МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Західноукраїнський національний університет
Факультет комп'ютерних інформаційних технологій
Кафедра інформаційно-обчислювальних систем і управління

ПАНЧАК Дмитро Вікторович

Метод генерування зображень з використанням генеративно-змагальних мереж / Method of Generating Image Using GAN

спеціальність: 122 - Комп'ютерні науки
освітньо-професійна програма - Комп'ютерні науки

Кваліфікаційна робота

Виконав студент групи
КНм-21
Д.В. Панчак
_____

Науковий керівник:
к.т.н., професор В.В. Кочан
_____

Кваліфікаційну роботу
допущено до захисту:
«___» _____ 20___ р.
Завідувач кафедри
_____ М.П. Комар

ТЕРНОПІЛЬ - 2022

Факультет комп'ютерних інформаційних технологій
Кафедра інформаційно-обчислювальних систем і управління
Освітній ступінь «магістр»
спеціальність: 122 – Комп'ютерні науки
освітньо-професійна програма – Комп'ютерні науки

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ М.П. Комар
«____» _____20__р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ
Панчак Дмитро Вікторович
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи
Метод генерування зображень з використанням генеративно-змагальних мереж / Method of Generating Image Using GAN
керівник роботи к.т.н., проофесор В.В. Кочан
затверджені наказом по університету від 31 грудня 2021 року № 606.

2. Строк подання студентом закінченої кваліфікаційної роботи 16 листопада 2022 року.

3. Вихідні дані до кваліфікаційної роботи: завдання на кваліфікаційну роботу студента, наукові статті, технічна література.

4. Основні питання, які потрібно розробити
- огляд предметної області;
- аналіз штучних нейроних мереж;
- огляд існуючих генеративно змагальних мереж;
- постановка задачі дослідження;
- аналіз архітектур та реалізацій для покращення генеративно змагальних мереж;
- розробка підходів для генерування реалістичних зображень;
- проведення експерементальних досліджень.

5. Перелік графічного матеріалу у роботі
- структурні схеми нейроних мереж;
- приклади згенерованих зображень

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 11 жовтня 2021 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів кваліфікаційної роботи	Примітка
1	Аналіз предметної області і постановка задач	12.2021 р. – 03.2022 р.	
2	Розробка архітектури мережі для генерації зображень	03.2022 р. – 05.2022 р.	
3	Програмна реалізація і експеременти	05.2022 р. – 11.2022 р.	
4	Повне завершення та представлення кваліфікаційної роботи на кафедру	16.11.2022 р.	

Студент _____ Д.В. Панчак
підпис

Керівник роботи _____ к.т.н., професор В.В. Кочан
підпис

РЕЗЮМЕ

Кваліфікаційна робота на тему «Метод генерування зображень з використанням генеративно-змагальних мереж» на здобуття освітнього ступеня «Магістр» зі спеціальності 122 «Комп’ютерні науки» освітньої програми «Комп’ютерні науки» написана обсягом в 93 сторінки і містить 41 ілюстрації, 3 додатки та 44 використаних джерел.

Метою даної кваліфікаційної роботи є розробка методу генерування зображень з використанням генеративно-змагальних мереж.

Методи досліджень: методи комп’ютерного моделювання, математичного програмування; системного аналізу; машинного навчання, штучного інтелекту.

Результати дослідження: полягають в отримані згенерованих реалістичних зображень певної роздільної здатності. Ці зображення можуть використовуватись в подальших дослідженнях, а саме в анонімізації медичних досліджень або інших.

Результати роботи можуть успішно застосовуватися для використання в будь-якій із багатьох сфер людської діяльності, де використовується зображення.

Ключові слова:ШТУЧНИЙ ІНТЕЛЕКТ, ШТУЧНІ НЕЙРОННІ МЕРЕЖІ, ЗГОРТКОВІ НЕЙРОННІ МЕРЕЖІ, ГЕНЕРАТИВНО ЗМАГАЛЬНІ МЕРЕЖІ

ABSTRACT

Qualification work on the topic «Method of Generating Image Using GAN» for Master's degree on speciality 122 «Computer Science» educational and professional program «Computer Science» is written on 100 pages and it contains 41 figures, 3 annexes and 44 sources.

The purpose of this qualification work is to develop a method of image generation using generative-competitive networks.

Research methods: methods of computer modeling, mathematical programming; system analysis; machine learning, artificial intelligence.

The results of the research: consist in the received generated realistic images of a certain resolution. These images may be used in further research, namely in anonymizing medical or other research.

The results of the work can be successfully applied for use in any of the many areas of human activity where images are used.

Keywords: ARTIFICIAL INTELLIGENCE, ARTIFICIAL NEURAL NETWORKS, CONVOLUTIONAL NEURAL NETWORKS, GENERATIVE COMPETITIVE NETWORKS

ЗМІСТ

ВСТУП

Актуальність теми. Довгий час однією з основних проблем алгоритмів машинного навчання з викладачем (англійське "supervised learning") є проблема даних. Зокрема, навчання нейронних мереж часто вимагає вибірки мільйонів елементів. Якщо розглядати як приклад навчання розпізнаванню образів, то мільйони зображень повинні бути достатньої якості і повинні бути відзначені, що вимагає ручної праці великої кількості людей. Таким чином, непрямі витрати на навчання нейронної мережі розпізнаванню зображень величезні.

Виходом із ситуації може стати алгоритм, здатний генерувати такі зразки без участі людини. Метою даної магістерської дисертації є саме такі алгоритми. За останні кілька років в цій області спостерігається очевидний прогрес завдяки появі генеративних змагальних мереж (GAN). Однак такі нейронні мережі вкрай складні в освоєнні, і питання синтезу зображень у високій роздільній здатності з їх допомогою до сих пір залишається відкритим.

Останні дослідження в області GAN спрямовані на реалізацію методів, які б принесли більшу стабільність в процес навчання такого роду мереж: численні типи нормалізації даних, такі як спектральна нормалізація, піксельна нормалізація і багато інших; різні функції помилок, такі як відстань Вассерштейна, відстань Кулбека-Лейблера тощо.

Також швидкість навчання не є маловажним фактором. Наприклад, розробка вчених NVIDIA «Прогресивно зростаючий GAN», за словами команди, навчена синтезувати зображення з роздільною здатністю 1024 пікселів на 8 максимальних відеокартах NVIDIA V100 (NVI1) протягом 2 днів (Pro).

У даній роботі буде проведений аналіз існуючих підходів в реалізації GAN. На основі найбільш вдалих з них буде реалізований алгоритм, метою якого є синтез зображень в досить високих дозволах. При цьому при виборі підходів, які будуть включені в якості компонентів в отриманий алгоритм, одним з фундаментальних факторів буде швидкість навчання.

Метою роботи є розробка методу генерування зображень з використанням генеративно-змагальних мереж.

Об’єктом дослідження є процес генерування реалістичних зображень за допомогою нейронних мереж а саме генеративно-змагальних.

Предметом дослідження є сукупність теоретико-методологічних та прикладних засад в штучному інтелекті.

Методи дослідження. У роботі використані методи методи комп’ютерного моделювання математичного програмування; системного аналізу; машинного навчання, штучного інтелекту.

Наукова новизна одержаних результатів. Вдосконалено метод генерування зображень з використанням генеративно-змагальних мереж, що дозволило покращити їх роздільну здатність на відміну від відомих рішень.

Практичне значення одержаних результатів полягає у реалізації генеративно-змагальних мереж з метою отримані згенерованих реалістичних зображень певної роздільної здатності.

Апробація результатів дослідження. Основні теоретичні положення роботи й практичні результати дослідження доповідалися й обговорювалися:

- Міжнародної наукової Інтернет-конференції «Інформаційне суспільство: технологічні, економічні та технічні аспекти становлення (випуск 73)», 2022 р.

- Науково-практичній конференції присвяченій пам'яті українського зв'язківця Тітко Валентина Михайловича, м. Київ, 16 листопада 2022 р.

Кваліфікаційна робота складається із вступу, трьох розділів, висновків, списку використаних джерел та додатків.

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ І ПОСТАНОВКА ЗАДАЧІ

1.1 Штучні нейронні мережі

Нейронні мережі (NN), також звані штучними нейронними мережами, є моделями машинного навчання (ML), які імітують принципи роботи людського мозку. NN складається з багатьох простих блоків обробки інформації, які називаються нейронами.

Нейрон, також званий перцептроном, є найпростішим НМ, розробленим у 1958 році Френком Розенблатом (1958) з використанням ідей, представлених МакКалохом і Піттсом у (1943). Нейрон, N_j, отримує n вхідних сигналів (x_i; i=1,...,n) через вхідні з'єднання, як дендрити в біологічних нейронах. Кожне з'єднання має вагу w_{ij} для модуляції відповідного входу xi. Зважена сума вхідних даних, додана з додаванням зсуву b_j (використовується для керування чистим входом до функції активації), відображається за допомогою функції активації ϕ() для генерації вихідних даних y_j нейрона, це математично представляється таким чином:

$$y_j = \varphi(z_j) = \varphi(\sum_{i=1}^{n} x_i w_{ij} + b_j)$$

Візуальне зображення персептрона знаходиться зліва на рисунку 1.1. Багатошаровий перцептрон (MLP), також відомий як багатошарова нейронна мережа (MNN), — це набір нейронів, організованих у шари. Кожен шар містить один або кілька нейронів. Крайній лівий і правий шари мережі називаються вхідним і вихідним шарами відповідно, а проміжні шари називаються прихованими шарами. Останні отримують свою назву, тому що виконані обчислення не бачать користувачі. Нейрони вхідного рівня не виконують жодної обробки отриманих даних, вони просто передають їх на наступний рівень як вхідні дані. Цей новий рівень обробляє вхідні дані та генерує вихідні дані, а потім передає їх наступному шару нейронів для повторення процесу. Архітектура MLP називається мережею прямого зв'язку, оскільки послідовні рівні переходять один в одного в прямому напрямку від входу до виходу. Архітектура прямого зв'язку передбачає, що всі

нейрони одного шару з'єднані з нейронами наступного рівня [1]. MLP з одним вхідним шаром, одним прихованим шаром і одним вихідним шаром показано праворуч на рисунку 1.1.

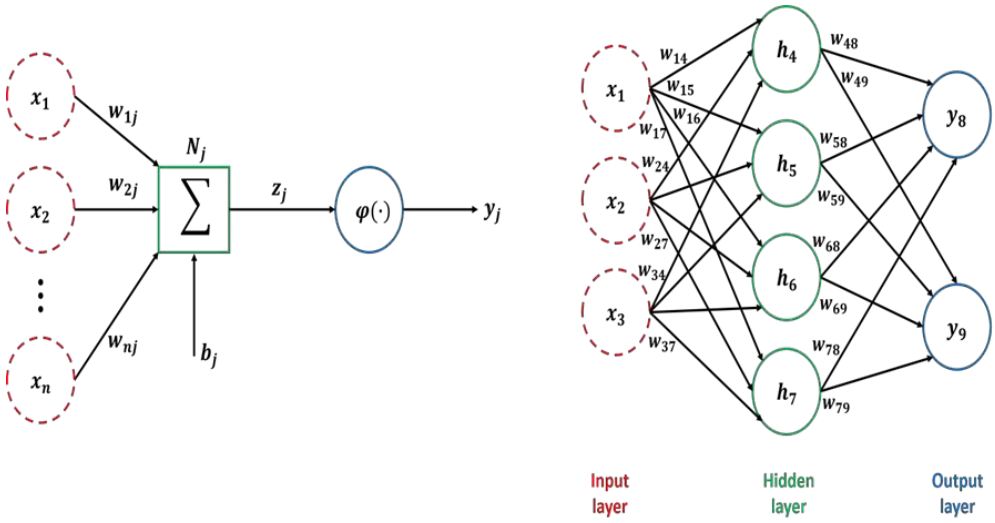


Рисунок 1.1 - Нейронна мережа. Зліва: архітектура Персептрон. Праворуч: архітектура багаторівневих персептронів. [2].

В останнє десятиліття прогрес і використання MNN у різних областях досліджень сформували тенденцію в машинному обігу під назвою «Глибоке навчання» (DL). Центральна ідея глибокого навчання полягає в використанні багатьох нелінійних рівнів обробки інформації для вилучення корисних функцій і перетворення для навчання під контролем або без нього. Глибока нейронна мережа, яка також називається DNN або тільки NN, може використовувати сотні навіть тисячі шарів, які спільно вивчають ієрархію представлень. Немає чіткого поділу між мілководними і глибокими НН за кількістю шарів. Однак будь-яка архітектура з більш ніж двома або трьома рівнями може вважатися глибокою [3].

Функції активації $\phi(\cdot)$ обмежують вихідний діапазон нейрона, і для демонстрації складної поведінки потрібна нелінійна функція. Функції активації є скалярними функціями. Коли нейрон передає ненульовий вихідний сигнал іншому нейрону, він вважається активованим.

Деякі функції активації, які найчастіше використовуються, будуть описані нижче.

Ідентифікаційна функція: Основною функцією активації є ідентифікаційна або лінійна функція, це означає, що функція виводить незмінний сигнал:

$$\varphi(v) = v$$

Ця функція не забезпечує нелінійності та часто використовується на вхідному та вихідному рівнях, коли ціль є реальним значенням [1]. Графічне зображення цієї функції показано на рисунку 1.2.

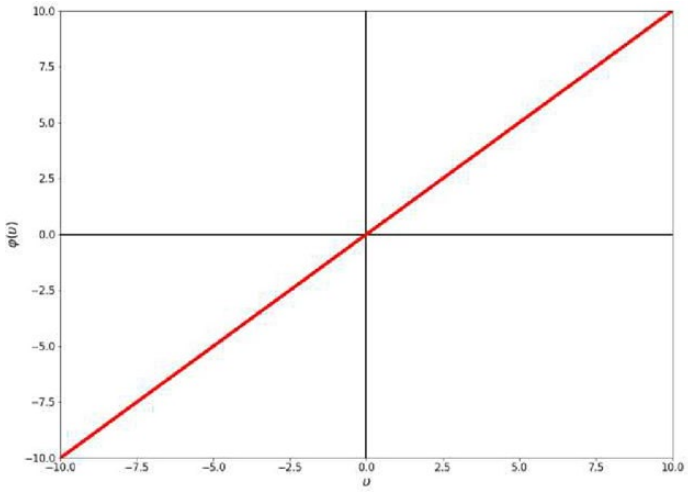


Рисунок 1.2 - Ідентифікаційна функція

Функція Sigmoid: активація Sigmoid, яка також називається логістичною функцією, виводить значення в діапазоні (0,1), що корисно для виконання обчислень, які слід інтерпретувати як ймовірності:

$$\varphi(v) = \frac{1}{1 + e^{-v}}$$

Крім того, це також корисно для створення функцій втрат, отриманих з моделей максимальної правдоподібності [1]. Графічне зображення цієї функції показано на рисунку 1.3.

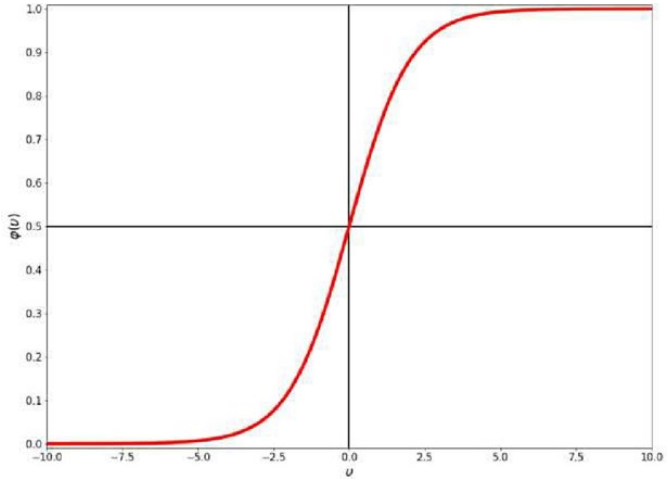


Рисунок 1.3 - Функція Sigmoid

Функція Softmax: Це узагальнення сигмоїдної функції, оскільки її можна застосовувати до безперервних даних (а не до двійкової класифікації) і може містити кілька меж прийняття рішень. Він використовується для представлення розподілу ймовірностей над дискретною змінною з k можливими класами [4]. Він визначається як:

$$\varphi(\overline{v})_i = \frac{e^{v_i}}{\sum_{j=1}^{k} e^{v_j}} \quad \forall i \in \{1, \ldots, k\}$$

Де $\bar{v} = [v_1, \ldots, v_k]$ представляє k виходів нейронів даного шару. Значення k, отримані з softmax, представляють ймовірності кожного k класу для прогнозування. Сума цих значень дорівнює одиниці, оскільки вони представляють багато класовий розподіл ймовірностей.

Функція Tanh: Tanh означає гіперболічний тангенс. Ця функція активації має форму, подібну до сигмоїдної функції, за винятком того, що вона вертикально пере масштабована в діапазоні [-1,1]:

$$\varphi(v) = \frac{e^{2v} - 1}{e^{2v} + 1}$$

Функція tanh є кращою, ніж сигмоїда, якщо бажано, щоб результати обчислень були як позитивними, так і негативними. Крім того, його середнє центрування та більший градієнт по відношенню до сигмоподібної функції призводить до легшого навчання. [1]. Графічне зображення цієї функції показано на рисунку 1.4.

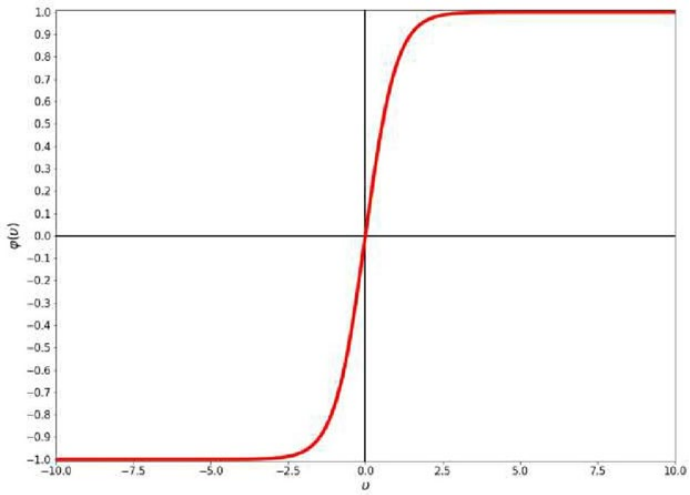


Рисунок 1.4 - Функція Tanh

Rectified Linear Unit: також називається ReLU, ця функція активації дозволяє уникнути проблем із насиченням сигмоїдної та tanh функцій. Це означає, що великі значення прив'язуються до 1,0, а малі – до -1 або 0 для tanh і sigmoid відповідно. Крім того, функції справді чутливі лише до змін навколо їхньої середньої точки введення, наприклад 0,5 для сигмоїди та 0,0 для tanh. Оскільки ReLU є майже лінійним, вони зберігають багато властивостей, завдяки яким лінійні моделі добре узагальнюються та легко оптимізуються за допомогою методів на основі градієнта (Гудфеллоу та ін. 2016). Його формула така:

$$\varphi(v) = \max\{v, 0\}$$

Графічне зображення цієї функції показано на рисунку 1.5.

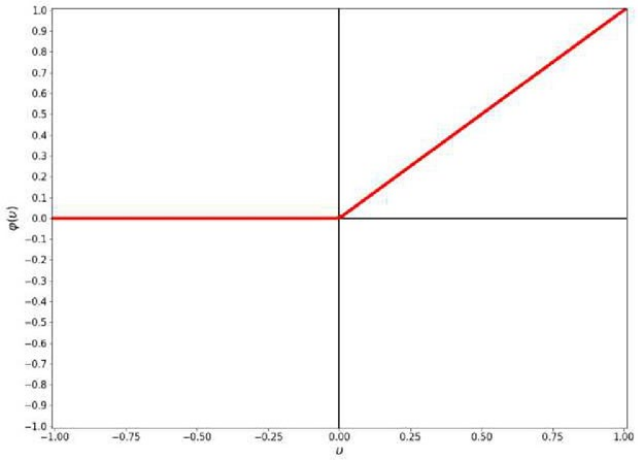


Рисунок 1.5 - Функція ReLU

Leaky ReLU: Ця функція є функцією ReLU з невеликим нахилом від'ємних значень замість повного нуля:

$$\varphi(v) = \left\{ \begin{array}{ll} \alpha \times (v), & \text{if } v < 0 \\ v, & \text{if } v \geq 0 \end{array} \right\}$$

Де α називається нахилом, який спочатку дорівнює 0,01. Ця функція намагається звести до мінімуму чутливість до проблеми ReLU, що вмирає. Ця проблема спричинена тим, що ReLU не можна безперервно диференціювати, коли υ = 0. Крім того, ReLU встановлює всі значення, менші за нуль, до нуля, і, отже, нейрони, які досягають великих від'ємних значень, не можуть відновитися після того, як застрягли на 0. Нейрон фактично гине, і, отже, проблема відома як вмираюча проблема ReLU. З нахилом Leaky ReLU результати трохи спадають. Теза полягає в тому, що ці невеликі кількості зменшують загибель активованих ReLU нейронів [4]. Графічне зображення цієї функції показано на рисунку 1.6.

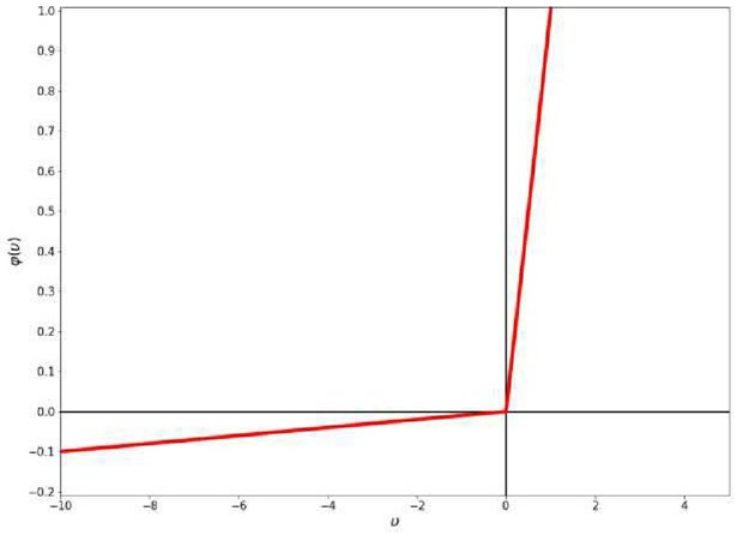


Рисунок 1.6 - Функція Leaky ReLU з нахилом (α) 0,01.

У парсептроні оригінальною функцією активації була сигмоїдна функція, оскільки вона є двійковим класифікатором. Пізніше, з використанням MLP, кожен шар нейронів міг використовувати ту саму або іншу функцію активації відповідно до потреб завдання. Використання нелінійних активацій відіграє фундаментальну роль у збільшенні потужності моделювання мережі. Якби мережа використовувала лише лінійні активації, вона не забезпечила б кращої потужності моделювання, ніж одношарова лінійна мережа [1].

Функції втрат, які також називаються функціями витрат або функціями помилок, кількісно визначають, наскільки близька дана NN до ідеального результату в її навчанні [5]. Це показники, засновані на спостережуваній помилці в прогнозах NN. Вибір функції втрат має вирішальне значення для визначення виходів таким чином, щоб він був чутливим до конкретної програми.

У наборі даних із задачі навчання під керівництвом N отриманих зразків можуть бути представлені як кортеж (X,Y). Де X представляє змінні ознаки, а Y представляє спостережуване значення або клас для прогнозування. Нехай Yˆ буде вихідним значенням мережі, яке також називається прогнозуванням. Позначення $h_{W,b}(\cdot)$ позначає NN як функцію, яка залежить від ваг (W) і зміщення (b) мережі. Таким чином:

$$h_{W,b}(X_i) = \hat{Y}_i$$

являє собою перетворення значень ознак i-ї вибірки в прогнозоване значення через NN.

Функція втрат представлена як L(W, b), підкреслюючи, що вартість або помилка прогнозованих значень мережі залежить виключно від її ваг і зміщень. Одним із основних підходів до функцій втрат є завдання класифікації. Однією з найбільш використовуваних функцій втрат для класифікації є втрата негативної логарифмічної ймовірності.

Оцінка максимальної правдоподібності — це спосіб знайти найкращі можливі параметри, які роблять спостережувані дані найбільш імовірними. У проблемах класифікації він прагне максимізувати ймовірність правильного прогнозування класу. У бінарній класифікації це описується наступним рівнянням:

$$P(Y_i|X_i;W,b) = (h_{W,b}(X_i))^{Y_i} \times (1 - h_{W,b}(X_i))^{1-Y_i}$$

Попереднє рівняння можна переписати так:

$$P(Y_i|X_i;W,b) = \prod_{i=1}^{N} (\hat{Y})^{Y_i} \times (1 - \hat{Y})^{1-Y_i}$$

При роботі з множенням ймовірностей корисно використовувати його логарифм. Це має на меті перетворити множення на суму ймовірностей. Крім того, монотонне зростання логарифма є цікавою властивістю при використанні ймовірностей. Таким чином, мінімізація негативної логарифмічної ймовірності еквівалентна максимізації ймовірності. Ось чому застосування логарифма до рівняння (2.10) отримує наступну функцію втрат, яка називається негативним логарифмом правдоподібних втрат (Гудфеллоу та ін. 2016):

$$L(W,b) = -\sum_{i=1}^{N} Y_i \times log(\hat{Y}_i) + (1 - Y_i) \times log(1 - \hat{Y}_i)$$

Перехресна ентропія є однією з основних функцій вартості, яка використовується для навчання під наглядом [7] Використовується для багато класової класифікації. Це представлено наступним рівнянням:

$$L(W,b) = -\sum_{n=1}^{N}\sum_{c=1}^{C} Y_{nc} \times log(\hat{Y}_{nc})$$

де C представляють класи контрольованої проблеми, Ync представляють основну істину для класу c входу n, а $\hat{Y}nc$ представляють передбачення того самого входу. У двійковій класифікації, де C = 2, крос-ентропія також може бути представлена як у рівнянні 2.11.

Мінімізація цієї функції вартості дозволяє застосувати метод, відомий як Gradient Descent, для навчання NN за допомогою Backpropagation з метою пошуку оптимальних параметрів, які дозволяють максимально знизити рівень помилок мережевих прогнозів (Khan et al. 2018).

1.2 Згорткові нейронні мережі

Згорткові нейронні мережі (CNN/ConvNets) є однією з найпоширеніших архітектур NN із багатьма програмами комп'ютерного зору (CV). Деякі з областей застосування включають класифікацію та сегментацію зображень, виявлення об'єктів, обробку відео, навіть використовуються поза CV, у таких завданнях, як обробка природної мови та розпізнавання мови [8]. Привабливою рисою CNN є його здатність використовувати просторову або часову кореляцію даних. Біологічним джерелом натхнення для CNN є зорова кора у тварин.

CNN — це багаторівнева ієрархічна мережа прямого зв'язку, дуже схожа на класичну NN, головна відмінність якої полягає в тому, що прихований нейрон пов'язаний лише з підмножиною нейронів попереднього рівня. Ця характеристика дозволяє неявно вивчати відповідні функції. Глибока архітектура моделі призводить до ієрархічного виділення ознак [9].

CNN є корисним класом моделей для завдань класифікації зображень. CNN вчиться зіставляти певне зображення з відповідною категорією, виявляючи низку абстрактних представлень ознак, починаючи від простіших і закінчуючи більш складними. Потім ці дискримінаційні особливості використовуються в мережі для правильного прогнозування вхідного зображення [9].

Топологія CNN розділена на кілька етапів навчання, що складаються з комбінації згорткових рівнів, нелінійних блоків обробки та рівнів. На рисунку 1.7 представлена загальна архітектура CNN. Далі ми розглянемо різні типи рівнів, які складають CNN.

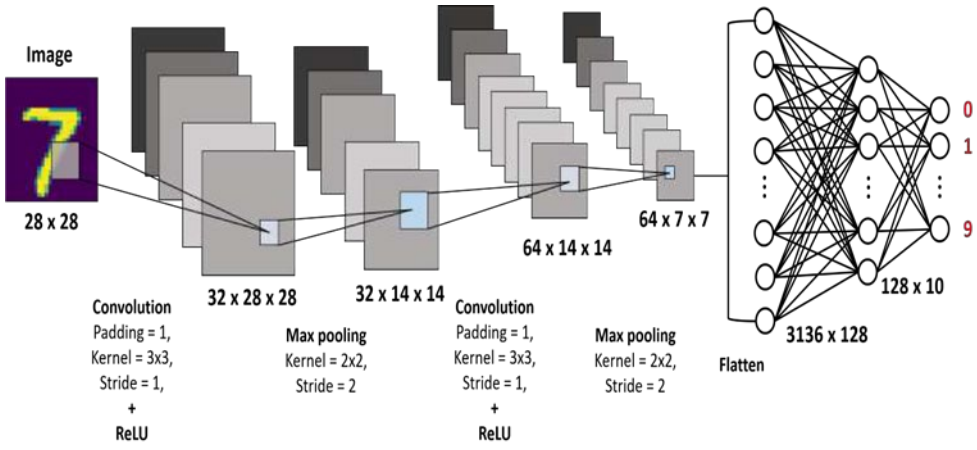


Рисунок 1.7 - Архітектура базового CNN.

CNN — це конкатенація різних типів рівнів, відповідальних за вивчення характеристик високого рівня для виконання таких завдань, як класифікація. Натхненний [10].

Згортковий рівень є основним будівельним блоком згорткової мережі, який виконує більшу частину важкої обчислювальної роботи. Він містить набір фільтрів, які згортаються з даними вхідними даними для створення вихідної карти ознак. Ці шари по суті є екстракторами функцій [11].

Кожен фільтр, який також називають ядром, у згортковому шарі є сіткою дискретних чисел, які називаються вагами. Подібно до звичайних нейронів у NN, ці ваги навчені вивчати функції, які дозволяють виконувати необхідну діяльність. Основна відмінність від NN полягає в тому, що ці нейрони підключені лише до невеликої області нейронів у шарі перед ними. Згорткове ядро працює шляхом

поділу зображення на невеликі частини, які зазвичай називають рецептивними полями. Поділ зображення на невеликі блоки допомагає виділити мотиви ознак, які ієрархічно створюють більш складні об’єкти.

Структура ядра має висоту та ширину і зазвичай має квадратну форму (висота = ширина). Наприклад, на малюнку 1.8 показано ядро висотою два і шириною два.

2	0
-1	3

Рисунок 1.8- Приклад фільтру 2x2.

Крім того, ядра мають третій вимір, який називається глибиною, не плутати з глибиною мережі. Цей розмір стосується кількості каналів у кожному шарі CNN, наприклад, кількості каналів основного кольору (наприклад, червоного, зеленого та синього на зображеннях RGB) у вхідному зображенні або кількості карт функцій у прихованих шарах [13]. Кожен фільтр має власні ваги для тренування, і кожен з них створить окрему карту функцій. Під час застосування фільтрів до зображення або попередніх карт об’єктів нові карти будуть меншими або рівними попереднім щодо висоти та ширини, але кількість нових карт об’єктів, глибина не обмежена. Кожен фільтр застосовує операцію згортки до попередніх карт функцій.

Операція згортки розміщує фільтр у кожній можливій позиції зображення (або прихованого шару), щоб фільтр повністю перекривався зображенням, і виконує скалярний добуток між фільтром і відповідною просторовою областю вхідного об’єму. Операція згортки визначається як:

$$h_{ijp}^{(q+1)} = (\sum_{r=1}^{F_q}\sum_{s=1}^{F_q}\sum_{k=1}^{d_q} w_{rsk}^{(p,q)} h_{i+r-1,j+s-1,k}^{(q)}) + b^{(p,q)}$$

$$\forall i \in \{1, ..., L_q - F_q + 1\} \quad \forall j \in \{1, ..., B_q - F_q + 1\} \quad \forall p \in \{1, ..., d_{q+1}\}$$

Де:

• $h^{(q)}$ представляє карту ознак q-го шару.

• F_q – висота та ширина фільтра. Пам'ятайте, що висота = ширина є звичайним.

• d_q – це кількість каналів фільтра, а також кількість карт функцій вхідного обсягу. Пам'ятайте, що глибина фільтра залежить від кількості карт функцій попереднього шару.

• $w^{p,q}$ представляє значення ваги p-го ядра на q-му рівні.

• L_q і B_q стосуються висоти (або довжини) і ширини (або ширини) вхідного об'єму, відповідно.

• $b^{(p,q)}$ відноситься до зсуву p-го ядра на q-му рівні.

• $L_q - F_q + 1$ і $B_q - F_q + 1$ представляють нові розміри (довжину та ширину) карти ознак шару q+1. Розміри після застосування згортки показано на рисунку 1.9.

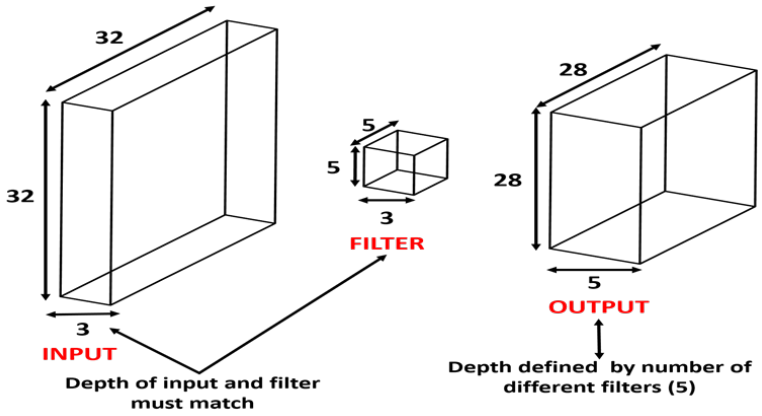


Рисунок 1.9 - Розміри в згортці.

Згортка між вхідним шаром розміром 32×32×3 і фільтром розміром 5×5×3 створює вихідний шар із просторовими розмірами 28×28. Глибина кінцевого результату залежить від кількості окремих фільтрів, а не від розмірів вхідного шару чи фільтра. Натхненний [1].

Як приклад операції згортки, на рисунку 1.10 показано згортання фільтра 3x3 над двовимірною сіткою.

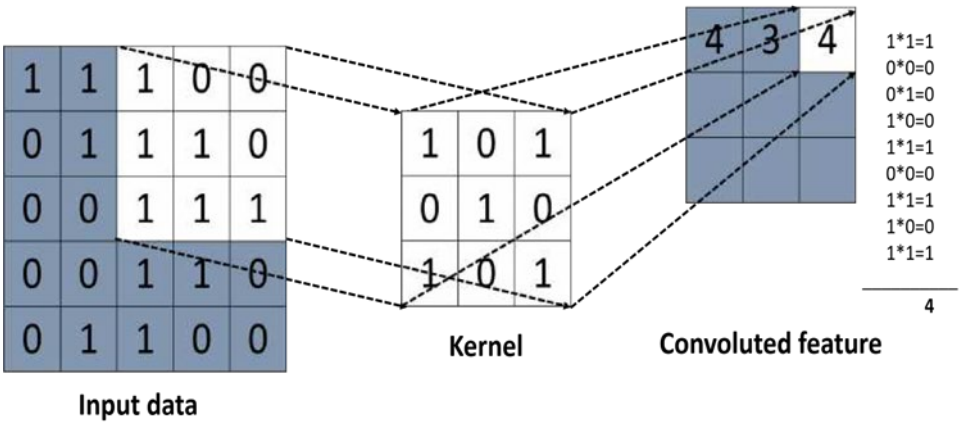


Рисунок 1.10 - Приклад операції згортки.

Окрім висоти, ширини та глибини фільтрів у згортковому шарі, для визначення згорткового шару також потрібні параметри кроку та відступу.

У наведеному вище описі згортки, щоб обчислити кожне значення вихідної карти ознак, фільтр робить крок 1 уздовж горизонтальної або вертикальної позиції, тобто вздовж стовпця або рядка вхідних даних відповідно. Цей крок називається кроком фільтра згортки, який може бути встановлений на інше (окрім 1) значення, якщо потрібно [9].

Крок налаштовує, як далеко пересунути фільтр, щоб створити карту функцій. Цей параметр може мати значення ковзання за довжиною та інше за шириною вхідного об’єму, але зазвичай однакове значення використовується для обох вимірів. Без використання заповнення згортка призводить до зменшення попередніх розмірів, що називається операцією підвибірки. рисунок 1.11 показано приклад згортки зі значеннями кроку один і два.

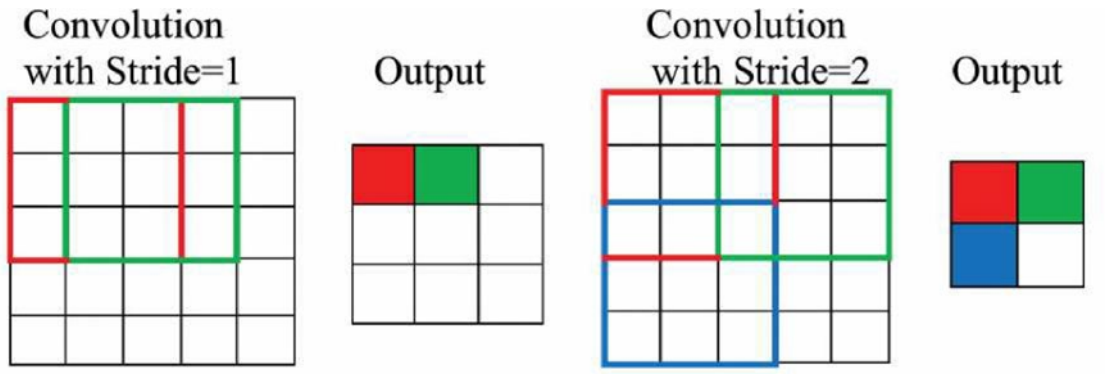


Рисунок 1.11 - Згортка зі значеннями кроку 1 і 2.

Одне спостереження полягає в тому, що операція згортки зменшує розмір q+1-го шару порівняно з розміром q-го шару. Цей тип зменшення розміру загалом небажаний, оскільки він може втрачати деяку інформацію вздовж кордонів зображення (або карти функцій, у випадку прихованих шарів). Цю проблему можна вирішити за допомогою доповнення. Заповнення контролює просторовий розмір вихідного об’єму, додаючи рядки та стовпці «пікселів» до карти вхідних функцій. Зазвичай ці нові значення є нулями, і в цьому випадку це називається доповненням нуля. Нульове заповнення представлено на малюнку 1.12.

Розміри вихідного об’єму після застосування згортки можна розрахувати за такими формулами:

$$L_{(q+1)} = \left\lfloor \frac{L_q - F_q + stride + 2 \times padding}{stride} \right\rfloor$$

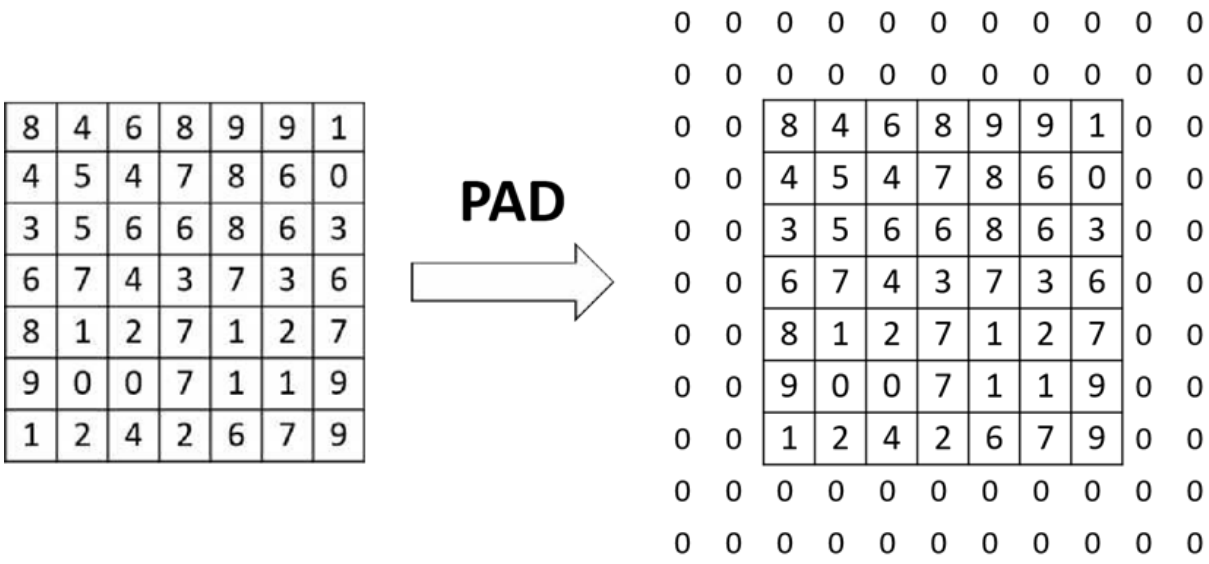


Рисунок 1.12 - Приклад заповнення нуля.

Значення заповнення дорівнює 2, тобто два рядки або стовпці додаються до кожної сторони карти функцій.

$$B_{(q+1)} = \left\lfloor \frac{B_q - F_q + stride + 2 \times padding}{stride} \right\rfloor$$

Пам’ятайте, що L_q і B_q — це довжина і ширина вхідного об’єму відповідно. Символ [.] представляє функцію підлоги. Як приклад, зображення у градаціях сірого (1 канал введення) розміром 64x64 пікселів, коли застосовано фільтр 3x3 із кроком 2 і відступом 1 (представляє один стовпець або рядок нулів, доданих кожним краєм), матиме вихідні розміри 32x32 на кожній карті об’єктів.

Шари об’єднання зазвичай вставляють між послідовними згортковими шарами. Використання згорткових шарів із шарами об’єднання робиться для поступового зменшення просторового розміру представлення даних і для допомоги в контролі надміру. Рівень об’єднання працює незалежно на кожному зрізі глибини вхідних даних [18].

Об’єднання або зменшення вибірки є цікавою локальною операцією. Він підсумовує подібну інформацію в околицях рецептивного поля та виводить домінуючу реакцію в цьому локальному регіоні [21]. Найпоширенішим налаштуванням шару об’єднання є застосування фільтрів 2×2 із кроком 2. Це призведе до зменшення частоти дискретизації кожного фрагмента глибини у вхідному об’ємі в два рази за довжиною та шириною, але вони зберігають кількість каналів, на відміну від згорткові фільтри.

Два основні рівні об’єднання, які використовуються в CNN, — це максимальне об’єднання та об’єднання середніх. Max-pool приймає максимальне значення рецептивного поля фільтра. У той же час Average-pooling бере середнє значення [17]. Продуктивність цих шарів показано на рисунку 1.13.

Рисунок 1.13 - Максимальне об'єднання та середнє об'єднання.

Шари активації додають нелінійність до карт функцій, отриманих за допомогою згорткових шарів. Вони застосовують величезну колекцію функцій активації.

Повністю зв’язані шари по суті відповідають багаторівневим персептронам. Кожен нейрон у повністю зв’язаному шарі щільно пов’язаний з усіма одиницями попереднього шару. Ці рівні беруть характеристики високого рівня, витягнуті набором згорткових, активаційних і об’єднаних рівнів CNN, і генерують прогнозоване значення Yˆ. Ці шари.використовуються лише в кінці CNN для виконання таких завдань, як класифікація .

Пакетна нормалізація (BN) використовується для вирішення проблем, пов’язаних із внутрішнім коваріаційним зсувом у картах функцій. Внутрішній коваріаційний зсув - це зміна розподілу значень прихованих шарів, яка уповільнює збіжність і вимагає ретельної ініціалізації параметрів. Крім того, він згладжує потік градієнта та діє як регулюючий фактор, що таким чином допомагає покращити узагальнення мережі [23]. Пакетна нормалізація представлена у вигляді:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

Де,

• x_i представляє карту характеристик і-го шару.

• $\widehat{x_i}$ — нормалізована карта ознак.

• μ_B і σ^2_B відображають середнє значення та дисперсію карти ознак для міні-серії зразків,

відповідно.

• ϵ є малою константою, щоб уникнути ділення на нуль для чисельної стабільності.

Пакетна нормалізація уніфікує розподіл значень карти ознак, встановлюючи для них нульове середнє значення та одиничну дисперсію. Пізніше нормалізована карта ознак масштабується за допомогою параметрів β і γ, щоб уникнути обмежень у потужності представлення згорткових шарів. Ця операція представлена у вигляді:

$$y_i = \gamma \times \hat{x}_i + \beta$$

де y_i представляє карту функцій перед активацією. На малюнку 1.14 показано представлення BN.

Пакетна нормалізація реалізована як шар і зазвичай використовується після згорткового шару.

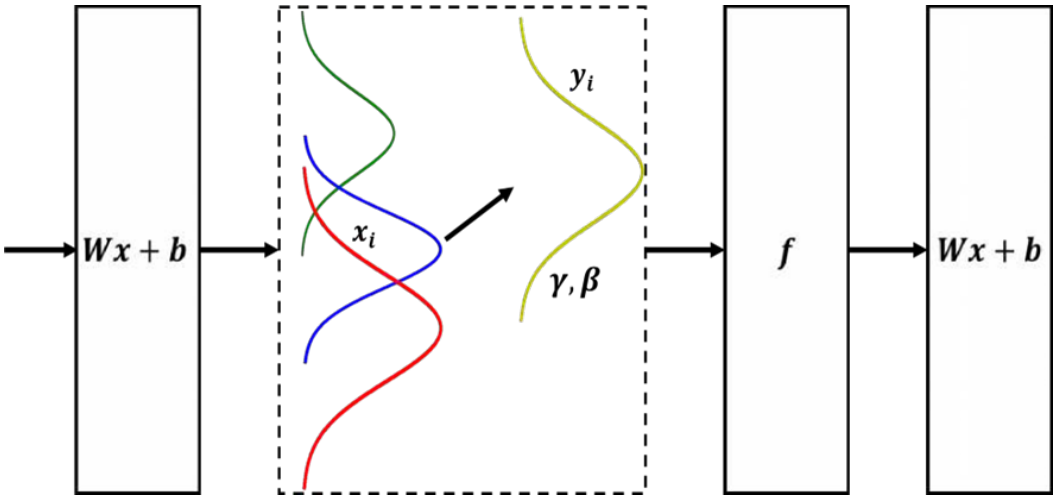


Рисунок 1.14 - Представлення пакетної нормалізації.

Різні розподіли значень (xi) для кожного зразка в шарі (Wx + b) перетворюються в нормалізовані та масштабовані сигнали (yi) з BN, які вводяться на шар активації (f). Натхненний.

Також називаються дробовими згортками. Цей тип шарів дуже подібний до згорткового шару в таких аспектах, як використання фільтрів, крок і відступ. Основна відмінність полягає в тому, що в той час як згорточні шари створюють карти функцій зі значень пікселів зображення, транспоновані згорточні шари відображають об'єкти в пікселі під час моделювання зображень, що є протилежністю того, що робить звичайний згортковий шар . Цей шар зазвичай виконується для підвищення дискретизації, тобто для створення вихідної карти ознак, яка має просторовий вимір, більший, ніж у вхідної карти ознак. Цей аспект транспонованих згорткових шарів дає змогу генерувати зображення на виході з нейронних мереж. Цей шар також відомий як рівень деконволюції, хоча він виконує інший процес, ніж деконволюція.

Процес, який цей шар застосовує для підвищення дискретизації карти функцій, — це операція згортки з фільтром над вхідним розміром, який має межі з доданими нульовими значеннями. Ці нульові межі мають функцію збільшення сприйнятливого поля фільтра і, отже, отримання карти ознак з більшою довжиною та шириною, ніж вхідний обсяг. Вихідний розмір ($O_{(q+1)}$) q+1-го шару, де довжина = ширина (O(q)) з фільтром розміру F_q, обчислюється за такою формулою:

$$O_{(q+1)} = (O_{(q)} - 1) \times stride + F_q - 2 \times padding$$

Наприклад, вхідний обсяг із $O_{(q)}$ 2, розміром ядра 3, кроком 1 і заповненням 0 матиме вихідний розмір ($O_{(q+1)}$) 4. На малюнку 1.15 показано цей приклад.

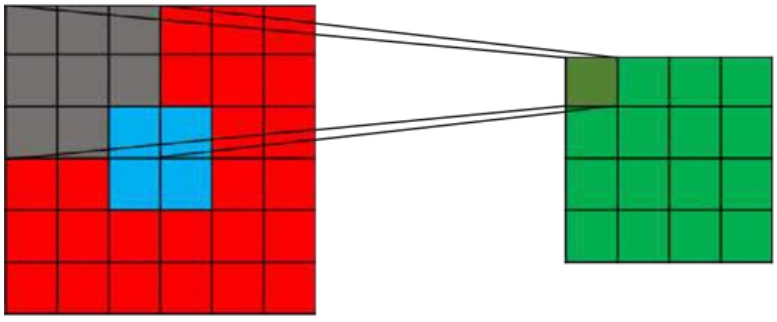


Рисунок 1.15- Приклад операції транспонованої згортки.

Операція згортки застосовується між фільтром 3x3 (заштрихована сітка) і картою функцій (синя сітка) з 2 нульовими межами (червона сітка) на кожній стороні для підвищення дискретизації та отримання нової карти функцій більшого розміру (зелена сітка). Натхненний.

На рисинку 1.16 показано два способи розуміння обчислення транспонованої згортки за допомогою попереднього прикладу.

Транспонований згортковий рівень не використовується в звичайній CNN, але є фундаментальним компонентом деконволюційних мереж, які використовуються в Deep Convolutional Generative Adversarial Networks, які будуть розглянуті пізніше.

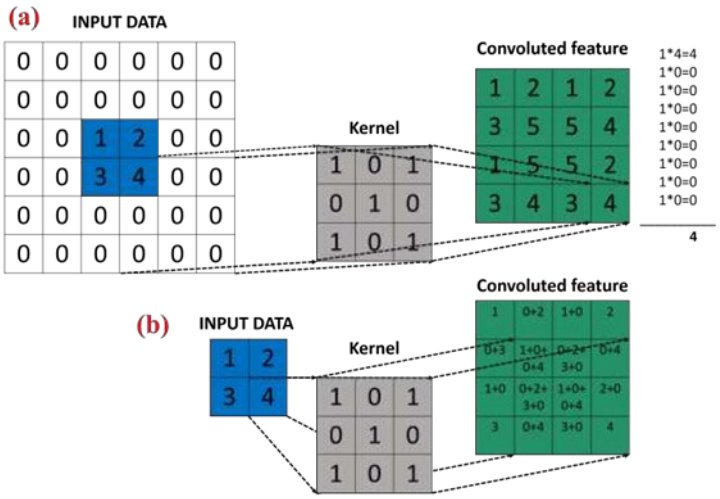


Рисунок 1.16- Два способи обчислення транспонованої згортки.

1.3 Історія розвитку нейронних мереж

Вперше така модель, як штучний нейрон, була запропонована в науковій роботі Уоррена Маккаллоха і Уолтера Пітца в 1943 році. У цій класичній ро-боті була описана логіка розрахунків в нейронних мережах на основі нейрофі-зіології і математичної логіки. Вчені довели, що мережа, що складається з безлічі одиничних нейронів і синаптичних зв'язків, теоретично здатна виконувати будь-які обчислення. Їх робота пробудила великий інтерес дослідницького співтовариства до цієї сфери.

Наступною важливою віхою став 1949 рік, який ознаменувався публікацією книги Д. Хебба «Організація поведінки», яка дає інтерпретацію процесу на-вчання з точки зору фізіології. Він висунув гіпотезу про те, що в міру вивчення різних завдань зв'язки в мозку постійно змінюються. Відомий навчальний постулат Хебба стверджує, що ефективність змінного синапсу між двома нейрона-ми зростає при повторній активації цих нейронів через даний синапс. Тобто він першим припустив, що навчання зводиться до зміни сили синаптичних зв'язків, що рівнозначно збільшенню ваги з'єднання штучного нейрона.

Середина 50-х років відома появою персептрона Розенблата , перша мо-дель одношарової нейронної мережі, що складається з датчиків, сигнали яких надходять в асоціативні елементи, які перетворюють цю інформацію і передають її реагуючим елементам.

За цим прослідував період спаду досліджень нейронних мереж. Дослідження нейронних мереж майже не розвивалися до тих пір, поки комп'ютери не досягли великих обчислювальних потужностей. Одним з важливих кроків, які стимулювали подальші дослідження, стала розробка в 1975 році Вербосом методу зворотного поширення помилки, що дозволило ефективно вирішити проблему навчання багатошарових мереж.

21 століття - це час постійних відкриттів і змін в цій області. Поява доста-тніх потужностей і вміння розпаралелювати розрахунки на відеокартах дозволило вдихнути життя в деякі забуті парадигми і підходи. Наприклад, в таких, як згорткові мережі. Зараз область нейромережевих технологій є одна з най-більш динамічно розвинутих напрямків в машинному навчанні.

У 2014 році Ян Гудфеллоу представив роботу під назвою Generative Adversarial Networks (GAN), в якій він описує абсолютно нову мережеву архітектуру. Ключова ідея, закладена в генеративні мережі, описана Гудфеллоу. Генеративні мережі є основним інструментом вирішення завдання, поставлено-го в даній магістерській дисертації. З 2014 року цей напрямок є одним з най-більш актуальних і динамічно розвиваються напрямків машинного навчання.

1.4 Генеративно змагальні мережі

Генеративно змагальні мережі (GANs) — це моделі глибокого навчання, які датуються 2014 роком (Гудфеллоу та ін. 2014). Ці моделі належать до набору генеративних моделей, гілки алгоритмів неконтрольованого навчання, відповідальних за відображення того, як були згенеровані дані. Моделі GAN навчаються з метою створення синтетичних даних, подібних до деякого набору реальних даних. Його функціонування складається з двох типів нейронних мереж, які стикаються через протилежні цілі (звідси термін «конкурентні»). Це конфронтаційне навчання, подібне до навчання з підкріпленням, дозволяє власне навчання мережі генерувати дані, подібні до тих, що належать до реального розподілу.

Є дві мережі, які складають GAN, а саме:

Генератор (G): Ця мережа відповідає за генерування даних, які дуже схожі на реальний набір, з яким її було навчено. Він приймає вектор випадкових значень шуму (z), взятий із простого попереднього розподілу (p_z), як вхідний і обробляє його через їхні внутрішні значення в мережі, доки не буде отримано вихідні дані ($G_{(z)}$), щоб ці нові дані були дуже подібний до такого ж складного розподілу вихідного набору.

Дискримінатор (D): ця модель має на меті класифікацію між реальними вибірками ($x \sim p_{data}(x)$), тобто даними, які належать до вихідного набору, та згенерованими вибірками ($G(z) \sim p_g(G(z))$), тобто створені генератором штучно. Їх вхідними даними є дані, справжні або підроблені, а їхнім виходом є ймовірності класифікації для обох класів через логістичну функцію.

Цю загальну структуру називають Vanilla GAN, оскільки вона представляє базову версію, з якої почалися більш складні архітектури, але яка продовжує поважати два фундаментальні блоки GAN. Візуальне представлення цієї базової структури показано на рисунку 1.17.

Навчання GAN описується як гра з нульовою сумою (також називається мінімакс) між двома гравцями з протилежними цілями; це Генератор і Дискримінатор. Взяття вектора випадкового шуму, взятого з попередніх

розподілів, напр. нормальний або рівномірний, ($z \sim p_z(z)$) як вхідні дані, які називають латентним вектором, генератор видає вибірки з більш складного розподілу ($G(z)$), метою якого є дорівнювати розподілу реального набору даних. Тим часом дискримінатор має завдання розрізнити реальні вибірки ($x \sim pdata(x)$) і згенеровані вибірки ($G(z) \sim p_g(G(z))$). У випадку реальних даних мета виходу дискримінатора ($D(\cdot)$) – бути близькою до 1. У сценарії підроблених даних ціль виходу дискримінатора – бути близькою до 0, тоді як генератор намагатиметься наблизитися до 1, тобто змусити дискримінатора класифікувати його творіння як справжні.

Функція вартості навчання GAN, також звана мінімаксними втратами, відображається наступним рівнянням:

$$\min_G \max_D \mathop{\mathbb{E}}_{x \sim p_{data}} [log(D(x))] + \mathop{\mathbb{E}}_{z \sim p_z} [log(1—D(G(z)))]$$

Попередня формула походить від перехресної ентропії між реальним і згенерованим розподілами.

Мережі навчаються одночасно, заохочуючи обидві моделі до постійного вдосконалення та адаптації. Для кожної ітерації виконується градієнтний крок із зворотним поширенням, щоб зменшити функцію вартості кожної мережі, оптимізуючи їхні внутрішні ваги. Оптимізація генератора розглядається як оптимізація розбіжності Дженсена–Шеннона, яка вимірює, як розподіл ймовірностей $G(z)$ (оцінений розподіл) розходиться з очікуваним розподілом ймовірностей pdata (реальний розподіл). Процес навчання повторюється протягом ряду ітерацій навчання, доки генератор не зійдеться, щоб зробити синтетичні дані невідрізними від реального набору даних, тобто $D(\cdot)$ дорівнює 0,5 для будь-якої вибірки, реальної чи ні, хоча цей теоретичний ідеал майже є ніколи не виконуються, і ітерації припиняються, коли якість творів є необхідною, визначеною за допомогою вимірювання або критеріїв практикуючого спеціаліста, який їх реалізує. Представлення цього процесу навчання наведено на малюнку 1.17.

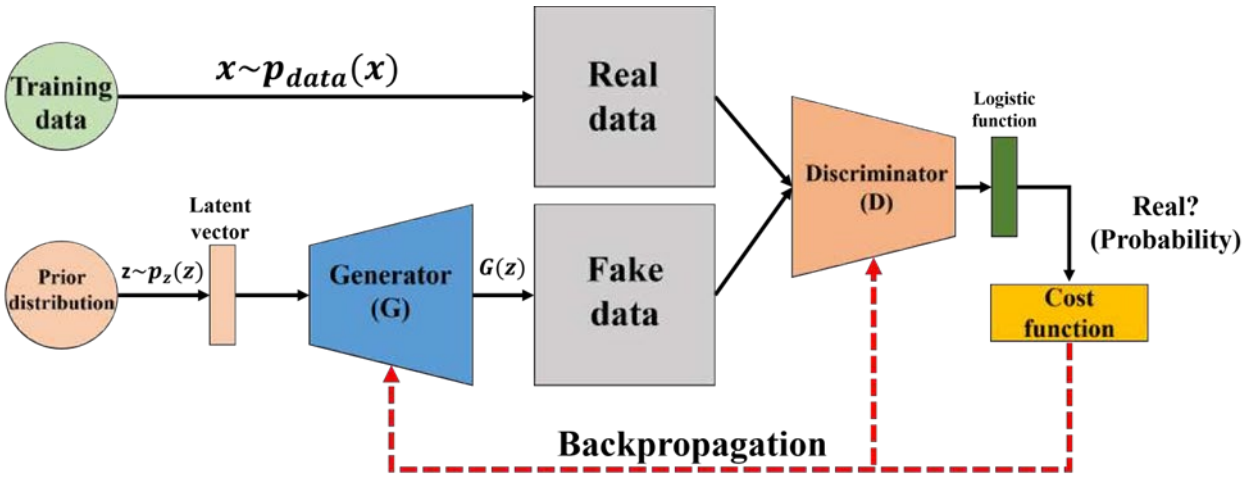


Рисунок 1.17- Навчання Vanilla GAN.

GAN використовувалися в багатьох різних завданнях і сферах, найбільш плідним був синтез зображень, їхньої області народження [27]. Однак великий вплив GAN як генеративної моделі найсучаснішого дозволив їм перейти до інших завдань, таких як передача нейронних стилів, синтез музики та відкриття ліків, щоб назвати кілька прикладів; наступні посилання пропонують детальний огляд сфер застосування GAN (включаючи вищезазначене), а також їх розвиток протягом багатьох років.

Навчання GAN є складним, оскільки має бути баланс між навичками генератора та дискримінатора. Якщо одна з мереж переважає, можуть виникнути проблеми з нестабільністю навчання.

Згортання режиму: ситуація, коли генератор може синтезувати лише невелику підмножину даних повного розподілу, оскільки навчання не дозволило узагальнити багатство варіантів вихідного розподілу. Як показано на малюнку 1.18 ліворуч, навчання з набором даних MNIST, який складається із зображень рукописних чисел від 0 до 9, не змогло охопити різноманіття набору, згорнувшись у генерації лише двох типів цифр. Праворуч від тієї ж фігури генератор створив комбінацію цифр, що перекривається.

Рисунок 1.18-. Приклад згортання режиму

Зникаючий градієнт: виникає, коли дискримінатор або генератор стає достатньо потужним, щоб спричинити незворотний дисбаланс у навчанні, і через невикористання адекватних функцій вартості, які дозволяють отримати адекватні градієнти навчання, які дозволяють протилежній мережі покращити свою продуктивність, таким чином викликаючи застарілий товариш. Як показано на рисунку 1.19, коли дискримінатор здатний легко виявляти між реальними та синтетичними зразками, тобто його втрати наближаються до нуля, виникає застій у градієнтах зворотного зв’язку з генератором, таким чином уникаючи покращення в навчанні генератора.

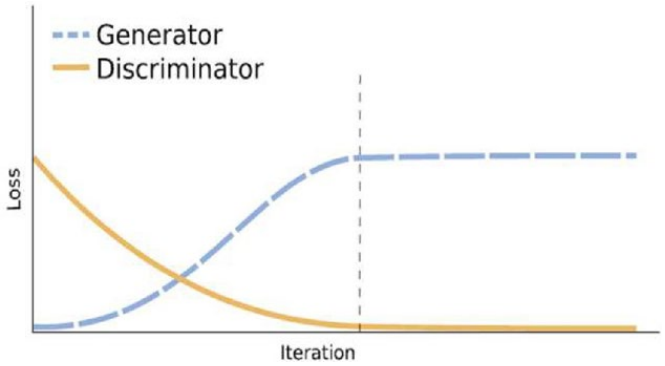


Рисунок 1.19- Приклад зникаючого градієнта.

1.5 Постановка задачі

З огляду на все вищесказане, зрозуміло, що успішний синтез зображень з високою роздільною здатністю все ж є невловимою метою. Існує ряд алгоритмів, які досить успішно показали себе в подібного роду завданнях. Однак при цьому

навчання нейронних мереж за цими алгоритмами навіть на самому продуктивному обладнанні є вкрай трудомістким завданням. З огляду на дану магістерську роботу, на основі представлених алгоритмів була поставлена задача реалізувати її сукупну генеративно-змагальну мережу для синтезу достовірних зображень в досить високих дозволах. Через високу складність завдання генерація зображень буде обмежена роздільною здатністю в 64 пікселя. Рішення повинно являти собою розумний компроміс між якістю отриманих зображень і швидкістю навчання, але першочерговою метою оптимізації буде якість зображень, отриманих від виходу мережі.

Висновоки до розділу 1

1. У цьому розділі розглядалися базові основи штучних нейронних мереж, згорткових нейронних мереж, а також генеративних змагальних мереж, їх варіантів і доповнень для підвищення їх стабільності.
2. Також було представлено внесок GAN у збільшення даних. Показано тематичні дослідження, розглянуте в цій дослідницькій роботі.
3. Було проведено постановку задачі для цієї роботи. Рішення повинно являти собою розумний компроміс між якістю отриманих зображень і швидкістю навчання, але першочерговою метою оптимізації буде якість зображень, отриманих від виходу мережі.

2. РОЗРОБКА АРХІТЕКТУРИ МЕРЕЖІ ДЛЯ ГЕНЕРАЦІЇ ЗОБРАЖЕНЬ

2.1 Архітектури та реалізації для покращення GAN

Деякі з найвидатніших досягнень на користь покращення стабільності мереж GAN, які мають відношення до розробки цього проекту, будуть пояснені нижче.

Глибокі згорткові генеративні змагальні мережі (DCGAN) були розроблені в (Radford et al. 2015) і орієнтовані на використання в зображеннях. Цей клас GAN було розроблено зосереджено на покращенні стабільності навчання порівняно з оригінальною моделлю GAN (Vanilla GAN), яка використовувала багаторівневі персептрони як для генератора, так і для дискримінатора. Процес навчання такий самий, як і в оригінальній моделі GAN, але в DCGAN і генератор, і дискримінатор натхненні архітектурою CNN.

Для дискримінатора використовується звичайний CNN (див. рисунок 1.7). Це пояснюється тим, що завдання класифікації дискримінатора не відрізняється від класичних завдань класифікації, для яких зазвичай використовуються CNN. Тим часом для генератора необхідна деконволюційна мережа. Цей тип мережі схожий на стандартну CNN з тією основною різницею, що будучи генеративною моделлю, вона відповідає за отримання зображень із високорівневих функцій, створених випадковим шумом. Деконволюційна мережа використовувала деконволюційні шари замість згорткових шарів, як CNN. Крім того, оскільки його метою є розширення карт функцій (підвищення дискретизації), а не їх зменшення (зменшення дискретизації), як у CNN, шари об’єднання не використовуються в його архітектурі. На рисунку 2.1 показана загальна архітектура генератора DCGAN.

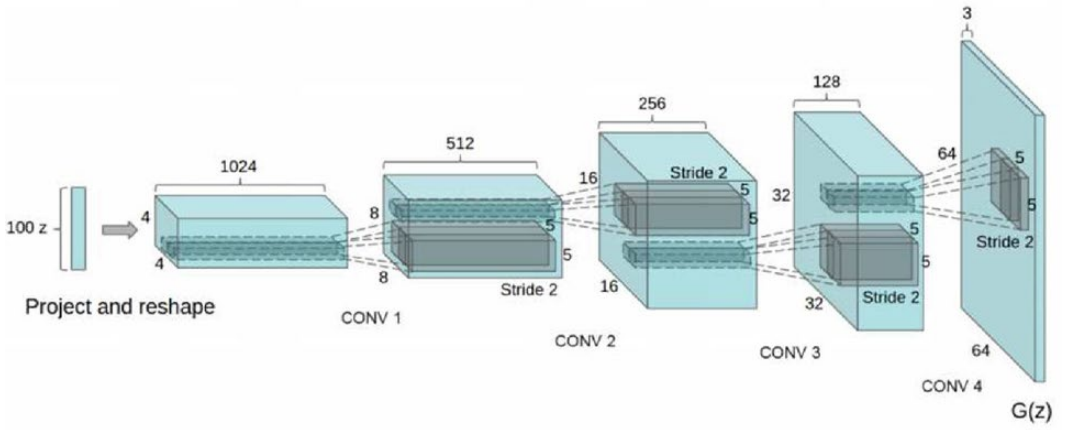


Рисунок 2.1 - Загальна архітектура генератора DCGAN

Як можна побачити, під час обробки випадкового введення шуму з серією деконволюційних шарів генеруються нові та більші карти функцій, що призводить до синтезу зображення RGB (3 канали) з роздільною здатністю 64x64 пікселів. Отримано з оригінальної статті DCGANs (Radford et al. 2015).

Ще одна відмінність від CNN полягає в рівнях активації. Зазвичай у CNN ReLU використовується як функція активації на кожному згортковому рівні, а функції сигмовидної або м'якої максимізації використовуються після повного зв'язаного рівня, який виконує класифікацію. У DCGAN рівні генератора також використовують ReLU, але на кінцевому рівні використовується функція Tanh, оскільки зображення, які використовуються для навчання, нормалізуються в діапазоні [-1,1], тому ж, що надає ця функція. Тоді як для дискримінатора активація LeakyReLU використовується на всіх рівнях, за винятком останнього, який використовує сигмоїдну функцію для отримання біноміальної ймовірності для виконання класифікації між підробленими та справжніми зображеннями.

З моменту свого створення DCGAN став базовою моделлю для синтезу зображень у багатьох творах найсучаснішого рівня. Завдяки чудовим спеціалізованим можливостям візуалізації та кращій стабільності навчання він допомагає зменшити такі проблеми, як збій режиму, хоча він не повністю вирішує типові проблеми GAN (Khan et al. 2018).

Conditional Generative Adversarial Networks (cGAN), створена в (Mirza and Osindero 2014), є розширенням оригінальної Vanilla GAN.

У Vanilla GAN, якщо навчальний набір містив кілька класів даних, його генерація залежно від латентного вектора, який надходить із випадкових значень, генерувала нові дані випадкового класу з тих, що існують у реальному наборі, тому його було виключено з користувач керує створеним класом даних.

У cGAN цей клас GAN обумовлюється використанням попередньої інформації для створення окремого класу даних. Це досягається шляхом додавання мітки y, яка представляє клас даних, до входу як генератора, так і дискримінатора. Завдяки цьому доповненню до оригінальної моделі можна контролювати клас згенерованих даних, а також робити більш точне розрізнення певного класу. Функція вартості цієї моделі є такою ж, як і для Vanilla GAN (див. рівняння 1.19) з додаванням міток класу (y), як показано в наступному рівнянні.

$$\min_G \max_D \mathop{\mathbb{E}}_{x\sim p_{data}} [log(D(\mathbf{x}|\mathbf{y}))] + \mathop{\mathbb{E}}_{z\sim p_z} [log(1—D(G(\mathbf{z}|\mathbf{y})))]$$

У генераторі попередній вхідний шум $p_z(z)$ і y об'єднується в спільне приховане представлення. Хоча в дискримінаторі обидва типи вхідних даних, x і G(z), також поєднуються з відповідними мітками y. На рисунку 2.2 представлена структура простої cGAN.

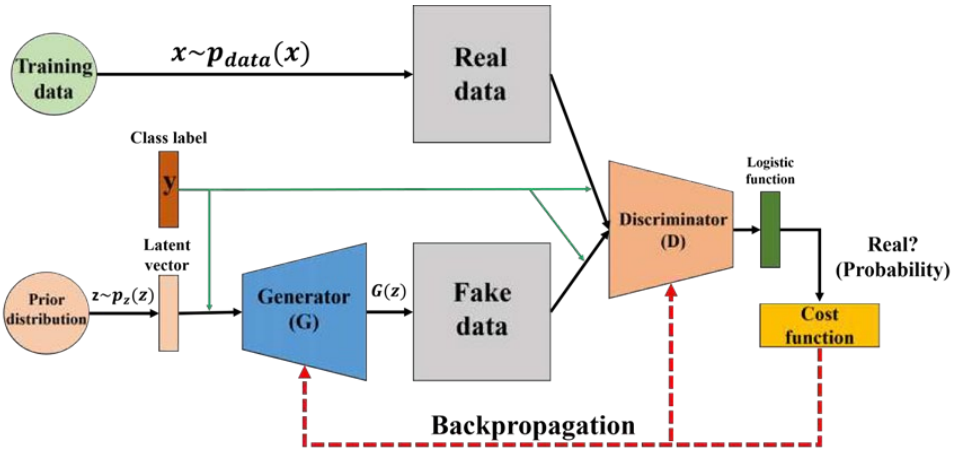


Рисунок 2.2 - Загальна структура cGAN.

Зелена стрілка позначає конкатенацію мітки y до входів GAN.

Об’єднавши cGAN і DCGAN, виходить модель, здатна контролювати клас створюваних зображень.

Структура cDCGAN відповідає оригіналу DCGAN (див. рис. 2.1). Механізми додавання мітки класу до різних входів мереж є такими:

Генератор: одноразовий вектор з’єднується з латентним вектором. Один гарячий вектор — це двійковий вектор, який має стільки позицій, скільки класів. Усі позиції містять нулі, за винятком позиції числа, яке вказує на клас, що кодується, яке містить одиницю. Цей вхід представлено на малюнку 2.3.

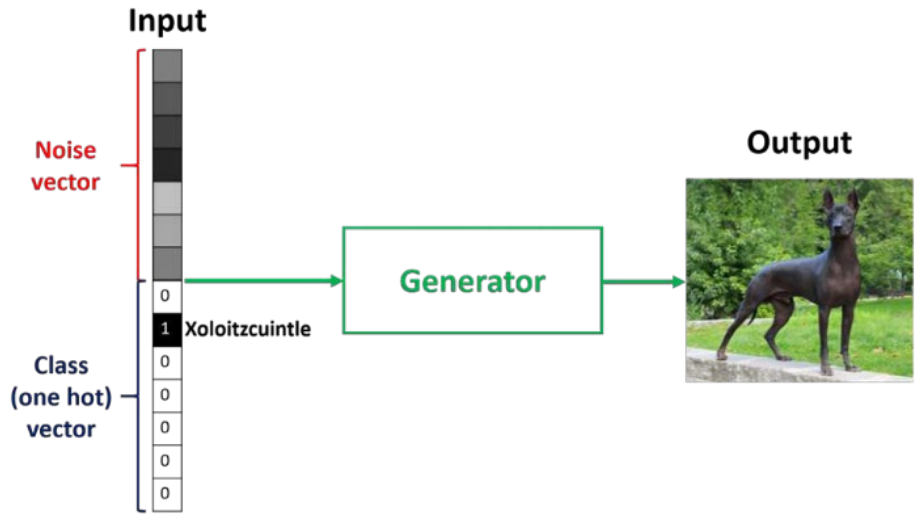


Рисунок 2.3 - Вхід генератора в cDCGAN.

Клас зображення (порода собаки), який потрібно згенерувати, представлений одним гарячим вектором, який об’єднаний із вектором шуму (латентний вектор).(Чжоу 2020).

Дискримінатор: стільки ж нових каналів однакової довжини й ширини зображення, скільки наявних класів, об’єднується із вхідними зображеннями. Ці нові канали (матриці) заповнюються нулями, за винятком каналу, який представляє номер класу, що кодується, який заповнюється одиницями. Цей вхід показаний на малюнку 2.4.

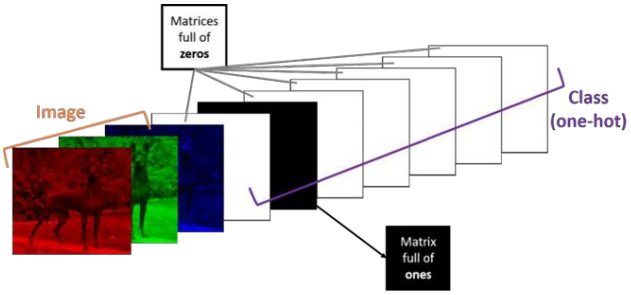


Рисунок 2.4 - Вхід дискримінатора в cDCGAN.

Клас зображення (порода собаки) представлено серією одноразових матриць, які з’єднані з вхідним зображенням. (Чжоу 2020).

Wasserstein-GAN (або також званий WGAN) є розширенням Vanilla GAN, розробленого в (Arjovsky та ін. 2017), який шукає альтернативний спосіб навчання моделі генератора для кращого наближення розподілу даних, що спостерігаються в заданому навчальному наборі даних. .

Замість того, щоб використовувати дискримінатор для класифікації або прогнозування ймовірності згенерованих зображень як справжніх чи підроблених, WGAN замінює модель дискримінатора на критику, яка оцінює реальність чи підробку даного зображення.

Ця зміна пов’язана з тим фактом, що функція вартості, яка використовується у Vanilla GAN, може мати градієнт, що зникає, коли насичена на екстремумах класифікації дискримінатора, тобто коли дискримінатор настільки хороший, що майже ідеально виявляє обидва випадки вибірки. Тому використовується нова функція вартості, яка перетворює дискримінатор на критика (як це називається в цій моделі), що дає бали реалістичності зображенням, отриманим генератором. Це дозволяє, незалежно від якості творінь генератора, він завжди отримує адекватний зворотний зв'язок.

На малюнку 2.5, взятому з оригінальної статті, показано відмінності між градієнтами, отриманими дискримінатором оригінального GAN і критиком WGAN під час навчання диференціації двох гауссів. Як видно, WGAN не має зникаючих градієнтів, як це має Vanilla GAN.

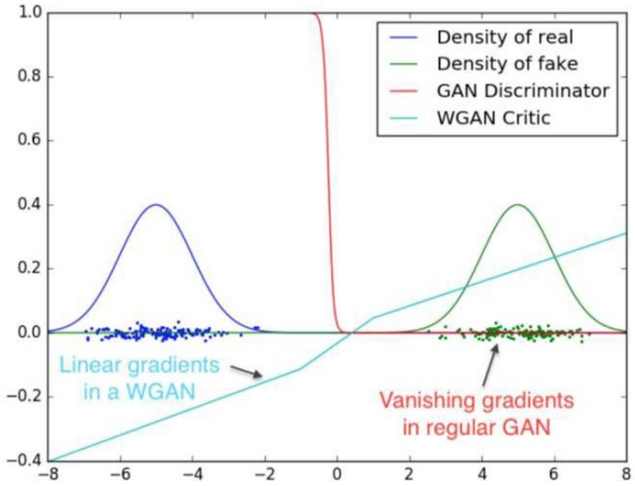


Рисунок 2.5 Градієнти WGAN і Vanilla GAN.(Arjovsky et al. 2017).

У WGAN відображено покращення (Gulrajani et al. 2017). Ця нова версія називається WGAN-GP. GP означає Gradient-Penalty, механізм для запобігання вибуху градієнта на критика.

WGAN-GP - це найсучасніша модель GAN, оскільки вона забезпечує кращу стабільність тренувань. Однак ці проблеми продовжують виникати. Крім того, він має дещо високі обчислювальні витрати, що подовжує час навчання.

Нормалізація ваги (Salimans and Kingma 2016) — це перепараметризація векторів ваги в нейронній мережі, яка дозволяє стабілізувати градієнти оновлення, таким чином уникаючи вибуху градієнта та прискорюючи їх навчання. Цей метод довів свою корисність у багатьох типах NN, де навчання можна легко дестабілізувати через велику кількість внутрішніх параметрів.

Спектральна нормалізація (Miyato et al. 2018) — це техніка нормалізації, яка використовується для стабілізації навчання дискримінатора. Цей метод нормалізує вагу для кожного шару (матриці ваг) відповідною спектральною нормою, також відомою як норма матриці, максимальне сингулярне значення матриці. За допомогою спектральної нормалізації ваги можна нормалізувати щоразу, коли вони оновлюються. Це створює мережу, яка пом’якшує проблеми вибуху градієнта і, таким чином, зменшує нестабільність у навчанні. Цей механізм показав подібні результати до WGAN-GP із меншим часом навчання.

Між цими двома методами спектральна нормалізація показала, що забезпечує більшу потужність представлення, не надто обмежуючи їх площу, як нормалізація

ваги, як підтверджено в оригінальній статті. Однак нормалізація ваги продовжує забезпечувати конкурентоспроможні результати при використанні на GAN (Xiang and Li 2017).

2.2 Початкова дистанція Фреше

Початкова відстань Фреше, також відома як FID (Heusel та ін. 2017), останніми роками виділяється як найсучасніший показник продуктивності в GAN. Хоча подібність між наборами зображень залишається відкритою проблемою в обробці зображень, FID є однією з останніх евристик, призначених для вирішення цієї проблеми. Це метрика для оцінки якості створених зображень, спеціально розроблена для оцінки продуктивності GAN (Lucic та ін. 2018).

FID використовує попередньо навчений Inception-v3 CNN (Szegedy et al. 2016) для виділення ознак реальних (x) і синтетичних (g) зображень. Зокрема, рівень кодування CNN (останній рівень об'єднання перед вихідною класифікацією зображень) використовується для захоплення властивостей комп'ютерного зору вхідного зображення, таким чином отримуючи вектор ознак із 2048 числових значень. Цей простір ознак інтерпретується як безперервний багатовимірний розподіл Гауса. Таким чином, на основі отриманих характеристик обчислюється відстань Фреше (також названа Вассертейном-2) між обома розподілами, використовуючи їх оцінене середнє (μ) і коваріацію (Σ). Чим нижчий цей показник, тим схожішими є два набори зображень, які дорівнюють нулю, коли вони рівні. Формула FID така:

$$FID(x, g) = \|\mu_x — \mu_g\|_2^2 + Tr(\Sigma_x + \Sigma_g — 2(\Sigma_x\Sigma_g)^{\frac{1}{2}})$$

де $\|\mu_x - \mu_g\|_2^2$ відноситься до суми квадратів різниці між двома середніми векторами, Tr(·) відноситься до операції лінійної алгебри трасування, тобто суми елементів вздовж головної діагоналі квадратної матриці та () — квадратний корінь із квадратної матриці, заданий як добуток двох коваріаційних матриць.

На рисунку 2.6, отриманому з оригінальної статті FID, показано приклади його збільшення залежно від рівня спотворень на зображеннях.

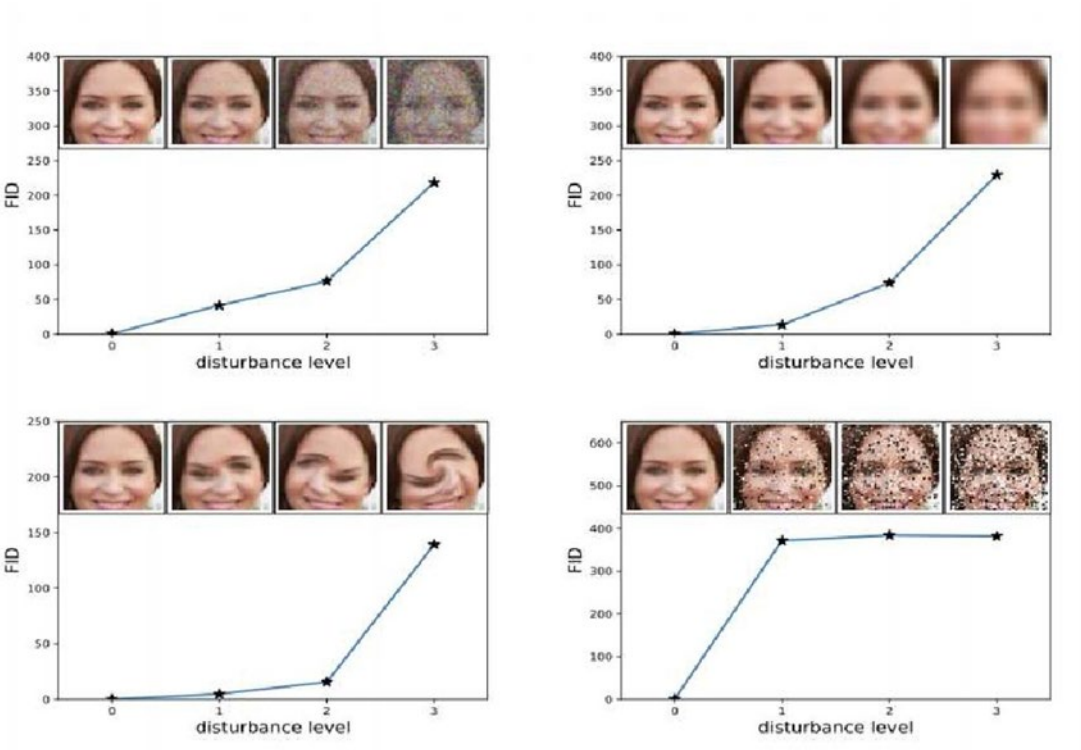


Рисунок 2.6- Оцінки FID з різними збуреннями.

Зліва направо, зверху вниз: гаусівський шум, гаусівське розмиття, закручені обличчя, шум солі та перцю. Отримано з оригінальної статті FID (Heusel et al. 2017).

2.2 Розробка підходу на основі SAGAN

Однією з найпоширеніших проблем навчання є «mode collapse». Результатом такої проблеми є те, що генератор завжди синтезує однакові, або майже ідентичні зображення. Це відбувається, зокрема, коли дискримінатор запізнюється з навчанням. В цьому випадку генератор знаходить якесь оптимальне зображення, яке завжди обманює дискримінатора. В результаті, незалежно від вхідного вектора шуму z, генератор буде синтезувати одне і те ж зображення. У зв'язку з цим всі останні дослідження сходяться на думці, що дискримінатора потрібно навчати швидше, ніж генератора. Це інтуїтивно зрозуміло, оскільки мережу розпізнавання спочатку потрібно навчити деяким шаблонам, перш ніж попросити розпізнати згенеровані зображення. Це міркування призвело до введення правила TTUR (two

time-scale update). У статті Martin Heusel наводяться докази впливу такого підходу на зближення до точки рівноваги по Нешу мінімакс гри дискримінатора і генератора

В основі мережі лежать три основні модулі – модуль «self-attention», модуль-генератор і модуль-дискримінатор. «self-attention» включається в якості додаткового шару, як в генераторі, так і в дискримінаторі. Самі модулі дискримінатора і генератора побудовані на основі глибоких згорткових мереж з тією різницею, що генератор використовує зворотні згорткові шари. Аналогічним чином для підвищення стійкості тренування, після кожного згортувального шару використовувався шар пакетної нормалізації.

Архітектура мережі зображена нижче на рисунку 2.7.

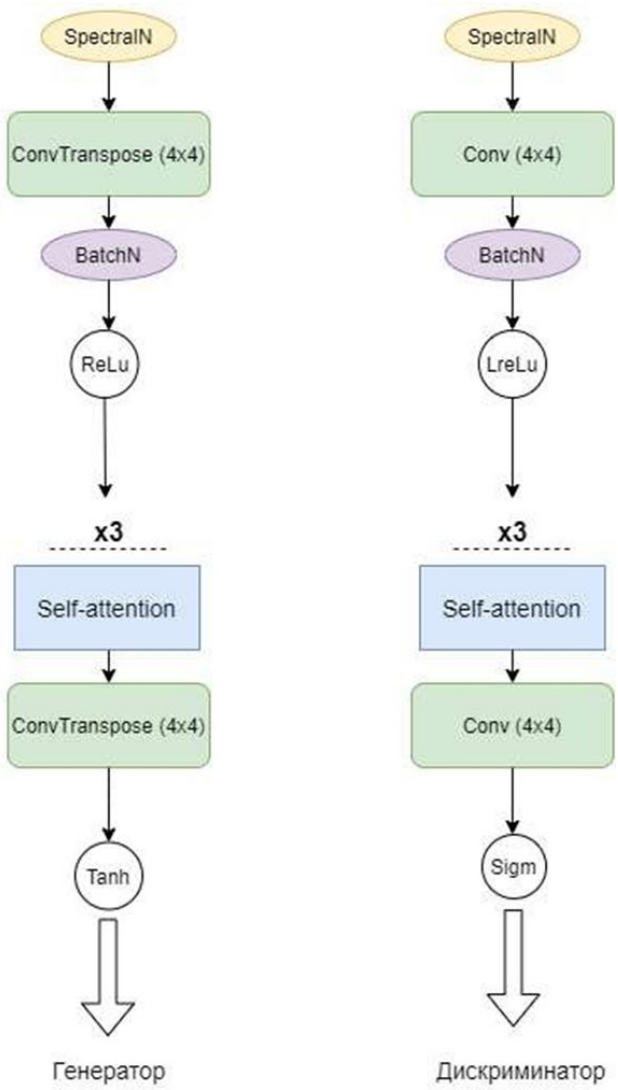


Рисунок 2.7 - Схема мережі в реалізації SAGAN

Як видно з малюнка, мережі практично симетричні.

Додаткові механізми оптимізації:

1. Згладжування міток при розрахунку значення помилки навчання (наприклад, замість 1 буде використовуватися випадкове число з сегмента [0.8,1]), що мінімізує обнулення градієнта для генератора, тобто стабілізує процес навчання. У багатьох роботах радять використовувати згладжування з одного боку.
2. Замість «pooling» шарів використовувалися конвуляційні з зрушенням (Alec Radford, 2016)
3. Була використана нормалізація партіями для генератора (такий підхід характерний для мережі SAGAN.), а також функція активації Relu для генератора і LeakyRelu для дискримінатора.
4. Використання оптимізатора Adam.

2.3 Розробка підходу на основі PGGAN

В результаті навчання SAGAN при дозволі 64 x 64 виникли певні труднощі, про які буде детально розказано в наступному розділі. Це показує, що класичному GAN, навіть при певних удосконаленнях, вкрай складно навчити синтезу зображень при досить високій роздільній здатності. Таким чином, необхідні деякі зміни і в самому процесі навчання. У 2018 році ряд дослідників з NVIDIA представили новий тип генеративної мережі під назвою Progressive Growing GAN (Tero Karras, 2018). У магістерській дисертації також була поставлена задача реалізації алгоритму, заснованого на цьому алгоритмі. Суть даного алгоритму полягає в зміні самої методики навчання генеративної мережі, в якій спочатку проводиться навчання для зображень в невеликих дозволах, починаючи з 4 пікселів. Потім, коли мережа достатньо навчена при низькій роздільній здатності, dводиться динамічне додавання нових шарів, що дозволяє збільшити розмірність зображення, що надходить; таким чином, новий етап навчання відбувається для зображень подвійного виміру. Процес покрокового подвоєння дозволу відбувається до тих пір, поки не буде досягнуто необхідного дозволу. Такий підхід дозволяє мережі спочатку вивчати складні структурні форми при невеликих

дозволах, а потім поступово деталізувати їх. Це також дозволяє значно скоротити час навчання, адже мережа зростає поетапно. Схема мережі PGGAN представлена на рисунок 2.2.

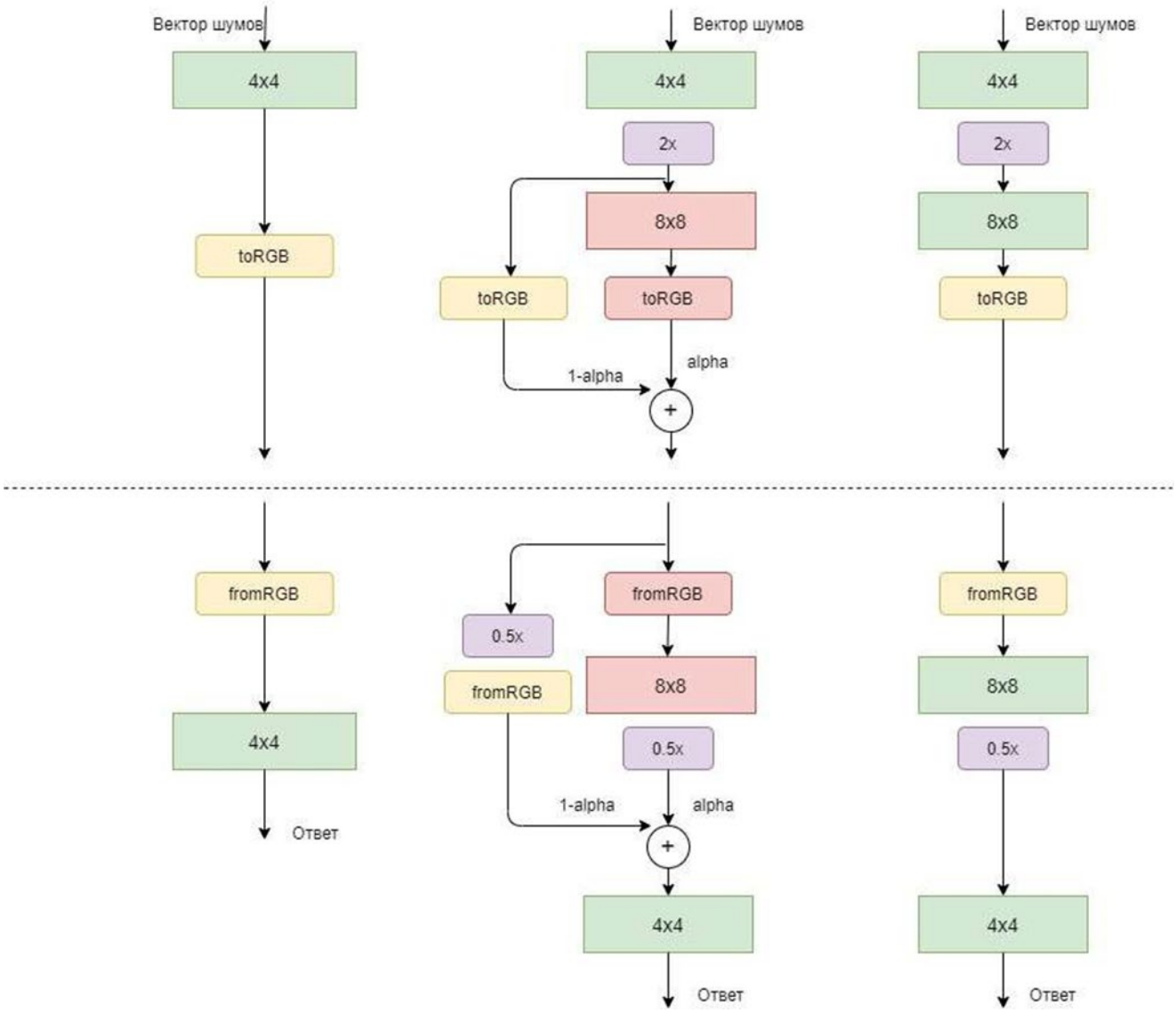


Рисунок. 2.8 - Схема роботи PGGAN

У процесі навчання можна виділити дві основні частини: стадію адаптації росту і стадію стабілізації.

Процес навчання починається з фази стабілізації – на цьому етапі мережа вчиться на зображеннях з мінімальним дозволом. Автори оригінальної статті виділили 800 тисяч ітерацій для кожного етапу. Далі мережа динамічно розширюється за рахунок додавання нового блоку. В якості блоку автори використовують два шари Conv3x3. Таким чином, в обидві мережі додається новий

блок: генератор і дискримінатор. У той же час вони спілкуються з уже працюючими блоками через шар upsample/downsample роздільної здатності для генератора та дискримінатора відповідно. Однак перебудова мережі відбувається поступово, завдяки стадії адаптації зростання. Робиться це шляхом введення альфа-параметра, який визначає, скільки даних доданого блоку буде використано в мережі. Для відображення «старої» частини мережі цей параметр буде дорівнює 1-alpha. На самому початку фази адаптації альфа дорівнює 0, тобто для мережі з моменту додавання нового блоку нічого не змінилося. З кожною ітерацією альфа-параметр збільшується рівномірно. Після етапу адаптації, коли весь потік даних проходить через новий шар, мережа «приклеюється», тобто видаляються непотрібні гілки, що підтримують стару структуру.

У мережі також використовується «mini sbatch discrimination» для підвищення варіативності синтезованих зображень, а також спектральної і пакетної нормалізації. Автори статті пропонують використовувати нормалізацію пікселів, але я віддав перевагу спектральній.

2.4 Оцінка якості результатів

Важливим моментом, який необхідно підкреслити, є правдоподібність образів. Думка конкретного індивіда не можна назвати правильною метрикою, тому в даній роботі я буду використовувати загальноприйняті показники. На даний момент найпопулярнішими метриками є оцінка початку (IS) і початкова відстань Frechet (FID). У своїй роботі я буду використовувати FID для оцінки правдоподібності образів.

Для розрахунку FID знову використовується попередньо навчена мережа для вилучення функцій з проміжних шарів. Розподіл даних за цими ознаками потім моделюється за допомогою багатофакторного розподілу Гауса з середнім значенням μ і коваріацією Σ. FID між реальними зображеннями x і згенерованими зображеннями g розраховується за такою формулою:

$$FID(x,g) = ||\mu_x - \mu_g||_2^2 + Tr\left(\Sigma_x + \Sigma_g - 2(\Sigma_x\Sigma_g)^{\frac{1}{2}}\right)$$

Де Tr− слід матриці, тобто сума всіх діагональних елементів

Чим нижче FID, тим вище якість згенерованих зображень.

Також варто відзначити, що офіційне значення оцінки якості генератора можливо тільки при використанні офіційних програмних пакетів, зокрема, це реалізація tensorflow для отримання балу FID. У даній роботі була використана реалізація Pyttorch. Його значення можуть мати деяку похибку порядку сотих або тисячних часток.

Висновки до розділу 2

1. В розділ 2 розгляну архітектури та реалізації генеративно-змагальних мереж. Після чого було вибрано дві мережі на основі яких будувалися дослідження у даній роботі.

2. У цьому розділі ми запропонували використовувати генеративні змагальні мережі самоуваги (SAGAN), які включають механізм самоуваги в структуру GAN. Запропонований модуль самоуваги ефективний при моделюванні далеких залежностей. Крім того, ми показуємо, що спектральна нормалізація, застосована до генератора, може стабілізувати навчання ГАН, а TTUR може прискорити навчання регульованих дискримінаторів..

3. Хоча якість наших результатів, як правило, висока в порівнянні з більш ранньою роботою над GAN, а навчання стабільне у великій роздільній здатності, існує довгий шлях до справжнього фотореалізму. Семантична чутливість і розуміння залежних від набору даних обмежень, таких як те, що певні об'єкти є прямими, а не кривими, залишає бажати кращого. Також є місце для вдосконалення мікроструктури зображень.

3. ПРОГРАМНА РЕАЛІЗАЦІЯ І ЕКСПЕРЕМЕНТИ

3.1 Опис апаратного забезпечення

Завдання, поставлене в магістерській дисертації, було вирішено за допомогою програмної платформи розробки Pytorch, яка є реалізацією відомої програмної платформи Torch для мови програмування Python. Враховуючи, що навчання генеративних змагальних мереж є обчислювально складним завданням, навчання мережі на процесорі перетворило б проблему на майже нерозв'язну. Таким чином, навчання всіх мереж проходило на відеокарті з використанням CUDA SDK (CUD). В рамках даної магістерської дисертації була використана одна з найпотужніших на даний момент відеокарт NVIDIA TESLA K80 (NVI) з 12Гб пам'яті. Згадана інфраструктура була розгорнута на віртуальному сервері Google Cloud. Ноутбук Jupyter (jup), встановлений на віртуальному сервері, використовувався в якості інтегрованого середовища розробки. Аналітика даних з різними графіками була проведена в пакеті Tensorboard (Ten)

3.2 Використовувані дані і їх підготовка

В якості набору даних, на якому проводився тренінг, використовувалися наступні два варіанти: фотографії облич знаменитостей з роздільною здатністю 64 пікселів і набір даних LSUN (LSU) з категорії церков з роздільною здатністю також 64 пікселя.

Для підготовки даних був реалізований клас Loader, який є обгорткою для стандартного torch.utils.data.DataLoader. Крім забезпечення інтерфейсу torch.utils.data.DataLoader, клас Loader реалізує ряд необхідних перетворень поверх зображень:

1) Масштабування зображення (у випадку з PGGAN). Це перетворення необхідно для того, щоб тренуватися з різною роздільною здатністю: від 4 до 64 пікселів. Як тип інтерполяції використовувався алгоритм пошуку «найближчого сусіда».

2) Переклад зображення в тензор – об'єкт, яким керує програмна платформа pytorch.

3) Нормалізація. Так як в генераторній мережі я часто використовував функцію тангенса активації як функцію останнього шару, то вихід генераторної мережі в даному випадку знаходився в діапазоні [-1,1], і відповідно, реальні зображення, що надходять на вхід дискримінатора, повинні бути з цього діапазону.

Також варто згадати, що я не використовував таку корисну операцію, як змішування даних при отриманні («shuffle»), так як ця операція сильно збільшила час роботи алгоритму.

3.3 Особливості програмної реалізації

Якості підтвердження складності поставленого завдання була реалізована класична модель GAN для відомої задачі генерації лиць знаменитостей в дозволі 32x32, що нижче потрібного дозволу. Мережа побудована за класичним глибоким згортковим шаблоном GAN (DCGAN).

Однак навіть при невеликому дозволі людське око легко розпізнає синтезоване зображення:

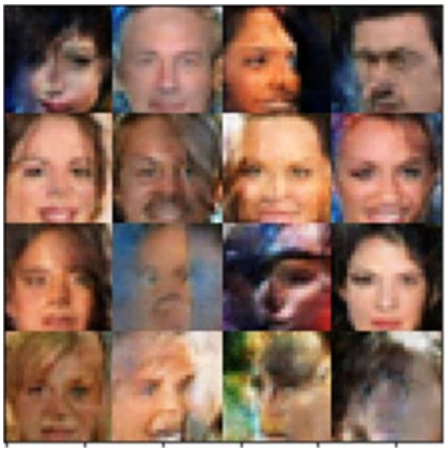


Рисунок 3.1- Згенеровані зображення обличчя

Метрика FID для оцінки якості згенерованих зображень в даному випадку становить 102, 35.

До складу мереж входять 4 основних модуля: завантажувач даних, який вже був описаний вище, клас генератора, клас дискримінатора, а також навчальний модуль, в якому відбуваються всі ітерації та оновлення ваг мережі.

Класи генераторів і дискримінаторів успадковують стандартний клас torch.nn.Module. З огляду на, що в структурі мереж чергуються повторювані модулі, вдалося при реалізації звести цей функціонал в окрему функцію. Приклад наведено на малюнку нижче:

```
def conv_block(self, first):
    block = []
    first_cov, second_cov = g_sizes[self.current_s]

    if first:
        block.append(nn.ConvTranspose2d(self.latent_size, first_cov, 4))
        block.append(nn.BatchNorm2d(first_cov))
        block.append(nn.LeakyReLU(0.2))
    else:
        block.append(nn.Conv2d(first_cov, first_cov, 3,padding = 1))
        block.append(nn.BatchNorm2d(first_cov))
        block.append(nn.LeakyReLU(0.2))

    block.append(nn.Conv2d(first_cov, second_cov, 3,padding = 1))
    block.append(nn.BatchNorm2d(second_cov))
    block.append(nn.LeakyReLU(0.2))
    return nn.Sequential(*block)
```

Рисунок 3.2- Приклад переносу блоку мережі PGGAN у функцію

Примітка про навчальний модуль: його завданням є отримання значення похибок мережі генератора і мережі-дискримінатора (на реальних і згенерованих даних) і оновлення ваг відповідно до отриманих значень градієнта. Цей модуль також збирає статистику тренувань в Tensorboard (Ten).

У мережі SAGAN є ряд унікальних деталей, зокрема модуль «self-attention», реалізація якого успадковує torch.nn.Module і містить кілька згорткових шарів 1x1 з власними вагами, а також виконує ряд матричних множень. Код для цього модуля показаний нижче на рисунку:

```
class attn(nn.Module):
    def __init__(self, insize):
        super(attn,self).__init__()
        self.insize = insize
        self.g_conv1to1 = nn.Conv2d(insize, insize//8, 1, 1, 0)
        self.f_conv1to1 = nn.Conv2d(insize, insize//8, 1, 1, 0)
        self.h_conv1to1 = nn.Conv2d(insize, insize, 1, 1, 0)
        self.gamma = nn.Parameter(torch.zeros(1))

    def forward(self, input):
        batchsize,channels,width ,height = input.size()

        f = self.f_conv1to1(input)
        g = self.g_conv1to1(input)
        h = self.h_conv1to1(input)

        f_processed = f.view(batchsize,-1, width*height).permute(0,2,1)
        g_processed = g.view(batchsize,-1, width*height)

        attention_matrix = \
        torch.nn.functional.softmax(torch.bmm(f_processed, g_processed \
                                              ),dim =-1)
        h_res = torch.bmm(h.view(batchsize,-1, width*height), attention_matrix.permute(0,2,1))
        h_res = h_res.view(batchsize, channels, width, height)
        return self.gamma* h_res + input
```

Рисунок 3.3- Модуль «self-attention»

Процес навчання мережі PGGAN принципово відрізняється від навчання класичних генеративних змагальних мереж тим, що складається з декількох фаз. Тому навчальний модуль виглядає по-іншому

```
d_optimizer = optim.Adam(D.parameters(), lr = lr, betas=(0.5, 0.999))
g_optimizer = optim.Adam(G.parameters(), lr = lr, betas=(0.5, 0.999))

while current_ext <= max_ext:
    step(g_optimizer, d_optimizer, False, D, G, loader)

    if current_ext < max_ext:
        reset_optimizer(g_optimizer, G)
        reset_optimizer(d_optimizer, D)

        D.extend()
        D = D.to(device)
        G.extend()
        G = G.to(device)
        current_ext *= 2

        loader = Data_Loader('celebs', current_ext, batch_size).loader()
        step(g_optimizer, d_optimizer, True, D, G, loader)

        D.stabilize()
        D = D.to(device)
        G.stabilize()
        G = G.to(device)
        reset_optimizer(g_optimizer, G)
        reset_optimizer(d_optimizer, D)
    else:
        break

step(g_optimizer, d_optimizer, False, D, G, loader)

torch.cuda.empty_cache()
```

Рисунок 3.4- Навчальний модуль мережі PGGAN

Як видно з коду, представленого на рис., спочатку відпрацьовується фаза стабілізації, після чого мережа зростає і відбувається навчання в режимі адаптації і т.д. При цьому після кожного етапу проводиться очищення градієнтів оптимізаторів. В самому кінці, як пропонується в оригінальній статті, виконується заключний другий етап стабілізації. Ще одним ключовим моментом є той факт, що правило TTUR погано працює для мережі PGGAN, тобто швидкість навчання повинна бути однаковою як для генератора, так і для дискримінатора.

Також для мереж PGAN з'явилися два методи, що реалізують адаптивний і стабілізаційний прохід:

```
def forward_adaptive(self, x, alpha):
    current_output = self.current_blocks(x)
    new_block_output = self.new_block(current_output)
    x = alpha * (self.new_to_rgb(new_block_output)) + (1.0-alpha) * self.current_to_rgb(current_output)
    return x

def forward(self,x):
    return self.current_to_rgb(self.current_blocks(x))
```

Рисунок 3.5- Два етапи навчання мережі .

Також варто відзначити, що вибір кількості ітерацій залежить від дозволу. Був введений коефіцієнт, що дозволяє довше тренуватися з великими дозволами.

3.4 Результати експериментів

На першому кроці я навчив мережу на наборі даних, що складається із зображень обличчя знаменитостей з роздільною здатністю 32 на 32 пікселя. Приклад граней, згенерованих тривіальним DCGAN без будь-яких оптимізацій, вже був наведений вище. У випадку з оптимізованою мережею результат виглядає краще навіть при невеликих періодах вибірки (епохи). Значення FID для згенерованих зображень становить 80,91, але у випадку зі звичайною мережею це значення було вище 100 (зображення на малюнку нижче здається розмитим, оскільки кожне зображення масштабується до роздільної здатності 64 x 64):

Рисунок 3.6 Зображення лиць, отримані в результаті роботи мережі SAGAN, і приклади реальних зображень лиць.

Як описано вище, більшість робіт з генеративних мереж використовують швидкість навчання (LR) 0,0001. У моїх експериментах використовувалися різні значення, і значення 0,0001 призвело до кращої продуктивності мережі. На графіку нижче показано зміну похибки генератора для двох варіантів швидкості навчання: 0,0001, 0,0002.

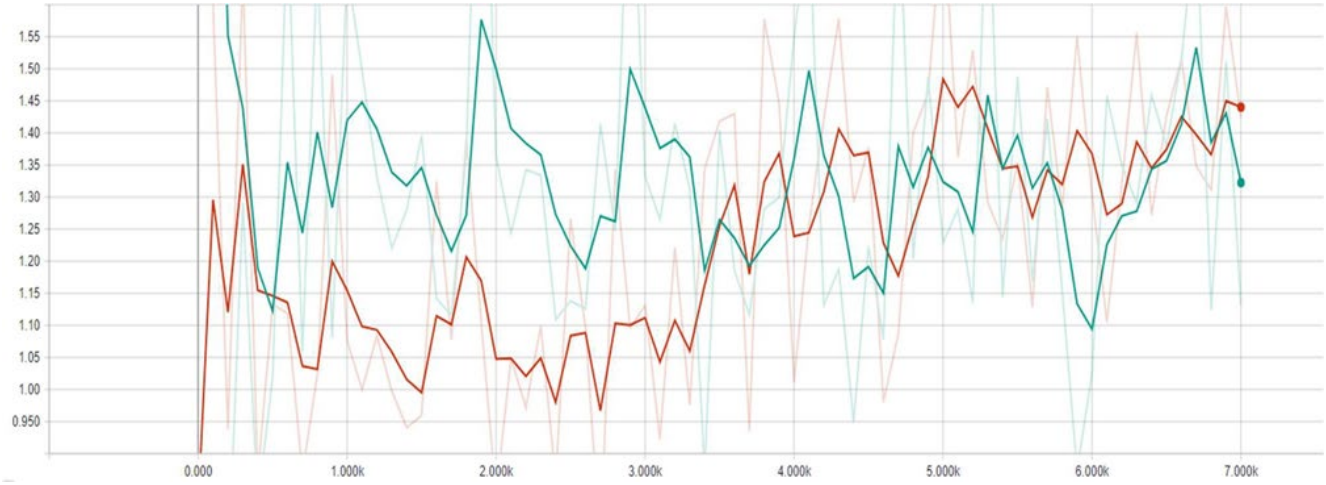


Рисунок 3.7- зелений показує похибку для lr = 0,0001, червоний – 0,0002

З графіків на рисунку 12 може здатися, що графік, відповідний lr = 0,0001, більш плавний, відповідно, і навчання відбувається передбачувано, але наявність шипів - всього лише ознака так званої зміни моди. Тобто це ознака мінливості синтезованих образів. При швидкості навчання lr = 0, 0002 вкрай легко домогтися колапсу режиму. Нижче наведено графік зміни помилок генераторної та дискримінаторі мережі для lr = 0,0001:

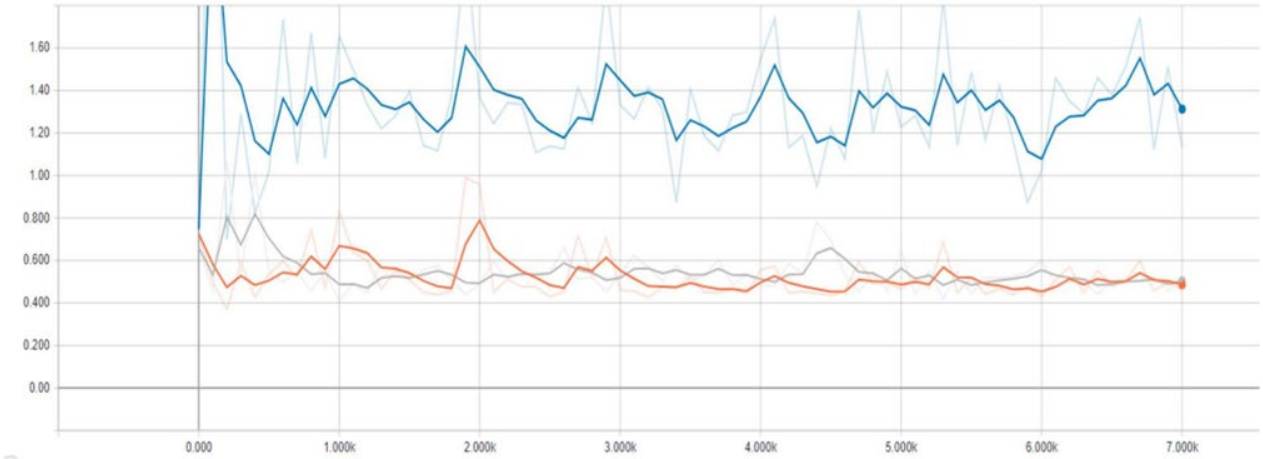


Рисунок 3.8 -Графік зміни помилок генератора і дискримінатора з фіксованим lr.Blue graph – генератор.

Як видно з графіка, обидва параметри, дискримінатора похибка на реальних даних і похибка дискримінатора на даних генератора, не показують істотних коливань і не коливаються поблизу нульових значень, що свідчить про стабільний процес навчання.

З урахуванням всіх проведених мною експериментів найкраще показала себе швидкість навчання lr = 0, 00009. На малюнку нижче показані аналогічні графіки похибок для різних значень lr [0,00008, 0,00009,0,0001]:

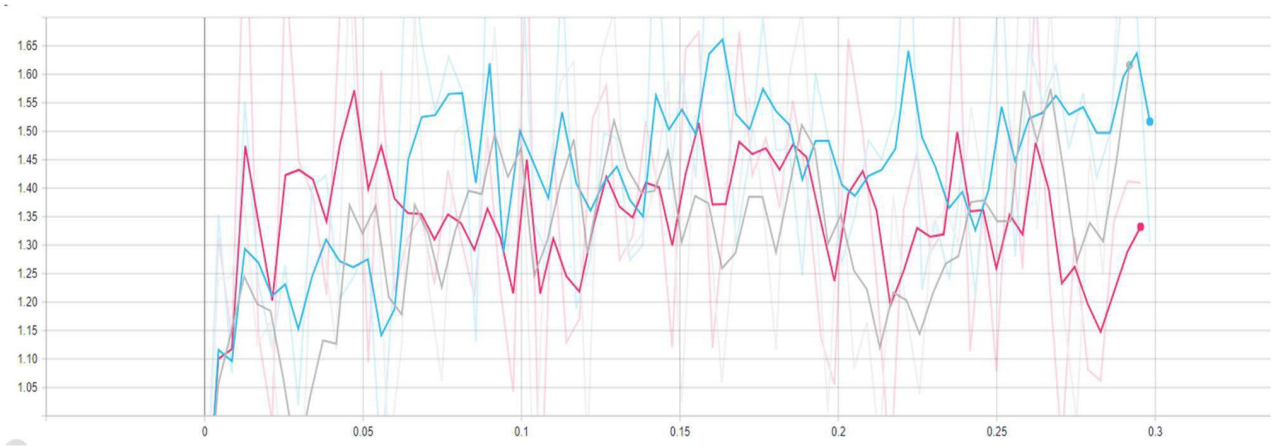


Рисунок 3.9- червоний колір позначає значення похибок при LR=0,00009, синій – 0,0001

Варто відзначити, що на такі роздільні здатності вплив шару self-attention мінімально за рахунок того, що дозвіл не велике. У той же час наявність спектральної нормалізації серйозно впливає на якість мережевого навчання. Нижче наведено графік зміни похибки осцилятора при наявності і відсутності спектральної нормалізації. Як бачите, при його відсутності похибка генератора зростає досить швидко:

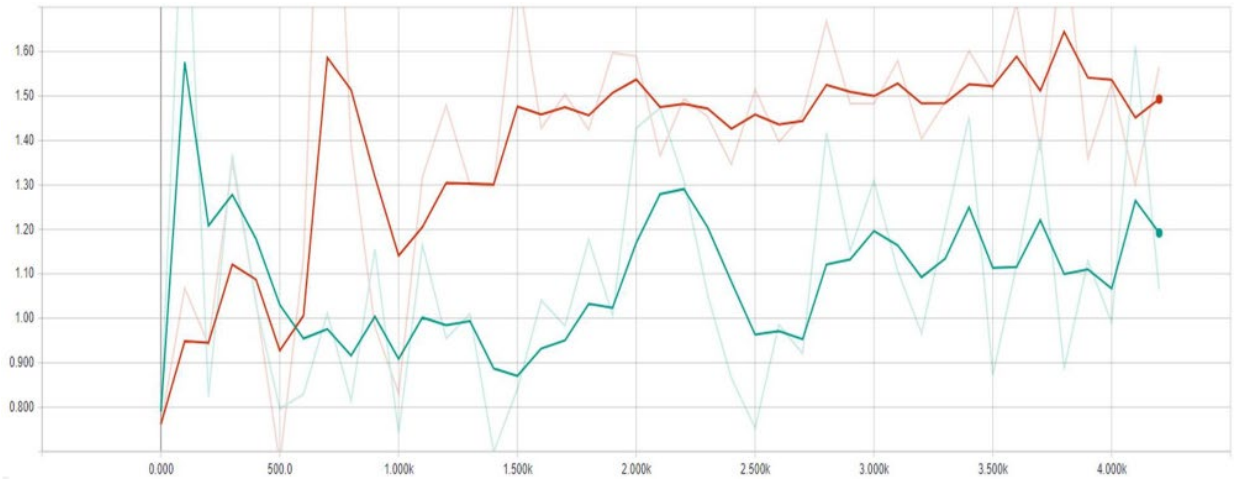


Рисунок 3.10- Помилка генератора в разі наявності (зелений) і відсутності (червоний) спектральної нормалізації.

Під час переходу до роздільної здатності 64 x 64 виникли проблеми з мережею. Головною складністю була ситуація «колапсу режиму». Стандартні рішення цієї проблеми, такі як зниження значення lr, виявилися неефективними.

Рисунок 3.11- «Mode-collapse» у разі синтезу зображень обличчя

Існує методика, яка, в деяких випадках, дозволяє впоратися з подібною проблемою: дискримінація по міні-партіях (мініпакетна дискримінація). Суть методу зводиться до того, що в момент проходження даних по мережі враховується не тільки один конкретний елемент, але і всі елементи мініпакета. Також розраховується нова статистика: міра подібності зображень, що надходять на вхід. Таким чином, мережа вчиться «штрафувати» випадки, коли потрапляє безліч подібних зображень, і саме це і відбувається в разі колапсу режиму. Цей прийом поліпшив якість генерованих зображень в дозволі 32 x 32, але не зміг істотно вплинути на здатність цієї мережі синтезувати зображення в дозволі 64 x 64.

В якості експерименту також були синтезовані зображення для відомого набору даних LSUN церковної категорії. У цьому прикладі проблеми «колапсу режиму» не спостерігалося, проте якість отриманих результатів було вкрай низьким. Метрика FID не використовувалася, тому що навіть неозброєним оком легко відрізнити синтезоване зображення від реального:

Рисунок 3.12- Приклади зображень церков, отриманих SAGAN на знімальному майданчику Дані LSUN

Навчання мережі в даному випадку займає чималий час: у випадку з 300к ітерація на кожному етапі являє собою приблизний час виконання в 4 години, враховуючи, що всі розрахунки проводилися на продуктивній відеокарті (NVI). Відповідно, я не проводив експериментів на заявлених авторами 800-тисячних ітераціях. В результаті в перших дослідах я отримав такі зображення:

Рисунок 3.13- Результат PGGAN

Згенеровані зображення все ще досить легко відрізнити від реальних зображень, однак їх якість помітно краще, ніж якість зображень, синтезованих за допомогою SAGAN.

Графік процесу навчання представлений нижче на зображенні:

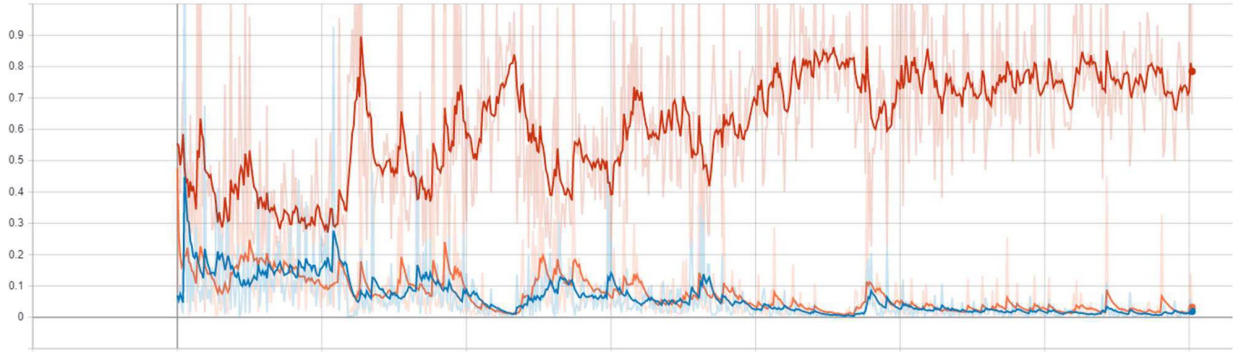


Рисунок3.14-Змініть помилку генератора (червоної лінії), а також дискримінатора при навчанні PGGAN

Як видно з графіка, при переході на новий дозвіл помилка генератора збільшується. Тут важливо «утримати» зменшення похибки дискримінатора і збільшення помилки генератора до тих пір, поки не буде досягнуто необхідного дозволу.

Цей тип мережі (PGGAN) вимагає більш ретельного занурення та дослідження. Можливо, при правильному підборі гіперпараметрів, функцій помилок, а також алгоритмів нормалізації та інших оптимізацій, зображення, що генеруються за допомогою PGGAN, буде вкрай складно відрізнити від реальних. В рамках цієї роботи подібні дослідження не проводилися через вкрай серйозних тимчасових і апаратних витрат на проведення експериментів.

В рамках даної магістерської дисертації ретельне дослідження впливу функції помилки не проводилося. У більшості випадків використовувалася стандартна «змагальна втрата» від torch.nn.BCELoss (у випадку з SAGAN) або середньоквадратична похибка кореня (у випадку з PGGAN). У той же час помилка Вассерштейна також була розрахована для PGGAN, але вона не показала себе правильною.

Таким чином: в досить близькі епохи похибка дискримінатора стала стрімко зменшуватися, в той час як похибка генератора росла. Варто також зазначити, що в обох випадках я використовував одностороннє згладжування цільових значень дискримінаторів.

Висновки до розділу 3

1. Вивчено велику кількість методів оптимізації на день побудови нейронних мереж для синтезу зображень. На основі останніх досліджень SAGAN і BigGAN був реалізований генеративний алгоритм змагальної мережі.

2. Експериментальним шляхом були отримані гіперпараметри, що дозволяють найбільш стабільно тренувати таку мережу. В результаті роботи цієї мережі були отримані високоякісні зображення в дозволі 32 пікселя і розрахована відповідна метрика FID, яка показала найкращі результати при використанні класичного DCGAN.

3. Однак також було показано, що при генерації зображень більш високої роздільної здатності в мережі на основі алгоритму SAGAN виникають проблеми з якістю одержуваних зображень.

4. В якості контрзаходу був проаналізований новий алгоритм генеративних мереж, що реалізує іншу методику навчання - PGGAN. На основі такого підходу була реалізована мережа, яка значно перевершує SAGAN за якістю згенерованих зображень у високій роздільній здатності.

5. Також був запропонований ряд напрямків для подальшого вивчення, які потенційно могли б значно підвищити якість і ефективність операцій GAN.

ВИСНОВКИ

В рамках даної роботи були проаналізовані новітні методи синтезу зображень при високій роздільній здатності. В результаті написання роботи отримані наступні основні результати.

1. У цьому розділі розглядалися базові основи штучних нейронних мереж, згорткових нейронних мереж, а також генеративних змагальних мереж, їх варіантів і доповнень для підвищення їх стабільності.

2. Також було представлено внесок GAN у збільшення даних. Показано тематичні дослідження, розглянуте в цій дослідницькій роботі.

3. Було проведено постановку задачі для цієї роботи. Рішення повинно являти собою розумний компроміс між якістю отриманих зображень і швидкістю навчання, але першочерговою метою оптимізації буде якість зображень, отриманих від виходу мережі.

4. В розділ 2 розгляну архітектури та реалізації генеративно-змагальних мереж. Після чого було вибрано дві мережі на основі яких будувалися дослідження у даній роботі.

5. У цьому розділі ми запропонували використовувати генеративні змагальні мережі самоуваги (SAGAN), які включають механізм самоуваги в структуру GAN. Запропонований модуль самоуваги ефективний при моделюванні далеких залежностей. Крім того, ми показуємо, що спектральна нормалізація, застосована до генератора, може стабілізувати навчання ГАН, а TTUR може прискорити навчання регульованих дискримінаторів..

6. Хоча якість наших результатів, як правило, висока в порівнянні з більш ранньою роботою над GAN, а навчання стабільне у великій роздільній здатності, існує довгий шлях до справжнього фотореалізму. Семантична чутливість і розуміння залежних від набору даних обмежень, таких як те, що певні об'єкти є прямими, а не кривими, залишає бажати кращого. Також є місце для вдосконалення мікроструктури зображень.

7. Вивчено велику кількість методів оптимізації на день побудови нейронних мереж для синтезу зображень. На основі останніх досліджень SAGAN і BigGAN був реалізований генеративний алгоритм змагальної мережі.

8. Експериментальним шляхом були отримані гіперпараметри, що дозволяють найбільш стабільно тренувати таку мережу. В результаті роботи цієї мережі були отримані високоякісні зображення в дозволі 32 пікселя і розрахована відповідна метрика FID, яка показала найкращі результати при використанні класичного DCGAN.

9. Однак також було показано, що при генерації зображень більш високої роздільної здатності в мережі на основі алгоритму SAGAN виникають проблеми з якістю одержуваних зображень.

10. В якості контрзаходу був проаналізований новий алгоритм генеративних мереж, що реалізує іншу методику навчання - PGGAN. На основі такого підходу була реалізована мережа, яка значно перевершує SAGAN за якістю згенерованих зображень у високій роздільній здатності.

11. Також був запропонований ряд напрямків для подальшого вивчення, які потенційно могли б значно підвищити якість і ефективність операцій GAN.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Sudarshan Adiga, Mohamed Adel Attia, Wei-Ting Chang, and Ravi Tandon. “ON THE TRADEOFF BETWEEN MODE COLLAPSE AND SAMPLE QUALITY IN GENERA- TIVE ADVERSARIAL NETWORKS”. In: 2018 IEEE Global Conference on Signal and In- formation Processing (GlobalSIP). 2018 IEEE Global Conference on Signal and Information Processing (GlobalSIP). Anaheim, CA, USA: IEEE, Nov. 2018, pp. 1184–1188. ISBN: 978-1-72811-295-4. DOI: 10 . 1109 / GlobalSIP . 2018 . 8646478. URL: https : / / ieeexplore.ieee.org/document/8646478/ (visited on 05/26/2021).

2 Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein GAN. Dec. 6, 2017. arXiv: 1701.07875 [cs, stat]. URL: http://arxiv.org/abs/1701.07875 (visited on 02/08/2021).

3 A Basheer and M Hajmeer. “Artificial Neural Networks: Fundamentals, Computing, Design, and Application”. In: Journal of Microbiological Methods. Neural Computting in Micrbiology 43.1 (Dec. 1, 2000), pp. 3–31. ISSN: 0167-7012. DOI: 10 . 1016 / S0167 - 7012(00) 00201 - 3. URL: https : / / www . sciencedirect . com / science / article/pii/S0167701200002013 (visited on 05/11/2021).

4 Ali Borji. Pros and Cons of GAN Evaluation Measures. Oct. 23, 2018. arXiv: 1802.03446 [cs]. URL: http://arxiv.org/abs/1802.03446 (visited on 02/10/2021).

5 Jason Brownlee. Deep Learning for Computer Vision: Image Classification, Object Detection, and Face Recognition in Python. Machine Learning Mastery, Apr. 4, 2019. 564 pp. Google Books: DOamDwAAQBAJ.

6 Jason Brownlee. How to Develop a Conditional GAN (cGAN) From Scratch. Machine Learn- ing Mastery. July 4, 2019. URL: https://machinelearningmastery.com/how- to- develop- a- conditional- generative- adversarial- network- from- scratch/ (visited on 02/09/2021).

7 Emily Denton, Soumith Chintala, Arthur Szlam, and Rob Fergus. Deep Generative Image Models Using a Laplacian Pyramid of Adversarial Networks. June 18, 2015. arXiv: 1506. 05751 [cs]. URL: http://arxiv.org/abs/1506.05751 (visited on 03/22/2021).

8 Vincent Dumoulin and Francesco Visin. “A Guide to Convolution Arithmetic for Deep Learning”. In: (Mar. 23, 2016). URL: https://arxiv.org/abs/1603.07285v2 (visited on 05/24/2021).

9 Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Networks. June 10, 2014. arXiv: 1406.2661 [cs, stat]. URL: http://arxiv.org/abs/ 1406.2661 (visited on 02/10/2021).

10 Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equi- librium. Jan. 12, 2018. arXiv: 1706.08500 [cs, stat]. URL: http://arxiv.org/ abs/1706.08500 (visited on 05/06/2021).

11 Tero Karras, Samuli Laine, and Timo Aila. “A Style-Based Generator Architecture for Generative Adversarial Networks”. In: (Dec. 12, 2018). URL: https://arxiv.org/ abs/1812.04948v3 (visited on 05/24/2021).

12 Keras: The Python Deep Learning API. URL: https : / / keras . io/ (visited on 05/16/2021).

13 Shaohui Liu, Yi Wei, Jiwen Lu, and Jie Zhou. An Improved Evaluation Framework for Gen- erative Adversarial Networks. July 19, 2018. arXiv: 1803.07474 [cs]. URL: http:// arxiv.org/abs/1803.07474 (visited on 05/13/2021).

14 Vishnu Makkapati and Arun Patro. “Enhancing Symmetry in GAN Generated Fash- ion Images”. In: Artificial Intelligence XXXIV. Ed. by Max Bramer and Miltos Petridis. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 405–410. ISBN: 978-3-319-71078-5. DOI: 10.1007/978-3-319-71078-5_34.

15 Mastit hos mjölkkor - SVA. URL: /djurhalsa/djursjukdomar-a-o/mastit-hos-mjolkkor/ (visited on 05/01/2021).

16 Matplotlib: Python Plotting — Matplotlib 3.4.2 Documentation. URL: https : / / matplotlib.org/ (visited on 05/16/2021).

17 Kyle McDonald. How to Recognize Fake AI-Generated Images. Medium. Dec. 14, 2018. URL: https://kcimc.medium.com/how- to- recognize- fake- ai- generated-images-4d1f6f9a2842 (visited on 05/25/2021).

18 Mehdi Mirza and Simon Osindero. Conditional Generative Adversarial Nets. Nov. 6, 2014. arXiv: 1411 . 1784 [cs, stat]. URL: http : / / arxiv . org / abs / 1411 . 1784 (visited on 02/18/2021).

19 NumPy. URL: https://numpy.org/ (visited on 05/16/2021).

20 Keiron O'Shea and Ryan Nash. An Introduction to Convolutional Neural Networks. Dec. 2, 2015. arXiv: 1511 . 08458 [cs]. URL: http : / / arxiv . org / abs / 1511 . 08458 (visited on 05/16/2021).

21 Augustus Odena, Vincent Dumoulin, and Chris Olah. "Deconvolution and Checker- board Artifacts". In: Distill 1.10 (Oct. 17, 2016), e3. ISSN: 2476-0757. DOI: 10.23915/ distill.00003. URL: http://distill.pub/2016/deconv- checkerboard (visited on 01/28/2021).

22 Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised Representation Learn- ing with Deep Convolutional Generative Adversarial Networks. Jan. 7, 2016. arXiv: 1511. 06434 [cs]. URL: http://arxiv.org/abs/1511.06434 (visited on 01/31/2021).

23 F. Rosenblatt. The Perceptron, a Perceiving and Recognizing Automaton Project Para. Cornell Aeronautical Laboratory, 1957. book. Google Books: P_XGPgAACAAJ.

24 Stuart J. Russell and Peter Norvig. Artificial Intelligence : A Modern Approach. Pren- tice Hall Series in Artificial Intelligence. Pearson Education Limited, 2016. ISBN: 978- 1-292-15396-4. URL: https : / / login . e . bibl . liu . se / login ? url = https : //search.ebscohost.com/login.aspx?direct=true&AuthType=ip,uid&db=cat00115a&AN=lkp.927144&lang=sv&site=eds-live&scope=site.

25 Martin Arjovsky and L´eon Bottou. Towards principled methods for training generative adversarial networks. In ICLR, 2017. Martin Arjovsky, Soumith Chintala, and L´eon Bottou. Wasserstein GAN. CoRR, abs/1701.07875, 2017.

26 Sanjeev Arora and Yi Zhang. Do GANs actually learn the distribution? an empirical study. CoRR, abs/1706.08224, 2017.

27 Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. CoRR, abs/1607.06450, 2016.

28 Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. In P. B. Sch¨olkopf, J. C. Platt, and T. Hoffman (eds.), NIPS, pp. 153–160. 2007.

29 David Berthelot, Tom Schumm, and Luke Metz. BEGAN: Boundary equilibrium generative adversarial networks. CoRR, abs/1703.10717, 2017.

30 Peter J. Burt and Edward H. Adelson. Readings in computer vision: Issues, problems, principles, and paradigms. chapter The Laplacian Pyramid As a Compact Image Code, pp. 671–679. 1987.

31 Qifeng ChenandVladlenKoltun. Photographic image synthesis with cascaded refinement networks. CoRR, abs/1707.09405, 2017.

32 Zihang Dai, Amjad Almahairi, Philip Bachman, Eduard H. Hovy, and Aaron C. Courville. Calibrating energy-based generative adversarial networks. In ICLR, 2017.

33 Emily L. Denton, Soumith Chintala, Arthur Szlam, and Robert Fergus. Deep generative image models using a Laplacian pyramid of adversarial networks. CoRR, abs/1506.05751, 2015.

34 Vincent Dumoulin, Ishmael Belghazi, Ben Poole, Alex Lamb, Martin Arjovsky, Olivier Mastropietro, and Aaron Courville. Adversarially learned inference. CoRR, abs/1606.00704, 2016.

35 Ishan P. Durugkar, Ian Gemp, and Sridhar Mahadevan. Generative multi-adversarial networks. CoRR, abs/1611.01673, 2016.

36 Bernd Fritzke. A growing neural gas network learns topologies. In G. Tesauro, D. S. Touretzky, and T. K. Leen (eds.), Advances in Neural Information Processing Systems 7, pp. 625–632. 1995.

37 Arnab Ghosh, Viveka Kulharia, Vinay P. Namboodiri, Philip H. S. Torr, and Puneet Kumar Dokania. Multi-agent diverse generative adversarial networks. CoRR, abs/1704.02906, 2017.

38 Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Networks. In NIPS, 2014.

39 Guillermo L. Grinblat, Lucas C. Uzal, and Pablo M. Granitto. Class-splitting generative adversarial networks. CoRR, abs/1709.07359, 2017.

40 Ishaan Gulrajani, Faruk Ahmed, Mart´ın Arjovsky, Vincent Dumoulin, and Aaron C. Courville. Improved training of Wasserstein GANs. CoRR, abs/1704.00028, 2017.

41 Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. CoRR, abs/1502.01852, 2015.

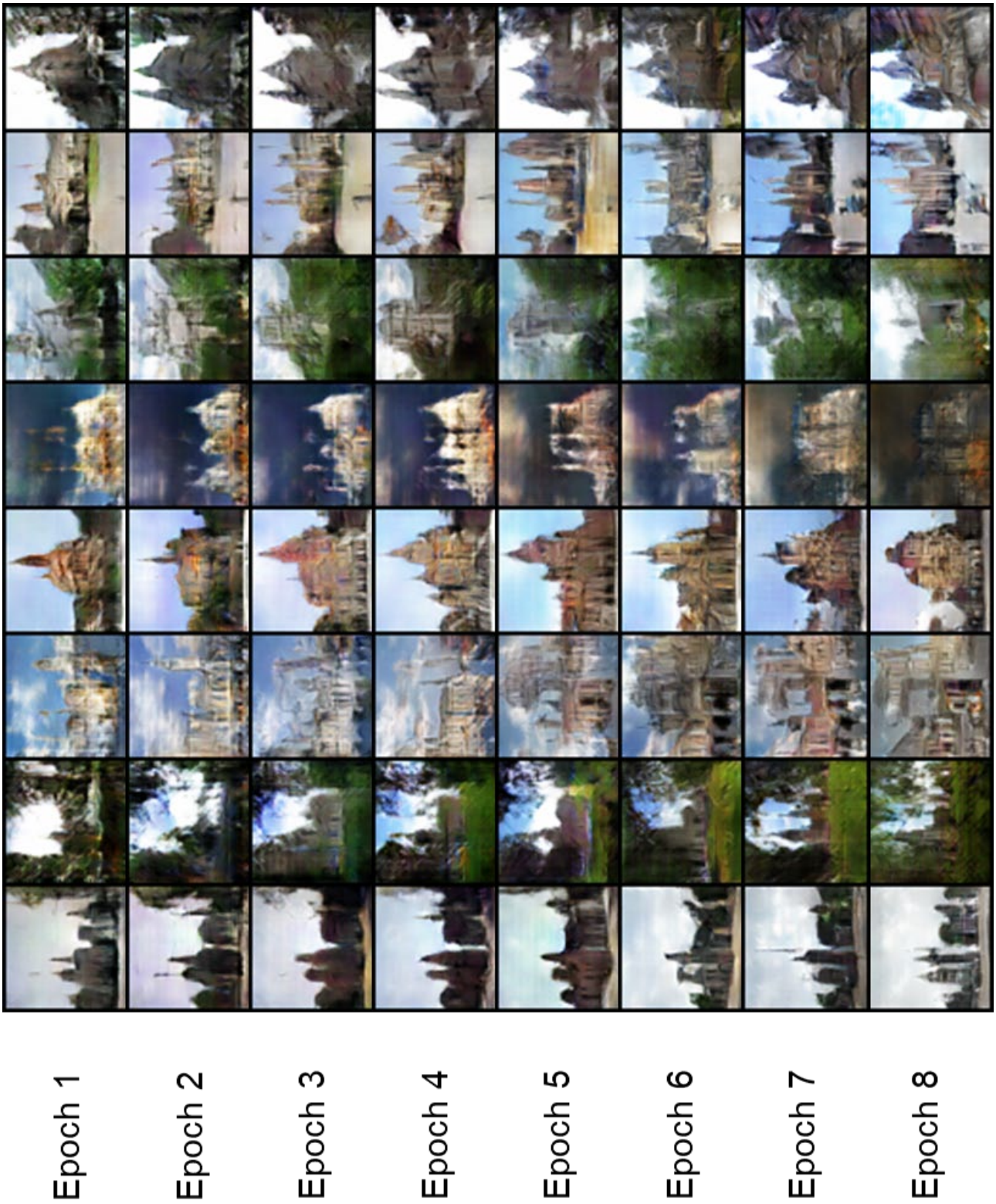
42 Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. GANstrained by a two time-scale update rule converge to a local Nash equilibrium. In NIPS, pp. 6626–6637. 2017.

43 Загальні рекомендації з підготовки, оформлення, захисту та оцінювання випускних кваліфікаційних робіт здобувачів вищої освіти першого «бакалаврського» і другого «магістерського» рівнів / За ред. доц. М.І. Шинкарика. Тернопіль: ТНЕУ, 2018. 67 с.

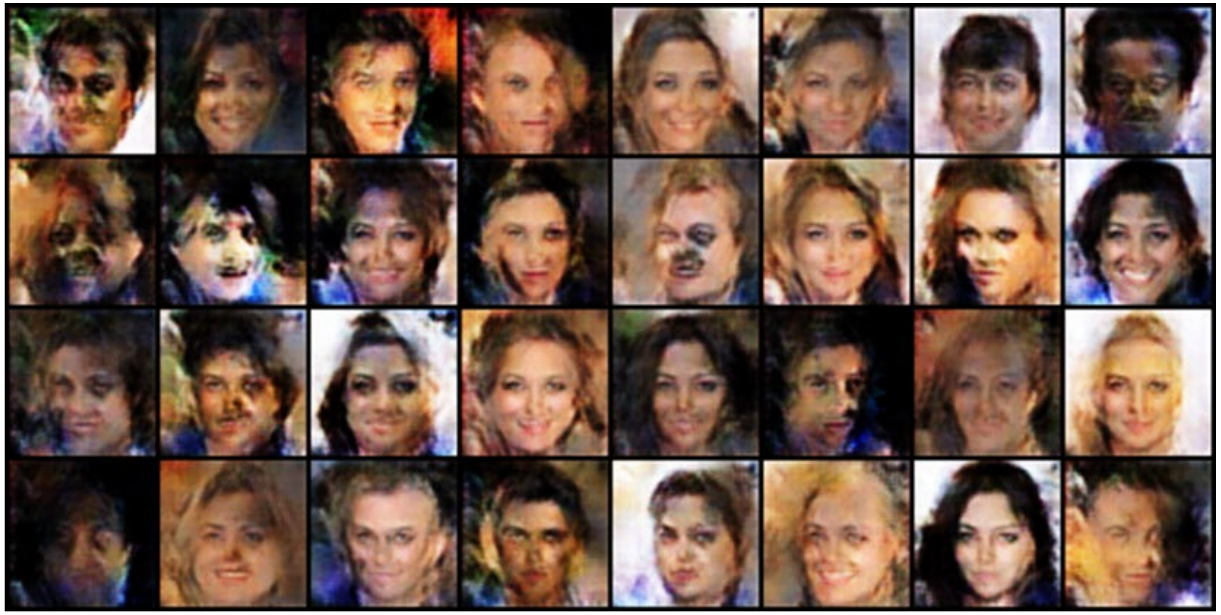
44 Комар М.П., Саченко А.О., Васильків Н.М. Методичні рекомендації до виконання кваліфікаційної роботи з освітньо-професійної програми «Комп’ютерні науки» спеціальності 122 «Комп’ютерні науки» за другим (магістерським) рівнем вищої освіти. Тернопіль: ЗУНУ, 2021. 32 с.

Додаток А

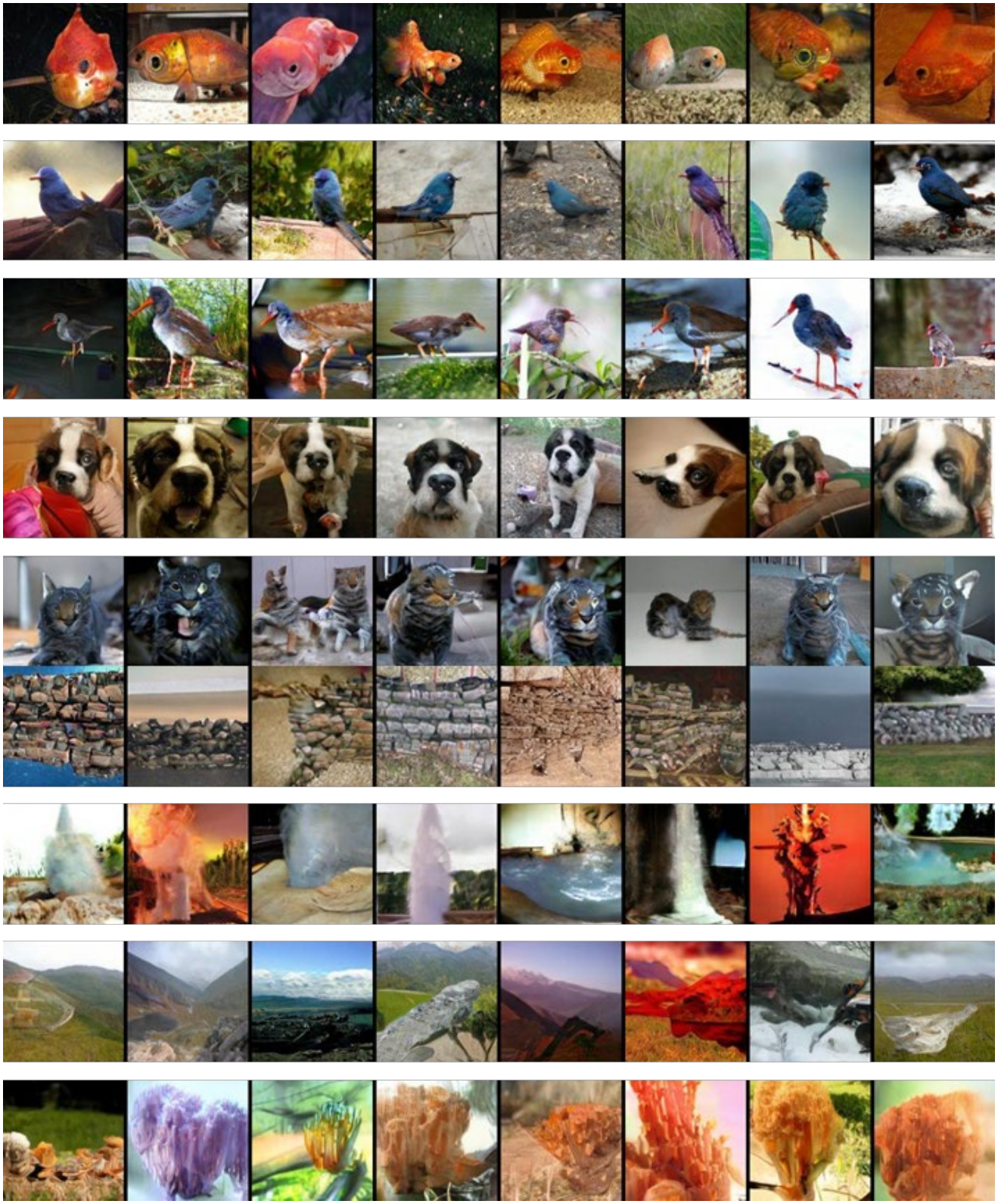
Приклади згенерованих зображеннь



Epoch 0.5



Epoch 8



Додаток Б

Код програмної реалізації

Файл sagan_models.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
from spectral import SpectralNorm
import numpy as np

class Self_Attn(nn.Module):
    """ Self attention Layer"""
    def __init__(self,in_dim,activation):
        super(Self_Attn,self).__init__()
        self.chanel_in = in_dim
        self.activation = activation

        self.query_conv = nn.Conv2d(in_channels = in_dim , out_channels = in_dim//8 , kernel_size= 1)
        self.key_conv = nn.Conv2d(in_channels = in_dim , out_channels = in_dim//8 , kernel_size= 1)
        self.value_conv = nn.Conv2d(in_channels = in_dim , out_channels = in_dim , kernel_size= 1)
        self.gamma = nn.Parameter(torch.zeros(1))

        self.softmax  = nn.Softmax(dim=-1) #
    def forward(self,x):
        """
            inputs :
```

```
            x : input feature maps( B X C X W X H)
        returns :
            out : self attention value + input feature
            attention: B X N X N (N is Width*Height)
        """
        m_batchsize,C,width ,height = x.size()
        proj_query  = self.query_conv(x).view(m_batchsize,-1,width*height).permute(0,2,1) # B X CX(N)
        proj_key =  self.key_conv(x).view(m_batchsize,-1,width*height) # B X C x (*W*H)
        energy =  torch.bmm(proj_query,proj_key) # transpose check
        attention = self.softmax(energy) # BX (N) X (N)
        proj_value = self.value_conv(x).view(m_batchsize,-1,width*height) # B X C X N

        out = torch.bmm(proj_value,attention.permute(0,2,1) )
        out = out.view(m_batchsize,C,width,height)

        out = self.gamma*out + x
        return out,attention

class Generator(nn.Module):
    """Generator."""

    def __init__(self, batch_size, image_size=64, z_dim=100, conv_dim=64):
        super(Generator, self).__init__()
        self.imsize = image_size
        layer1 = []
        layer2 = []
        layer3 = []
        last = []

        repeat_num = int(np.log2(self.imsize)) - 3
```

```
        mult = 2 ** repeat_num # 8
        layer1.append(SpectralNorm(nn.ConvTranspose2d(z_dim, conv_dim * mult, 4)))
        layer1.append(nn.BatchNorm2d(conv_dim * mult))
        layer1.append(nn.ReLU())

        curr_dim = conv_dim * mult

        layer2.append(SpectralNorm(nn.ConvTranspose2d(curr_dim, int(curr_dim / 2), 4,
2, 1)))
        layer2.append(nn.BatchNorm2d(int(curr_dim / 2)))
        layer2.append(nn.ReLU())

        curr_dim = int(curr_dim / 2)

        layer3.append(SpectralNorm(nn.ConvTranspose2d(curr_dim, int(curr_dim / 2), 4,
2, 1)))
        layer3.append(nn.BatchNorm2d(int(curr_dim / 2)))
        layer3.append(nn.ReLU())

        if self.imsize == 64:
            layer4 = []
            curr_dim = int(curr_dim / 2)
            layer4.append(SpectralNorm(nn.ConvTranspose2d(curr_dim, int(curr_dim / 2),
4, 2, 1)))
            layer4.append(nn.BatchNorm2d(int(curr_dim / 2)))
            layer4.append(nn.ReLU())
            self.l4 = nn.Sequential(*layer4)
            curr_dim = int(curr_dim / 2)

        self.l1 = nn.Sequential(*layer1)
        self.l2 = nn.Sequential(*layer2)
```

```
        self.l3 = nn.Sequential(*layer3)

        last.append(nn.ConvTranspose2d(curr_dim, 3, 4, 2, 1))
        last.append(nn.Tanh())
        self.last = nn.Sequential(*last)

        self.attn1 = Self_Attn( 128, 'relu')
        self.attn2 = Self_Attn( 64,  'relu')

    def forward(self, z):
        z = z.view(z.size(0), z.size(1), 1, 1)
        out=self.l1(z)
        out=self.l2(out)
        out=self.l3(out)
        out,p1 = self.attn1(out)
        out=self.l4(out)
        out,p2 = self.attn2(out)
        out=self.last(out)

        return out, p1, p2


class Discriminator(nn.Module):
    """Discriminator, Auxiliary Classifier."""

    def __init__(self, batch_size=64, image_size=64, conv_dim=64):
        super(Discriminator, self).__init__()
        self.imsize = image_size
        layer1 = []
        layer2 = []
        layer3 = []
```

```
last = []

layer1.append(SpectralNorm(nn.Conv2d(3, conv_dim, 4, 2, 1)))
layer1.append(nn.LeakyReLU(0.1))

curr_dim = conv_dim

layer2.append(SpectralNorm(nn.Conv2d(curr_dim, curr_dim * 2, 4, 2, 1)))
layer2.append(nn.LeakyReLU(0.1))
curr_dim = curr_dim * 2

layer3.append(SpectralNorm(nn.Conv2d(curr_dim, curr_dim * 2, 4, 2, 1)))
layer3.append(nn.LeakyReLU(0.1))
curr_dim = curr_dim * 2

if self.imsize == 64:
    layer4 = []
    layer4.append(SpectralNorm(nn.Conv2d(curr_dim, curr_dim * 2, 4, 2, 1)))
    layer4.append(nn.LeakyReLU(0.1))
    self.l4 = nn.Sequential(*layer4)
    curr_dim = curr_dim*2
self.l1 = nn.Sequential(*layer1)
self.l2 = nn.Sequential(*layer2)
self.l3 = nn.Sequential(*layer3)

last.append(nn.Conv2d(curr_dim, 1, 4))
self.last = nn.Sequential(*last)

self.attn1 = Self_Attn(256, 'relu')
self.attn2 = Self_Attn(512, 'relu')
```

```
    def forward(self, x):
        out = self.l1(x)
        out = self.l2(out)
        out = self.l3(out)
        out,p1 = self.attn1(out)
        out=self.l4(out)
        out,p2 = self.attn2(out)
        out=self.last(out)

        return out.squeeze(), p1, p2
```

Файл data_loader.py

```
import torch
import torchvision.datasets as dsets
from torchvision import transforms

class Data_Loader():
    def __init__(self, train, dataset, image_path, image_size, batch_size, shuf=True):
        self.dataset = dataset
        self.path = image_path
        self.imsize = image_size
        self.batch = batch_size
        self.shuf = shuf
        self.train = train

    def transform(self, resize, totensor, normalize, centercrop):
        options = []
        if centercrop:
            options.append(transforms.CenterCrop(160))
        if resize:
```

```
            options.append(transforms.Resize((self.imsize,self.imsize)))
        if totensor:
            options.append(transforms.ToTensor())
        if normalize:
            options.append(transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)))
        transform = transforms.Compose(options)
        return transform

    def load_lsun(self, classes='church_outdoor_train'):
        transforms = self.transform(True, True, True, False)
        dataset = dsets.LSUN(self.path, classes=[classes], transform=transforms)
        return dataset

    def load_celeb(self):
        transforms = self.transform(True, True, True, True)
        dataset = dsets.ImageFolder(self.path+'/CelebA', transform=transforms)
        return dataset

    def loader(self):
        if self.dataset == 'lsun':
            dataset = self.load_lsun()
        elif self.dataset == 'celeb':
            dataset = self.load_celeb()

        loader = torch.utils.data.DataLoader(dataset=dataset,
                                             batch_size=self.batch,
                                             shuffle=self.shuf,
                                             num_workers=2,
                                             drop_last=True)
        return loader
```

Файл trainer.py

```
import os
import time
import torch
import datetime

import torch.nn as nn
from torch.autograd import Variable
from torchvision.utils import save_image

from sagan_models import Generator, Discriminator
from utils import *

class Trainer(object):
    def __init__(self, data_loader, config):

        # Data loader
        self.data_loader = data_loader

        # exact model and loss
        self.model = config.model
        self.adv_loss = config.adv_loss

        # Model hyper-parameters
        self.imsize = config.imsize
        self.g_num = config.g_num
        self.z_dim = config.z_dim
        self.g_conv_dim = config.g_conv_dim
        self.d_conv_dim = config.d_conv_dim
        self.parallel = config.parallel

        self.lambda_gp = config.lambda_gp
```

```
self.total_step = config.total_step
self.d_iters = config.d_iters
self.batch_size = config.batch_size
self.num_workers = config.num_workers
self.g_lr = config.g_lr
self.d_lr = config.d_lr
self.lr_decay = config.lr_decay
self.beta1 = config.beta1
self.beta2 = config.beta2
self.pretrained_model = config.pretrained_model

self.dataset = config.dataset
self.use_tensorboard = config.use_tensorboard
self.image_path = config.image_path
self.log_path = config.log_path
self.model_save_path = config.model_save_path
self.sample_path = config.sample_path
self.log_step = config.log_step
self.sample_step = config.sample_step
self.model_save_step = config.model_save_step
self.version = config.version

# Path
self.log_path = os.path.join(config.log_path, self.version)
self.sample_path = os.path.join(config.sample_path, self.version)
self.model_save_path = os.path.join(config.model_save_path, self.version)

self.build_model()

if self.use_tensorboard:
    self.build_tensorboard()
```

```
    # Start with trained model
    if self.pretrained_model:
        self.load_pretrained_model()

def train(self):

    # Data iterator
    data_iter = iter(self.data_loader)
    step_per_epoch = len(self.data_loader)
    model_save_step = int(self.model_save_step * step_per_epoch)

    # Fixed input for debugging
    fixed_z = tensor2var(torch.randn(self.batch_size, self.z_dim))

    # Start with trained model
    if self.pretrained_model:
        start = self.pretrained_model + 1
    else:
        start = 0

    # Start time
    start_time = time.time()
    for step in range(start, self.total_step):

        # ================== Train D ================== #
        self.D.train()
        self.G.train()

        try:
            real_images, _ = next(data_iter)
```

```
except:
    data_iter = iter(self.data_loader)
    real_images, _ = next(data_iter)

# Compute loss with real images
# dr1, dr2, df1, df2, gf1, gf2 are attention scores
real_images = tensor2var(real_images)
d_out_real,dr1,dr2 = self.D(real_images)
if self.adv_loss == 'wgan-gp':
    d_loss_real = - torch.mean(d_out_real)
elif self.adv_loss == 'hinge':
    d_loss_real = torch.nn.ReLU()(1.0 - d_out_real).mean()

# apply Gumbel Softmax
z = tensor2var(torch.randn(real_images.size(0), self.z_dim))
fake_images,gf1,gf2 = self.G(z)
d_out_fake,df1,df2 = self.D(fake_images)

if self.adv_loss == 'wgan-gp':
    d_loss_fake = d_out_fake.mean()
elif self.adv_loss == 'hinge':
    d_loss_fake = torch.nn.ReLU()(1.0 + d_out_fake).mean()

# Backward + Optimize
d_loss = d_loss_real + d_loss_fake
self.reset_grad()
d_loss.backward()
self.d_optimizer.step()

if self.adv_loss == 'wgan-gp':
    # Compute gradient penalty
```

```
            alpha = torch.rand(real_images.size(0), 1, 1, 1).cuda().expand_as(real_images)
            interpolated = Variable(alpha * real_images.data + (1 - alpha) *
fake_images.data, requires_grad=True)
            out,_,_ = self.D(interpolated)

            grad = torch.autograd.grad(outputs=out,
                                       inputs=interpolated,
                                       grad_outputs=torch.ones(out.size()).cuda(),
                                       retain_graph=True,
                                       create_graph=True,
                                       only_inputs=True)[0]

            grad = grad.view(grad.size(0), -1)
            grad_l2norm = torch.sqrt(torch.sum(grad ** 2, dim=1))
            d_loss_gp = torch.mean((grad_l2norm - 1) ** 2)

            # Backward + Optimize
            d_loss = self.lambda_gp * d_loss_gp

            self.reset_grad()
            d_loss.backward()
            self.d_optimizer.step()

        # ==================== Train G and gumbel ==================== #
        # Create random noise
        z = tensor2var(torch.randn(real_images.size(0), self.z_dim))
        fake_images,_,_ = self.G(z)

        # Compute loss with fake images
        g_out_fake,_,_ = self.D(fake_images)  # batch x n
        if self.adv_loss == 'wgan-gp':
```

```
            g_loss_fake = - g_out_fake.mean()
        elif self.adv_loss == 'hinge':
            g_loss_fake = - g_out_fake.mean()

        self.reset_grad()
        g_loss_fake.backward()
        self.g_optimizer.step()

        # Print out log info
        if (step + 1) % self.log_step == 0:
            elapsed = time.time() - start_time
            elapsed = str(datetime.timedelta(seconds=elapsed))
            print("Elapsed [{}], G_step [{}/{}], D_step[{}/{}], d_out_real: {:.4f}, "
                  " ave_gamma_l3: {:.4f}, ave_gamma_l4: {:.4f}".
                  format(elapsed, step + 1, self.total_step, (step + 1),
                         self.total_step , d_loss_real.data[0],
                         self.G.attn1.gamma.mean().data[0],
self.G.attn2.gamma.mean().data[0] ))

        # Sample images
        if (step + 1) % self.sample_step == 0:
            fake_images,_,_ = self.G(fixed_z)
            save_image(denorm(fake_images.data),
                       os.path.join(self.sample_path, '{}_fake.png'.format(step + 1)))

        if (step+1) % model_save_step==0:
            torch.save(self.G.state_dict(),
                       os.path.join(self.model_save_path, '{}_G.pth'.format(step + 1)))
            torch.save(self.D.state_dict(),
                       os.path.join(self.model_save_path, '{}_D.pth'.format(step + 1)))
```

```
    def build_model(self):

        self.G = Generator(self.batch_size,self.imsize, self.z_dim, self.g_conv_dim).cuda()
        self.D = Discriminator(self.batch_size,self.imsize, self.d_conv_dim).cuda()
        if self.parallel:
            self.G = nn.DataParallel(self.G)
            self.D = nn.DataParallel(self.D)

        # Loss and optimizer
        # self.g_optimizer = torch.optim.Adam(self.G.parameters(), self.g_lr, [self.beta1, self.beta2])
        self.g_optimizer = torch.optim.Adam(filter(lambda p: p.requires_grad, self.G.parameters()), self.g_lr, [self.beta1, self.beta2])
        self.d_optimizer = torch.optim.Adam(filter(lambda p: p.requires_grad, self.D.parameters()), self.d_lr, [self.beta1, self.beta2])

        self.c_loss = torch.nn.CrossEntropyLoss()
        # print networks
        print(self.G)
        print(self.D)

    def build_tensorboard(self):
        from logger import Logger
        self.logger = Logger(self.log_path)

    def load_pretrained_model(self):
        self.G.load_state_dict(torch.load(os.path.join(
            self.model_save_path, '{}_G.pth'.format(self.pretrained_model))))
        self.D.load_state_dict(torch.load(os.path.join(
            self.model_save_path, '{}_D.pth'.format(self.pretrained_model))))
        print('loaded trained models (step: {})..!'.format(self.pretrained_model))
```

```
    def reset_grad(self):
        self.d_optimizer.zero_grad()
        self.g_optimizer.zero_grad()

    def save_sample(self, data_iter):
        real_images, _ = next(data_iter)
        save_image(denorm(real_images), os.path.join(self.sample_path, 'real.png'))
```

Код PGGAN

Файл trai.py

```
import os
import time
import numpy as np
import tensorflow as tf

import config
import tfutil
import dataset
import misc

#----------------------------------------------------------------------------
# Choose the size and contents of the image snapshot grids that are exported
# periodically during training.

def setup_snapshot_image_grid(G, training_set,
    size    = '1080p',      # '1080p' = to be viewed on 1080p display, '4k' = to be viewed on 4k display.
    layout  = 'random'):    # 'random' = grid contents are selected randomly, 'row_per_class' = each row corresponds to one class label.
```

```
    # Select size.
    gw = 1; gh = 1
    if size == '1080p':
        gw = np.clip(1920 // G.output_shape[3], 3, 32)
        gh = np.clip(1080 // G.output_shape[2], 2, 32)
    if size == '4k':
        gw = np.clip(3840 // G.output_shape[3], 7, 32)
        gh = np.clip(2160 // G.output_shape[2], 4, 32)

    # Fill in reals and labels.
    reals = np.zeros([gw * gh] + training_set.shape, dtype=training_set.dtype)
    labels = np.zeros([gw * gh, training_set.label_size], dtype=training_set.label_dtype)
    for idx in range(gw * gh):
        x = idx % gw; y = idx // gw
        while True:
            real, label = training_set.get_minibatch_np(1)
            if layout == 'row_per_class' and training_set.label_size > 0:
                if label[0, y % training_set.label_size] == 0.0:
                    continue
            reals[idx] = real[0]
            labels[idx] = label[0]
            break

    # Generate latents.
    latents = misc.random_latents(gw * gh, G)
    return (gw, gh), reals, labels, latents

#----------------------------------------------------------------------------
# Just-in-time processing of training images before feeding them to the networks.
```

```
def process_reals(x, lod, mirror_augment, drange_data, drange_net):
    with tf.name_scope('ProcessReals'):
        with tf.name_scope('DynamicRange'):
            x = tf.cast(x, tf.float32)
            x = misc.adjust_dynamic_range(x, drange_data, drange_net)
        if mirror_augment:
            with tf.name_scope('MirrorAugment'):
                s = tf.shape(x)
                mask = tf.random_uniform([s[0], 1, 1, 1], 0.0, 1.0)
                mask = tf.tile(mask, [1, s[1], s[2], s[3]])
                x = tf.where(mask < 0.5, x, tf.reverse(x, axis=[3]))
        with tf.name_scope('FadeLOD'): # Smooth crossfade between consecutive levels-of-detail.
            s = tf.shape(x)
            y = tf.reshape(x, [-1, s[1], s[2]//2, 2, s[3]//2, 2])
            y = tf.reduce_mean(y, axis=[3, 5], keepdims=True)
            y = tf.tile(y, [1, 1, 1, 2, 1, 2])
            y = tf.reshape(y, [-1, s[1], s[2], s[3]])
            x = tfutil.lerp(x, y, lod - tf.floor(lod))
        with tf.name_scope('UpscaleLOD'): # Upscale to match the expected input/output size of the networks.
            s = tf.shape(x)
            factor = tf.cast(2 ** tf.floor(lod), tf.int32)
            x = tf.reshape(x, [-1, s[1], s[2], 1, s[3], 1])
            x = tf.tile(x, [1, 1, 1, factor, 1, factor])
            x = tf.reshape(x, [-1, s[1], s[2] * factor, s[3] * factor])
        return x

#----------------------------------------------------------------------------
# Class for evaluating and storing the values of time-varying training parameters.
```

```
class TrainingSchedule:
    def __init__(
        self,
        cur_nimg,
        training_set,
        lod_initial_resolution  = 4,        # Image resolution used at the beginning.
        lod_training_kimg      = 600,     # Thousands of real images to show before doubling the resolution.
        lod_transition_kimg     = 600,      # Thousands of real images to show when fading in new layers.
        minibatch_base          = 16,       # Maximum minibatch size, divided evenly among GPUs.
        minibatch_dict          = {},       # Resolution-specific overrides.
        max_minibatch_per_gpu   = {},       # Resolution-specific maximum minibatch size per GPU.
        G_lrate_base            = 0.001,    # Learning rate for the generator.
        G_lrate_dict            = {},       # Resolution-specific overrides.
        D_lrate_base            = 0.001,    # Learning rate for the discriminator.
        D_lrate_dict            = {},       # Resolution-specific overrides.
        tick_kimg_base          = 160,      # Default interval of progress snapshots.
        tick_kimg_dict              = {4: 160, 8:140, 16:120, 32:100, 64:80, 128:60, 256:40, 512:20, 1024:10}): # Resolution-specific overrides.

        # Training phase.
        self.kimg = cur_nimg / 1000.0
        phase_dur = lod_training_kimg + lod_transition_kimg
        phase_idx = int(np.floor(self.kimg / phase_dur)) if phase_dur > 0 else 0
        phase_kimg = self.kimg - phase_idx * phase_dur

        # Level-of-detail and resolution.
        self.lod = training_set.resolution_log2
```

```
        self.lod -= np.floor(np.log2(lod_initial_resolution))
        self.lod -= phase_idx
        if lod_transition_kimg > 0:
            self.lod -= max(phase_kimg - lod_training_kimg, 0.0) / lod_transition_kimg
        self.lod = max(self.lod, 0.0)
        self.resolution = 2 ** (training_set.resolution_log2 - int(np.floor(self.lod)))

        # Minibatch size.
        self.minibatch = minibatch_dict.get(self.resolution, minibatch_base)
        self.minibatch -= self.minibatch % config.num_gpus
        if self.resolution in max_minibatch_per_gpu:
            self.minibatch = min(self.minibatch, max_minibatch_per_gpu[self.resolution] * config.num_gpus)

        # Other parameters.
        self.G_lrate = G_lrate_dict.get(self.resolution, G_lrate_base)
        self.D_lrate = D_lrate_dict.get(self.resolution, D_lrate_base)
        self.tick_kimg = tick_kimg_dict.get(self.resolution, tick_kimg_base)

#----------------------------------------------------------------------------
# Main training script.
# To run, comment/uncomment appropriate lines in config.py and launch train.py.

def train_progressive_gan(
    G_smoothing         = 0.999,    # Exponential running average of generator weights.
    D_repeats           = 1,        # How many times the discriminator is trained per G iteration.
    minibatch_repeats   = 4,        # Number of minibatches to run before adjusting training parameters.
    reset_opt_for_new_lod   = True,         # Reset optimizer internal state (e.g. Adam moments) when new layers are introduced?
```

```
    total_kimg              = 15000,        # Total length of the training, measured in thousands of real images.
    mirror_augment          = False,        # Enable mirror augment?
    drange_net              = [-1,1],       # Dynamic range used when feeding image data to the networks.
    image_snapshot_ticks    = 1,            # How often to export image snapshots?
    network_snapshot_ticks  = 10,           # How often to export network snapshots?
    save_tf_graph           = False,        # Include full TensorFlow computation graph in the tfevents file?
    save_weight_histograms  = False,        # Include weight histograms in the tfevents file?
    resume_run_id           = None,         # Run ID or network pkl to resume training from, None = start from scratch.
    resume_snapshot         = None,         # Snapshot index to resume training from, None = autodetect.
    resume_kimg             = 0.0,          # Assumed training progress at the beginning. Affects reporting and training schedule.
    resume_time             = 0.0):         # Assumed wallclock time at the beginning. Affects reporting.

    maintenance_start_time = time.time()
    training_set = dataset.load_dataset(data_dir=config.data_dir, verbose=True, **config.dataset)

    # Construct networks.
    with tf.device('/gpu:0'):
        if resume_run_id is not None:
            network_pkl = misc.locate_network_pkl(resume_run_id, resume_snapshot)
            print('Loading networks from "%s"...' % network_pkl)
            G, D, Gs = misc.load_pkl(network_pkl)
        else:
            print('Constructing networks...')
```

```
        G = tfutil.Network('G', num_channels=training_set.shape[0], resolution=training_set.shape[1], label_size=training_set.label_size, **config.G)
        D = tfutil.Network('D', num_channels=training_set.shape[0], resolution=training_set.shape[1], label_size=training_set.label_size, **config.D)
        Gs = G.clone('Gs')
    Gs_update_op = Gs.setup_as_moving_average_of(G, beta=G_smoothing)
  G.print_layers(); D.print_layers()

  print('Building TensorFlow graph...')
  with tf.name_scope('Inputs'):
    lod_in          = tf.placeholder(tf.float32, name='lod_in', shape=[])
    lrate_in        = tf.placeholder(tf.float32, name='lrate_in', shape=[])
    minibatch_in    = tf.placeholder(tf.int32, name='minibatch_in', shape=[])
    minibatch_split = minibatch_in // config.num_gpus
    reals, labels   = training_set.get_minibatch_tf()
    reals_split     = tf.split(reals, config.num_gpus)
    labels_split    = tf.split(labels, config.num_gpus)
  G_opt = tfutil.Optimizer(name='TrainG', learning_rate=lrate_in, **config.G_opt)
  D_opt = tfutil.Optimizer(name='TrainD', learning_rate=lrate_in, **config.D_opt)
  for gpu in range(config.num_gpus):
    with tf.name_scope('GPU%d' % gpu), tf.device('/gpu:%d' % gpu):
      G_gpu = G if gpu == 0 else G.clone(G.name + '_shadow')
      D_gpu = D if gpu == 0 else D.clone(D.name + '_shadow')
      lod_assign_ops = [tf.assign(G_gpu.find_var('lod'), lod_in), tf.assign(D_gpu.find_var('lod'), lod_in)]
      reals_gpu = process_reals(reals_split[gpu], lod_in, mirror_augment, training_set.dynamic_range, drange_net)
      labels_gpu = labels_split[gpu]
      with tf.name_scope('G_loss'), tf.control_dependencies(lod_assign_ops):
        G_loss = tfutil.call_func_by_name(G=G_gpu, D=D_gpu, opt=G_opt, training_set=training_set, minibatch_size=minibatch_split, **config.G_loss)
```

```
            with tf.name_scope('D_loss'), tf.control_dependencies(lod_assign_ops):
                D_loss = tfutil.call_func_by_name(G=G_gpu, D=D_gpu, opt=D_opt, training_set=training_set, minibatch_size=minibatch_split, reals=reals_gpu, labels=labels_gpu, **config.D_loss)
            G_opt.register_gradients(tf.reduce_mean(G_loss), G_gpu.trainables)
            D_opt.register_gradients(tf.reduce_mean(D_loss), D_gpu.trainables)
    G_train_op = G_opt.apply_updates()
    D_train_op = D_opt.apply_updates()

    print('Setting up snapshot image grid...')
    grid_size, grid_reals, grid_labels, grid_latents = setup_snapshot_image_grid(G, training_set, **config.grid)
    sched = TrainingSchedule(total_kimg * 1000, training_set, **config.sched)
    grid_fakes = Gs.run(grid_latents, grid_labels, minibatch_size=sched.minibatch//config.num_gpus)

    print('Setting up result dir...')
    result_subdir = misc.create_result_subdir(config.result_dir, config.desc)
    misc.save_image_grid(grid_reals, os.path.join(result_subdir, 'reals.png'), drange=training_set.dynamic_range, grid_size=grid_size)
    misc.save_image_grid(grid_fakes, os.path.join(result_subdir, 'fakes%06d.png' % 0), drange=drange_net, grid_size=grid_size)
    summary_log = tf.summary.FileWriter(result_subdir)
    if save_tf_graph:
        summary_log.add_graph(tf.get_default_graph())
    if save_weight_histograms:
        G.setup_weight_histograms(); D.setup_weight_histograms()

    print('Training...')
    cur_nimg = int(resume_kimg * 1000)
    cur_tick = 0
```

```
    tick_start_nimg = cur_nimg
    tick_start_time = time.time()
    train_start_time = tick_start_time - resume_time
    prev_lod = -1.0
    while cur_nimg < total_kimg * 1000:

        # Choose training parameters and configure training ops.
        sched = TrainingSchedule(cur_nimg, training_set, **config.sched)
        training_set.configure(sched.minibatch, sched.lod)
        if reset_opt_for_new_lod:
            if np.floor(sched.lod) != np.floor(prev_lod) or np.ceil(sched.lod) != np.ceil(prev_lod):
                G_opt.reset_optimizer_state(); D_opt.reset_optimizer_state()
        prev_lod = sched.lod

        # Run training ops.
        for repeat in range(minibatch_repeats):
            for _ in range(D_repeats):
                tfutil.run([D_train_op, Gs_update_op], {lod_in: sched.lod, lrate_in: sched.D_lrate, minibatch_in: sched.minibatch})
                cur_nimg += sched.minibatch
            tfutil.run([G_train_op], {lod_in: sched.lod, lrate_in: sched.G_lrate, minibatch_in: sched.minibatch})

        # Perform maintenance tasks once per tick.
        done = (cur_nimg >= total_kimg * 1000)
        if cur_nimg >= tick_start_nimg + sched.tick_kimg * 1000 or done:
            cur_tick += 1
            cur_time = time.time()
            tick_kimg = (cur_nimg - tick_start_nimg) / 1000.0
            tick_start_nimg = cur_nimg
```

```
        tick_time = cur_time - tick_start_time
        total_time = cur_time - train_start_time
        maintenance_time = tick_start_time - maintenance_start_time
        maintenance_start_time = cur_time

        # Report progress.
        print('tick %-5d kimg %-8.1f lod %-5.2f minibatch %-4d time %-12s sec/tick %-7.1f sec/kimg %-7.2f maintenance %.1f' % (
            tfutil.autosummary('Progress/tick', cur_tick),
            tfutil.autosummary('Progress/kimg', cur_nimg / 1000.0),
            tfutil.autosummary('Progress/lod', sched.lod),
            tfutil.autosummary('Progress/minibatch', sched.minibatch),
            misc.format_time(tfutil.autosummary('Timing/total_sec', total_time)),
            tfutil.autosummary('Timing/sec_per_tick', tick_time),
            tfutil.autosummary('Timing/sec_per_kimg', tick_time / tick_kimg),
            tfutil.autosummary('Timing/maintenance_sec', maintenance_time)))
        tfutil.autosummary('Timing/total_hours', total_time / (60.0 * 60.0))
        tfutil.autosummary('Timing/total_days', total_time / (24.0 * 60.0 * 60.0))
        tfutil.save_summaries(summary_log, cur_nimg)

        # Save snapshots.
        if cur_tick % image_snapshot_ticks == 0 or done:
            grid_fakes = Gs.run(grid_latents, grid_labels, minibatch_size=sched.minibatch//config.num_gpus)
            misc.save_image_grid(grid_fakes, os.path.join(result_subdir, 'fakes%06d.png' % (cur_nimg // 1000)), drange=drange_net, grid_size=grid_size)
        if cur_tick % network_snapshot_ticks == 0 or done:
            misc.save_pkl((G, D, Gs), os.path.join(result_subdir, 'network-snapshot-%06d.pkl' % (cur_nimg // 1000)))

        # Record start time of the next tick.
```

```
            tick_start_time = time.time()

    # Write final results.
    misc.save_pkl((G, D, Gs), os.path.join(result_subdir, 'network-final.pkl'))
    summary_log.close()
    open(os.path.join(result_subdir, '_training-done.txt'), 'wt').close()

#----------------------------------------------------------------------------
# Main entry point.
# Calls the function indicated in config.py.
if __name__ == "__main__":
    misc.init_output_logging()
    np.random.seed(config.random_seed)
    print('Initializing TensorFlow...')
    os.environ.update(config.env)
    tfutil.init_tf(config.tf_config)
    print('Running %s()...' % config.train['func'])
    tfutil.call_func_by_name(**config.train)
    print('Exiting...')
#----------------------------------------------------------------------------
```

Додаток В

Апробація отриманих результатів

Панчак Дмитро Вікторович

магістр спеціальності 122 “Комп’ютерні науки”

Західноукраїнський національний університет

Метод генерування зображень за допомогою GAN-мереж

Алгоритми GAN виникли в 2014 році [1] і з тих пір були виділені як потенційні альтернативи для збільшення даних і відсутніх проблем з даними, серед іншого, завдяки їх видатним можливостям генерувати реалістичні екземпляри даних, особливо зображення.

На сьогоднішній день багато дослідників продовжують вивчати свої можливості і розширювати свої області потенціального застосування, що дає позитивний погляд на цю технологію. Зокрема, до цього проєкту призвело порушене питання, в якому йдеться про доцільність використання систем GAN для генерації підроблених даних, не обов'язково зображень, що імітують атрибути приватного набору даних. Якщо можливо, ця згенерована машина буде дуже корисним інструментом, оскільки вона дозволить необмежену кількість подібних до вихідних даних без шкоди для конфіденційності оригінальних елементів, уникаючи будь-якого потенційного ризику витоку конфіденційної інформації. Застосування цього інструменту, як це буде видно пізніше, може варіюватися від освітніх цілей до наукових симуляцій та досліджень, оскільки конфіденційні дані з будь-якої галузі можуть бути доступні без ризику витоку приватних даних

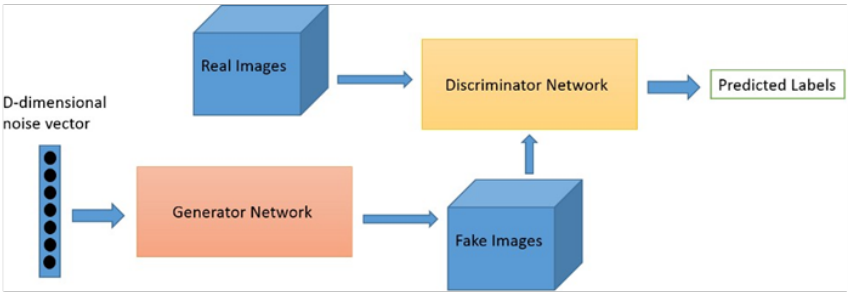


Рис. 1: Базова архітектура GAN.

Для того щоб відповісти на питання, виставлене в попередніх абзацах, в першу чергу необхідно було б зрозуміти, як працює GAN. Далі буде здійснено ретельне перерахування загальних і конкретних цілей проєкту.

Генеративні змагальні мережі (GAN) - це системи, засновані на стратегії min-

max, де стикаються два алгоритми: один алгоритм генерує дані (генератор), а інший дискримінує підроблені та реальні дані (дискримінатор)

Мета генератора полягає в тому, щоб максимізувати помилку дискримінатора, тоді як дискримінатор хоче її мінімізувати. Це ітераційний процес, який закінчується, коли помилка дискримінатора становить 0,5, що означає, що вона не вдається 50% разів, базова помилка в бікласифікації. Ми можемо думати про GAN як про «Гру в кішки та мишки» між поліцейським та грошовим фальшивомонетником [2], де фальшивомонетник (генератор) намагається обдурити поліцейського (дискримінатора), створюючи нескінченну петлю, де обидва гравці продовжують вдосконалюватися чистою конкуренцією. На рисунку 1 представлена базова система GAN.

Як описано, для того, щоб генерувати фальшиве зображення, нам завжди потрібне джерело «творчості», яке в даному випадку походить від випадкового вектора шуму (seed). З іншого боку, для того, щоб мати можливість розрізняти реальні та підроблені зображення (щоб модель дискримінатора могла надсилати до моделі генератора, що робить не так), необхідна база даних реальних зображень.

Отже, цільовою функцією повної мережі є наступне:

$$\min_G \max_D V(D,G) = E_{x \sim P_{data}(x)}[\log D(x)] + E_{z \sim P_z(z)}[\log(1 - D(G(z)))] \qquad (1.1)$$

Цей вираз представляє значення (V), яке є функцією як дискримінатора D, так і генератора G. Мета полягає в тому, щоб максимізувати втрати дискримінатора (D) і мінімізувати втрати генератора (G). Значення V - це сума очікуваної ймовірності журналу для реальних і згенерованих даних. Імовірності (ймовірності) є дискримінаційними виходами для реальних або породжених ім-віків. Зверніть увагу, що вихід дискримінатора для згенерованого зображення віднімається від 1 перед тим, як взяти журнал. Максимізація отриманих значень призводить до оптимізації параметрів дискримінатора таким чином, щоб він навчився правильно ідентифікувати як реальні, так і підроблені дані.

Також необхідно пояснити, що:

1. P_{data}: представляє розподіл реальних даних.

P_z: Представляє розподіл шуму (зазвичай це розподіл Гауса), з якого ми

1. можемо створити підроблене зображення.
2. x і z: представляють зразки з кожного відповідного простору.
3. E_x та E_z: Представляють очікувану ймовірність журналу з різних виходів як реальних, так і згенерованих зображень.
4. *Функція D* виводить дійсне число, коливається між 0 і 1, що представляє імовірність для даних, що є реальними (1) або підробленими (0). З іншого боку, функція G виводить згенерований зразок або екземпляр.

Крім того, для того, щоб навчити генератора та дискримінатора, помилки на їх виходах поширюються назад у моделі. Ці помилки поширюються як градієнти наступних функцій втрати.

Правило оновлення для дискримінатора:

$$\nabla_{\theta_d}\frac{1}{m}\sum_{i=1}^{m}[\log D(x^{(i)}) + \log(1 - D(G(z^{(i)})))] \qquad (1.2)$$

Правило оновлення для генератора

$$\nabla_{\theta_g}\frac{1}{m}\sum_{i=1}^{m}\log(1 - D(G(z^{(i)}))) \qquad (1.3)$$

де m представляє загальну кількість зразків, протестованих в партії перед оновленням обох моделей, а θ_d і θ_g - ваги кожної моделі. Варто зазначити, що в наступних розділах цього проекту ми будемо посилатися на ці помилки, які ми розраховуємо градієнт як *втрати*.

Оскільки GAN - це всього лише системна структура, вибір елементів для складання цієї системи (генератора і дискримінатора) залежить від користувача. У цьому випадку буде використовуватися найпопулярніший варіант: використання згорткової нейронної мережі (CNN) для дискримінатора і транспонованого CNN для генератора. Для того, щоб цей проект був зосереджений на структурі GAN, CNN не будуть детально пояснюватися.

Тепер, коли було описано призначення, елементи та цільову функцію, будуть перераховані етапи тренувального процесу. Кожна петля цього тренувального процесу називається епохою. Усередині цієї епохи всі реальні зразки, як правило, доступні зображення, будуть оброблятися партіями: кожна партія є випадковою підмножиною фіксованого розміру (називається розміром

партії) з набору даних реальних зразків. Після кожної партії помилки від невідповідності між виходами дискримінатора та генератора та їх очікуваними значеннями знову поширюються на моделі і, отже, використовуються для навчання моделей. Кроки всередині кожної партії:

1. Пакетні реальні образи підживлюють їх у дискримінатор.
2. Помилки з виводу дискримінатора розраховуються, знаючи, що в ідеалі ці виходи повинні бути всі 1, оскільки всі ці зображення реальні.
3. Пакетні підроблені зображення генеруються з пакетними векторами шуму.
4. Пакетні підроблені зображення подаються в дискримінатор.
5. Помилки з виводу дискримінатора розраховуються, знаючи, що в ідеалі ці виходи повинні бути всі 0, оскільки всі ці зображення є підробленими.
6. Обидві помилки, розраховані у дискримінатора, тепер поширюються на його модель.
7. Кроки 3 і 4 повторюються з новими векторами шуму.
8. Цього разу помилки розраховуються для генератора, знаючи, що в ідеалі всі виходи з дискримінатора повинні бути 1, оскільки ці зображення повинні ідеально імітувати реальні .
9. Ці помилки тепер поширюються на модель генератора.
10. Почніть знову з чергової партії з цієї епохи.

Як буде показано в наступному розділі, ці системи можуть збільшити свою складність, додавши нові елементи і функції в основну структуру. Найбільш поширеним додатковим елементом енкодер (зазвичай інший CNN) для живлення генератора не випадковим вектором шуму, а іншим джерелом інформації, що дозволяє системі генерувати реальні зображення, наприклад, з описового тексту або іншого зображення.

Ще одним фактором, який додає складності системі, є необхідність тонкої настройки конфігурації для різних елементів GAN з метою полегшення процесу її навчання та оптимізації.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-

Панчак Дмитро Вікторович, Студент
Західноукраїнський національний університет, м. Тернопіль

РОЗРОБКА ПІДХОДУ ГЕНЕРУВАННЯ ЗОБРАЖЕННЯ НА ОСНОВІ SAGAN

Однією з найпоширеніших проблем навчання є «mode collapse». Результатом такої проблеми є те, що генератор завжди синтезує однакові, або майже ідентичні зображення. Це відбувається, зокрема, коли дискримінатор запізнюється з навчанням. В цьому випадку генератор знаходить якесь оптимальне зображення, яке завжди обманює дискримінатора. В результаті, незалежно від вхідного вектора шуму z, генератор буде синтезувати одне і те ж зображення. У зв'язку з цим всі останні дослідження сходяться на думці, що дискримінатора потрібно навчати швидше, ніж генератора. Це інтуїтивно зрозуміло, оскільки мережу розпізнавання спочатку потрібно навчити деяким шаблонам, перш ніж попросити розпізнати згенеровані зображення. Це міркування призвело до введення правила TTUR (two time-scale update). У статті Martin Heusel наводяться докази впливу такого підходу на зближення до точки рівноваги по Не-шу мінімакс гри дискримінатора і генератора

В основі мережі лежать три основні модулі – модуль «self-attention», модуль-генератор і модуль-дискримінатор. «self-attention» включається в якості додаткового шару, як в генераторі, так і в дискримінаторі. Самі модулі дискримінатора і генератора побудовані на основі глибоких згорткових мереж з тією різницею, що генератор використовує зворотні згорткові шари. Аналогіч-ним чином для підвищення стійкості тренування, після кожного згортувального шару використовувався шар пакетної нормалізації.

Архітектура мережі зображена нижче на рисунку 1

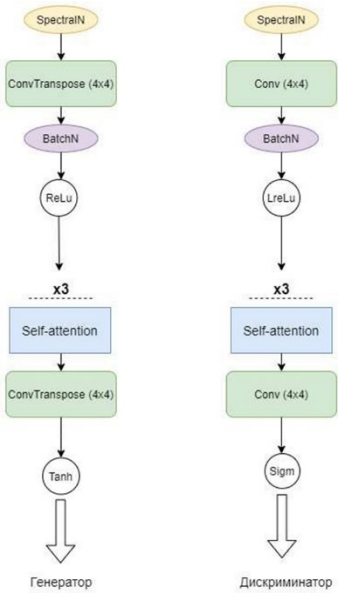


Рис. 1 Схема мережі в реалізації SAGAN

Як видно з малюнка, мережі практично симетричні.

Додаткові механізми оптимізації:

1. Згладжування міток при розрахунку значення помилки нав-чання (наприклад, замість 1 буде використовуватися випадкове число з сегмента [0.8,1]), що мінімізує обнулення градієнта для генератора, тоб-то стабілізує процес навчання. У багатьох роботах радять використову-вати згладжування з одного боку.

2. Замість «pooling» шарів використовувалися конвуляційні з зрушенням (Alec Radford, 2016)

3. Була використана нормалізація партіями для генератора (такий підхід характерний для мережі SAGAN.), а також функція активації Relu для генератора і LeakyRelu для дискримінатора.

4. Використання оптимізатора Adam.

Література

1. A. P. Parikh, O. Täckström, D. Das, and J. Uszkoreit, “A decomposable attention model for natural language inference,” in EMNLP, 2016. 51, 52

2. J. Cheng, L. Dong, and M. Lapata, “Long short-term memory-networks for machine reading,” in EMNLP, 2016.