ВСТУП

Бакалаврська робота присвячена аналізу та оптимізації процесів розробки дизайну монограм.

Головним завданням є розробка нейромережі і навчання її взаємодії з даними для зміни і надбудови з огляду на можливості генерування зображень. Кращим рішенням архітектури такої нейромережі - перцептрон який дозволяє створити набір «асоціацій» між вхідними стимулами та необхідною реакцією на виході. В біологічному плані це відповідає перетворенню, наприклад, зорової інформації у фізіологічну відповідь рухових нейронів, що і буде застосовано для рішення моєї задачі.

Об’єктом дослідження даної роботи є методи взаємодії безпосередньо нейромережі та дизайну монограм.

Предметом дослідження виступають можливості покращення сприйняття візуального, а також досвіду, використовуючи можливості штучного інтелекту.

Мета роботи передбачає розробку системи, що буде проводити генерацію зображення за допомогою технології GAN.

Поставлена мета вимагає рішення низки завдань:

- Аналіз існуючих рішень в дизайні;
- Аналіз популярних способів створення дизайну;
- Пошук методів створення монограм;
- Пошук методів взаємодії штучного інтелекту з дизайном;
- Розробка системи підтримки прийняття рішення;
- Провести порівняльний аналіз.

Практична значущість результатів дослідження може полягати в можливості подальшого розвитку сфери дизайну логотипів.

1. Аналіз рішень та методологій у сфері дизайну

1.1 Аналіз сфери створення логотипів

На просторах мережі можна наштовхнутися на думки про повне зникнення дизайнерів завдяки нейромережам, моя думка не суміжна з авторами таких статей.

Для того щоб нейромережа повністю замінила творчість, потрібно навчити її повноцінному мисленні, виходячи за рамки, але на даний момент вона може виключно генерувати, а не створювати повністю, художник / дизайнер представляє картину, а нейромережа повинна розуміти звідки їй брати - первинне джерело генерування дизайну.

У такому випадку досить одного варіанту, щоб отримати результат який цілком перевищує очікування. З інтеграцією нейромереж і штучного інтелекту дизайнери можуть виступати не як творці, а як куратори творчого процесу. Так, наприклад, давно на службу дизайнерам поставлений Autodesk Dreamcatcher, який використовує алгоритмічні методи, щоб створювати різні абстрактні дизайни.

Розроблена технологія генерації все ще працює місцями з помилками, дуже часто в дизайні деяких моделей можна спостерігати порожнє місце і не стикування, так само часто варіанти запропоновані системою можуть бути перекручені в виробництві, в такому випадку потрібен оператор який контролює роботу нейромережі.

«Ідея повністю замінити дизайнера алгоритмом звучить футуристично, але сутність не в цьому. Дизайнери продуктів допомагають втілити ідею сирого продукту в добре продуманий інтерфейс з твердими принципами взаємодії, продуманої інформаційної архітектурою і візуальним стилем, допомагаючи компанії досягти своїх бізнес-цілей і зміцнити свій бренд»- Джон Кідл.

1.2 Актуальність генеративних технологій у дизайні

Вперше сама модель технологій була запропонована в статті NeurIPS 2014 року Яном Гудфеллоу і його колегами.

Перспектива глибокого навчання полягає у відкритті багатих ієрархічних моделей, які представляють розподіл ймовірностей щодо видів даних, що зустрічаються в програмах штучного інтелекту, таких як природні зображення, звукові форми, що містять мову, та символи в корпусах природної мови.

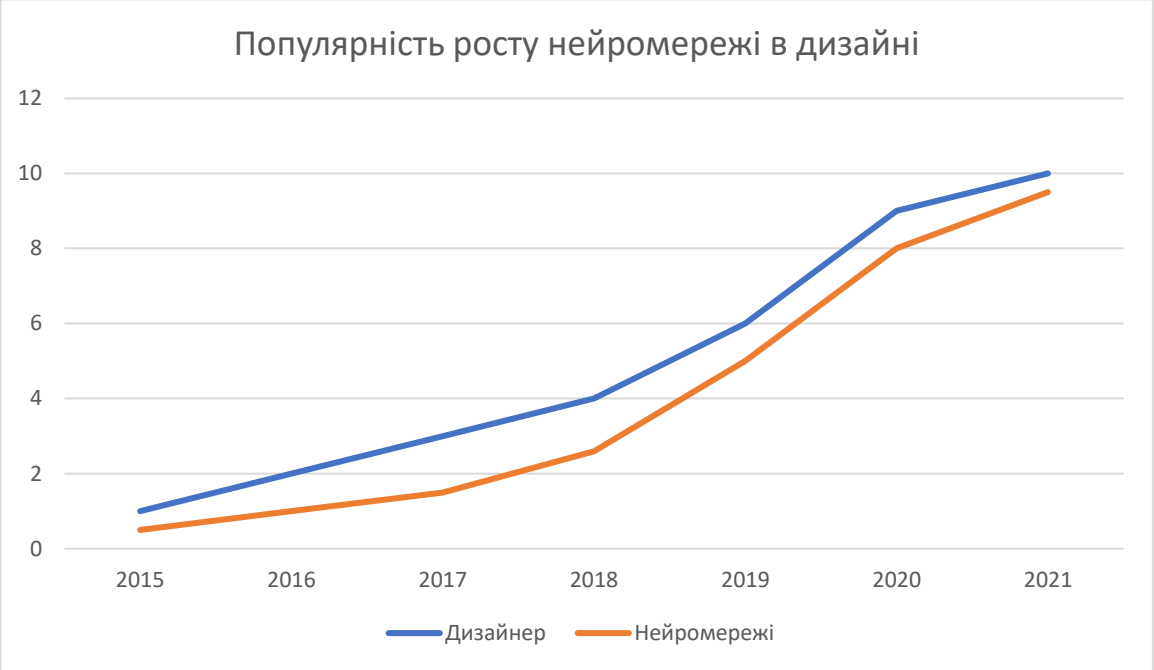


Рисунок 1.1 – Діаграма популярності нейромережі

Згідно діаграми с початку 2017 року, коли з'явилася перша програма The Grid, вона вміє генерувати модулі сайту, після чого багато людей подумали про вимирання професії дизайнера, але не дивлячись на це багато хто знайшов можливість керувати технологією, експериментувати і відкривати для себе нові напрямки.

1.3 Вплив нейромережевих технологій на створення дизайну

Більшість нейромереж не зможуть замінити дизайнера, у якого виникає первісна думка про подання картинки, а самій нейромережі потрібні вхідні дані для роботи, якщо говорити про GAN тип нейромереж, то вони як правило працюють з генератором і дискримінатором. Саме через це наразі виникає велика кількість в потребі співпраці між дизайнером та технологіями.

Цікава думка провідних спеціалістів:

Якщо ж говорити про впровадження в процес розробки дизайну нейромереж - це дасть можливість нескінченного вибору. Так, багато дизайнерів-одинаки або дизайн-студії надають вибір замовнику, але мова йде всього лише про декілька варіантів. Нейромережа може згенерувати нескінченно-можливу кількість варіацій на підставі закладених параметрів.

<u>У майбутньому з'являться:</u>

Дизайнери, які не використовують нейромережі, які стануть в галузі унікальними фахівцями, так званими «майстрами ручної роботи»;

Деяка категорія дизайнерів, ведуча розробку з використанням запозичених нейромереж в якості інструменту або ПЗ;

Дизайнери, які почнуть створювати свої власні нейромережі, здатні генерувати дизайн з індивідуальним стилем.

- Світлана Феллоу.

На сьогодні я розглядаю сервіси нейромереж як величезну допомогу з генерацією ідей і підтримку у виконанні рутинної роботи. Нейромережа може накидати безліч варіантів дизайну, але вибирати і задавати тон - саме людина. У мережі (поки) немає волі і поняття контексту соціального простору, контексту особливості часу, поняття специфіки замовника.

- Алла Сэлмон.

Більшість сервісів зараз пропонують робити повністю роботу за клієнта або ж дизайнера який до них звертається, слід вважати що краще дати можливість генерації варіантів з вже готової інформації, у роботі розглянуто навчання нейромережі за допомогою даних рукописного введення EMNIST.

	Дизайнер	Нейромережа
Час	У прийнятті рішень, дизайнеру потрібна велика кількість часу.	Залежить від потужності комплектуючих ПК на якому працює нейромережу, але зараз є багато способів зменшення часу на роботу.
Кошторис	Дизайнери бувають різні, але їх робота набагато дорожче, так як кожен варіант дизайну буде враховуватися в вартість.	Вартість технології може окупитися, не дивлячись на вартість.
Варіативність	Дизайнер може запропонувати в роботі певну кількість варіантів, можливості дизайнера обмежені людськими факторами.	У нейромережі немає обмежень в варіативності, оскільки вони добре справляються з великим потоком генерації даних.
Якість	Якісніше роботу робить безпосередньо дизайнер, можливість у процесі роботи змінювати сам дизайн.	Технології навчання нейромереж з легкістю вийшли на рівень якісної обробки інформації і можуть видавати зображення з високою роздільною здатністю, але після тривалого навчання.

Таблиця 1.1 порівняння Дизайнера і нейромережі

2. Використані технології

Нейромережі, які вміють генерувати зображення, музику, текст, аиеористовую генеративно-асоціативні технологі. GAN складається з двох нейромереж, одна з них генерує дані, інша порівнює їх з реальними, є приклад хороших результатів і досліджень такі як:

CycleGAN – переробка одного зображення відповідно до стилю інших зображень.

У нейромережі задіяно 2 генератора (G і F) і 2 дискриминатора (X і Y).

- Генератор G вчиться перетворювати зображення X в зображення Y.
- Генератор F вчиться перетворювати зображення Y в зображення X.
- Дискримінатор Dx передає зображення X і згенероване зображення X (F (Y)).
- Дискримінатор Dy створює відмінне зображення Y і згенероване зображення Y (G (X)).

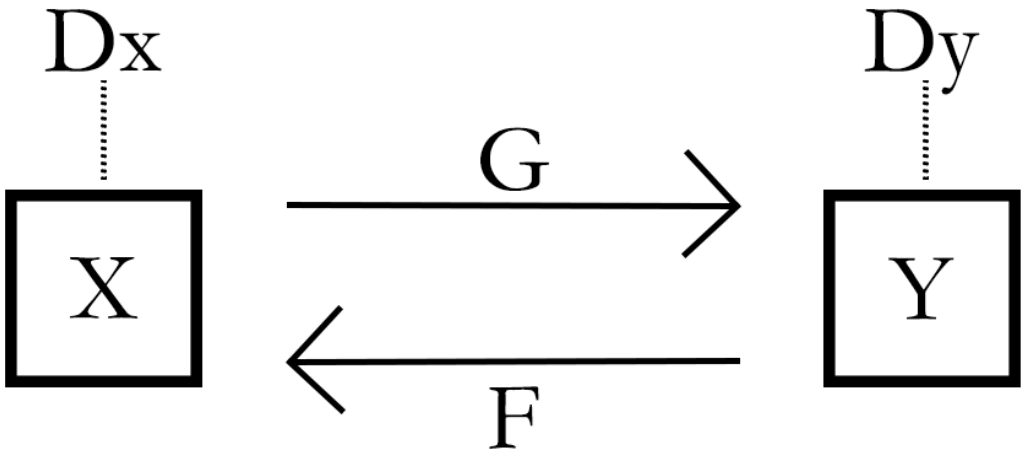


Рисунок 2.1 - Алгоритм роботи CycleGAN

CycleGAN використовує втрату послідовності циклу, щоб забезпечити навчання без дублювання даних. Іншими словами, його можна переміщати з одного домену в інший без однозначного зіставлення між вихідним та цільовим доменами.

StyleGAN

StyleGAN намагається відокремити атрибути зображень високого рівня такі як положення обличчя від випадкових факторів, наприклад: зачіска, родимки, веснянки.

Новий метод запропонований Nvidia, архітектура генератора перероблена таким чином, що відкриває новий спосіб управління процесом створення зображення. Генератор StyleGAN постійний з початку дослідження і заспокоює стиль кожного згорткового шару зображення.

Ціль архітектури віднайти приховані фактори варіацій, за для більш можливого контролю.

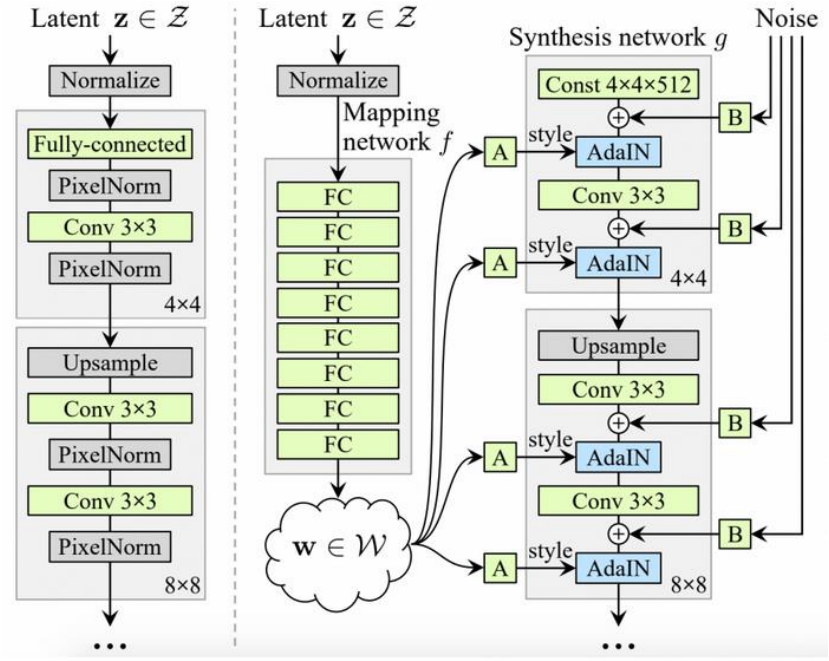


Рисунок 2.2 – Алгоритм роботи StyleGAN

Архітектура базується на ідеї розміщення вхідного секретного коду для доступу до прихованого проміжного, що може мати значний вплив на те, як виражаються змінні. Прихований простір посередині не підпадає під такі обмеження, тому його можна знову відкрити.

Ця операція виконується з використанням нелінійного перетворення, що призводить до модифікованого прихованого вектора. Модифікований вектор використовує афінне перетворення для більшої адаптації до різних стилів.

Аффінне перетворення - рівень або простір, в собі, при якому сторони стають паралельними лініями, паралельними перетинам, перетини в перетинаючі.

Можливості афінних перетворень

Такі методи використвоують для швидких маніпуляцій з перетвореннями зображень та їх векторів як у двомірному, так і тривимірному просторі.

- Поворот осі

  Початкова матриця: $\begin{pmatrix} cos(\alpha) & sin(\alpha) \\ -sin(\alpha) & cos(\alpha) \end{pmatrix}$

  Перетворюється в: $\begin{cases} x' = xcos\alpha + ysin\alpha \\ y' = -xsin\alpha + ycos\alpha \end{cases}$

- Розтягнення осі

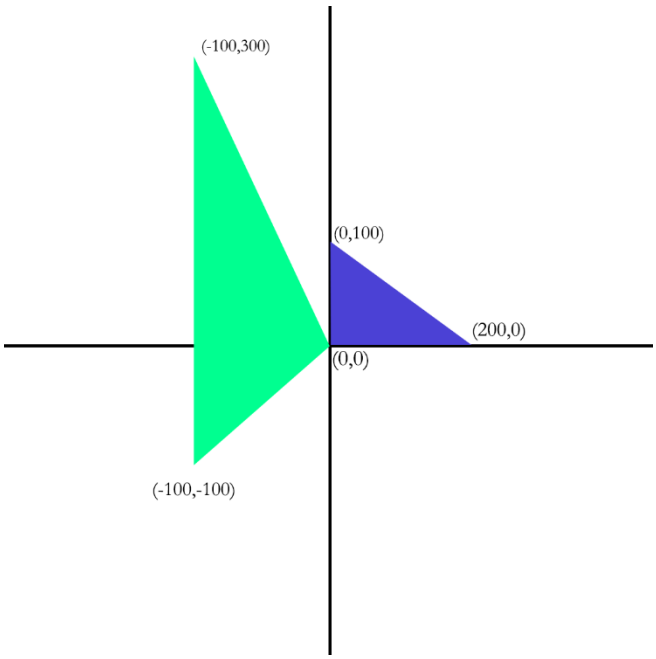
  Початкова матриця: $\begin{pmatrix} 1/k_x & 0 \\ 0 & 1/k_y \end{pmatrix}$

  Нові координати: $\begin{cases} x' = x/k_x \\ y' = y/k_y \end{cases}$

- Зсув осі

  Початкова матриця не змінює вхідні координати.

  Система відображена настипним чином $\begin{cases} x' = x - dx \\ y' = y - dy \end{cases}$



Середовище розробки Conda

Conda - це міжплатформенний пакет та менеджер середовища, який встановлює та управляє пакетами conda із сховища Anaconda, а також з хмари Anaconda. Пакети Conda - це двійкові файли. Не потрібно мати компіляторів для їх встановлення.

Крім того, пакети conda не обмежуються лише програмним забезпеченням Python. Вони також можуть містити бібліотеки C або C ++, пакети R або будь-яке інше програмне забезпечення.

	Conda	**Pip**
Керує	Двійкові файли	Колесо або джерело
Предоставлення компілятору	Ні	Так
Типи пакетів	Любі	Тільки python
Створення середовища	Так, вбудовано	Ні, тільки з venv
Перевірка залежностей	Так	Ні

Таблиця 2.1 – порівняння PIP та Conda

Конда та піп часто вважаються майже однаковими. Хоча деякі функції цих двох інструментів перекриваються, вони були розроблені і повинні використовуватися для різних цілей. Pip - рекомендований Python Packaging Authority інструмент для встановлення пакетів з індексу пакетів Python, PyPI. Pip встановлює програмне забезпечення Python, упаковане як колеса або дистрибутив. Останній може вимагати, щоб система мала сумісні

компілятори та, можливо, бібліотеки, встановлені перед викликом pip для успіху.

Це підкреслює ключову різницю між conda та pip. Pip встановлює пакети Python, тоді як conda встановлює пакети, які можуть містити програмне забезпечення, написане будь-якою мовою. Наприклад, перед використанням pip, інтерпретатор Python повинен бути встановлений через менеджер системних пакетів або завантаживши та запустивши інсталятор. Conda, з іншого боку, може встановлювати пакети Python, а також інтерпретатор Python безпосередньо.

Ще одна ключова різниця між двома інструментами полягає в тому, що conda має можливість створювати ізольовані середовища, які можуть містити різні версії Python та / або встановлені в них пакети. Це може бути надзвичайно корисним при роботі з інструментами науки про дані, оскільки різні інструменти можуть містити суперечливі вимоги, які можуть перешкоджати їх інсталяції в єдине середовище. Pip не має вбудованої підтримки середовищ, а, скоріше, залежить від інших інструментів, таких як virtualenv або venv, для створення ізольованих середовищ.

Jupyter notebook

Jupyter - зручний інструмент для створення аналітичних звітів, що дозволяє зберігати код, зображення, формули та графіки. Також підтримує більшість мов програмування, з легкістю може бути налаштований на будь якому сервері, лише необхідний доступ за ssh або http.

Головні особливості Jupyter:

- Експорт блокнотів.
- Побудова графіків.
- Magic-команди.
- Виконання shell-команд.
- %run для виконання коду на Python.
- Аналіз Big Data.
- Повторне підключення до ядра.

Різниця між Jupyter та Pycharm

Блокнот Jupyter - це середовище IDE з відкритим кодом, яке використовується для створення документів Jupyter, які можна створювати та надавати спільний доступ до живих кодів. Крім того, це веб-інтерактивне обчислювальне середовище. Блокнот Jupyter може підтримувати різні мови, популярні в науці про дані, такі як Python, Julia, Scala, R тощо.

Pycharm - це IDE, розроблена JetBrains і створена спеціально для Python. Він має різні функції, такі як аналіз коду, інтегрований тестер одиниць, інтегрований налагоджувач Python, підтримка веб-фреймворків тощо. Pycharm особливо корисний у машинному навчанні, оскільки підтримує такі бібліотеки, як Pandas, Matplotlib, Scikit-Learn, NumPy тощо.

Jupyter	Pycharm
Блокнот Jupyter - це веб-інтерактивна обчислювальна платформа.	Pycharm - це розумний редактор коду.
Поєднує в собі живий код, рівняння, текст розповіді, візуалізації, інтерактивні інформаційні панелі та інші засоби масової інформації.	Редактор забезпечує підтримку Python, JavaScript, CoffeeScript, TypeScript, CSS, популярну мову шаблонів тощо.
Його можна класифікувати як інструмент глибокого вивчення даних.	PyCharm згруповано під IDE

Забезпечує виконання рядкового коду за допомогою блоків.	Забезпечує розумне автоматичне заповнення.
Забезпечує підтримку вбудованих графічних графіків.	Забезпечує інтелектуальний аналіз коду.
Він може бути тематичним і підтримує ядро.	Це потужний рефакторинг, інтеграція virtualenv та інтеграція Git.
Він дуже гнучкий у порівнянні з pycharm.	Не гнучкий, як у порівнянні з jupyter та повільний запуск.
Такі інструменти, як GitHub, Python, Dropbox, Scala, TensorFlow тощо, інтегровані з jupyter.	Такі інструменти як Django, Anaconda, Wakatime, Kite тощо, інтегровані з Pycharm.
Компанії, такі як Ruangguru, Delivery Hero SE, trivago, Intuit, Hepsiburada тощо використовують Jupyter.	Такі компанії, як Lyft, Bepro Company, trivago, Hepsiburada, Picnic Technologies тощо використовують Pycharm.

Таблиця 2.2 – порівняння Jupyter та Pycharm

Діаграма 2.1 – частина користувачів Jupyter

Проект Jupyter є наступником більш раннього проекту IPython Notebook. Хоча Jupyter Notebooks можна використовувати з багатьма різними мовами програмування, увагу буде приділено Python, оскільки він є найбільш поширеним варіантом використання.

Фреймворк PyTorch

PyTorch, розроблений командою Facebook в 2017 році фреймворк Deep Learning, створений в першу чергу для Python.

Наразі існує два найпопулярніших фремворк - платформи для глибокого навчання PyTorch та Tensorflow.

TensorFlow - це дуже потужна і зріла бібліотека для глибокого навчання з потужними можливостями візуалізації та кількома варіантами для розробки моделей високого рівня. Він має готові до розгортання параметри та підтримку мобільних платформ. PyTorch, з іншого боку, все ще є молодим фреймворком з більш сильним рухом спільноти, і він більш зручний для Python.

TensorFlow - це платформа глибокого навчання з відкритим кодом, створена розробниками в Google і випущена в 2015 році. Офіційне дослідження опубліковано в статті “TensorFlow: широкомасштабне машинне навчання на гетерогенних розподілених системах”.

Зараз TensorFlow широко використовується компаніями, стартапами та бізнес-фірмами для автоматизації речей та розробки нових систем. Він користується своєю репутацією завдяки своїй розподіленій навчальній підтримці, масштабованим параметрам виробництва та розгортання, а також підтримці різних пристроїв, таких як Android.

PyTorch - одна з останніх платформ глибокого навчання, яка була розроблена командою Facebook і відкрита на GitHub. PyTorch набирає популярності своєю простотою, простотою використання, динамічним обчислювальним графіком та ефективним використанням пам'яті.

Спочатку нейронні мережі використовувались для вирішення простих класифікаційних задач, таких як розпізнавання рукописних цифр або ідентифікація реєстраційного номера автомобіля за допомогою камер. Але завдяки найновішим фреймворкам та високим обчислювальним графічним процесорам NVIDIA (GPU), ми можемо тренувати нейронні мережі на байтах даних та вирішувати набагато складніші проблеми.

Кілька помітних досягнень включають досягнення найсучасніших показників набору даних IMAGENET за допомогою згорткових нейронних мереж, реалізованих як у TensorFlow, так і в PyTorch. Навчена модель може бути використана в різних додатках, таких як виявлення об’єктів, семантична сегментація зображень тощо.

Хоча архітектуру нейронної мережі можна реалізувати на будь-якому з цих фреймворків, результат буде не однаковим. Навчальний процес має безліч параметрів, які залежать від рамок.

Наприклад, якщо проводиться навчання набору даних на PyTorch, є можливість вдосконалити навчальний процес, використовуючи графічні процесори, коли вони працюють на CUDA (серверна база C ++). У TensorFlow надає доступ до графічних процесорів, але він використовує власне вбудоване прискорення графічного процесора, тому час навчання цих моделей завжди буде різнитися залежно від обраної структури.

Обчислювальний графік - це абстрактний спосіб опису обчислень як спрямованого графіка.

Графік - це структура даних, що складається з вузлів (вершин) і ребер. Це набір вершин, з'єднаних попарно напрямленими ребрами.

Коли оброблюється код у TensorFlow, графіки обчислень визначаються статично. Усі зв’язки із зовнішнім світом здійснюються через об’єкт tf.Session та tf.Placeholder, які є тензорами, які будуть замінені зовнішніми даними під час виконання. Основною перевагою наявності обчислювального графіка є можливість планування паралелізму або залежності, що робить навчання швидшим та ефективнішим.

Загалом, фреймворк тісніше інтегрований з мовою Python і більшу частину часу почувається ріднішим. Отже, PyTorch - це більше «пітонічний» фреймворк, і TensorFlow відчуває себе абсолютно новою мовою.

Вони значно різняться в областях програмного забезпечення на основі використовуваної структури. TensorFlow забезпечує спосіб реалізації динамічного графіку за допомогою бібліотеки під назвою TensorFlow Fold, але PyTorch має його вбудований.

Розподілене навчання

Однією з основних особливостей, яка відрізняє PyTorch від TensorFlow, є паралельність даних. PyTorch оптимізує продуктивність, користуючись перевагами власної підтримки асинхронного виконання з Python. У TensorFlow доведеться вручну кодувати та тонко налаштовувати кожну операцію, що виконується на певному пристрої, щоб дозволити розподілене навчання. Однак є можливість відтворити все в TensorFlow з PyTorch, але вам потрібно докласти більше зусиль.

Візуалізація

Що стосується візуалізації тренувального процесу, TensorFlow бере на себе ініціативу. Візуалізація допомагає розробнику відстежувати навчальний

процес та налагоджувати більш зручним способом. Бібліотека візуалізації TenforFlow називається TensorBoard. Розробники PyTorch використовують Visdom, однак функції, які надає Visdom, дуже мінімалістичні та обмежені, тому TensorBoard набирає бал у візуалізації навчального процесу.

Розгортання виробництва

Що стосується впровадження навчених моделей у виробництво, TensorFlow є безперечним переможцем. Він надає можливості безпосередньо розгортати моделі в TensorFlow, використовуючи службу TensorFlow, яка є фреймворком, що використовує REST Client API.

У PyTorch ці виробничі розгортання стали простішими в обробці, ніж у останній стабільній версії 1.0, але вони не передбачають жодної основи для розгортання моделей безпосередньо в інтернеті. Обов'язково доведеться використовувати Flask або Django як сервер. Отже, обслуговування TensorFlow може бути кращим варіантом, якщо продуктивність викликає занепокоєння.

Визначення простої нейронної мережі в PyTorch та TensorFlow

У PyTorch нейронна мережа буде класом, і за допомогою пакета torch.nn імпортує необхідні шари для побудови архітектури. Усі шари спочатку оголошуються в методі __init __ (), а потім у методі forward (), як вхідний x обходить до всіх шарів у мережі.

Нещодавно Keras, нейромережевий фреймворк, який використовує TensorFlow як бекенд, був об'єднаний у TF Repository. Відтоді синтаксис оголошення шарів у TensorFlow був схожий на синтаксис Keras. Спочатку оголошується змінна та призначається її тип архітектури, яку оголошуємо, в даному випадку архітектурі "Sequential ()". Далі додаються шари безпосередньо послідовно, використовуючи метод model.add (). Тип шару можна імпортувати з tf.layers.

Висновок порівняння фреймворків

TensorFlow - це дуже потужна і зріла бібліотека для глибокого навчання з потужними можливостями візуалізації та кількома варіантами для розробки моделей високого рівня. Він має готові до розгортання параметри та підтримку мобільних платформ. PyTorch, з іншого боку, все ще є молодим фреймворком з більш сильним рухом спільноти, і він більш зручний для Python.

У дипломній роботі доцільно використовувати PyTorch щоб забезпечити гнучкість навчання системи та можливість оптимізувати процес розробки.

2.1 Архітектура нейромережі

Модель перцептрону

В якості дискрімінативної моделі нейромережі буде використовуватися перцептрон.

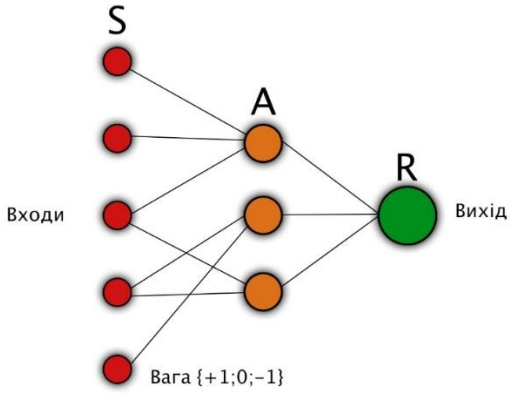


Рисунок 2.3 - Структура нейромережі – перцептрон

Багатошаровий персептрон - нейронна мережа, що складається з шарів, кожен з яких складається з елементів – нейронів, їх моделей. Ці елементи бувають трьох типів: сенсорні (вхідні, S), асоціативні (вивчення «прихованих» шарів, A) та реактивні (вихідні, R).

Цей тип персептрона називають багатошаровим не тому, що він складається з декількох шарів, оскільки вхідний і вихідний рівні не можуть бути створені в коді, а тому, що вони мають кілька навчальних (A) шарів.

Модель нейрона - це мережевий елемент, який має кілька входів, кожен з яких має вагу. Коли нейрон отримує сигнал, він множить сигнали на вагу, додає отримані значення, а потім передає результат іншому нейрону або на виході мережі.

Багатошаровий персептрон відрізняється. Його функція сигмоїдна, вона відображає значення в діапазоні від 0 до 1.

функція вартості визначається як

$$C0 = (a (L) - y) 2.$$

Активація останнього нейрона задається зваженою сумою, а вірніше масштабується функцією від зваженої суми:

$$a (L) = \sigma (w (L) a (L-1) + b (L)).$$

Для стислості зважену суму можна позначити буквою з відповідним індексом, наприклад z (L):

$$a\ (L) = \sigma\ (z\ (L)).$$

зміна C0 залежить від зміни a (L), що в свою чергу залежить від зміни z (L), яке і залежить від w (L). Відповідно правилом взяття подібних похідних, шукане значення визначається твором наступних приватних похідних:

$$\partial C0 / \partial w\ (L) = \partial z\ (L) / \partial w\ (L) \bullet \partial a\ (L) / \partial z\ (L) \bullet \partial C0 / \partial a\ (L)$$

Визначення похідних

$$\partial C0 / \partial a\ (L) = 2\ (a\ (L) - y)$$

Тобто похідна пропорційна різниці між поточним значенням активації і бажаним.

Середня похідна в ланцюжку є просто похідною від масштабується функції:

$$\partial a\ (L) / \partial z\ (L) = \sigma\ '(z\ (L))$$

Останній множник це похідна від зваженої суми:

$$\partial z\ (L) / \partial w\ (L) = a\ (L-1)$$

Таким чином, відповідну зміну визначається тим, наскільки активований попередній нейрон.

$$\partial C_0/\partial w^{(L)} = 2(a^{(L)} - y)\ \sigma'(z^{(L)})\ a^{(L-1)}$$

При використані багатошарового перцептрону все буде виглядати аналогічно, просто додасться додатковий нижній індекс, що відображає номер нейрона всередині шару, а у ваг з'являться подвійні нижні індекси, наприклад, jk, що відображають зв'язок нейрона j з одного шару L з іншим нейроном k в шарі L-1.

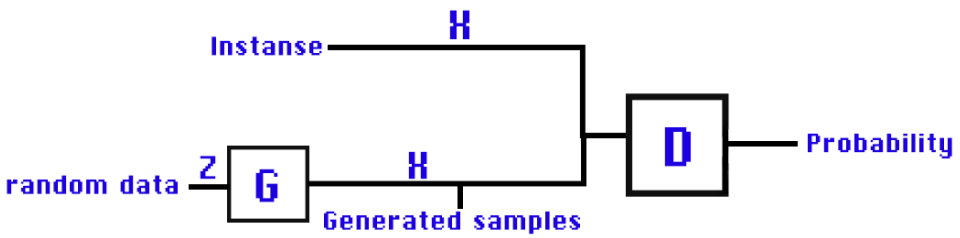


Рисунок 2.4 – Структура GAN нейромережі

2.2 Метод оптимізації нейронної мережі

Nesterov Accelerated Gradient

Ідея методів з накопиченням імпульсу, полягає в пошуках оцінки середнього значення, тому використовується формула експонціальним середнім значенням.

$$v_t = \gamma v_{t-1} + (1 - \gamma)x$$

Рисунок 2.4 – формула експонціального середнього значення

Графічне представлення накопленого імпульсу

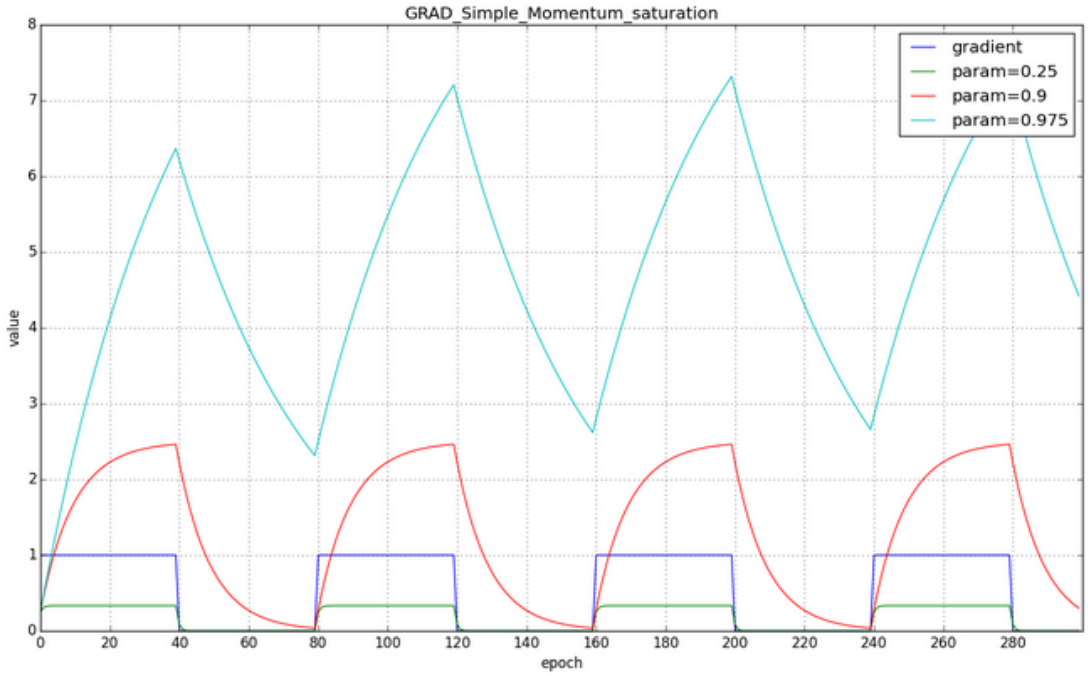


Рисунок 2.5 – градієнт накоплених значень

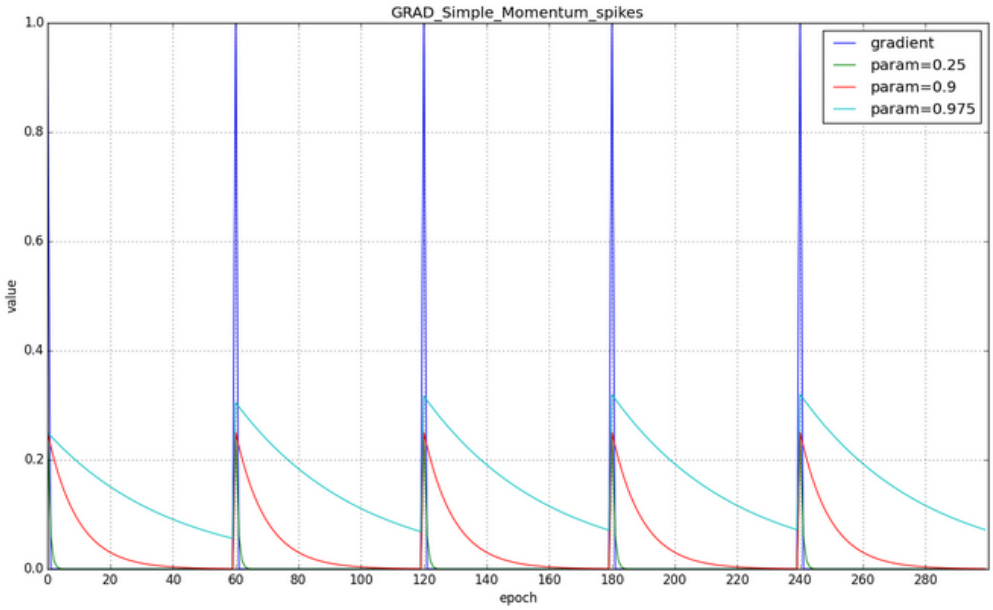


Рисунок 2.6 – зубковий градієнт накоплених значень

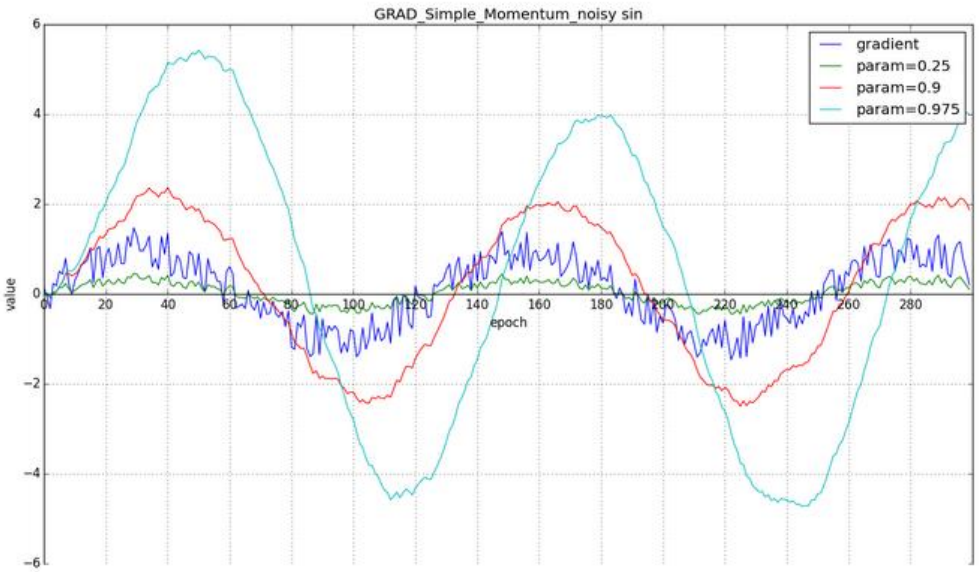


Рисунок 2.7 – градієнт шуму накоплених значень

Проте цього недостатньо, якщо змістити градієнт функції втрат в точку за формулою :

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$

$$\theta = \theta - v_t$$

Рисунок 2.8 – формула зміненої точки

$$v_t = \gamma v_{t-1} + \frac{\eta}{3}\left(\nabla_\theta J(\theta - \frac{\gamma v_{t-1}}{3}) + \nabla_\theta J(\theta - \frac{2\gamma v_{t-1}}{3}) + \nabla_\theta J(\theta - \gamma v_{t-1})\right)$$

Рисунок 2.9 – формула стабілізації градієнта розподіленого на декількох точках

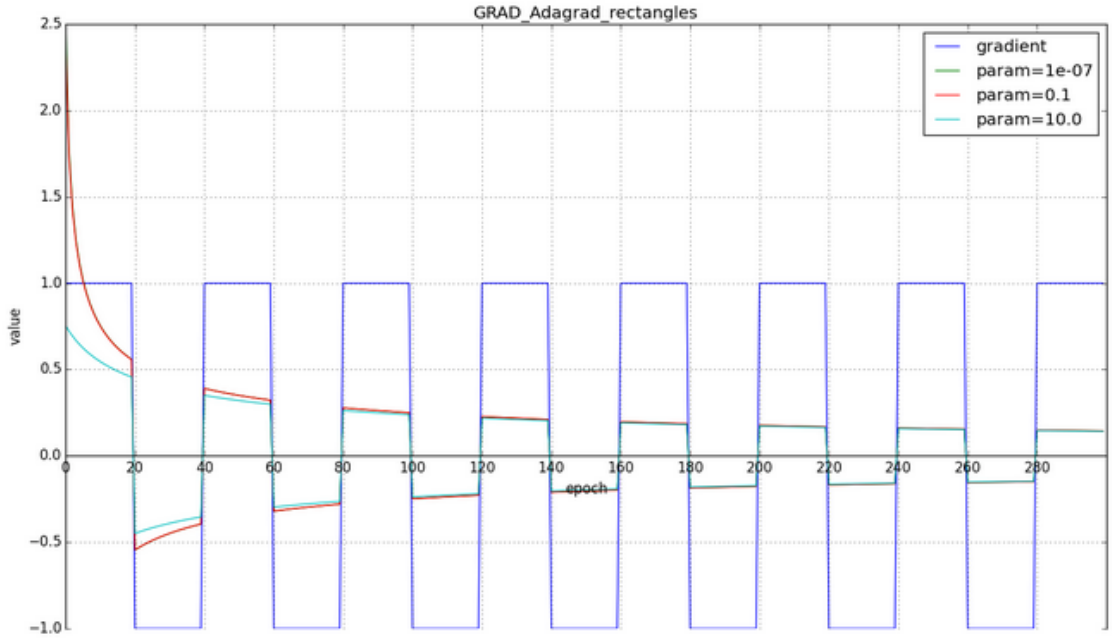
Adagrad

Модифікація стохастичного градієнтного спуску з окремою для кожного параметру швидкістю навчання.

$$G_t = G_t + g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} g_t$$

Рисунок 2.10 – формула оновлення
Gt – сума квадратів оновлення



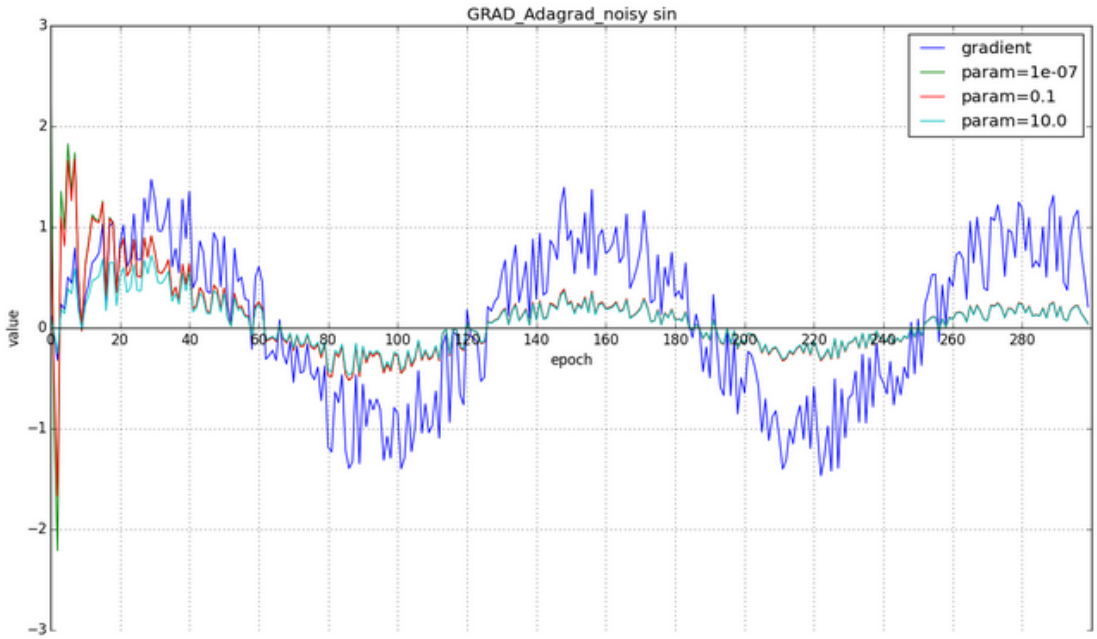


Рисунок 2.10 – графіки навчання за допомогою Adagrad

Головною ідеєю є використання пропозицій зменшення оновлення елементів, відсутня потрібність влучно підбирати швидкість навчання.

RMSProp

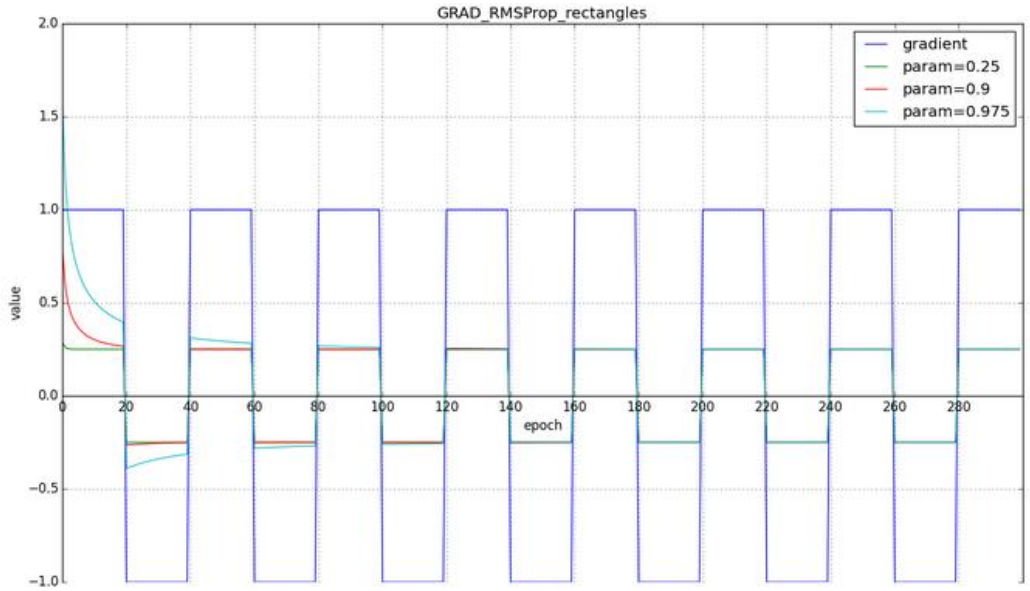
Модифікуючи ідею Adagrad: все так же буде оновлювати менше ваги, які занадто часто оновлюються, але замість повної суми оновлень, використовувати усереднений по історії квадрат градієнта.

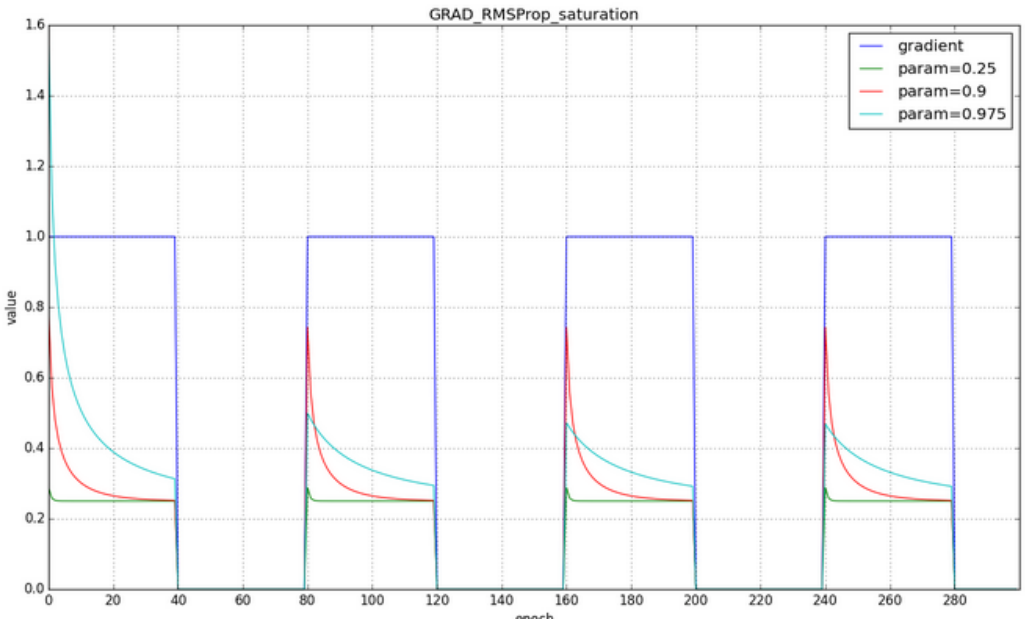
$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

$$RMS[g]_t = \sqrt{E[g^2]_t + \epsilon}$$

Рисунок 2.11 – Основні формули RMSProp





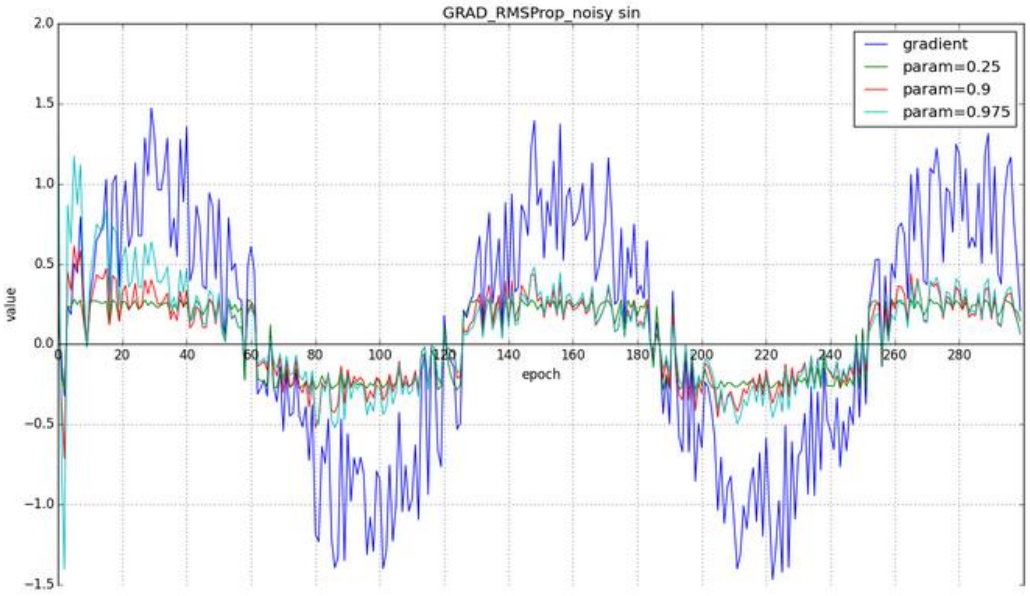


Рисунок 2.12 – графіки швидкості оновлення

Adadelta

Схожий на RMSProp, головно. Різницею є додавання в чисельник, стабілізуючий член пропорціональний RMS.

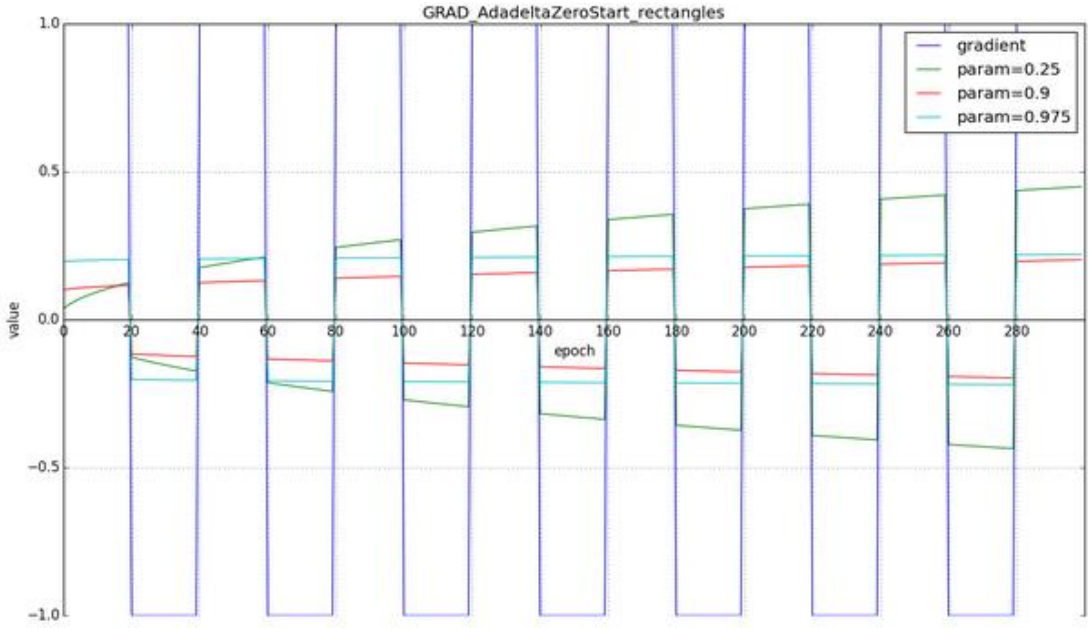
$$\Delta\theta = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$

$$\theta_{t+1} = \theta_t - \frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2$$

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon}$$

Рисунок 2.13 – Основні формули Adadelta



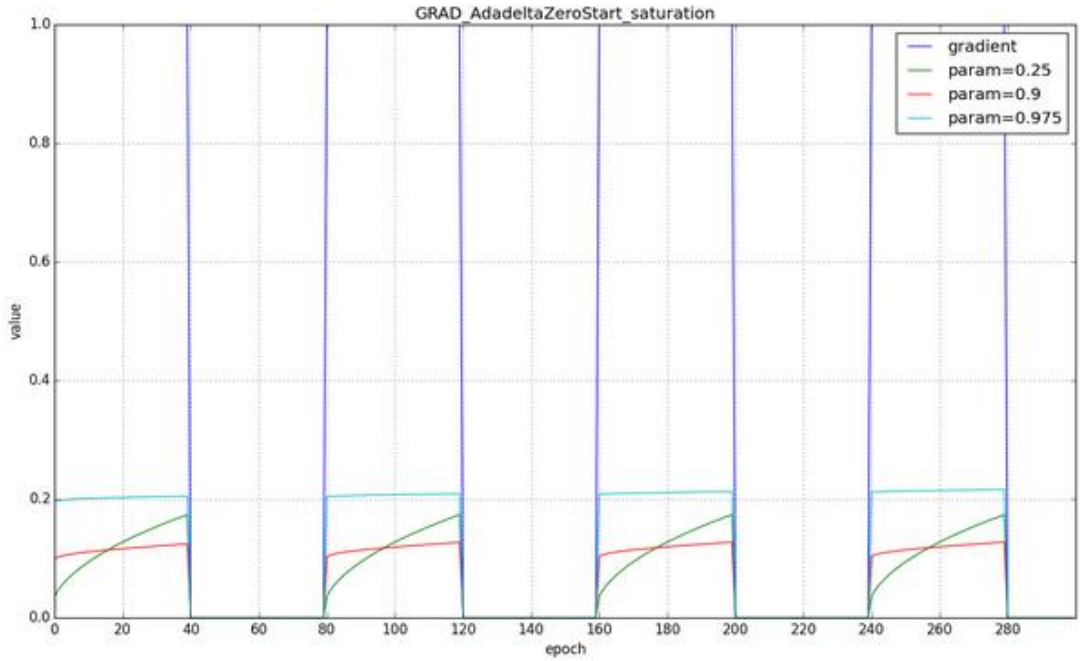
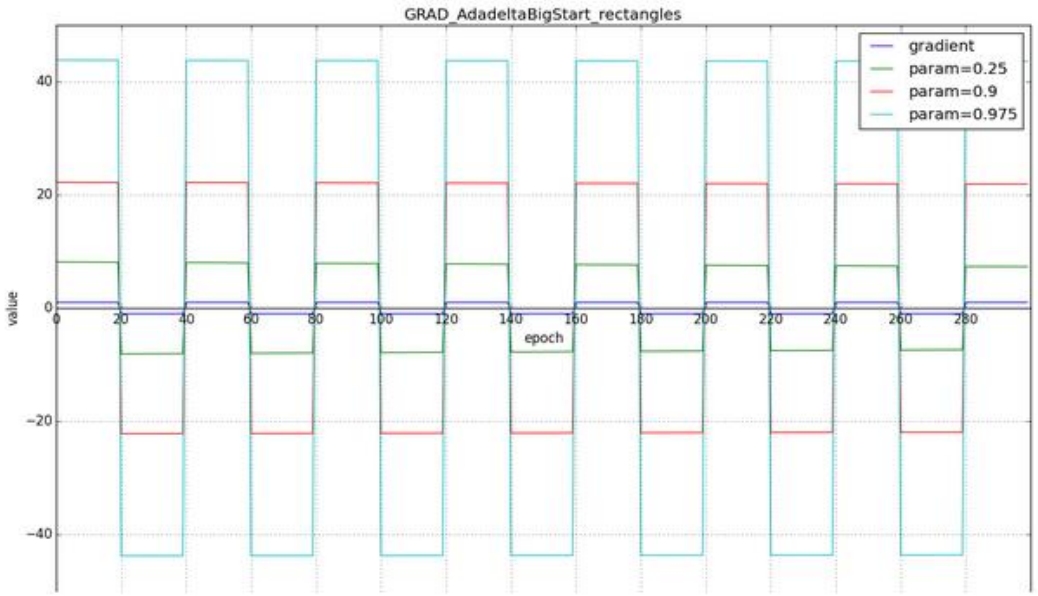


Рисунок 2.14 – Графіки нульового початкового RMS



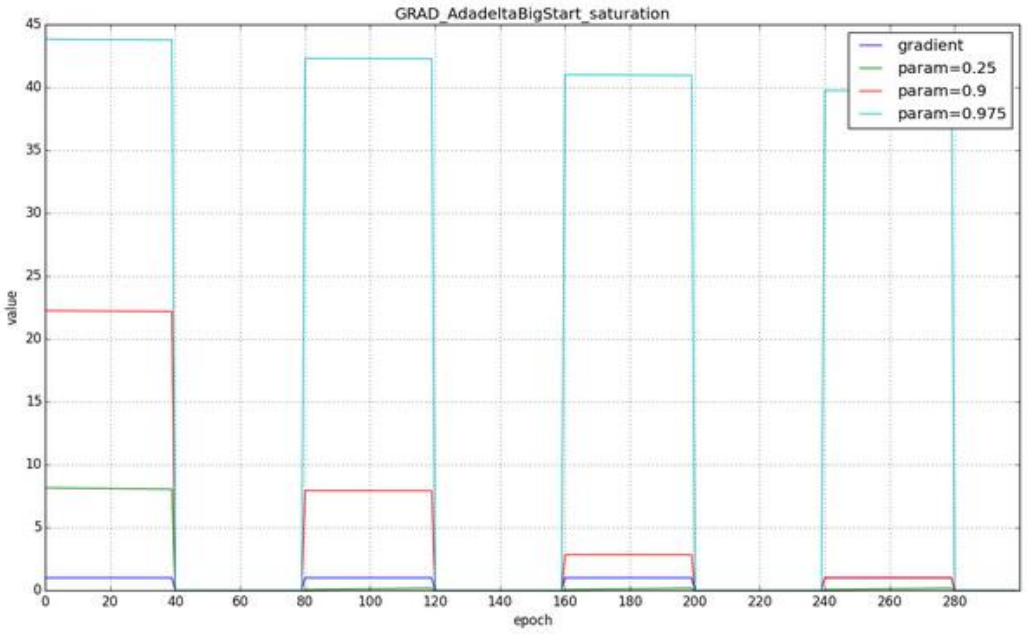


Рисунок 2.15 – Графіки для більшого початкового значення RMS

Для RMSProp і Adadelta, як і для Adagrad не потрібно дуже точно підбирати швидкість навчання - досить приблизного значення.

Adam

Для навчання моделей дискриминатора і генератора був реалізований алгоритм стохастичного градієнтного спуску Adam - adaptive moment estimation, оптимізаційний алгоритм. Він поєднує в собі і ідею накопичення руху і ідею слабшого поновлення ваг для типових ознак.

$$m_w^{(t+1)} \leftarrow \beta_1 m_w^{(t)} + (1-\beta_1)\nabla_w L^{(t)}$$
$$v_w^{(t+1)} \leftarrow \beta_2 v_w^{(t)} + (1-\beta_2)(\nabla_w L^{(t)})^2$$

$$\hat{m}_w = \frac{m_w^{(t+1)}}{1-\beta_1^{t+1}}$$

$$\hat{v}_w = \frac{v_w^{(t+1)}}{1-\beta_2^{t+1}}$$

$$w^{(t+1)} \leftarrow w^{(t)} - \eta \frac{\hat{m}_w}{\sqrt{\hat{v}_w} + \epsilon}$$

Рисунок 2.16 - Основні формули Adam

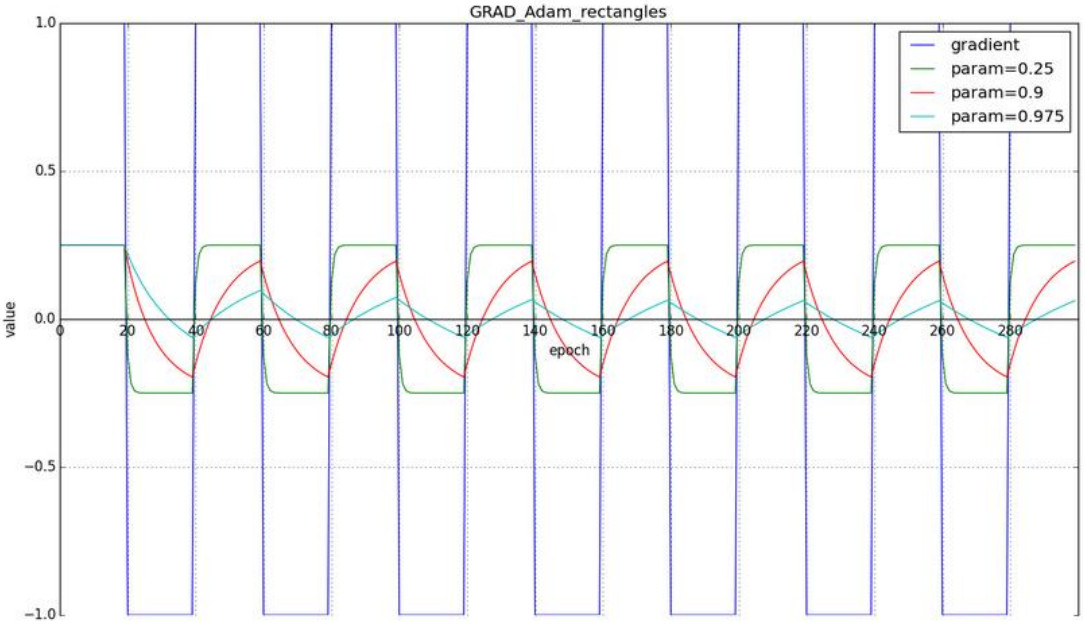


Рисунок 2.17 – приклад синхронізації у графіках

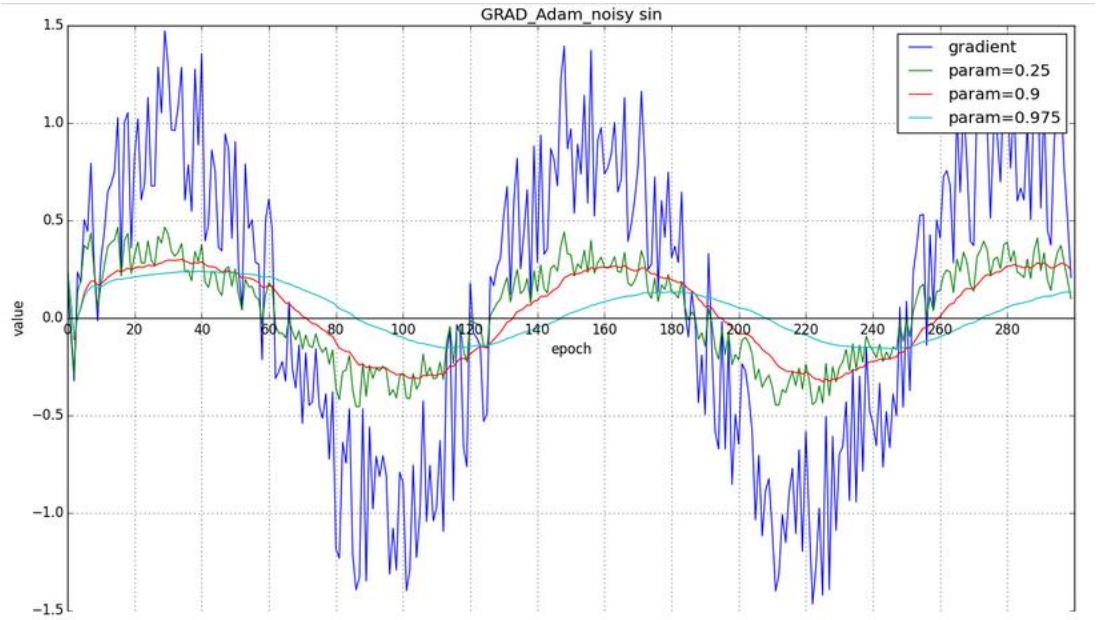


Рисунок 2.18 – графік оновлення синусу

Активаційна функція нейрона

Головним завданням є можливість навчати нейромережа якомога швидше не втрачаючи якості, для цього підходить активаційна функція типу ReLu.

$$A(x) = \max(0, x)$$

значення x повертається якщо воно позитивно і нуль в іншому випадку.

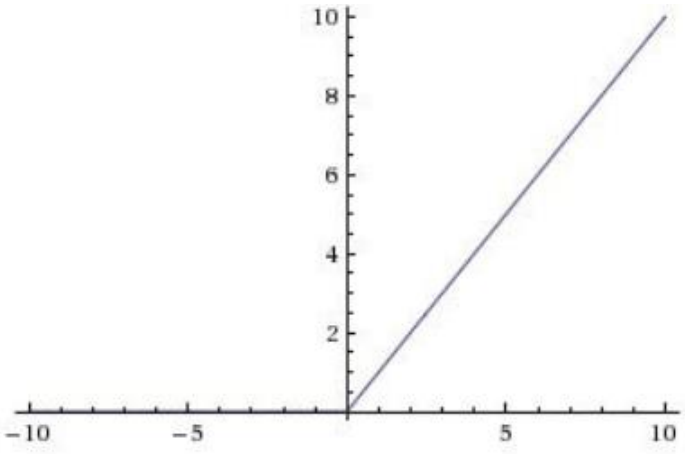


Рис 2.19 - Схема функції ReLu.

На перший погляд здається, що ReLu має всі ті ж проблеми, що і лінійна функція, так як ReLu лінійна в першому квадраті. ReLu далеко нелінійна за своїм виглядом, а комбінація ReLu також буде нелінійною (Така функція є дійсно якісним аппроксіматором, так як будь-яка функція може бути апроксимована комбінацією ReLu, це досить зручно але не означає можливостей використовувати її завжди).

Це означає, що слої можуть бути накладені один на одного. Область допустимих значень ReLu - [0, inf).

Наступний пункт – плавність активації. Якщо взяти нейромережу з великою кількістю нейронів. Використання сигмоїда або гіперболічного тангенсу буде означати за собою активацію всіх нейронів по черзі та буде досить напруженою.

Це означає, що майже всі активації повинні бути оброблені для опису виходу мережі. Іншими словами, активація щільна, що призводить до сильних затрат у часі. Тому ліпше залишити активованими декілька нейронів, це зробило б їх активації розрідженими і ефективними. ReLu дозволяє це зробити.

Якщо представити мережу із випадково ініціалізованими вагами, в якій приблизно половина активацій рівні 0, а вже відомо що ReLu повертає 0 для від'ємних значень x. У такій мережі буде задіяно менше нейронів, безпосередньо сама мереже стане легшою.

ReLu бездоганно підходить за всіма параметрами але це не так, через те, що частина ReLu вдає із себе горизонтальну лінію (для негативних значень X), градієнт на цій частині дорівнює 0.

Через рівності нулю градієнта, ваги не будуть коригуватися під час спуску. Це означає, що перебуваючи в такому стані нейрони не реагуватимуть на зміни в помилки/вхідних даних із-за проблеми що градієнт буде дорівнювати нулю, нічого не зміниться.

Таке явище називається проблемою ReLu (Dying ReLu problem). Через ризик виникнення проблеми деякі нейрони просто вимикаються і не будуть відповідати, роблячи значну частину нейромережі пасивною. Однак існують варіації, які допомагають цю проблему уникнути. Наприклад, має сенс замінити горизонтальну частину функції на лінійну.

Якщо вираз для лінійної функції задається виразом $y = 0.01x$ для області $x < 0$, лінія злегка відхилиться від горизонтального положення.

Основна ідея - зробити графік нерівним нулю і поступово відновлювати його під час тренування нейромережі. ReLu менш вибаглива до обчислювальних ресурсів, ніж гіперболічний тангенс або сигмоїд, так як виробляє більш прості математичні операції. Тому має сенс використовувати ReLu при створенні глибоких нейронних мереж.

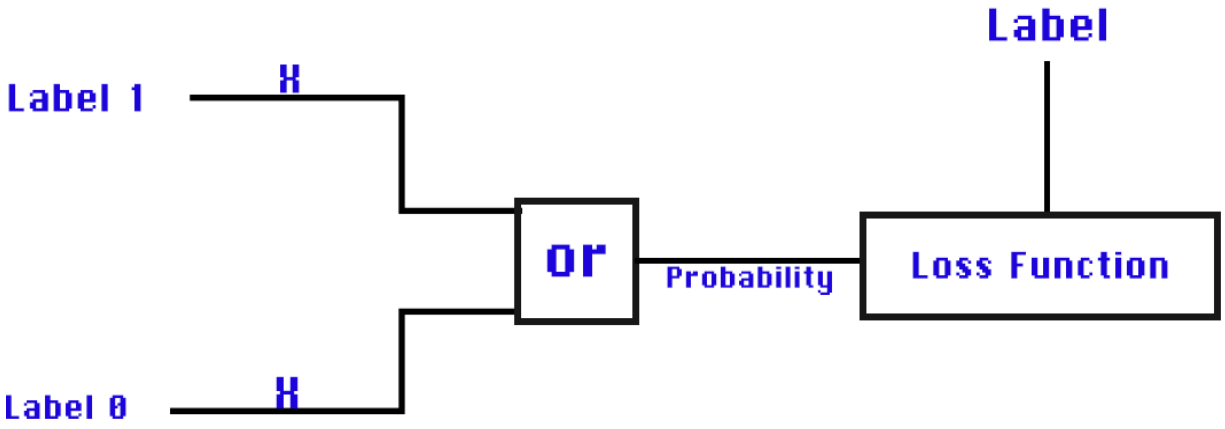


Рис. 2.8 - процес навчання дискримінатору

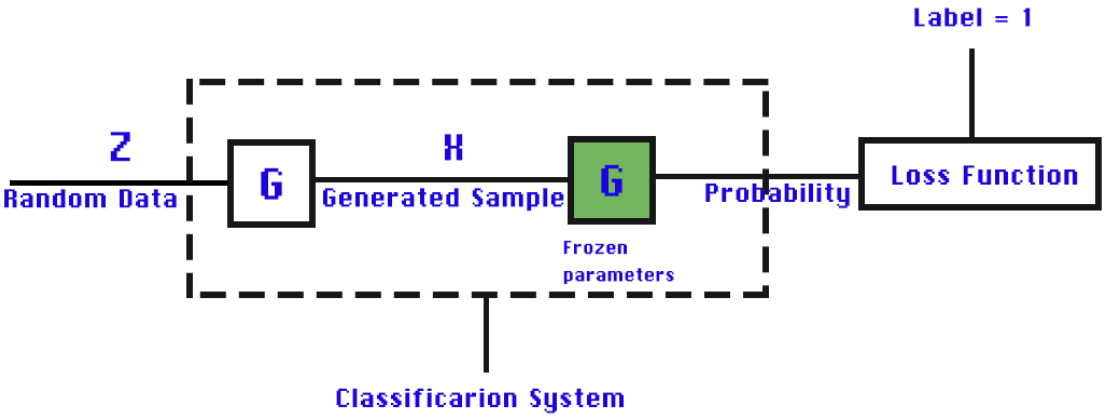


Рисунок 2.8 – процес навчання генератору

PyTorch

Для глибокого навчання нейромережі використовувався фреймворк PyTorch

Torchvision та torchvision.transforms – імпортовані для переробки інформації, що зберігаються в файлах зображення.

Число 125 використано для ініціалізації генератора випадкових чисел, знадобиться для задання початкових ваг нейронної мережі.

При навчанні на центральному процесорі (CPU) на кожну епоху буде відходити багато часу. Але для нормального результату генерації зображень потрібно більше 50-60 епох.

Щоб скоротити час навчання нейромережі, бажано використовувати графічний процесор (GPU).

Для того щоб код працював незалежно від потужності девайсу на якому навчається нейромережа, потрібно створити об’єкт device, який буде вказувати або на центральний процесор, або на графічний процесор при його наявності.

За можливостей використання cuda, процес навчання прискориться залежно

CUDA	CPU
На обробку п’яти епох навчання моделі було витрачено 10 хвилин	На обробку п’яти епох навчання моделі було витрачено 30 хвилин
На обробку сотні епох навчання було витрачено декілька годин	На обробку сотні епох навчання було витрачено чотири години

Таблиця 2.3 – порівняння використання графічного процесору та центрального

3. Розробка нейромережі

Підготовка вхідних даних

Для використання даних потрібно ввести функцію transform, функція складається з двох частин:

- transforms.ToTensor() – перетворює дані в тензор для PyTorch
- transforms.Normalize() – перетворює діапазон коефіцієнтів, котрі знаходяться в діапазоні від 0 до 1
- datasets.EMNIST – дає можливість завантажити дані для навчання нейромережі використовуючи бібліотеку EMNIST.
- download = True – дає гарант що при першому запуску коду набір даних EMNIST обов’язково буде завнтажено і збережено.
- train_dataset – для створення завантажувача даних.

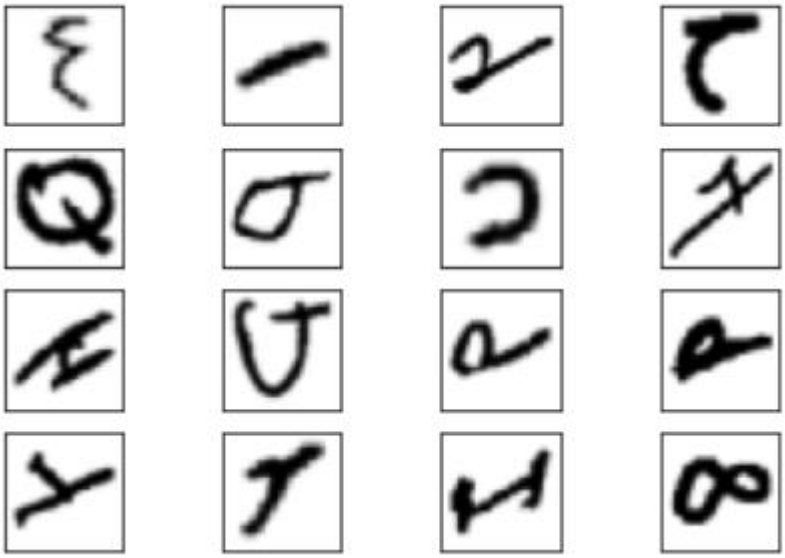


Рисунок 3.1 - Візуальне представлення вхідних даних

Реалізація Дискримінатору

Дискримінатором є нейронна мережа багатошарового перцептрона, яка приймає зображення розміром 28 × 28 пікселів і знаходить ймовірність того, що зображення належить реальним навчальних даних.

Для введення коефіцієнтів зображення в нейронну мережу перцептрону, необхідно їх представити у векторі так, щоб нейронна мережа завойовувала вектор, що складається з 784 коефіцієнтів (28 × 28 = 784).

Векторизація відбувається в першому рядку методу forward () - виклик x.view () перетворює форму вхідного тензора.

Вихідна форма тензора *x*x - 32 × 1 × 28 × 28, де 32 - розмір партії. Після перетворення форма *x*x стає рівною 32 × 784, причому кожен рядок являє коефіцієнти зображення навчального набору.

Щоб запустити модель дискримінатору з використанням графічного процесора, потрібно створити його екземпляр і пов'язати з об'єктом пристрою за допомогою методу to ()

Реалізація Генератору

Спочатку потрібно створити клас Generator, успадковані від nn.Module, потім визначити архітектуру нейронної мережі, і, нарешті, створити екземпляр об'єкта Generator

Генератор включає два прихованих шару з нейронами з активаційною функцією ReLU, а на виході шар з двома нейронами з лінійною функцією активації.

Таким чином, вихідні дані будуть складатися з двох елементів, що мають значення в діапазоні від -∞ до + ∞, яке буде представляти (x1, x2). Це означає що у генератора немає ніяких обмежень - він повинен всьому навчитися сам.

Вихідні коефіцієнти повинні знаходитися в інтервалі від -1 до 1. Тому на виході генератора найкраще використати гіперболічну функцію активації Tanh ()

Навчання моделей

- Lr (learning rate) – швидкість навчання, використовується для адаптації ваги мережі
- Num_epochs - задає кількість епох, означає скільки повторень процесу навчання буде виконано з використанням всього датасету.

Змінною loss_function призначається функція логістичної функції втрат (бінарної перехресної ентропії) - BCELoss ().

Це та функція втрат, яку краще всього використовувати для навчання моделей. Вона підходить як для навчання дискримінатора, так і для генератора, так як він подає свої дані на вхід дискримінатора.

Правила поновлення ваг (навчання моделі) в PyTorch реалізовані в модулі torch.optim.

Для навчання моделей дискриминатора і генератора був реалізований алгоритм стохастичного градієнтного спуску Adam(*Adaptive Moment Estimation*).

Цикл

Отримуємо реальні зразки поточних даних з завантажувача даних і призначаємо їх змінній real_samples.

Перший вимір в розмірності масиву має кількість елементів, рівний batch_size. Це стандартний спосіб організації даних в PyTorch, де кожен рядок тензора представляє один зразок з пакету.

Використано torch.ones (), щоб створити ярлики зі значенням 1 для реальних зразків і призначити ярлики змінній real_samples_labels.

Генерація зразків, зберігаючи випадкові дані у hidden_spatial_samples, які потім передаються генератору для отримання generate_samples. Для міток згенерованих зразків потрібно використати нулі torch.zeros (), які ми зберігаємо в gens_samples_labels.

Залишилось поєднати реальні та згенеровані зразки, мітки та зберегти відповідно у all_samples та all_samples_labels.

Тренування нейромережі

Дискримінатор

У PyTorch важливо очищати значення градієнту на кожному етапі навчання. Краще робити це за допомогою методу zero_grad ().

- Обчислюємо вихідні дискримінаційні дані, використовуючи навчальні дані all_samples.
- Необхідно обчислити значення функції втрат, використовуючи вивід дискримінатора output_discriminator та мітку all_samples_labels.
- Обчислити градієнти для відновлення ваги за допомогою функції loss_discriminator.backward ().
- Знайти оновлену шкалу дискримінації, застосувавши optimizer_discriminator.step ().

Підготувавши дані для навчання генератора. Рандомізовані дані зберігаються в hidden_samples_samples, кількість рядків відповідає - batch_size. Використано два стовпці, щоб зіставити дані з двовимірними даними на вході генератора.

Генератор

- Перед початком тренування моделі генератору потрібно очистити градієнти, використовуючи метод zero_grad ().

- Потім наступний крок генератор latent_space_samples і зберегти його вихідні дані у generated_samples.

- Передати вихідні дані генератора - дискримінатору і зберегти його вихідні дані у output_discriminator_generated, що буде використано як вихідні дані всієї моделі.

- Обчислити функцію збитків, використовуючи висновки системи класифікації, які зберігаються в мітках output_discriminator_generated та real_samples_labels, рівних 1.

– Обчислити градієнт та оновити вагу генератора. Слід пам’ятати, що, коли тренуємо генератор, то підтримуємо дискримінацію ваги в замороженому стані.

– В останніх рядках циклу відбувається виведення значення функцій втрат дискримінатора та генератора в кінці кожної епохи.

Перевірка результатів нейромережі

Щоб побудувати згенеровані зразки, якщо дані обробляються на графічному процесорі, вони повинні бути відправлені назад у CPU.

Все, що потрібно зробити, це викликати метод cpu (). Як і раніше, перед витягуванням даних потрібно викликати метод detach ().

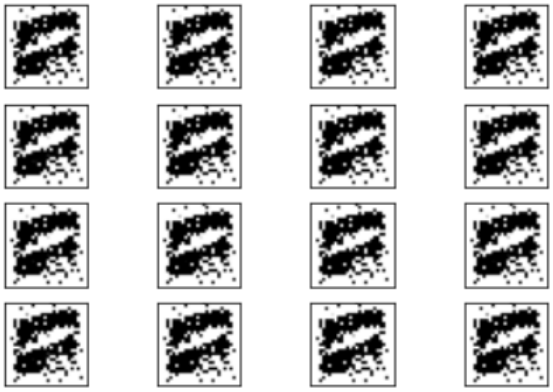


Рисунок 3.2 - навчання моделі після 5-ти епох.

Рисунок 3.3 - після 50-ти епох навчання.

ВИСНОВОК

Результати роботи, проведеної в бакалаврській роботі, що проваджують можливості оптимізації створення монограм та навчання нейромережі. Було проведено аналіз існуючих рішень, на основі якого впроваджено нові можливості.
Виконання задач кваліфікаційної роботи, привело до створення нейромережі та пошуків рішення проблем глибокого навчання.

- Проаналізовано архітектури генеративних технологій.

- Проведено порівняльний аналіз фреймворків.

- Знайдено можливості забезпечення співпраці середовища розробки з мовою програмування.

В якості удосконалення запропоновано використовувати алгоритм стохастичного градієнтного спуску Adam з використанням фреймворку PyTorch, та розробка нейромережі з генерацією зображень.

Запропоновану модель нейромережі можна розвивати надалі, використовуючи зображення з великою роздільною здатністю. Впроваджувати створення власних датасетів та надання її на генерацію до нейромережі.