Застосування генетичного алгоритму та спроба замінити Convolutional Neural Network на Cellular Automata

Андрій Кривий, Нікіта Леник, Владислав Шимановський, Іван Максимчук

1 Мета проєкту

- Використання клітинних автоматів замість згорткових нейронних мереж
- Пошук альтернативного підходу до обробки зображень
- Застосування генетичних алгоритмів для оптимізації правил клітинних автоматів

2 Огляд структури проекту

Проект складається з 2-х мікрофреймворків та 3-х додаткових модулів:

- Фреймворк `nml` — відповідає за побудову нейромереж, операції над тензорами та їх виконання як на CPU, так і на GPU
- Фреймворк `genetic` — відповідає за генетичний алгоритм (а саме селекцію, мутації та кросовер) та залежить від тензорів та параметрів у `nml`
- Модуль `loader` — відповідає за завантаження, семплинг та попередню обробку датасетів
- Модуль `project` — імплементація коду відповідального за цикл генетичного алгоритму, містить обчислення фітнесу та запуск самого алгоритму
- Модуль `handlers` — імплементації обробників, які використовуються у `project` для: збереження проміжних результатів навчання, виведення інформації у термінал та збереження інформації у `csv` таблиці

3 Використані технології (залежності)

- NumPy [3] для матричних обчислень на CPU
- Numba [6] для JIT-компіляції

- Numba-CUDA для GPU-прискорення
- Python як основна мова програмування
- Інші бібліотеки лише для завантаження датасетів

4 Мікрофреймворк `nml`

Цей мікрофреймворк був значною мірою натхненним гігантами індустрії: *PyTorch* [1] та *TensorFlow*[2]. Використано для імплементації підтримки цілочисельних тензорів, які використовуються у шарі `CellularAutomata`

4.1 Загальна архітектура

Фреймворк містить 6 основних компонент:

- `Tensor` — тензори, багатовимірні масиви
- `Layer` — шари для опису моделі
- `Unit` — скомпільовані шари, на яких виконуються обчислення на конкретній моделі
- `Sequential` — послідовний опис моделі (контейнер шарів)
- `Model` — скомпільована модель під конкретний пристрій
- `cpu/gpu` — модулі для імплементації конкретних операцій над тензорами

Код можна розбити на 3 частини:

1. Опис моделі — API інтерфейс для користувача фреймворка: `Tensor`, `Parameter`, `Sequential`, `Layer`, те, що створює користувач
2. Побудова моделі — об'єкти які будуються з опису: `Unit`, `Model`, `DeferredResults`, те, що користувач використовує, але сам не створює
3. Виконання моделі — код для виконання різних операцій над тензорами, міститься у підмодулях `cpu` та `gpu`, або використовується з бібліотеки `numpy`, користувач напряму не взаємодіє з цим кодом, внутрішня імплементація

4.2 Доступні шари:

- `Input` — вхідний шар
- `CellularAutomata` — шар клітинного автомата
- `Linear` — шар лінійного перетворення
- `ReLU`, `Softmax`, `Tanh`, `Sigmoid`, `PReLU`, `LeakyReLU` — шари активацій
- `Cast`, `Reshape`, `Flatten` — шари маніпуляції тензорів

4.3 Експериментальний шар клітинного автомата

Дискретна модель, що складається з квадратної сітки клітинок, кожна з яких може мати різний стан. Кожна клітинка змінює свій стан залежно від станів сусідів за певними правилами. Сітка "загортається"по краях, тобто крайні клітинки взаємодіють із протилежними. Клітинки можуть взаємодіяти з різною кількістю сусідів, і для цього в проєкті реалізовано кілька методів пошуку сусідів. Найпростіший — це околиця фон Неймана, де кожна клітинка враховує лише чотирьох сусідів по вертикалі та горизонталі (зверху, знизу, зліва, справа). Околиця Мура розширює цю ідею, дозволяючи клітинці враховувати ще й діагональних сусідів, тобто всього вісім сусідів навколо. Також використовується розширена околиця, яка включає сусідів на більшій відстані, наприклад, другий рівень по прямим напрямкам, що дає до 12 сусідів. Принцип роботи кожного методу полягає у визначенні індексів сусідніх клітинок відносно поточної, з урахуванням замикання сітки по краях, і подальшому аналізі їхніх станів для прийняття рішення про зміну стану самої клітинки. З інформації про сусідів та теперішню клітину будується число переходу, яке є індексом у векторі-таблиці пошуку для нового стану. Саме таблиця пошуку є тензором-параметром цього шару.

4.4 Апаратне прискорення

- CPU-реалізації використовують Numba [6] для JIT-компіляції
- GPU-реалізації використовують CUDA через Numba CUDA
- Автоматичне виявлення доступності GPU

4.5 Система відкладених результатів

- `DeferredResults` (`nml/model.py`) дозволяє асинхронно обробляти результати
- Особливо корисно для GPU-обчислень, де можна перекрити передачу даних і обчислення

4.6 Робочий процес

1. Визначення вхідних даних (`Input`)
2. Конфігурація архітектури моделі через послідовність шарів (`Sequential`)
3. Побудова моделі для конкретного пристрою (`build` з параметром `device`)
4. Виконання обчислень (`infer`)
5. Отримання результатів (`wait` на об'єкті `DeferredResults`)

4.7 Приклад використання

```
model = Sequential(
    Input(shape=(28, 28), dtype=np.uint8),
    CellularAutomata(rule_bitwidth=1, neighborhood="moore_1"),
    Flatten(),
    Cast(dtype=np.float32),
).build(device=Device.GPU)
```

```
results = model(input_tensor)
output = results.wait()
```

Фреймворк дозволяє ефективно виконувати нейромережеві обчислення, автоматично адаптуючись до наявного апаратного забезпечення.

5 Мікрофреймворк `genetic` та додаткові модулі

5.1 Оцінка якості рішень

Наш фреймворк підтримує наступні метрики для оцінки якості:

- **Accuracy** — частка правильних передбачень серед усіх:

$$\text{Accuracy} = \frac{\text{Кількість правильних передбачень}}{\text{Загальна кількість}}$$

- **Cross-Entropy** — логарифмічна функція втрат, яка оцінює відстань між справжніми й передбаченими розподілами ймовірностей:

$$\text{CE} = -\sum_{i=1}^{n} y_i \log(\hat{y}_i)$$

- **Mean Probability** — середня ймовірність правильного класу серед усіх передбачень:

$$\frac{1}{n}\sum_{i=1}^{n} P(\text{правильний}_i)$$

- **Combined** — комбінована метрика, яка об'єднує точність та середню ймовірність із ваговими коефіцієнтами:

$$\text{Combined} = w_1 \cdot \text{Accuracy} + w_2 \cdot \text{MeanProbability}$$

- **Balanced Accuracy** — зважена точність для класифікацій з незбалансованими класами (враховує точність по кожному класу окремо).

$$\text{Balanced Accuracy} = \frac{1}{K}\sum_{k=1}^{K} \frac{TP_k}{TP_k + FN_k}$$

де K — кількість класів, TP_k — кількість істинно позитивних випадків для класу k, FN_k — хибно негативних.

5.2 Хромосоми та операції над ними

- `Crossover` (`genetic/crossover.py`) — клас, що реалізує різні методи кросоверу:
  - `single_point` — одноточковий кросовер
  - `two_point` — двоточковий кросовер
  - `uniform` — рівномірний кросовер
  - `none` — без кросоверу

- `Mutation` (`genetic/mutation/base.py`) — базовий клас для мутацій з реалізаціями:
  - `GaussianMutation` (`genetic/mutation/gaussian.py`) — додає гаусів шум до хромосом
  - `GaussianScaledMutation` (`genetic/mutation/gaussian_scaled.py`) — адаптивний гаусів шум, де сила мутації масштабується відносно стандартного відхилення

5.3 Селекція

Різні стратегії вибору батьківських особин:

- `BestSelection` (`genetic/selection/best.py`) — вибір найкращих особин
- `RankSelection` (`genetic/selection/rank.py`) — ймовірність вибору пропорційна рангу моделі
- `RouletteSelection` (`genetic/selection/roulette.py`) — ймовірність вибору пропорційна фітнесу
- `TournamentSelection` (`genetic/selection/tournament.py`) — турнірний відбір

5.4 Конвеєри обробки

- `ChromosomePipeline` (`genetic/chromosome.py`) — обробляє хромосоми, застосовуючи кросовер та мутацію
- `GenomePipeline` (`genetic/genome.py`) — обробляє цілий геном, використовуючи селекцію та `ChromosomePipeline`

6 Апаратна оптимізація

Фреймворк підтримує розрахунки на різних обчислювальних пристроях:

- CPU-реалізації у директорії `genetic/cpu/`
- GPU-реалізації з використанням CUDA у директорії `genetic/gpu/`

6.1 Робочий процес

Типовий процес роботи з фреймворком:

1. Ініціалізація популяції
2. Оцінка фітнес-функції для кожної особини
3. Селекція батьків за допомогою обраної стратегії
4. Створення нащадків через кросовер
5. Застосування мутацій для внесення різноманітності
6. Формування нового покоління
7. Повторення циклу до досягнення умови зупинки

6.2 Гіперпараметри навчання

1. Розмір популяції
2. Сила елітизму
3. Сила мутацій
4. Частота мутацій
5. Тип кросоверу
6. Тип відбору

7 Завантаження датасетів у модулі `loader`

7.1 Структура модуля

Модуль `loader` розділений на наступні компоненти:

- `core` — містить функціональність квантизації для CPU та GPU:
  - `quantize_cpu.py` — реалізує `CPUStateDownSampler` для зменшення розрядності даних на CPU
  - `quantize_gpu.py` — реалізує `CUDAStateDownSampler` для зменшення розрядності даних на GPU
- `manager` — містить класи для управління даними:
  - `data_manager.py` — загальний менеджер даних
  - `downloader.py` — завантаження даних з мережі
  - `sklearn_loader.py` — інтеграція з датасетами scikit-learn [5]

7.2 Головні компоненти

7.2.1 Клас DataManager

- Забезпечує завантаження та обробку даних із зазначеного шляху
- Підтримує зберігання даних як на CPU, так і на GPU
- Дозволяє згрупувати дані в батчі для навчання моделей
- Реалізує квантизацію даних з вказаною розрядністю (`bit_width`)

7.2.2 Клас Downloader

- Завантажує датасет MNIST з інтернету
- Розпаковує стиснуті файли (.gz) у бінарний формат
- Перетворює бінарні дані у формат NumPy [3] для подальшої обробки
- Підтримує декілька дзеркал для завантаження даних

7.2.3 Клас SklearnBalancedDataLoader

- Завантажує датасет рукописних цифр scikit-learn [5] (`load_digits`)
- Забезпечує збалансовану вибірку (рівномірний розподіл класів)
- Підтримує зміну розміру зображень (`resize_to`)
- Квантизує дані під визначену розрядність (`rule_bitwidth`)
- Дозволяє налаштувати різні комбінації обробки та зберігання на CPU/GPU

7.3 Підтримувані датасети

1. **MNIST** — датасет рукописних цифр від 0 до 9
   - 60,000 зображень для тренування
   - 10,000 зображень для тестування
   - Зображення розміром 28×28 пікселів
2. **Scikit-learn digits** — вбудований датасет рукописних цифр scikit-learn [5]
   - Менший за розміром датасет (1,797 зображень)
   - Використовується для швидкого прототипування
   - Зображення розміром 8×8 пікселів

7.4 Функціональні особливості

- **Квантизація даних** — зниження розрядності зображень для зменшення обчислювальних витрат
- **Балансований семплінг** — гарантує рівномірне представлення всіх класів у батчі
- **Паралельна обробка** — підтримка як CPU, так і GPU для обробки та зберігання даних
- **Препроцесинг** — автоматичне перетворення у формат one-hot encoding для міток класів
- **Інтеграція з NML** — безшовна інтеграція з Neural Machine Library для подальшого використання в моделях

7.5 Робочий процес

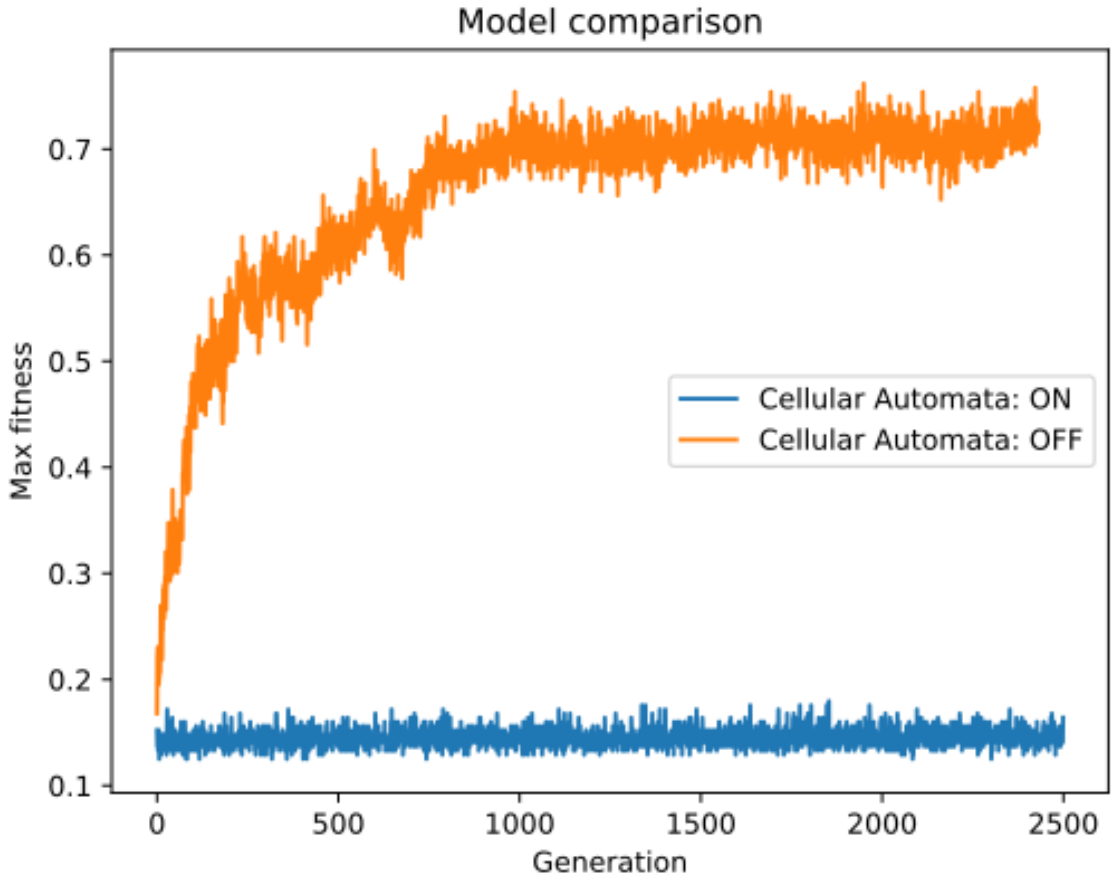
1. Завантаження необхідного датасету (MNIST або scikit-learn [5] digits)
2. Попередня обробка даних (зміна розміру, квантизація)
3. Формування збалансованих батчів для навчання та тестування
4. Передача даних на відповідний обчислювальний пристрій (CPU або GPU)
5. Інтеграція з моделями машинного навчання через клас `DeferredResults`

8 Експерименти

Було проведено величезну кількість експериментів з використаням даних двох мікрофреймворків. Як основна модель для них використовувалась наступна:

```
sequential = Sequential(
    Input((8, 8), np.dtype("uint8")),
    Flatten(),
    Cast(np.dtype("float32")),
    Linear(512),
    ReLU(),
    Linear(10),
    Softmax(),
)
```

Дана модель була побудована на основі моделі з вільного доступу [4]. Проте на певних етапах вона була змінена

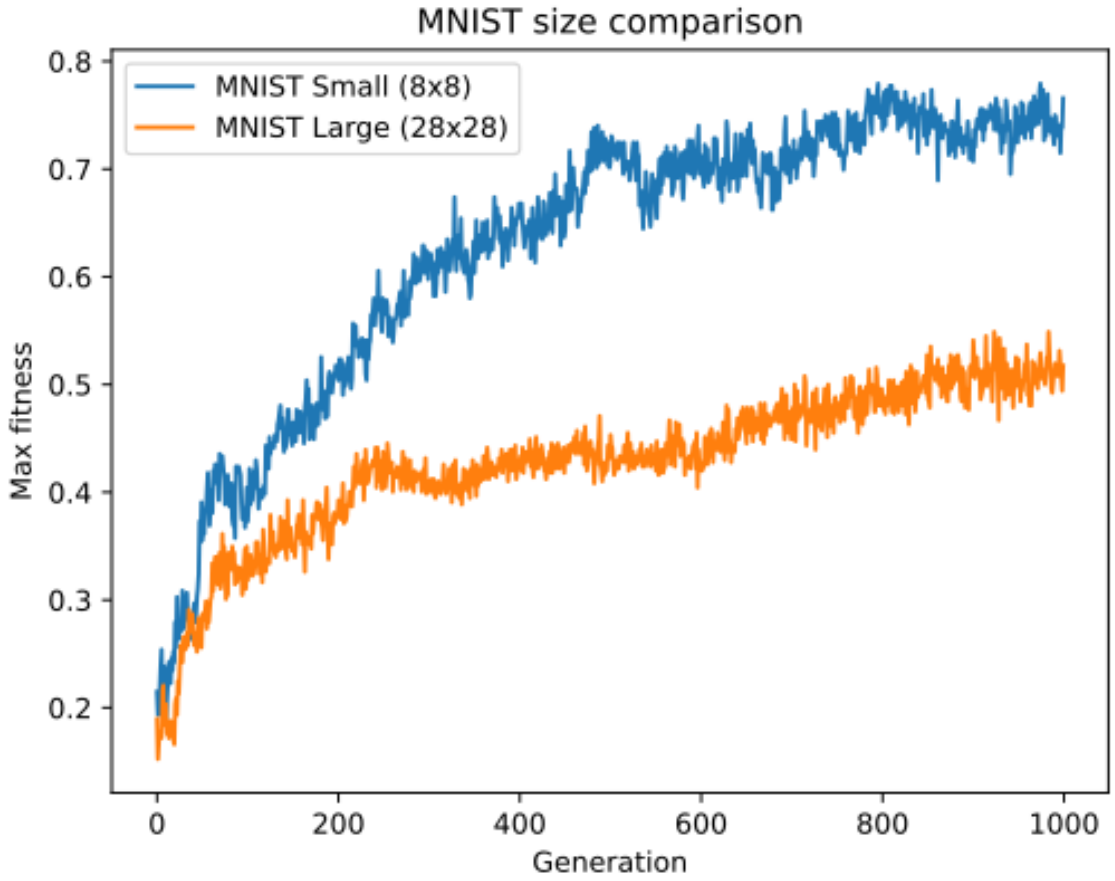


8.1 Порівняння з скінченним автоматом та без

Була також створена модель:

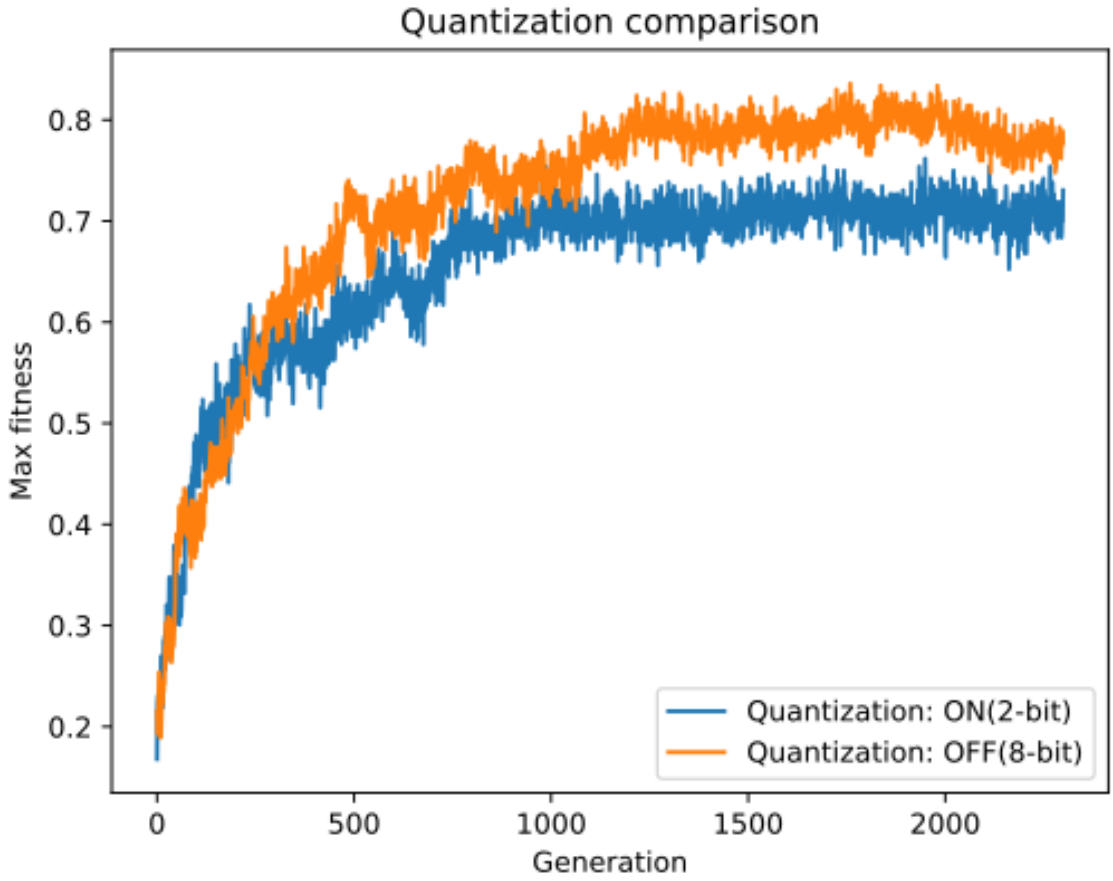
```
sequential = Sequential(
    Input((8, 8), np.dtype("uint8")),
    CellularAutomata(
        rule_bitwidth=2,
        neighborhood="von_neumann_1",
        iterations=1,
    ),
    CellularAutomata(
        rule_bitwidth=2,
        neighborhood="von_neumann_1",
        iterations=1,
    ),
    Flatten(),
    Cast(np.dtype("float32")),
    Linear(512),
    ReLU(),
    Linear(10),
    Softmax(),
)
```

І рис. 8.1 показує результат виконання навчання обох моделей



8.2 Порівняння за розміром датасету

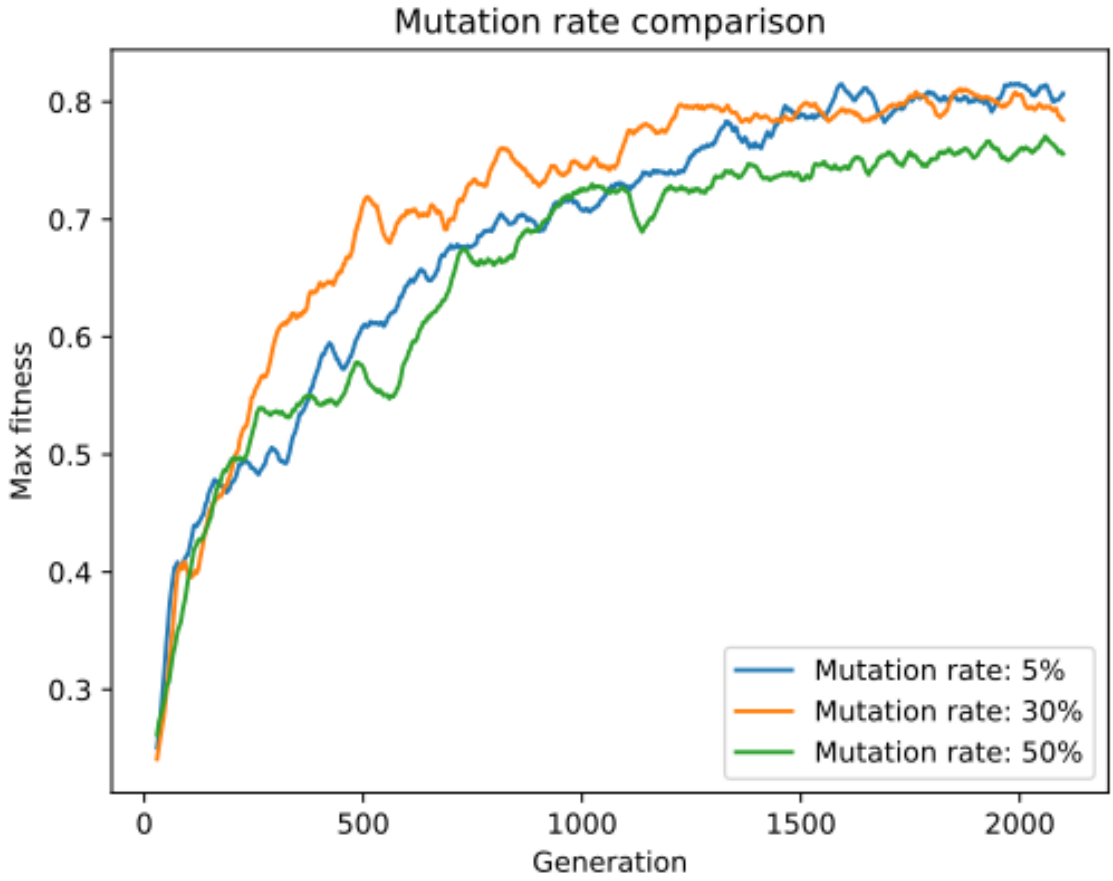
Було порівняно навчання на виликому датасеті `MNIST` та малому. Що показано на рис. 8.2



8.3 Порівняння за квантизацією

Було порівняно ефективність моделі при квантизації до 2-х бітів та вимкненій (8 бітів). Що показано на рис. 8.3

Можна побачити, що втрачається частина інформації.



8.4 Порівняння за рівнем мутації

Було порівняно швидкість навчання при ймовірностях мутацій 5%, 30% та 50%. Що показано на рис. 8.4 (зауважте, використано згладжування rolling average з розміром вікна 30)

9 Результати

9.0.1 Вплив Cellular Automata

Модель не здатна функціонувати, пошуковий простір надто великий для вектора-таблиці пошуку, тому генетичний алгоритм не може навіть почати спускатись у рельєфі фітнес-функції.

9.0.2 Вплив розміру зображення

- Потребує додаткових обчислювальних ресурсів
- Модель справляється гірше через зростання поушкового простору (45% проти 80%)
- На більших зображення модель стабільніша

9.0.3 Вплив квантизації

- Підвищена квантизація (q8) зберігає більше деталей зображення

- Сприяє покращенню максимального фітнесу
- Збільшує стабільність моделі

9.1 Висновок

У цьому проєкті було продемонстровано альтернативний підхід до обробки зображень, заснований на поєднанні клітинних автоматів та генетичних алгоритмів. Розроблена система дозволяє будувати адаптивні моделі без використання традиційних згорткових нейронних мереж, що відкриває нові можливості для дослідження в області обчислювального інтелекту. Створені мікрофреймворки забезпечують гнучкість, модульність та ефективну реалізацію як на CPU, так і на GPU, а також спрощують експериментування з різними конфігураціями алгоритмів.

Наші експерименти, на жаль, лише продемонстрували незданість клітинних автоматів замінити згорткові нейронні мережі через великий простір пошуку та неможливість використання градієнтного спуску. Але через брак часу та технічних ресурсів ми змогли лише перевірити низку конфігурацій, тому ця тема потребує подальших досліджень.

Подальші дослідження можуть бути спрямовані на розширення типів автоматів, вдосконалення фітнес-функцій та оптимізацію параметрів еволюційного процесу.

Джерела

[1] *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. NeurIPS 2019. Посилання: `https://pytorch.org/`

[2] *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015. Посилання: `https://www.tensorflow.org/`

[3] *Array programming with NumPy*. Nature, 585, 357–362 (2020). Посилання: `https://numpy.org/`

[4] Nipun Manral, *MLP Training for MNIST Classification (GitHub Repository)*. Посилання: `https://github.com/nipunmanral/MLP-Training-For-MNIST-Classification`

[5] *Scikit-learn: Machine Learning in Python*. Journal of Machine Learning Research, 12, pp. 2825–2830, 2011. Посилання: `https://scikit-learn.org/`

[6] Numba: A LLVM-based Python JIT compiler. In Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC (pp. 1–6). ACM. Посилання: `https://numba.pydata.org/`