вигляді багатьох форм, наприклад, як відеопослідовність, зображення з різних камер або тривимірними даними з медичного сканера.

Як технологічна дисципліна, комп'ютерний зір прагне застосувати теорії та моделі комп'ютерного зору до створення систем комп'ютерного зору. Прикладами таких систем можуть бути:

- системи керування процесами (промислові роботи, автономні транспортні засоби);
- системи відеоспостереження;
- системи організації інформації (наприклад, для індексації баз даних зображень);

- системи моделювання об'єктів або оточуючого середовища (аналіз медичних зображень, топографічне моделювання);
- системи взаємодії (наприклад, пристрої введення для систем людино-машинної взаємодії).

Комп'ютерний зір також може бути описаний як доповнення (але не обов'язково протилежність) біологічному зору. У біології вивчається зорове сприйняття людини і різноманітних тварин, в результаті створюються моделі роботи таких систем в термінах фізіологічних процесів. Комп'ютерний зір, з іншого боку, вивчає і описує системи комп'ютерного зору, які виконано апаратно або програмно. Міждисциплінарний обмін між біологічним та комп'ютерним зором виявився досить продуктивним для обох наукових галузей. Підрозділи комп'ютерного зору включають відтворення дій, виявлення подій, стеження, розпізнавання образів, відновлення зображень. [1]

1.2 Типові задачі комп’ютерного зору

Кожна з областей застосування комп'ютерного зору, що були описані вище, пов'язана з низкою задач; більш чи менше гарно визначені проблеми

вимірювання чи обробки можуть бути вирішені з використанням багатьох методів. Деякі приклади типових задач комп'ютерного зору представлені нижче [1].

1.2.1 Розпізнавання

Класична задача в комп'ютерному зорі, машинному зорі та обробці зображень— це визначення того, чи містять відеодані деякий характерний об'єкт, особливість чи активність. Ця задача може бути достовірно і легко вирішена людиною, але досі не вирішена задовільно в комп'ютерному зорі в загальному випадку: випадкові об'єкти у випадкових ситуаціях.

Існуючі методи вирішення цієї задачі ефективні тільки для окремих об'єктів, таких як прості геометричні об'єкти (наприклад, багатокутники), людські обличчя,

друковані чи рукописні символи, автомобілі і лише у визначених умовах, зазвичай це певне освітлення, фон і положення об'єкта відносно камери.

В літературі описане різноманіття проблем розпізнавання:

Розпізнавання: один чи декілька попередньо заданих чи вивчених об'єктів або класів об'єктів можуть бути розпізнані, зазвичай разом з їх двовимірним положенням на зображенні чи тривимірним положенням в сцені.

Ідентифікація: розпізнається індивідуальний екземпляр об'єкта. Приклади: ідентифікація визначеного людського обличчя чи відбитка пальців чи автомобіля.

Виявлення: відеодані перевіряються на наявність визначеної умови. Наприклад, виявлення можливих неправильних клітин чи тканин в медичних зображеннях. Виявлення, що базується на відносно простих і швидких

обчисленнях, іноді використовується для знаходження невеликих ділянок в зображенні, що аналізується, які потім аналізуються за допомогою заходів, що потребують більше ресурсів, для отримання правильної інтерпретації.

Існує кілька спеціалізованих задач, що базуються на розпізнаванні, наприклад:

Пошук зображень за вмістом: знаходження всіх зображень, що мають певний зміст, серед великого набору зображень. Вміст може бути визначено різними шляхами, наприклад в термінах схожості з конкретним зображенням (знайти всі зображення, що схожі на дане зображення), чи в термінах високорівневих критеріїв пошуку, що вводяться як текстові дані (знайти всі зображення на яких зображено багато будинків, які зроблені взимку і на яких нема машин).

Оцінка положення та орієнтації: визначення положення чи орієнтації визначеного об'єкта відносно камери. Прикладом застосування цієї техніки може бути сприяння руки робота при вилученні об'єктів з конвеєра на лінії складання.

Оптичне розпізнавання символів: розпізнавання символів на зображеннях друкованого чи рукописного тексту, зазвичай для переведення останнього в текстовий формат, найбільш зручний для редагування чи зберігання (до прикладу, ASCII) [1].

1.2.2 Рух

Розглянемо декілька задач, що пов'язані з оцінкою руху, в яких послідовність зображень (відеодані) обробляється для знаходження швидкості кожної точки зображення або навіть самої камери, яка й відтворює зйомку. Прикладами таких задач є:

· одометрія — визначення руху камери (її переміщення і обертання) в тривимірному просторі на основі серії знімків;

· стеження, тобто, слідкування за переміщенням об'єкта (наприклад, хмар чи покупців магазинів);

· оптичний потік — визначення руху кожної точки зображення відносно площини зображення, тобто видимий рух, що є результатом руху як самої точки так і камери [1].

1.2.3 Відновлення зображень

Задача відновлення зображень — це видалення шумів (шум датчика, розмитість об'єкта, що рухається тощо). Найпростішим підходом до вирішення цієї задачі є різноманітні типи фільтрів, таких як фільтри низьких чи середніх частот. Складніші методи використовують для уявлення того, як повинні виглядати ті або інші ділянки зображення, і на основі цього їх перетворення.

Більш високий рівень видалення шумів досягається за допомогою таких методів: первинний аналіз відеоданих на наявність різноманітних структур, таких як лінії чи межі, управління процесом фільтрації на основі цих даних [1].

1.2.3.1 Методи усунення шуму

Залежно від розглянутої моделі шуму складність вирішення такого завдання може істотно змінюватися. У практичних завданнях, як правило, має сенс використовувати моделі імпульсного і адитивного гаусівського шуму, так як вони близькі до найбільш поширених у реальному світі шумів. Адитивний шум

Гауса заснований на використанні нормально розподіленої випадкової величини з нульовим математичним очікуванням, значення якої додаються до кожного пікселя зображення. Ця модель описує шум, який природним чином виникає при захопленні зображення цифровими сенсорами, і для якого існують добре вивчені способи шумозаглушення - високу ефективність мають класичні лінійні фільтри, наприклад фільтр Вінера [13] однак, разом з шумом фільтрації схильні також дрібні деталі, і в контексті даної задачі може виникнути необхідність використання нелінійних методів, таких як алгоритми анізотропної дифузії, білатеральні і трілатеральні фільтри [14].

Перераховані вище методи базуються на локалізованої оцінці градієнта зображення, наявності контурів і дрібних деталей, що в подальшому дозволяє послабити згладжування цих ділянок і зберегти більшу кількість деталей.

Імпульсний шум виражається в неправильному (фіксованому або випадковому) значенні частини пікселів зображення. Як правило, подібний шум виникає через помилки при передачі інформації. Для такої моделі ефективними є ранжуючі фільтри [15], метою яких є виявлення імпульсної помилки і коригування її з використанням оцінки на базі наявних даних, при цьому неушкоджені пікселі залишаються недоторканими. У якості більш загального рішення, що підходить для придушення як гаусівських, так і імпульсних шумів, використовуються адаптивні алгоритми, що використовують деяку околицю пікселя для визначення виду і коригування шуму, властивого центру цієї околиці.

1.3 Згорткові нейронні мережі

Для того, щоб обробляти такі складні об’єкти як фотографії, використовуються згорткові мережі. Вони дозволяють нам спростити

зображення, а таким чином і зменшити об’єм інформації, що отримує для обробки наш комп’ютер. Наприклад, згорткові мережі можуть усереднити значення дев’ятьох пікселів, що знаходяться один поряд із іншим, або відкидають червоний колір на фотографіях, тому що він неважливий для надання потрібного результату.

Ці мережі – це додатковий шар нейронів на вході, який обробляє вхідну інформацію для зрозумілого та спрощеного вигляду мережі.

Винахідники згорткових мереж надихалися біологічними процесами, в яких схему з'єднання нейронів відтворювали відповідно до організації зорової кори тварин. Окремі нейрони кори реагують на стимули лише в обмеженій області зорового поля, відомій як рецептивне поле. А далі рецептивні поля різних нейронів частково перекриваються таким чином, що вони покривають усе зорове поле.

У своїй основі згорткову нейронну мережу можна розглядати як нейронну мережу, що користується безліччю ідентичних копій одного і того ж нейрону. У свою чергу це надає мережі можливості мати обмежену кількість параметрів при обчисленні великих моделей.

Рис. 1.1. Вигляд згорткової нейронної мережі

Прийом із декількома копіями одного і того ж нейрону має близьку аналогію з абстракцією функцій в математиці та інформатиці. При програмуванні функція пишеться один раз і потім повторно використовується, тоюто, не потрібно писати один і той же код безліч разів в різних місцях, що прискорює виконання програми і зменшує кількість помилок. Аналогічно згорткова нейронна мережа, одного разу навчивши нейрон, використовує його у інших місцях алгоритму, що в свою чергу полегшує навчання моделі і мінімізує помилки.

Згорткова нейронна мережа використовує порівняно мало попередньої обробки, в порівнянні з іншими алгоритмами класифікування зображень. Це означає, що мережа навчається фільтрів, що в традиційних алгоритмах

розроблялися вручну. Ця незалежність у конструюванні ознак від апріорних знань та людських зусиль є великою перевагою [1].

1.3.1 Конструкція

Згорткова нейронна мережа складається з шарів входу та виходу, а також із декількох прихованих шарів. Приховані шари згорткових мереж зазвичай

складаються зі згорткових шарів, агрегувальних шарів, повноз'єднаних шарів та шарів нормалізації.

Такий процес в нейронних мережах називають як згортку за домовленістю. Із математичної точки зору він є більше взаємною кореляцією, ніж згорткою. Це має значення лише для індексів у матриці, й відтак які ваги на якому індексі розташовуються.

Згорткові шари

Згорткові шари застосовують до входу операцію згортки, передаючи результат до наступного шару. Згортка імітує реакцію окремого нейрону на зоровий стимул.

Кожен згортковий нейрон обробляє дані лише для свого рецептивного поля.

Хоч повноз'єднані нейронні мережі прямого поширення й можливо застосовувати як для навчання ознак, так і для класифікування даних, застосування цієї архітектури до зображень є непрактичним. Було би необхідним дуже велике число нейронів, навіть у поверхневій (протилежній до глибинної) архітектурі, через дуже великі розміри входу, пов'язані з зображеннями, де кожен піксель є відповідною змінною. Наприклад, повноз'єднаний шар для маленького зображення розміром

100 × 100 має 10 000 ваг. Алгоритм згортки дає змогу розв'язати цю проблему, оскільки вона зменшує кількість вільних параметрів, дозволяючи мережі бути глибшою за меншої кількості параметрів. Наприклад, незалежно від розміру зображення, області замощування розміру 5 × 5, кожна з одними й тими ж спільними вагами, вимагають лише 25 вільних параметрів. Таким чином, це розв'язує проблему зникання або вибуху градієнтів у тренуванні традиційних багатошарових нейронних мереж з багатьма шарами за допомогою зворотного поширення.

Агрегувальні шари

Згорткові мережі можуть включати шари локального або глобального агрегування, які об'єднують виходи кластерів нейронів одного шару до іншого нейрону наступного шару. Наприклад, максимізаційне агрегування використовує максимальне значення з кожного з кластерів нейронів попереднього шару. Іншим

прикладом є усереднювальне агрегування, що використовує усереднене значення з кожного з кластерів нейронів попереднього шару.

Повноз'єднані шари

Повноз'єднані шари з'єднують кожен нейрон одного шару з кожним нейроном наступного шару. Це, в принципі, є тим самим, що й традиційна нейронна мережа багатошарового перцептрону.

Ваги

Згорткові нейронні мережі використовують спільні ваги в згорткових шарах, і це означає, що для кожного рецептивного поля шару використовується один і той самий фільтр (банк ваг); це зменшує обсяг необхідної пам'яті та покращує показники продуктивності [1].

1.4 Тонке налаштування

Для багатьох задач, що потрібно реалізувати нейронним мережам, тренувальних даних доступно мало у вільному доступі. А згорткові нейронні мережі зазвичай вимагають великої кількості тренувальних даних, щоби запобігати перенавчанню. Поширеною методикою є тренувати мережу на ширшому наборі даних з пов'язаної області визначення. Щойно параметри мережі зійшлися, виконується додатковий етап тренування із застосуванням даних з області визначення для тонкого налаштування ваг мережі. Це дозволяє згортковим мережам успішно застосовуватися до задач з невеликими тренувальними наборами.

Для тонкого налаштування мережі необхідно виконати наступні дії: ·

Замінити класифікатор попередньо навченої нейронної мережі новим класифікатором, відповідним під нашу задачу;

· Зупинити згорткові шари попередньо навченої нейронної мережі. В результаті ці шари не будуть навчатися;

· Провести навчання складовою мережі з новим класифікатором із новим

набором даних.

· «Розморозити» кілька шарів згорткової частини попередньо навченою нейронною мережею.

· Довчити мережу з розмороженими згортковими шарами на новому наборі даних. Саме цей етап і називається тонким налаштуванням (fine tuning). Тонке налаштування можна проводити тільки після того, як навчений новий класифікатор. У класифікаторі ваги призначаються випадковим чином, тому на перших етапах навчання сигнал про помилку буде дуже великий. Якщо цей сигнал поширюється в згорткову частину мережі, то результат попереднього

навчання може бути втрачено, тому що ваги нейронів будуть сильно змінюватися. Коли ж навчання класифікатора на новому наборі даних завершено, то таких сильних змін ваг вже не буде, і можна переходити до навчання згорткової частини.

1.5 Породжуючі моделі

На відміну від дискримінантних моделей, що використовуються для завдань класифікації або регресії, порожуючі моделі обчислюють розподіл вірогідностей за допомогою навчальних прикладів. Виокремлюючи з цього багатомірного розподілу, породжуючі моделі створюють нові образи, подібні до навчальних. Це означає, що породжуюча модель, навчена на реальних зображеннях осіб, може синтезувати нові зображення подібних осіб. Детальніше про будову таких моделей розказав Ян Гудфеллоу у 2014 році у своєму навчальному курсі з NIPS. Описана ним архітектура, генеративна змагальна мережа (GAN), особливо популярна зараз у науковому світі, оскільки дозволяє наближатись до неконтрольованого навчання. Сети GAN складаються з двох нейромереж: генератору, який отримує на вході випадковий шум і повинний синтезувати вміст (наприклад, зображення), і дискримінатор, який навчається за допомогою зображень і повинен відділяти реальні зображення від несправжніх, створених генератором.

Змагальні мережі можна зрозуміти як гру, по правилам якої генератор повинний поступово навчитися створювати з шумів такі зображення, які

дискримінатор не зможе відрізнити від реальних. Поступово цей підхід став застосовуватися для найрізноманітніших типів даних та завдань.

1.6 Генеративна змагальна мережа

Генеративні змагальні мережі (Generative adversarial networks, GANs) — це клас алгоритмів штучного інтелекту, що використовуються в навчанні без учителя, реалізовані системою двох штучних нейронних мереж, які змагаються одна з одною в рамках гри з нульовою сумою. Вони були запроваджені Яном Ґудфелоу в 2014 році. Ця методика дозволяє створювати фотографії, які для побіжного огляду людиною виглядають як справжні та мають багато реалістичних елементів (хоча в тестах люди можуть відрізнити реальні зображення від згенерованих у багатьох випадках).

Одна мережа генерує кандидатів (генератор), а інша оцінює їх (дискримінатор). [3][4][5][6] Як правило, генеративна мережа навчається будувати відповідності з латентного простору до певного розподілу даних, тоді як дискримінаційна мережа розрізняє представників справжнього розподілу даних та кандидатів, вироблених генератором. Метою тренувальної мережі є збільшення частоти помилок дискримінаційної мережі (тобто «обдурити» дискримінатор шляхом створення нових синтезованих екземплярів, які повинні походити на представників справжнього розподілу даних). [3][7]

На практиці заздалегідь відомий набір даних використовують як початкові навчальні данні для дискримінатора. Навчання дискримінатора передбачає забезпечення його зразками з набору даних, доки він не досягне певного рівня точності. Зазвичай генератор на початку отримує випадково відбирані дані із заздалегідь визначеного латентного простору [4] (наприклад, за допомогою багатовимірного нормального розподілу. Після цього зразки, синтезовані генератором, оцінюються дискримінатором. Метод зворотного поширення

помилки застосовується в обох мережах[5], так що генератор створює кращі

зображення, тоді як дискримінатор стає більш кваліфікованим при визначенні синтезованих зображень.[8] Генератор, як правило, є деконволюційною нейронною мережею, а дискримінатор — згортковою нейронною мережею.

1.6.1 Створення зображень

Припустимо, що у нас є велика колекція зображень: наприклад 1,2 мільйона зображень в наборі даних ImageNet. Слід, однак, мати на увазі, що в майбутньому це може бути велика колекція зображень або відеороликів, отриманих з Інтернету або від пошукових роботів. Якщо ми змінимо розмір кожного зображення так, щоб його ширина і висота дорівнювали 256 (номінальний розмір), наш набір даних буде являти собою один великий блок пікселів розміром 1.200.000x256x256x3 (приблизно 200 Гб). Нижче наведено кілька прикладів зображень з цього набору даних:

Рис. 1.2. Приклад даних із бази ImageNet

Ці зображення є прикладами того, як виглядає наш візуальний світ, і ми називаємо їх «зразками з істинного розподілу даних». Тепер потрібно створити генеративную модель, яку можна було б навчати генерувати подібні зображення з нуля. Зокрема, генеративної моделлю в цьому випадку може бути одна велика нейронна мережа, що виводить зображення, які ми називаємо «зразками з моделі».

1.6.2 DCGAN

Однією з таких новітніх моделей є мережа DCGAN, розроблена вченими на чолі з А. Редфором (модель показана на малюнку нижче). Ця мережа приймає в якості вхідних даних 100 випадкових чисел, отриманих з рівномірного розподілу (ми називаємо їх кодом або прихованими змінними, на схемі вони позначені червоним кольором) і виводить зображення (в даному випадку зображення розміром 64x64x3 справа, позначені зеленим кольором). Покрокова зміна коду тягне за собою відповідні зміни генеруючих зображень. Це означає, що модель досить вивчила властивості зображень, щоб описати, як виглядає світ, а не просто запам'ятати деякі приклади.

Мережа (позначена жовтим) складається з таких стандартних компонентів згортальної нейронної мережі, як шари деконволюції (результат розгортання згортальних шарів), повнозв'язні шари і т.д.:

Рис. 1.3. Схема алгоритму побудови мережі DCGAN

DCGAN ініціалізується випадковими вагами, тому випадковий код, підключений до мережі, генерує повністю випадкове зображення. Однак очевидно,

що в мережі мільйони параметрів, які ми можемо змінити, і наша мета -- навчитися ставити ці параметри так, щоб зразки, створювані мережею з випадкових кодів, були схожими на дані, що використовувалися для її навчання. Інакше кажучи, ми хочемо, щоб розподіл моделі відповідав істинному розподілу даних в просторі зображень.

1.6.3 Навчання генеративної моделі

Припустимо, що ми використовували тільки що ініціалізовувану мережу для генерації 200 зображень, починаючи кожен раз з нового випадкового коду. Питання в тому, як налаштувати параметри мережі, щоб спонукати її в майбутньому створювати кілька більш правдоподібних зразків? Причому мова

йде не про звичайне контрольоване налаштування. Для цих 200 згенерованих зображень у нас немає ніяких явних бажаних вимог, крім того, щоб вони виглядали як справжні. Один із вдалих підходів до вирішення цього завдання - слідувати алгоритму генеративно-змагальної мережі (Generative adversarial network, GAN). Для цього потрібно ввести другу мережу-дискримінатор (як правило, це стандартна згорткова нейронна мережа), яка спробує визначити, чи є вхідне зображення справжнім або згенероване.

Рис. 1.4. Схема алгоритму роботи генератора і дискримінатора

Наприклад, можна вводити в мережу-дискримінатор 200 згенерованих і 200 реальних зображень і навчати її подібно стандартному класифікатором розрізняти два види зображень за джерелом. Хитрість в тому, щоб крім цього застосувати метод зворотного поширення помилки до обох мереж, дискримінатору і генератору, і таким чином дізнатися, як змінити параметри генератора, щоб він міг ускладнити дискримінаторові розпізнавання своїх 200 зразків. Таким чином, ці дві мережі вступають в протиборство: дискримінатор

намагається відрізнити реальні зображення від підроблених, а генератор намагається створити образи, які діскрімінатор прийме за реальні. В результаті мережа-генератор навчиться виводити зображення, відрізнити для дискриминатора від справжніх.

До порівнянні цих розподілів існує ще кілька підходів, про які ми коротко розповімо нижче. Але перед цим ми б хотіли візуально уявити суть процесу навчання на прикладі наступної анімації, яка демонструє зразки з генеративної моделі.

В обох випадках мережу-генератор починає з гучних і хаотичних зразків, але з часом зосереджується і отримує більш правдоподібну статистику зображення:

Рис. 1.5. Мережа VAE вчиться генерувати зображення

1.6.4. Більш загальне формулювання побудови DCGAN

У математичному сенсі це може бути набір даних прикладів *x1,...,xn* в якості зразків з істинного розподілу даних. У наведеному нижче прикладі синя область показує частину простору зображення, яке з високою ймовірністю (при перевищенні деякого порога) містить реальні зображення, а чорними точками позначені точки наших даних (кожна з них є одним зображенням в нашому наборі даних). Тепер наша модель вже описує розподіл *^p(x)* (зелений колір), яке визначається імпліцитно шляхом імпорту точок з нормального (гаусівського) розподілу з нульовим середнім значенням (червоний) і їх меппінга в нейронної (детермінованою) мережі, тобто в нашій генеративній моделі (жовтий).

Наша мережа - це функція з параметрами θ, і зміна цих параметрів буде міняти і налаштовувати генеруючий розподіл зображень. Наступна наша мета - знайти параметри θ, що створюють розподіл близько відповідають істинному розподілу даних (наприклад, що характеризується невеликою втратою даних (дивергенцією Кульбака-Лейблера). Таким чином, можна уявити, що зелений розподіл починається з випадкових значень, а потім в процесі навчання ітеративно змінює параметри θ і прагне розтягнути і стиснути його, щоб домогтися більш точної відповідності синьому розподілу.

Рис. 1.6. Принцип навчання мережі

1.6.5 Три підходи до навчання мережі

Більшість генеративних моделей засновані на описаному вище базовому принципі, але мають безліч відмінностей. Далі розглядаємо ці відмінності на трьох відомих прикладах методів навчання генеративних моделей.

У генеративно-змагальних мережах (GAN), вже згаданих вище, процес навчання являє собою гру між двома різними мережами: мережею-генератором (див. вище) і мережею-дискримінатором, яка прагне визначити, чи є об'єкти прикладами з істинного розподілу $p(x)$ або розподілу моделі $\hat{p}(x)$. Кожен раз, коли діскрімінатор зауважує різницю між двома розподілами, генератор трохи корегує свої параметри, щоб різниця стала менш помітною. В результаті (теоретично), генератор навчиться точно відтворювати справжнє розподіл даних, а дискримінатор, перевіряючи випадкові зразки з двох розподілів, не їхня зможе відрізнити.

Варіаційні автокодери (VAE) дозволяють формалізувати цю проблему в рамках імовірнісних графічних моделей , де ми максимізували нижню межулогарифмічного правдоподібності даних.

У той же час авторегресійні моделі, такі як PixelRNN , вчать мережу моделювати умовний розподіл кожного окремого пікселя з урахуванням

попередніх пікселів (у напрямку вліво і вгору). Цей процес аналогічний включенню пікселів зображення в багатошарову рекуррентную мережу char-rnn , але рекурентні нейрсеті (RNN) працюють із зображенням не в одному напрямку, а в двох - по горизонталі і по вертикалі.

Усі ці підходи мають свої плюси і мінуси. Зокрема, варіаційні автокодери дозволяють не тільки здійснити навчання, а й ефективно виконати байєсовський висновок у складних імовірнісних графічних моделях з прихованими змінними (як приклади щодо складних з недавно розроблених моделей можна розглянути DRAW або Attend Infer Repeat). Однак зразки, згенеровані цими моделями, часто виявляються злегка розмитими. Найчіткіші зображення в даний момент вдається отримати за допомогою мереж GAN, але їх складніше оптимізувати через нестійку динаміку навчання. PixelRNNs відрізняються досить простим і стабільним процесом навчання (softmax loss) і дозволяють домогтися найвищого на сьогоднішній день логарифмічної правдоподібності (тобто правдоподібності генеруючих даних). Але ці системи виявляються неефективними в процесі вибірки (семплінгу) і зазнають труднощів при наданні кодів в низькій роздільній здатності для зображень. Кожна з цих моделей являє перспективну галузь наукових досліджень, і нам треба простежити, як далеко зайдуть ці дослідження в майбутньому.

Як уже згадувалося вище, GAN - вельми багатообіцяюче сімейство генеративних моделей, так як GAN, на відміну від інших методів, здатні генерувати дуже чисті і чіткі зображення і освоювати коди, що містять цінну інформацію про текстури цих зображень. Однак головний принцип GAN - гра, яку ведуть дві мережі, і це важлива (і непроста) умова їх рівноваги. Наприклад, обидві мережі можуть зазнавати труднощів у виборі між двома рішеннями, а мережа-генератор може перестати виконувати свої завдання. У своїй роботі Тім Саліманс, Ієн Гудфеллоу, Войцех Заремба і їх колеги представили ряд нових методів підвищення ефективності навчання GAN. Ці методи дозволяють нам масштабувати GAN і отримувати вдалі зразки розміром 128x128 з ImageNet:

Рис. 1.7. Справжні зображення ImageNet

Рис. 1.8.
Згенеровані зображення за допомогою мережі GAN

РОЗДІЛ 2

СТРУКТУРА ПОРОДЖУЮЧОЇ НЕЙРОННОЇ МЕРЕЖІ

У нашому прикладі ми будемо використовувати базу даних зображень Cat Dataset на kaggle.com. Набір даних CAT містить понад 9000 зображень котів. Для кожного зображення є анотації голови кота з дев'ятьма точками, двома для очей, одним для роту і шістьма для вух.

Дані анотацій зберігаються у файлі з назвою відповідного зображення плюс "кішка" в кінці. Для кожного зображення кота існує один файл анотацій. Для кожного файлу анотацій дані анотацій зберігаються в такій послідовності:

- Кількість балів (за замовчуванням 9)
- Ліве око
- Праве око
- Рот
- Ліве вухо-1
- Ліве вухо-2
- Ліве вухо-3
- Праве вухо-1
- Праве вухо-2
- Праве вухо-3

Виконала Літ. Арк. Аркушів Бовт М.О.

Керівник Породжуюча модель 29
Моденов Ю.Б. 35
Консульт. Райчев
побудови образів І.Е. УС-211М 122

Н.
контр.

Рис. 2.1 Приклад фотографій, що надає база даних Cat Dataset

Для того, щоб обробити такі великі об’єми даних, я буду користуватися

виділеним сервером з відеокартою GPU.

```
## Download CAT dataset
# wget -nc http://www.simoninithomas.com/data/cats.zip

## Setting up folder
unzip cats-dataset.zip -d cat_dataset
unzip cats.zip
mv cat_dataset/CAT_00/* cat_dataset
```



```
rmdir cat_dataset/CAT_00
mv cat_dataset/CAT_01/* cat_dataset
rmdir cat_dataset/CAT_01
mv cat_dataset/CAT_02/* cat_dataset
rmdir cat_dataset/CAT_02
mv cat_dataset/CAT_03/* cat_dataset
rmdir cat_dataset/CAT_03
mv cat_dataset/CAT_04/* cat_dataset
rmdir cat_dataset/CAT_04
mv cat_dataset/CAT_05/* cat_dataset
rmdir cat_dataset/CAT_05
mv cat_dataset/CAT_06/* cat_dataset
rmdir cat_dataset/CAT_06

## Error correction
```

rm cat_dataset/00000003_019.jpg.cat

mv 00000003_015.jpg.cat cat_dataset/00000003_015.jpg.cat

Removing outliers

Corrupted, drawings, badly cropped, inverted, impossible to tell it's a cat, blocked face

cd cat_dataset

rm 00000004_007.jpg 00000007_002.jpg 00000045_028.jpg 00000050_014.jpg 00000056_013.jpg 00000059_002.jpg 00000108_005.jpg 00000122_023.jpg 00000126_005.jpg 00000132_018.jpg 00000142_024.jpg 00000142_029.jpg 00000143_003.jpg 00000145_021.jpg 00000166_021.jpg 00000169_021.jpg 00000186_002.jpg 00000202_022.jpg 00000208_023.jpg 00000210_003.jpg 00000229_005.jpg 00000236_025.jpg 00000249_016.jpg 00000254_013.jpg 00000260_019.jpg 00000261_029.jpg 00000265_029.jpg 00000271_020.jpg 00000282_026.jpg 00000316_004.jpg 00000352_014.jpg 00000400_026.jpg 00000406_006.jpg 00000431_024.jpg 00000443_027.jpg 00000502_015.jpg 00000504_012.jpg 00000510_019.jpg 00000514_016.jpg 00000514_008.jpg 00000515_021.jpg 00000519_015.jpg 00000522_016.jpg 00000523_021.jpg 00000529_005.jpg 00000556_022.jpg 00000574_011.jpg 00000581_018.jpg 00000582_011.jpg 00000588_016.jpg 00000588_019.jpg 00000590_006.jpg 00000592_018.jpg 00000593_027.jpg 00000617_013.jpg 00000618_016.jpg 00000619_025.jpg 00000622_019.jpg 00000622_021.jpg 00000630_007.jpg 00000645_016.jpg 00000656_017.jpg 00000659_000.jpg 00000660_022.jpg 00000660_029.jpg 00000661_016.jpg 00000663_005.jpg 00000672_027.jpg 00000673_027.jpg 00000675_023.jpg 00000692_006.jpg 00000800_017.jpg 00000805_004.jpg 00000807_020.jpg 00000823_010.jpg 00000824_010.jpg 00000836_008.jpg 00000843_021.jpg 00000850_025.jpg 00000862_017.jpg 00000864_007.jpg 00000865_015.jpg 00000870_007.jpg 00000877_014.jpg 00000882_013.jpg 00000887_028.jpg 00000893_022.jpg

00000907_013.jpg 00000921_029.jpg 00000929_022.jpg 00000934_006.jpg
00000960_021.jpg 00000976_004.jpg 00000987_000.jpg 00000993_009.jpg

38

00001006_014.jpg 00001008_013.jpg 00001012_019.jpg 00001014_005.jpg
00001020_017.jpg 00001039_008.jpg 00001039_023.jpg 00001048_029.jpg
00001057_003.jpg 00001068_005.jpg 00001113_015.jpg 00001140_007.jpg
00001157_029.jpg 00001158_000.jpg 00001167_007.jpg 00001184_007.jpg
00001188_019.jpg 00001204_027.jpg 00001205_022.jpg 00001219_005.jpg
00001243_010.jpg 00001261_005.jpg 00001270_028.jpg 00001274_006.jpg
00001293_015.jpg 00001312_021.jpg 00001365_026.jpg 00001372_006.jpg
00001379_018.jpg 00001388_024.jpg 00001389_026.jpg 00001418_028.jpg
00001425_012.jpg 00001431_001.jpg 00001456_018.jpg 00001458_003.jpg
00001468_019.jpg 00001475_009.jpg 00001487_020.jpg

rm 00000004_007.jpg.cat 00000007_002.jpg.cat 00000045_028.jpg.cat
00000050_014.jpg.cat 00000056_013.jpg.cat 00000059_002.jpg.cat
00000108_005.jpg.cat 00000122_023.jpg.cat 00000126_005.jpg.cat
00000132_018.jpg.cat 00000142_024.jpg.cat 00000142_029.jpg.cat
00000143_003.jpg.cat 00000145_021.jpg.cat 00000166_021.jpg.cat
00000169_021.jpg.cat 00000186_002.jpg.cat 00000202_022.jpg.cat
00000208_023.jpg.cat 00000210_003.jpg.cat 00000229_005.jpg.cat
00000236_025.jpg.cat 00000249_016.jpg.cat 00000254_013.jpg.cat
00000260_019.jpg.cat 00000261_029.jpg.cat 00000265_029.jpg.cat
00000271_020.jpg.cat 00000282_026.jpg.cat 00000316_004.jpg.cat
00000352_014.jpg.cat 00000400_026.jpg.cat 00000406_006.jpg.cat
00000431_024.jpg.cat 00000443_027.jpg.cat 00000502_015.jpg.cat
00000504_012.jpg.cat 00000510_019.jpg.cat 00000514_016.jpg.cat
00000514_008.jpg.cat 00000515_021.jpg.cat 00000519_015.jpg.cat
00000522_016.jpg.cat 00000523_021.jpg.cat 00000529_005.jpg.cat

39

00000556_022.jpg.cat 00000574_011.jpg.cat 00000581_018.jpg.cat
00000582_011.jpg.cat 00000588_016.jpg.cat 00000588_019.jpg.cat

00000590_006.jpg.cat 00000592_018.jpg.cat 00000593_027.jpg.cat
00000617_013.jpg.cat 00000618_016.jpg.cat 00000619_025.jpg.cat
00000622_019.jpg.cat 00000622_021.jpg.cat 00000630_007.jpg.cat
00000645_016.jpg.cat 00000656_017.jpg.cat 00000659_000.jpg.cat
00000660_022.jpg.cat 00000660_029.jpg.cat 00000661_016.jpg.cat
00000663_005.jpg.cat 00000672_027.jpg.cat 00000673_027.jpg.cat
00000675_023.jpg.cat 00000692_006.jpg.cat 00000800_017.jpg.cat
00000805_004.jpg.cat 00000807_020.jpg.cat 00000823_010.jpg.cat
00000824_010.jpg.cat 00000836_008.jpg.cat 00000843_021.jpg.cat
00000850_025.jpg.cat 00000862_017.jpg.cat 00000864_007.jpg.cat
00000865_015.jpg.cat 00000870_007.jpg.cat 00000877_014.jpg.cat
00000882_013.jpg.cat 00000887_028.jpg.cat 00000893_022.jpg.cat
00000907_013.jpg.cat 00000921_029.jpg.cat 00000929_022.jpg.cat
00000934_006.jpg.cat 00000960_021.jpg.cat 00000976_004.jpg.cat
00000987_000.jpg.cat 00000993_009.jpg.cat 00001006_014.jpg.cat
00001008_013.jpg.cat 00001012_019.jpg.cat 00001014_005.jpg.cat
00001020_017.jpg.cat 00001039_008.jpg.cat 00001039_023.jpg.cat
00001048_029.jpg.cat 00001057_003.jpg.cat 00001068_005.jpg.cat
00001113_015.jpg.cat 00001140_007.jpg.cat 00001157_029.jpg.cat
00001158_000.jpg.cat 00001167_007.jpg.cat 00001184_007.jpg.cat
00001188_019.jpg.cat 00001204_027.jpg.cat 00001205_022.jpg.cat
00001219_005.jpg.cat 00001243_010.jpg.cat 00001261_005.jpg.cat
00001270_028.jpg.cat 00001274_006.jpg.cat 00001293_015.jpg.cat

00001312_021.jpg.cat 00001365_026.jpg.cat 00001372_006.jpg.cat
00001379_018.jpg.cat 00001388_024.jpg.cat 00001389_026.jpg.cat
00001418_028.jpg.cat 00001425_012.jpg.cat 00001431_001.jpg.cat
00001456_018.jpg.cat 00001458_003.jpg.cat 00001468_019.jpg.cat
00001475_009.jpg.cat 00001487_020.jpg.cat

cd ..

```
## Preprocessing and putting in folders for different image sizes
mkdir cats_bigger_than_64x64
mkdir cats_bigger_than_128x128
python preprocess_cat_dataset.py
```

2.1 GPU сервер

Графічний процесор (Graphics Proccesing Unit, GPU) — окремий пристрій персонального комп'ютера або ігрової приставки, виконує графічний рендеринг. Сучасні графічні процесори дуже ефективно обробляють і зображують комп'ютерну графіку, завдяки спеціалізованій конвеєрній архітектурі вони набагато ефективніші в обробці графічної інформації, ніж типовий центральний процесор.

Графічний процесор в сучасних відеоадаптерах використовується як прискорювач тривимірної графіки, але в деяких випадках його можна використовувати і для обрахунків (GPGPU). Обчислювальними особливостями в порівнянні із CPU (центральний процесор) є:

· архітектура, максимально націлена на збільшення швидкості обчислень текстур та складних графічних об'єктів;

· обмежений список команд.

Висока обчислювальна потужність GPU пояснюється особливостями архітектури. Сучасні CPU містять невелику кількість ядер , тоді як графічний процесор спочатку створювався як многопоточная структура з безліччю ядер. Різниця в архітектурі обумовлює і різницю в принципах роботи. Якщо архітектура CPU передбачає послідовну обробку інформації, то GPU історично призначався для обробки комп'ютерної графіки, тому розрахований на масивно паралельних обчислень .

Кожна з цих двох архітектур має свої переваги. CPU краще працює з

послідовними завданнями. При великому обсязі оброблюваної інформації очевидна перевага має GPU. Умова тільки одне - в завданні повинен спостерігатися паралелізм.

Сучасні моделі графічних процесорів (в складі відеоадаптера) можуть повноцінно застосовуватися для загальних обчислень (див. GPGPU). Прикладами таких процесорів можуть служити чіпи 5700XT (від AMD) або GTX 1660 Super (від NVIDIA).

2.2 Вхідні дані

Перед тим, як перейти до самого процесу опису створення мережі, давайте ознайомимося з загальними рекомендаціями, якими далі скористаємося. DCGAN - модифікація алгоритму GAN, в основі якого лежать згорткові нейронні мережі (CNN). Завдання пошуку зручного представлення ознак на

великих обсягах нерозмічених даних є однією з найбільш активних сфер досліджень, зокрема уявлення зображень і відео. Одним із зручних способів пошуку уявлень може бути DCGAN. Використання згортальних нейронних мереж безпосередньо не давало хороших результатів, тому було внесено обмеження на шари згорток. Ці обмеження і лежать в основі DCGAN:

· заміна всіх пулінгова шарів на страйдінгові згортки (strided convolutions) в дискримінаторі і частково-страйдінгові згортки (fractional strided-convolutions) в генераторі, що дозволяє мережам знаходити відповідні зниження і підвищення розмірностей;

· використання батчінгової нормалізації для генератора і дискримінатора, тобто нормалізація входу так, щоб середнє значення дорівнювало нулю і дисперсія дорівнювала одиниці. Не варто використовувати батч-нормалізацію для вихідного шару генератора і вхідного шару дискримінатора.

· видалення всіх повнозв'язних прихованих рівнів для глибших

архітектур;

· використання ReLU як функції активації в генераторі для всіх шарів, крім останнього, де використовується tanh;

· використання LeakyReLU як функції активації в дискримінаторі для всіх шарів.

Приступимо до побудови: створимо заглушки для вхідних даних: inputs_real для дискримінатора і inputs_z для генератора. Зверніть увагу, що у нас буде дві швидкості навчання (learning rates), окремо для генератора і дискримінатора.

DCGAN'и дуже чутливі до гіперпараметрів, тому дуже важливо тонко їх налаштувати.

```
def model_inputs(real_dim, z_dim):
def discriminator(x, is_reuse=False, alpha = 0.2):
''' Build the discriminator network.

Arguments
---------
x : Input tensor for the discriminator
n_units: Number of units in hidden layer
reuse : Reuse the variables with tf.variable_scope
alpha : leak parameter for leaky ReLU

Returns
-------
out, logits:
'''
```

```
with tf.variable_scope("discriminator", reuse = is_reuse):

    # Input layer 128*128*3 --> 64x64x64
    # Conv --> BatchNorm --> LeakyReLU
    conv1 = tf.layers.conv2d(inputs = x,
    filters = 64,
    kernel_size = [5,5],
```



```
    strides = [2,2],
    padding = "SAME",

kernel_initializer=tf.truncated_normal_initializer(stddev=0.02),
    name='conv1')

    batch_norm1 = tf.layers.batch_normalization(conv1,
    training = True,
    epsilon = 1e-5,
    name = 'batch_norm1')

    conv1_out = tf.nn.leaky_relu(batch_norm1, alpha=alpha,
name="conv1_out")


    # 64x64x64--> 32x32x128
    # Conv --> BatchNorm --> LeakyReLU
    conv2 = tf.layers.conv2d(inputs = conv1_out,
    filters = 128,
    kernel_size = [5, 5],
    strides = [2, 2],
    padding = "SAME",
```

kernel_initializer=tf.truncated_normal_initializer(stddev=0.02),
name='conv2')

batch_norm2 = tf.layers.batch_normalization(conv2,
training = True,
epsilon = 1e-5,
name = 'batch_norm2')
conv2_out = tf.nn.leaky_relu(batch_norm2, alpha=alpha,
name="conv2_out")

 # 32x32x128 --> 16x16x256
 # Conv --> BatchNorm --> LeakyReLU
conv3 = tf.layers.conv2d(inputs = conv2_out,
filters = 256,
kernel_size = [5, 5],
strides = [2, 2],
padding = "SAME",

kernel_initializer=tf.truncated_normal_initializer(stddev=0.02),
name='conv3')

batch_norm3 = tf.layers.batch_normalization(conv3,
training = True,
epsilon = 1e-5,
name = 'batch_norm3')

conv3_out = tf.nn.leaky_relu(batch_norm3, alpha=alpha,
name="conv3_out")

```
        # 16x16x256 --> 16x16x512
        # Conv --> BatchNorm --> LeakyReLU
        conv4 = tf.layers.conv2d(inputs = conv3_out,
        filters = 512,
        kernel_size = [5, 5],
        strides = [1, 1],
        padding = "SAME",

kernel_initializer=tf.truncated_normal_initializer(stddev=0.02),
        name='conv4')

        batch_norm4 = tf.layers.batch_normalization(conv4,
        training = True,
        epsilon = 1e-5,
        name = 'batch_norm4')
        conv4_out = tf.nn.leaky_relu(batch_norm4, alpha=alpha,
name="conv4_out")
```



```
        # 16x16x512 --> 8x8x1024
        # Conv --> BatchNorm --> LeakyReLU
        conv5 = tf.layers.conv2d(inputs = conv4_out,
        filters = 1024,
        kernel_size = [5, 5],
        strides = [2, 2],
        padding = "SAME",
```

```
kernel_initializer=tf.truncated_normal_initializer(stddev=0.02),
        name='conv5')

         batch_norm5 = tf.layers.batch_normalization(conv5,
        training = True,
         epsilon = 1e-5,
         name = 'batch_norm5')
         conv5_out = tf.nn.leaky_relu(batch_norm5, alpha=alpha,
name="conv5_out")



         # Flatten it
         flatten = tf.reshape(conv5_out, (-1, 8*8*1024))
         # Logits
         logits = tf.layers.dense(inputs = flatten,
         units = 1,
```



```
         activation = None)


         out = tf.sigmoid(logits)

        return out, logits
```

2.3 Дискримінатор і генератор

Ми використовуємо tf.variable_scope з двох причин. По-перше, щоб бути впевненими, що імена всіх змінних починаються з generator / discriminator. Пізніше це нам допоможе при навчанні двох нейромереж. По-друге, ми будемо повторно використовувати ці мережі з різними вхідними даними:

· Ми навчимо генератор, а потім візьмемо зразок згенерованих їм

зображень.

· У дискримінаторі будемо спільно використовувати змінні для фальшивих і справжніх вхідних зображень.

2.3.1 Дискримінатор

Рис. 2.2.

Схема побудови дискримінатора

Давайте створимо дискримінатор. У якості вхідних даних він бере реальне або фальшиве зображення і у відповідь видає 0 або 1.

Кілька приміток:

· Нам потрібно подвоїти розмір фільтра в кожному згортковому шарі.

· Не рекомендується використовувати знижувальну вибірку (downsampling). Замість цього застосовуються лише згорткові шари субдискретизації (strided convolutional layers).

· У кожному шарі використовуємо пакетну нормалізацію (batch normalization) (за винятком вхідного шару), оскільки це знижує коваріаційний зрушення (covariance shift).

· У якості функції активації скористаємося Leaky ReLU, це допоможе уникнути ефекту «зникаючого» градієнту.

def discriminator(x, is_reuse=False, alpha = 0.2):

''' Build the discriminator network.

Arguments

----

x : Input tensor for the discriminator

n_units: Number of units in hidden layer

reuse : Reuse the variables with tf.variable_scope

alpha : leak parameter for leaky ReLU

Returns

----

out, logits:

'''

with tf.variable_scope("discriminator", reuse = is_reuse):

*# Input layer 128*128*3 --> 64x64x64*

 # Conv --> BatchNorm --> LeakyReLU

conv1 = tf.layers.conv2d(inputs = x,

filters = 64,

kernel_size = [5,5],

strides = [2,2],

padding = "SAME",

kernel_initializer=tf.truncated_normal_initializer(stddev=0.02),

```
        name='conv1')
```



```
        batch_norm1 = tf.layers.batch_normalization(conv1,
        training = True,
        epsilon = 1e-5,
        name = 'batch_norm1')

        conv1_out = tf.nn.leaky_relu(batch_norm1, alpha=alpha,
name="conv1_out")


        # 64x64x64--> 32x32x128
        # Conv --> BatchNorm --> LeakyReLU
        conv2 = tf.layers.conv2d(inputs = conv1_out,
        filters = 128,
        kernel_size = [5, 5],
        strides = [2, 2],
        padding = "SAME",

kernel_initializer=tf.truncated_normal_initializer(stddev=0.02),
        name='conv2')

        batch_norm2 = tf.layers.batch_normalization(conv2,
        training = True,
        epsilon = 1e-5,
        name = 'batch_norm2')
```



```
        conv2_out = tf.nn.leaky_relu(batch_norm2, alpha=alpha,
name="conv2_out")
```

```
        # 32x32x128 --> 16x16x256
        # Conv --> BatchNorm --> LeakyReLU
        conv3 = tf.layers.conv2d(inputs = conv2_out,
        filters = 256,
        kernel_size = [5, 5],
        strides = [2, 2],
        padding = "SAME",

kernel_initializer=tf.truncated_normal_initializer(stddev=0.02),
        name='conv3')

        batch_norm3 = tf.layers.batch_normalization(conv3,
        training = True,
        epsilon = 1e-5,
        name = 'batch_norm3')
        conv3_out = tf.nn.leaky_relu(batch_norm3, alpha=alpha,
name="conv3_out")
```



```
        # 16x16x256 --> 16x16x512
        # Conv --> BatchNorm --> LeakyReLU
        conv4 = tf.layers.conv2d(inputs = conv3_out,
        filters = 512,
        kernel_size = [5, 5],
        strides = [1, 1],
        padding = "SAME",
```

```
kernel_initializer=tf.truncated_normal_initializer(stddev=0.02),
        name='conv4')

         batch_norm4 = tf.layers.batch_normalization(conv4,
        training = True,
         epsilon = 1e-5,
         name = 'batch_norm4')
         conv4_out = tf.nn.leaky_relu(batch_norm4, alpha=alpha,
name="conv4_out")



         # 16x16x512 --> 8x8x1024
         # Conv --> BatchNorm --> LeakyReLU
         conv5 = tf.layers.conv2d(inputs = conv4_out,
         filters = 1024,
         kernel_size = [5, 5],
```



```
         strides = [2, 2],
         padding = "SAME",

kernel_initializer=tf.truncated_normal_initializer(stddev=0.02),
        name='conv5')

         batch_norm5 = tf.layers.batch_normalization(conv5,
        training = True,
         epsilon = 1e-5,
         name = 'batch_norm5')
         conv5_out = tf.nn.leaky_relu(batch_norm5, alpha=alpha,
name="conv5_out")
```

```
# Flatten it
flatten = tf.reshape(conv5_out, (-1, 8*8*1024))
# Logits
logits = tf.layers.dense(inputs = flatten,
units = 1,
activation = None)

out = tf.sigmoid(logits)

return out, logits
```


2.3.2 Генератор

Рис. 2.3. Схема побудови генератора

Наступним кроком ми створюємо генератор. Ми пам'ятаємо, що він бере в якості вхідних даних вектор шуму (z) і завдяки транспонованим згортковим шарам (transposed convolution layers) створює фальшиве зображення.

Ткаож на кожному шарі ми зменшуємо розмір фільтра вдвічі, і також вдвічі збільшуємо розмір картинки.

Найкраще генератор працює при використанні tanh (гіперболічна функція) в якості вихідної функції активації (output activation function).

```
def generator(z, output_channel_dim, is_train=True):
 ''' Build the generator network.
```

Arguments

----

z : Input tensor for the generator

output_channel_dim : Shape of the generator output

n_units : Number of units in hidden layer

reuse : Reuse the variables with tf.variable_scope

alpha : leak parameter for leaky ReLU

Returns

----

out:

'''

```
with tf.variable_scope("generator", reuse= not is_train):
# First FC layer --> 8x8x1024
fc1 = tf.layers.dense(z, 8*8*1024)

# Reshape it
fc1 = tf.reshape(fc1, (-1, 8, 8, 1024))

# Leaky ReLU
fc1 = tf.nn.leaky_relu(fc1, alpha=alpha)


# Transposed conv 1 --> BatchNorm --> LeakyReLU #
8x8x1024 --> 16x16x512
trans_conv1 = tf.layers.conv2d_transpose(inputs = fc1,
filters = 512,
kernel_size = [5,5],
strides = [2,2],
```

```
        padding = "SAME",
```



```
    kernel_initializer=tf.truncated_normal_initializer(stddev=0.02),
                                name="trans_conv1")

        batch_trans_conv1 = tf.layers.batch_normalization(inputs =
trans_conv1, training=is_train, epsilon=1e-5, name="batch_trans_conv1")
        trans_conv1_out = tf.nn.leaky_relu(batch_trans_conv1,
alpha=alpha, name="trans_conv1_out")


        # Transposed conv 2 --> BatchNorm --> LeakyReLU  #
        16x16x512 --> 32x32x256
        trans_conv2 = tf.layers.conv2d_transpose(inputs =
trans_conv1_out,
        filters = 256,
        kernel_size = [5,5],
        strides = [2,2],
        padding = "SAME",

    kernel_initializer=tf.truncated_normal_initializer(stddev=0.02),
                                name="trans_conv2")

        batch_trans_conv2 = tf.layers.batch_normalization(inputs =
trans_conv2, training=is_train, epsilon=1e-5, name="batch_trans_conv2")
```



```
        trans_conv2_out = tf.nn.leaky_relu(batch_trans_conv2,
alpha=alpha, name="trans_conv2_out")


        # Transposed conv 3 --> BatchNorm --> LeakyReLU  #
```

```
        32x32x256 --> 64x64x128
        trans_conv3 = tf.layers.conv2d_transpose(inputs =
trans_conv2_out,
        filters = 128,
        kernel_size = [5,5],
        strides = [2,2],
        padding = "SAME",

 kernel_initializer=tf.truncated_normal_initializer(stddev=0.02),
                        name="trans_conv3")

        batch_trans_conv3 = tf.layers.batch_normalization(inputs =
trans_conv3, training=is_train, epsilon=1e-5, name="batch_trans_conv3")
        trans_conv3_out = tf.nn.leaky_relu(batch_trans_conv3,
alpha=alpha, name="trans_conv3_out")


        # Transposed conv 4 --> BatchNorm --> LeakyReLU  #
        64x64x128 --> 128x128x64
```



```
        trans_conv4 = tf.layers.conv2d_transpose(inputs =
trans_conv3_out,
        filters = 64,
        kernel_size = [5,5],
        strides = [2,2],
        padding = "SAME",

 kernel_initializer=tf.truncated_normal_initializer(stddev=0.02),
                        name="trans_conv4")
```

```
        batch_trans_conv4 = tf.layers.batch_normalization(inputs =
trans_conv4, training=is_train, epsilon=1e-5, name="batch_trans_conv4")
        trans_conv4_out = tf.nn.leaky_relu(batch_trans_conv4,
alpha=alpha, name="trans_conv4_out")


        # Transposed conv 5 --> tanh
        # 128x128x64 --> 128x128x3
        logits = tf.layers.conv2d_transpose(inputs = trans_conv4_out,
       filters = 3,
        kernel_size = [5,5],
        strides = [1,1],
        padding = "SAME",

kernel_initializer=tf.truncated_normal_initializer(stddev=0.02),
```



```
        name="logits")

        out = tf.tanh(logits, name="out")

        return out
```

2.3.3 Втрати в генераторі та дискримінаторі

Оскільки ми одночасно навчаємо генератор і дискримінатор, нам потрібно обчислити втрати для обох нейромереж. Діскрімінатор повинен видавати 1, коли він «вважає» зображення справжнім, і 0, якщо зображення фальшиве. Відповідно до цього і потрібно налаштувати втрати. Втрата дискриминатора обчислюється як сума втрат для справжнього і фальшивого зображення:

```
d_loss=d_loss_real+d_loss_fake
```

де d_loss_real - це втрата, коли діскрімінатор вважає зображення фальшивим, а насправді воно справжнє. Обчислюється так:

· Використовуємо d_logits_real, всі мітки рівні 1 (тому що всі дані справжні).

· labels = tf.ones_like(tensor) * (1 - smooth). Скористаємося label smoothing: зменшимо значення міток з 1,0 до 0,9, щоб допомогти дискримінатору узагальнювати краще.

· d_loss_fake - це втрата, коли дискрімінатор вважає зображення справжнім, а насправді воно фальшиве.

· Використовуємо d_logits_fake, всі мітки рівні 0.

Для втрати генератора використовується d_logits_fakez дискримінатора. На цей раз все мітки рівні 1, тому що генератор хоче обдурити дискримінатор.

```
def model_loss(input_real, input_z, output_channel_dim, alpha):
    """
    Get the loss for the discriminator and generator
    :param input_real: Images from the real dataset
    :param input_z: Z input
    :param out_channel_dim: The number of channels in the output image
    :return: A tuple of (discriminator loss, generator loss)
    """
    # Generator network here
    g_model = generator(input_z, output_channel_dim)
    # g_model is the generator output

    # Discriminator network here
    d_model_real, d_logits_real = discriminator(input_real,
alpha=alpha)
```

d_model_fake, d_logits_fake = discriminator(g_model,is_reuse=True, alpha=alpha)

 # Calculate losses

d_loss_real = tf.reduce_mean(

tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_real, labels=tf.ones_like(d_model_real))) d_loss_fake = tf.reduce_mean(

tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_fake, labels=tf.zeros_like(d_model_fake))) d_loss = d_loss_real + d_loss_fake

g_loss = tf.reduce_mean(
tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_fake, labels=tf.ones_like(d_model_fake)))
return d_loss, g_loss

2.3.4 Оптимізатори

Після обчислення втрат потрібно окремо оновити генератор і дискримінатор. Для цього за допомогою tf.trainable_variables() створимо список всіх змінних, визначених в нашому графі.

def model_optimizers(d_loss, g_loss, lr_D, lr_G, beta1):
"""
Get optimization operations
:param d_loss: Discriminator loss Tensor

:param g_loss: Generator loss Tensor
:param learning_rate: Learning Rate Placeholder

```
    :param beta1: The exponential decay rate for the 1st moment in the
optimizer
    :return: A tuple of (discriminator training operation, generator
training operation)
    """
    # Get the trainable_variables, split into G and D parts
    t_vars = tf.trainable_variables()
    g_vars = [var for var in t_vars if var.name.startswith("generator")]
    d_vars = [var for var in t_vars if
var.name.startswith("discriminator")]

    update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
    # Generator update
    gen_updates = [op for op in update_ops if
op.name.startswith('generator')]

    # Optimizers
    with tf.control_dependencies(gen_updates):
    d_train_opt = tf.train.AdamOptimizer(learning_rate=lr_D,
beta1=beta1).minimize(d_loss, var_list=d_vars)
    g_train_opt = tf.train.AdamOptimizer(learning_rate=lr_G,
beta1=beta1).minimize(g_loss, var_list=g_vars)
```



```
    return d_train_opt, g_train_opt
```

2.3.5 Навчання

Тепер реалізуємо навчальну функцію. Ідея досить проста: · Зберігаємо нашу модель раз в п'ять періодів (epoch). · Зберігаємо картинку в папці з зображеннями через кожні 10 навчених пакетів (batches).

· Через кожні 15 періодів відображаємо g_loss, d_lossi згенероване зображення. Це потрібно тому, що обраний Jupyter notebook може давати збої при відображенні занадто великої кількості картинок.

· Або можемо безпосередньо генерувати справжні зображення, завантажуючи збережену модель (це заощадить 20 годин навчання).

```
def train(epoch_count, batch_size, z_dim, learning_rate_D,
learning_rate_G, beta1, get_batches, data_shape, data_image_mode, alpha):
    """
    Train the GAN
    :param epoch_count: Number of epochs
    :param batch_size: Batch Size
    :param z_dim: Z dimension
    :param learning_rate: Learning Rate
    :param beta1: The exponential decay rate for the 1st moment in the
optimizer
    :param get_batches: Function to get batches
    :param data_shape: Shape of the data
```



```
    :param data_image_mode: The image mode to use for images ("RGB"  or
"L")
    """
    # Create our input placeholders
    input_images, input_z, lr_G, lr_D = model_inputs(data_shape[1:],
z_dim)

    # Losses
    d_loss, g_loss = model_loss(input_images, input_z, data_shape[3],
alpha)

    # Optimizers
```

```
d_opt, g_opt = model_optimizers(d_loss, g_loss, lr_D, lr_G, beta1)
i = 0

version = "firstTrain"
with tf.Session() as sess:
sess.run(tf.global_variables_initializer())

# Saver
 saver = tf.train.Saver()

num_epoch = 0

if from_checkpoint == True:
```



```
saver.restore(sess, "./models/model.ckpt")

show_generator_output(sess, 4, input_z, data_shape[3],
data_image_mode, image_path, True, False)

else:
for epoch_i in range(epoch_count):
num_epoch += 1

if num_epoch % 5 == 0:

# Save model every 5 epochs
#if not os.path.exists("models/" + version):  #
os.makedirs("models/" + version)
save_path = saver.save(sess, "./models/model.ckpt")
print("Model saved")
```

```
        for batch_images in get_batches(batch_size):  #
        Random noise
        batch_z = np.random.uniform(-1, 1, size=(batch_size,  z_dim))

        i += 1

        # Run optimizers
```



```
        _ = sess.run(d_opt, feed_dict={input_images:
batch_images, input_z: batch_z, lr_D: learning_rate_D})
        _ = sess.run(g_opt, feed_dict={input_images:
batch_images, input_z: batch_z, lr_G: learning_rate_G})

        if i % 10 == 0:
        train_loss_d = d_loss.eval({input_z: batch_z,  input_images:
batch_images})
                train_loss_g = g_loss.eval({input_z: batch_z})

        # Save it
        image_name = str(i) + ".jpg"
        image_path = "./images/" + image_name
        show_generator_output(sess, 4, input_z, data_shape[3],
data_image_mode, image_path, True, False)

        # Print every 5 epochs (for stability overwize the jupyter  notebook will
bug)
        if i % 1500 == 0:

        image_name = str(i) + ".jpg"
        image_path = "./images/" + image_name
```

print("Epoch {}/{}...".format(epoch_i+1, epochs), "Discriminator Loss: {:.4f}...".format(train_loss_d), "Generator Loss: {:.4f}".format(train_loss_g))

show_generator_output(sess, 4, input_z, data_shape[3], data_image_mode, image_path, False, True)

return losses, samples

Все це можна запустити прямо на своєму комп'ютері, але виконання займатиме приблизно 10 років. Тому краще скористатися хмарними GPU сервісами на зразок AWS або FloydHub. Я навчала цю DCGAN протягом 20 годин на Microsoft Azure і їх Deep Learning Virtual Machine .

2.4 Результат роботи породжуючої нейронної мережі DCGAN

Отже, у процесі роботи ми отримуємо даний датасет з абсолютно новими зображеннями котів, які не є реальними, але якщо не знати того, що їх «намалювала» нейронна мережа, то можна запросто вирішити, що це реальні, існуючі персонажі. Погляньте:

Рис. 2.4 Результат роботи породжуючої нейронної мережі

Ми бачимо на цих фото достатньо помітний «шум», це і є той самий шум z, який був на вході генеративної мережі. Звичайно, спочатку фотографії були

настільки спотворені ним, що мережа Дискримінатор з легкістю могла визначити, що це фальшиві дані. Проте у процесі багатьох ітерацій мережа суперниця навчилася створювати нові образи настільки, що цих котів і людина може вважати реальними.

ВИСНОВКИ

У ході дипломного проектування була створена породжуюча нейронна інформаційна мережа, що створює абсолютно реальні образи котів, попередньо навчившись на базі даних із фотографіями створінь.

Програмна реалізація даної нейронної мережі базується на використанні технологій глибинного навчання представлених у вигляді бібліотек та спеціальних пакетів, що дозволяють програмістам проектувати архітектури мереж та налаштовувати параметри.

Генерування нових образів було відтворено на базі бібліотеки Keras і допоміжного графічного процесору GPU Microsoft Azure і Deep Learning Virtual Machine, що дозволило значно скоротити час на виконання даного алгоритму і збільшило обчислювальну продуктивність завдяки використанню графічного процесору.

На початку моделювання ми розробили чітку послідовність дій. Розробили два алгоритми для наших нейронних мереж, що змагаються між собою: генеративна нейронна мережа і дискримінативна мережа. Далі відтворили потрібні кроки для роботи мережі: описали втрати і оптимізатори. Наприкінці описали процес самого навчання мереж.

Помітно значне зменшення часу навчання нейронної мережі із використанням графічних прискорювачів. Графічні процесори (GPU) краще підходять для машинного навчання, ніж центральні процесори (CPU). Технічні особливості GPU допомагають виконати одночасно безліч матричних операцій, які використовуються саме для навчання нейронних мереж. Для прискорення роботи за зменшення обчислюваної складності при навчанні базової моделі

згорткової нейронної мережі використовувалась платформа Deep Learning Virtual Machine.

Таким чином, процес навчання мережі зайняв 20 годин.

Дана мережа відкриває нескінчений потенціал для подальшого застосування.

Такі процеси лежать у основі алгоритмів, що створюють новий контент і дані. Наприклад, створюють картинки для інтернет-магазинів, аватари для ігр, відеокліпи, згенеровані автоматично, виходячи з музикального біту доріжки, чи навіть віртуальних ведучих для ТБ-каналу. Найчастіше такі мережі використовуються для створень надреалістичних зображень.

Цей підхід також використовується для автоматичного редагування у деяких смартфонах чи програмах. Він дозволяє змінювати вираз обличчя на фото, кількість зморшок, змінювати день на ніч або зістарювати зображення.

Подібні алгоритми можуть давати доволі цікаві результати в майбутньому. Наприклад, допомагатимуть створювати послідовність нот таким чином, що отримуємо мелодію. Чи створюватимуть зображення по заданому тексту. Вони також мужуть покращувати роздільну здатність зображень, що з початку мали низьку роздільну здатність.

Отже, у процесі роботи над розробкою породжуючої моделі побудови образів ознайомилася з алгоритмом генеративної змагальної нейронної мережі. Описала процес її створення та роботи над великим об'ємом даних, а саме – зображень. Показала, які образи дана мережа може будувати після тривалого двадцятигодинного процесу навчання.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ

1. Бовт М. О. Нейронна інформаційна мережа дипл. проект ... ст-тки комп. наук: 6.050101 / Нац. унів. авіац. України. Київ, 2019. 53 с.

2. Salimans, Tim; Goodfellow, Ian; Zaremba, Wojciech; Cheung, Vicki; Radford, Alec; Chen, Xi (2016). «Improved Techniques for Training GANs». arXiv:1606.03498 [cs.LG].

3. Goodfellow, Ian J.; Pouget-Abadie, Jean; Mirza, Mehdi; Xu, Bing; Warde Farley,

David; Ozair, Sherjil; Courville, Aaron; Bengio, Yoshua (2014). «Generative Adversarial Networks». arXiv:1406.2661 [stat.ML].

4. Thaler, SL, US Patent 05659666, Device for the autonomous generation of useful information, 08/19/1997.

5. Thaler, SL, US Patent, 07454388, Device for the autonomous bootstrapping of useful information, 11/18/2008.

6. Thaler, SL, The Creativity Machine Paradigm, Encyclopedia of Creativity, Invention, Innovation, and Entrepreneurship, (ed.) E.G. Carayannis, Springer Science+Business Media, LLC, 2013.

7. Luc, Pauline; Couprie, Camille; Chintala, Soumith; Verbeek, Jakob (2016-11- 25). Semantic Segmentation using Adversarial Networks. NIPS Workshop on Adversarial Training, Dec , Barcelona, Spain 2016. Bibcode:2016arXiv161108408L. arXiv:1611.08408.

8. Andrej Karpathy, Pieter Abbeel, Greg Brockman, Peter Chen, Vicki Cheung, Rocky Duan, Ian Goodfellow, Durk Kingma, Jonathan Ho, Rein Houthooft, Tim

Salimans, John Schulman, Ilya Sutskever, And Wojciech Zaremba. Generative Models. OpenAI. Процитовано April 7, 2016.