Львівський Національний Університет імені Івана Франка

Факультет електроніки та комп’ютерних технологій

Кафедра системного проектування

Курсова робота

на тему

Дослідження різних варіацій генетичного алгоритму навчання нейромереж змін топології нейромережі нейроеволюцією (NEAT)

Роботу виконав:

Ульянов Олексій

Студент ФЕІм-12м

Науковий керівник:

Шувар Роман Ярославович

завідувач кафедри

системного проектування

Львів 2023

Зміст

1. Анотація....3
Abstract....5
2. Вступ....7
2.1 Об’єкт теми дослідження....7
2.2 Актуальність теми дослідження....7
2.3 Мета дослідження....9
2.4 Завдання дослідження....11
3. Огляд засобів і технологій використаних в роботі....13
3.1 Загальний перелік технологій....13
3.2 Технології для 3д симулятора....16
3.3 Технології для машинного навчання....17
4. Встановлення завдання проєкту....19
4.1 Очікуваний результат....19
4.2 Архітектура нейронної мережі....21
4.3 Навчання нейронної мережі....23
4.4 Визначення ефективності мережі. Фітнес....25
4.5 Hypercube-based NEAT....27
4.6 Compositional Deep NEAT....28
4.7 ES-HyperNEAT....30
4.8 NEAT з пошуком новизни....32
4.9 Багатоцільовий NEAT....34
4.10 Вибір одного алгоритму модифікації NEAT - HyperNEAT....36
5. Опис виконання проєкту....37
5.1 Нейронна мережа....37
5.2 Збереження та відтворення нейронних мереж....41
5.3 Генетичний алгоритм: NEAT....44
5.4 Покращений генетичний алгоритм: HyperNEAT....51
6. Навчання нейронної мережі....56
6.1 Вибір параметрів для NEAT....56
6.2 Вибір параметрів для HyperNEAT....58
6.3 Створення треку для навчання....61
6.4 Складніша задача з важчим треком....63
6.5 Процес навчання на звичайному NEAT....65
6.6 Навчання з HyperNEAT....67
6.7 Порівняння результатів NEAT і HyperNEAT....70
8. Висновок....72
9. Джерела....74
10. Додаток....75

1. Анотація

У цій курсовій роботи ми дослідили вплив різних варіацій та змін NEAT (Neuroevolution of Augmenting Topologies)[1] на результат навчання нейронних мереж. NEAT - це потужний алгоритм, який поєднує нейронні мережі та генетичні алгоритми для еволюції складних мережевих структур. Впроваджуючи та аналізуючи різні варіації NEAT, ми прагнемо зрозуміти, як ці модифікації впливають на продуктивність, збіжність та надійність еволюційних нейронних мереж. Наше дослідження фокусується на HyperNEAT[16], CoDeepNEAT[20], ES-HyperNEAT[21], NEAT з пошуком новизни[22] та багатоцільовому NEAT[23]. Втім задля визначення реального обсягу роботи, ми порівняли між собою алгоритми, та зупинились на реалізації HyperNEAT.

Методологія дослідження передбачає реалізацію кожного варіанту в середовищі Unity3D з використанням C# для кодування нейронних мереж та алгоритмів NEAT. В якості експериментального завдання використовується симуляція водіння автомобіля з метою навчання автомобіля автономному керуванню. Кожна варіація тестується за допомогою декількох ітерацій еволюції, вимірюючи ключові показники ефективності, такі як пройдена відстань, середня швидкість та успішне виконання завдань.

Результати дослідження демонструють порівняльну ефективність варіації HyperNEAT з точки зору його здатності розвивати ефективні структури нейронних мереж. Аналіз включає оцінку швидкості збіжності, складності мережі та адаптивності до змін навколишнього середовища. Крім того, для вивчення структурної еволюції нейронних мереж протягом поколінь використовуються інструменти візуалізації.

Результати цього дослідження дають цінну інформацію про сильні та слабкі сторони HyperNEAT, пропонує рекомендації щодо вибору та впровадження конкретних модифікацій NEAT на основі бажаних цілей еволюції нейронної мережі. Дослідження сприяє розумінню того, як варіація NEAT, HyperNEAT може впливати на ефективність та результативність еволюції нейронних мереж, особливо в контексті завдань автономного водіння.

Програму реалізували з використанням рушія Unity3d, мова програмування — C#, у середовищі JetBrains Riders.

Наведено результати реалізації програми.

Опис роботи містить:

75 сторінки, 40 рисунків, 1 додаток та 23 бібліографічних найменувань за переліком посилань.

Ключові слова: NEAT, нейроеволюція, нейронні мережі, HyperNEAT, CoDeepNEAT, ES-HyperNEAT, пошук новизни, багатоцільовий NEAT, автономне керування, Unity3D, C#

Abstract

In this course work, we investigated the effect of different variations and changes of NEAT (Neuroevolution of Augmenting Topologies)[1] on the training result of neural networks. NEAT is a powerful algorithm that combines neural networks and genetic algorithms to evolve complex network structures. By implementing one and analyzing different variations of NEAT, we aim to understand how these modifications affect the performance, convergence, and robustness of evolutionary neural networks. Our research focuses on HyperNEAT[16], CoDeepNEAT[20], ES-HyperNEAT[21], NEAT with novelty search[22], and multi-objective NEAT[23]. However, in order to determine the real scope of work, we compared the algorithms and focused on the HyperNEAT implementation.

The research methodology involves the implementation of each option in the Unity3D environment using C# to code neural networks and NEAT algorithms. As an experimental task, a car driving simulation is used to train a car for autonomous driving. Each variation is tested using several evolution iterations, measuring key performance indicators such as distance traveled, average speed, and successful task completion.

The results of the study demonstrate the comparative effectiveness of HyperNEAT variations in terms of its ability to evolve efficient neural network structures. The analysis includes an assessment of convergence speed, network complexity, and adaptability to environmental changes. In addition, visualization tools are used to study the structural evolution of neural networks over generations.

The results of this study provide valuable information about the strengths and weaknesses of HyperNEAT, and offer recommendations for selecting and implementing specific NEAT modifications based on the desired goals of neural network evolution. The study contributes to the understanding of how a variation of NEAT, HyperNEAT, can affect the efficiency and effectiveness of neural network evolution, especially in the context of autonomous driving tasks.

The program was implemented using the Unity3d engine, C# programming language, in the JetBrains Riders environment.

The results of the program implementation are presented.

The description of the work contains:

75 pages, 40 figures, 1 appendix and 23 bibliographic references.

Keywords: NEAT, neuroevolution, neural networks, HyperNEAT, CoDeepNEAT, ES-HyperNEAT, novelty search, multi-purpose NEAT, autonomous control, Unity3D, C#

2. Вступ

2.1 Об’єкт теми дослідження

Еволюція штучних нейронних мереж зробила революцію в галузі машинного навчання та штучного інтелекту, уможлививши розробку інтелектуальних систем, здатних вирішувати складні завдання. Традиційні методи навчання нейронних мереж часто покладаються на градієнтні алгоритми оптимізації, які мають обмеження при роботі з великорозмірними та нелінійними проблемами. Як альтернативний підхід, алгоритми нейроеволюції, які поєднують генетичні алгоритми з нейронними мережами, з'явилися як перспективні методи для еволюції структур нейронних мереж та оптимізації їх вагових коефіцієнтів.

Одним із впливових алгоритмів нейроеволюції є NEAT (Neuroevolution of Augmenting Topologies), запропонований Стенлі та Міккулайненом у 2002 році. NEAT представляє унікальну методологію, яка дозволяє одночасно еволюціонувати вагам і структурам нейронної мережі. Явно кодуючи шаблони зв'язків нейронних мереж, NEAT дозволяє досліджувати різноманітні мережеві топології та сприяє еволюції складних та ефективних рішень для широкого кола проблем.

2.2 Актуальність теми дослідження

В останні роки NEAT привернув до себе значну увагу і успішно застосовується в різних сферах, включаючи робототехніку, ігри, оптимізацію та системи управління. Здатність до автономного пошуку та розвитку мережевих архітектур зробила NEAT особливо придатним для задач, які вимагають адаптивної та інтелектуальної поведінки.

Хоча NEAT забезпечує потужну основу для еволюції нейронних мереж, дослідники вивчили і запропонували кілька варіацій і модифікацій для підвищення його продуктивності та можливостей. Ці модифікації спрямовані на вирішення конкретних проблем або використання додаткових концепцій для подальшого підвищення ефективності та результативності NEAT в еволюції нейронних мереж.

Ця робота зосереджена на дослідженні та аналізі цих варіацій NEAT, щоб отримати всебічне розуміння їхнього впливу на процес еволюції нейронних мереж. Зокрема, ми прагнемо оцінити продуктивність, швидкість збіжності та надійність різних варіацій NEAT при застосуванні до задачі водіння автомобіля в середовищі моделювання Unity3D. Вивчаючи поведінку та характеристики еволюційних нейронних мереж у кожній варіації, ми прагнемо виявити сильні та слабкі сторони, надаючи цінну інформацію для вибору та впровадження конкретних модифікацій NEAT у практичних додатках.

2.3 Мета дослідження

Метою цього дослідження є вивчення та порівняння різних варіацій та змін NEAT в контексті еволюції нейронних мереж для задач автономного водіння. Вивчаючи ці варіації, ми прагнемо пролити світло на їхній вплив на продуктивність, збіжність та адаптивність нейронних мереж, що еволюціонують.

Основними завданнями цього дослідження є наступні:

1. Оцінити продуктивність: Ми прагнемо оцінити продуктивність нейронних мереж, створених з використанням різних варіацій NEAT у симуляції водіння автомобіля. Для вимірювання ефективності еволюційних мереж будуть використовуватися такі показники, як пройдена відстань, середня швидкість та успішне виконання завдань з водіння. Порівнюючи продуктивність різних варіантів NEAT, ми прагнемо визначити, які варіанти призводять до кращих результатів з точки зору ефективності водіння.
2. Аналіз швидкості збіжності: Розуміння швидкості збіжності варіантів NEAT має вирішальне значення для оцінки їхньої ефективності. Ми розглянемо, як швидко еволюціонували нейронні мережі досягають задовільного рівня продуктивності протягом поколінь. Порівнюючи швидкість збіжності різних варіацій NEAT, ми можемо визначити, які з них сприяють швидшій збіжності і, отже, більш ефективній еволюції мережі.
3. Оцініть надійність та адаптивність: Надійність і адаптивність є критично важливими якостями в автономних системах. Ми дослідимо, наскільки добре еволюціонували нейронні мережі з кожної варіації NEAT адаптуються до мінливого середовища або сценаріїв в симуляції водіння автомобіля. Ця оцінка проллє світло на здатність еволюційних мереж узагальнювати свою поведінку під час водіння та ефективно справлятися з несподіваними ситуаціями.

4. Дослідження мережевих структур: На додаток до оцінки продуктивності, ми дослідимо структурну еволюцію нейронних мереж між поколіннями в кожній варіації NEAT. Візуалізуючи та аналізуючи зміни в топології мережі, ми прагнемо отримати уявлення про те, як різні варіації впливають на складність, модульність та схеми зв'язків еволюційних мереж. Цей аналіз надасть цінну інформацію про архітектурні характеристики та механізми, що лежать в основі еволюції мереж.

2.4 Завдання дослідження

Цілі цього дослідження полягають у наступному:

1. Дослідити найпопулярніші модифікації алгоритму NEAT, порівняти їх, та обрати один для подальшої реалізації, щоб порівняти його результати з базовим NEAT.

2. Реалізувати та інтегрувати обраний алгоритм модифікації NEAT у симулятор водіння автомобіля в Unity3D з використанням C#. Це передбачає кодування нейронних мереж та алгоритмів NEAT, забезпечуючи їх належну функціональність у середовищі симуляції.

3. Оцінити та порівняти продуктивність еволюційних нейронних мереж у обраній варіації NEAT. Для оцінки ефективності еволюційних мереж будуть використовуватися такі показники, як пройдена відстань, середня швидкість та успішне виконання завдань з водіння. Порівнюючи результати, ми прагнемо визначити чи модифікацій NEAT здатна покращити результат навчання.

4. Проаналізувати та порівняти швидкість збіжності варіанта NEAT з базовим алгоритмом. Швидкість збіжності - це швидкість, з якою нейронні мережі досягають задовільного рівня продуктивності протягом поколінь еволюції.

5. Оцінити надійність та адаптивність еволюційних нейронних мереж у відповідь на зміну середовища або сценаріїв в рамках симуляції водіння автомобіля. Ця оцінка дасть уявлення про те, наскільки добре еволюціонували мережі узагальнюють поведінку водія та ефективно справляються з несподіваними ситуаціями. Порівнюючи адаптивність між різними варіантами NEAT, ми прагнемо визначити варіанти, які дають більш надійні та адаптивні мережі.

6. Дослідити та проаналізувати структурну еволюцію нейронних мереж через покоління для обраної варіації NEAT. Цей аналіз передбачає вивчення складності, модульності та моделях зв'язків. Вивчаючи

архітектурні характеристики еволюційних мереж, ми прагнемо отримати уявлення про вплив різних варіацій NEAT на мережеві структури.

7. Надати цінну інформацію та рекомендації щодо вибору та впровадження конкретних варіацій NEAT в контексті еволюції нейронних мереж для завдань автономного водіння. Результати цього дослідження допоможуть дослідникам і практикам у виборі найбільш підходящих варіацій NEAT на основі бажаних цілей еволюції мережі.

Досягаючи цих цілей дослідження, ми прагнемо зробити внесок у розуміння сильних сторін, обмежень і компромісів, пов'язаних з різними варіаціями NEAT.

У наступних розділах цієї роботи ми надамо ретельний огляд відповідної літератури, включаючи NEAT та його варіації. Потім ми представимо використану методологію, описавши налаштування симуляції водіння автомобіля в Unity3D, деталі реалізації обраного варіанту NEAT та використані метрики оцінювання. Потім ми представимо та обговоримо результати, отримані в ході наших експериментів, після чого проведемо всебічний аналіз отриманих висновків. Насамкінець ми підсумуємо основні результати, обговоримо їхні наслідки та пропонуємо майбутні напрямки для подальших досліджень.

3. Огляд засобів і технологій використаних в роботі

3.1 Загальний перелік технологій

У цьому розділі ми надаємо огляд інструментів і технологій, що використовуються для розробки та аналізу системи автономного водіння на основі NEAT. Вибір відповідних інструментів і технологій має вирішальне значення для успішного впровадження та оцінки системи. Тут ми описуємо ключові інструменти та технології, що використовувалися в нашій роботі:

Ігровий рушій Unity3D[3]:

Наше середовище розробки базується на ігровому рушію Unity3D (версія 2020.1.15f1 Personal). Unity3D надає надійну та універсальну платформу для створення інтерактивних симуляцій та віртуальних середовищ. Він пропонує широку підтримку для рендерингу графіки, фізичного моделювання та управління активами. Ми використовуємо Unity3D для створення середовища симуляції водіння автомобіля, що дозволяє нам оцінити продуктивність розвинених нейронних мереж в реалістичному та інтерактивному середовищі.

Мова програмування C#:

Unity3D сумісна з мовою програмування C#, і ми широко використовуємо C# для реалізації нейронних мереж, алгоритмів NEAT та інших необхідних функцій. C# - це потужна об'єктно-орієнтована мова програмування, яка надає широкий спектр бібліотек та фреймворків для ефективної розробки. Використовуючи C# у нашій реалізації, ми можемо скористатися перевагами її багатої екосистеми та легко інтегрувати її з Unity3D.

Vehicle Physics Pro[4]:

Для програмування фізики автомобіля в середовищі Unity3D ми використовуємо бібліотеку Vehicle Physics Pro (версія 1.2). Vehicle Physics Pro пропонує повний набір функцій фізичного моделювання, розроблених спеціально для динаміки транспортних засобів. Включаючи Vehicle Physics Pro в нашу реалізацію, ми можемо точно моделювати поведінку і рух автомобіля, підвищуючи реалістичність симуляції водіння.

Easy Roads 3D[5]:

Для створення та моделювання доріг у середовищі Unity3D ми використовуємо бібліотеку Easy Roads 3D (версія 3.1.4). Easy Roads 3D надає інтуїтивно зрозумілі інструменти та функції для проектування та створення реалістичних дорожніх мереж. Використовуючи цю бібліотеку, ми можемо ефективно створювати дорожні структури та визначати шляхи руху для автономного автомобіля в симуляції.

Tableau[6]:

Для візуалізації та аналізу даних, отриманих під час експериментів, ми використовуємо Tableau, потужний інструмент візуалізації та аналізу даних. Tableau дозволяє нам створювати інформативні та інтерактивні візуалізації, що дають змогу отримати уявлення про зібрані дані. Завдяки широким можливостям Tableau ми можемо ефективно аналізувати показники продуктивності нейронних мереж, що еволюціонували, та представляти результати у чіткій та лаконічній формі.

JetBrains Rider[7]:

Вся розробка коду для цього проекту виконується за допомогою інтегрованого середовища розробки (IDE) JetBrains Rider (версія 2020.2.4). JetBrains Rider надає потужне та зручне середовище для кодування різними мовами програмування, включаючи C#. Він пропонує розширені

можливості редагування коду, інструменти налагодження та безшовну інтеграцію з системами контролю версій.

Git з GitHub та Git LFS[8]:

Для контролю версій та співпраці ми використовуємо Git як систему контролю версій. Ми розміщуємо наше сховище коду на GitHub і використовуємо Git LFS (Large File Storage) для ефективної роботи з великими бінарними файлами. Git та GitHub дозволяють безперешкодно співпрацювати, відстежувати версії та легко керувати змінами коду протягом усього процесу розробки.

Використовуючи ці інструменти та технології, ми забезпечуємо надійний та ефективний робочий процес для впровадження, аналізу та візуалізації системи автономного водіння на основі NEAT. Вони дозволяють нам створити реалістичне середовище для симуляції водіння, моделювати фізику і дорожню структуру, ефективно аналізувати дані, записувати візуальні демонстрації і підтримувати контроль версій для ефективної співпраці.

У наступних розділах ми заглибимося в деталі методології реалізації, експериментальних установок та методів аналізу, демонструючи, як ці інструменти та технології використовуються в нашому дослідженні.

3.2 Технології для 3д симулятора

3D-симулятор, обраний для цього дослідження, - це Unity3D, широко використовуваний ігровий рушій, який надає фреймворк для запуску скриптів на C# або JavaScript. Unity3D пропонує значні переваги при роботі з тривимірною фізикою завдяки потужній логіці та зручним інструментам для роботи з тривимірною графікою. Рішення про використання Unity3D зумовлене його придатністю для створення реалістичного середовища для симуляції водіння.

Для точного моделювання фізики автомобілів у середовищі Unity3D ми використали бібліотеку Vehicle Physics Pro. Ця бібліотека відома своїми потужними можливостями, що дозволяють точно налаштовувати різні фізичні параметри для наближення до реальних автомобілів. Вона дозволяє програмувати окремі педалі, такі як педаль акселерації, педаль гальма та педаль зчеплення. Крім того, він надає можливість програмувати ручні, автоматичні та напівавтоматичні трансмісії, різні системи приводу, функції двигуна, а також такі фактори, як тертя, опір коченню, аеродинаміка тощо. Використовуючи Vehicle Physics Pro, ми мали на меті створити високоточну фізичну модель автомобіля для нашої симуляції автономного водіння.

Для побудови дорожньої мережі в рамках симуляції ми використали бібліотеку Easy Roads 3D. Ця бібліотека пропонує зручний інтерфейс та зручні інструменти для побудови доріг на основі серії точок. Вона дозволяє створювати складні дорожні структури, включаючи багаторівневі дороги та перехрестя. Easy Roads 3D спрощує процес моделювання доріг, забезпечуючи ефективне проектування та інтеграцію в середовище Unity3D.

Використовуючи Unity3D, Vehicle Physics Pro та Easy Roads 3D, ми створили комплексну систему моделювання, яка полегшила розробку

реалістичного середовища для водіння. Ця комбінація інструментів надала нам необхідні можливості для програмування точної фізики автомобіля, моделювання складних дорожніх мереж та створення захоплюючого досвіду симуляції для навчання та оцінки нейронних мереж в контексті автономного водіння.

3.3 Технології для машинного навчання

Реалізація нейронних мереж та генетичного алгоритму для цього дослідження буде проводитися за допомогою нативної мови C# без використання додаткових бібліотек. Такий підхід дозволяє підвищити гнучкість та прозорість кодової бази, що дає змогу глибше зрозуміти основні операції, які виконуються над матрицями.

Для підтримки матричних операцій в рамках реалізації ми розробимо власну полегшену бібліотеку матриць. Створюючи власну бібліотеку, ми зможемо адаптувати її до потреб нашого дослідження, забезпечуючи сумісність з реалізаціями нейронних мереж та генетичних алгоритмів. Така бібліотека матриць забезпечує необхідний функціонал для маніпуляцій з матрицями, таких як множення, додавання, віднімання та інші операції, що мають вирішальне значення для навчання нейронних мереж та роботи генетичних алгоритмів.

Рішення розробити власну бібліотеку матриць відображає нашу мету мати більший контроль над реалізацією та сприяти більш повному розумінню компонентів нейронних мереж та генетичних алгоритмів. Створюючи власну бібліотеку, ми можемо налаштовувати та оптимізувати матричні операції відповідно до вимог алгоритмів, тим самим покращуючи продуктивність і забезпечуючи прозорий аналіз основних обчислень.

Таким чином, нейронні мережі та компоненти генетичних алгоритмів будуть реалізовані без використання зовнішніх бібліотек, що дозволяє глибше зрозуміти основні процеси. Крім того, буде розроблена

спеціальна бібліотека матриць для полегшення ефективної роботи з матрицями і збереження гнучкості та прозорості протягом усього процесу реалізації.

4. Встановлення завдання проєкту

4.1 Очікуваний результат

Основною метою цього дослідження є розробка бібліотеки нейронних мереж та їх навчання за допомогою алгоритму NEAT, і його однієї варіації покращення. Після завершення проекту ми очікуємо отримати навчену нейронну мережу, яка демонструє кваліфікаційні навички водіння в симульованому середовищі. Ця навчена нейронна мережа буде зберігатися у форматі JSON, готова до використання для завдань автономного водіння.

Зокрема, ми прагнемо досягти наступного очікуваного результату:

1. **Розробка бібліотеки нейронних мереж:**
   Наша мета - створити з нуля комплексну та універсальну бібліотеку нейронних мереж, реалізовану на мові C#. Ця бібліотека буде включати в себе основні функціональні можливості для побудови, навчання та оцінки нейронних мереж. Розробляючи власну бібліотеку, ми можемо адаптувати її до специфічних вимог алгоритму NEAT і налаштовувати її за потреби для досягнення оптимальної продуктивності та гнучкості.
2. **Навчання нейронної мережі за допомогою NEAT та обраною модифікацією:**
   Використовуючи реалізовану бібліотеку нейронних мереж, ми застосуємо алгоритм NEAT (Neuroevolution of Augmenting Topologies) для навчання нейронної мережі для автономного водіння. NEAT поєднує еволюційні обчислення зі штучними нейронними мережами, що дозволяє еволюціонувати топології мережі та вагу зв'язків. За допомогою ітеративної еволюції ми прагнемо підвищити ефективність водіння нейронної мережі шляхом оптимізації її структури та параметрів.

3. **Генерація навченої нейронної мережі:**

   Після успішного навчання наша мета - отримати навчену нейронну мережу, яка демонструє професійні навички навчання. Навчена нейронна мережа буде фіксувати набуті знання та стратегії для ефективного автономного водіння в змодельованому середовищі. Ми зберігатимемо навчену нейронну мережу у форматі JSON, що забезпечує її сумісність та портативність для подальшого використання.

4. **Демонстрація задовільних результатів водіння:**

   Кінцевим очікуваним результатом є спостереження за тим, як навчена нейронна мережа рухається на достатній швидкості, точно проходячи повороти в змодельованому середовищі. Навчена мережа повинна продемонструвати компетентні навички прийняття рішень, демонструючи свою здатність керувати автомобілем автономно та ефективно вирішувати завдання водіння.

Досягнувши цих очікуваних результатів, ми прагнемо зробити свій внесок у розвиток систем автономного водіння з використанням нейронних мереж, навчених за алгоритмом NEAT. Результати цього дослідження дадуть уявлення про можливості та обмеження реалізованої бібліотеки нейронних мереж, а також про ефективність алгоритму NEAT у покращенні ефективності водіння.

4.2 Архітектура нейронної мережі

Наша нейронна мережа має традиційну архітектуру прямого поширення, що включає три типи шарів: вхідний, вихідний та прихований. Кожен шар складається з нейронів, причому в кожному шарі присутній принаймні один нейрон. Нейрони в межах шару повністю з'єднані з нейронами в наступному і попередньому шарах, за винятком вхідного і вихідного шарів. Вхідний шар отримує зовнішні входи, такі як дані з датчиків, тоді як вихідний шар з'єднується з інтерфейсом керування автомобілем, який діє як привід системи. Кожен зв'язок між нейронами асоціюється зі скалярною вагою, і кожен нейрон використовує функцію активації.

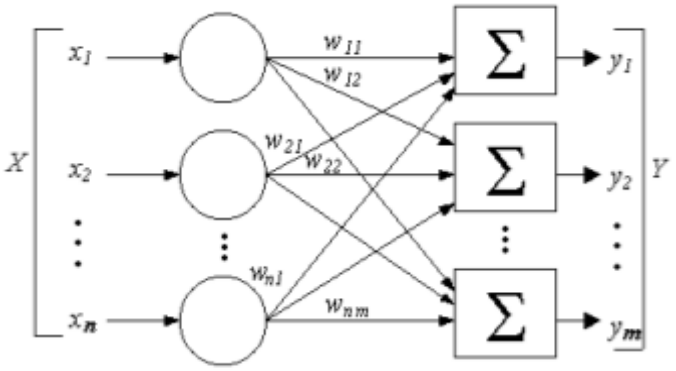

Рис. 4.2.1 Схема будови одношарової нейронної мережі

Під час роботи нейронної мережі значення датчиків слугують вхідними даними для мережі. Значення наступного шару обчислюються шляхом підсумовування добутків ваг зв'язків, пов'язаних з кожним нейроном, на відповідні значення попереднього шару. Додатково застосовується функція активації для отримання кінцевого значення. Цей процес виконується рекурсивно для всіх нейронів, поки не будуть отримані значення нейронів вихідного шару. Згодом ці значення передаються на виконавчий механізм, що призводить до виконання бажаної зовнішньої дії.

Крім того, кожен нейрон включає в себе нейрон "зміщення", який утримує постійне значення. При обчисленні значення нейрона, це значення

зміщення додається, щоб відповідним чином змістити рівень його активації. Цей нейрон зсуву відіграє важливу роль у забезпеченні додаткового рівня гнучкості та контролю нейронної мережі.

Суть навчання нейронної мережі полягає у визначенні оптимальних ваг для всіх нейронів, включаючи нейрон зміщення. Процес навчання включає в себе коригування значень ваг для того, щоб гарантувати, що для ряду вхідних сценаріїв мережа видає бажані результати на виході. Ітеративно змінюючи ваги, нейронна мережа вчиться апроксимувати складні зв'язки між вхідними та вихідними даними та покращує свою продуктивність.

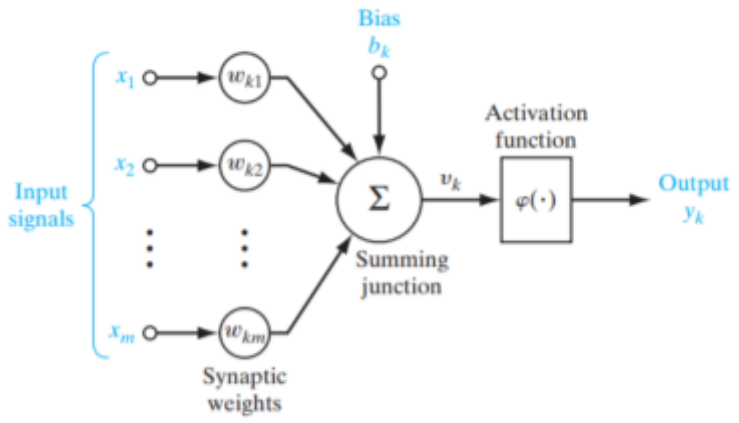

Рис. 4.2.2 Схема роботи нейронної мережі

4.3 Навчання нейронної мережі

Для навчання ми реалізували алгоритм NEAT (Neuro-Evolution of Augmenting Topologies - нейроеволюція доповнюючих топологій)[11]. NEAT - це генетичний алгоритм, який черпає натхнення в біологічних процесах, таких як генетика та природний відбір, що робить його ефективним підходом для навчання нейронних мереж.

Як згадувалося раніше, основною метою навчання нейронних мереж є визначення оптимальних ваг для всіх нейронів. Однак використання методів грубої сили для вичерпного пошуку цих ваг було б не практичним і трудомістким, особливо для великомасштабних систем. Щоб вирішити цю проблему і прискорити процес навчання, зменшивши при цьому випадковість, було розроблено алгоритм NEAT.

NEAT використовує принципи біології та еволюції, запозичуючи термінологію та концепції з цих галузей. Давайте заглибимося в деякі з ключових понять NEAT.

У контексті генетичного алгоритму кожен екземпляр нейронної мережі представляє певний результат і називається фенотипом. У контексті навчання фенотип часто називають частковим генотипом. Результатом генотипу є агент, який у нашому випадку представляє машину або автопілот - об'єкт, який ми прагнемо навчити.

Згідно з NEAT[12], навчання відбувається ітеративно, кожна ітерація називається популяцією або поколінням. Кожне покоління складається з певної кількості генотипів, які є основними одиницями алгоритму.

Іншим важливим поняттям в рамках NEAT є поняття гена, який є складовою частиною генотипу. Таким чином, генотип складається з декількох генів, кожен з яких відповідає за певний аспект поведінки або функціональності нейронної мережі.

Фундаментальним аспектом алгоритму NEAT є фітнес і пов'язана з ним функція фітнесу. Фітнес - це реальна скалярна величина, яка кількісно визначає ефективність агента. Агенти з вищими значеннями фітнесу

вважаються більш ефективними. Фітнес-функція оцінює ефективність агента на основі його поведінки, і побудова відповідної фітнес-функції є вирішальним кроком у навчанні нейронної мережі.

На додаток до цих концепцій, NEAT включає в себе два життєво важливих процеси: кросовер і мутацію. Кросинговер передбачає створення нового генотипу, який називається дочірнім, шляхом поєднання генів двох батьківських генотипів. Частина генів вибирається від першого батька, а інша частина - від другого. У нашому випадку ми спростимо модель, розглядаючи всі фенотипи як одностатеві і не використовуючи хромосоми або статі.

Мутація - це процес, який вносить випадковість під час кросинговеру шляхом випадкової зміни генів. Цей процес дозволяє створювати нові гени і дає можливість досліджувати альтернативні рішення і потенційно відкривати більш оптимальні конфігурації [13].

4.4 Визначення ефективності мережі. Фітнес

Як ми вже говорили раніше, фітнес є фундаментальним і дуже важливим поняттям, адже він визначає “чого ми хочемо від нашого штучного інтелекту”. Отже, спочатку ми маємо дати відповідь на питання “що штучний інтелект має зробити? якими факторами можемо описати його результат? наскільки для нас є важливим чи пріоритетним той чи інший фактор?”

Ми очікуємо від нашого штучного інтелекту, щоб він міг керувати машиною з точки A в точку B. Оскільки в умовах задачі ми відмовились від створення навігаційної системи, то шлях A-B, ми розбиваємо відрізки *n* точками C_1, C_2, ... , C_{n-1}, C_n = B. Тому наш шлях виглядатиме після розбиття: A-C_1-C_2-...-C_{n-1}-B. Тобто ми очікуємо, що автомобіль зможе доїхати від точки A, до точки C_1, потім від точки C_1 до точки C_2, …, потім від точки C_{n-2} до точки C_{n-1}, і нарешті з точки C_{n-1} до точки B.

Перший фактор і найголовніший це фактор пройденої відстані. Він буде показувати, наскільки наш агент наблизився до наступної точки. Для шляху A-C_1 будемо рахувати його за формулою:

$$S_{A-C_1} = \left|\overline{AC_1} - \overline{PC_1}\right|, \tag{4.4.1}$$

де P*(x, y, z)* - точка поточної позиції агента, а $S_{A\text{-}C1}$ - пройдений шлях відстані між точками A і C_1.

Далі для середніх точок:

$$S_{C_{i-1}-C_i} = \left|\overline{C_{i-1}C_i} - \overline{PC_i}\right|, \ (i = \overline{1, n-1}), \tag{4.4.2}$$

І для останнього шляху:

$$S_{C_{n-1}-B} = \left|\overline{C_{n-1}B} - \overline{PB}\right|, \tag{4.4.3}$$

І загальний пройдений шлях A-B, *S* буде виглядати наступним чином:

$$S = S_{A-C_1} + \sum_{i=1}^{n-1} S_{C_{i-1}-C_i} + S_{C_{n-1}-B} = \left|\overline{AC_1} - \overline{PC_1}\right| + \sum_{i=1}^{n-1} |\overline{C_{i-1}C_i} - \overline{PC_i}| + + \left|\overline{C_{n-1}B} - \overline{PB}\right|. \quad (4.4.4)$$

Наступний фактор - це фактор середньої швидкості. Оскільки для нас важливо як швидко автомобіль буде рухатись між точками, то на потрібно визначити фактор середньої швидкості. Її будемо рахувати так:

$$< v > = S / \Delta t, \quad (4.4.5)$$

де <v> - середня швидкість, S - весь пройдений шлях, Δt - часовий проміжок між початком руху і поточним часом.

Нехай це будуть наші основні фактори фітнесу, з ними тепер можемо визначити фітнес-функцію *F,* яка залежить від параметрів машини *P*:

$$F(P) = S \cdot w_S + < v > \cdot w_{<v>}, \quad (4.4.6)$$

де w_S *і* $w_{<v>}$ - коефіцієнти “важливості” факторів пройденого шляху і середньої швидкості відповідно.

Також є випадки коли ми будемо вважати, що агент не впорався зі задачею і будемо перезапускати, залишивши останній фітнес. В нашій задачі це випадки, коли агент покине дорогу, або здійснить зіткнення корпусом, або не буде показувати оптимальної ефективності (ΔF) в скінченний період часу (Δt), або якщо загальний час, виділений на задачу, буде надто великим (t_{max}).

4.5 Hypercube-based NEAT

Тепер розглянемо першу модифікацію NEAT, відому як HyperNEAT[17] (NEAT на основі гіперкуба, з англ. “Hypercube-based NEAT”). HyperNEAT розширює можливості NEAT за рахунок використання іншого методу еволюції топології нейронної мережі. На відміну від NEAT, який розвиває мережі з прямими зв'язками, HyperNEAT працює на іншому рівні, розвиваючи патерни, які диктують зв'язки всередині нейронної мережі. Він використовує концепцію під назвою "гіперкуб" (Hypercube substrate), де нейрони розташовані у сіткоподібній структурі. Розвиваючи зв'язки між вузлами цієї сітки, HyperNEAT може створювати складні та ефективні топології нейронних мереж.

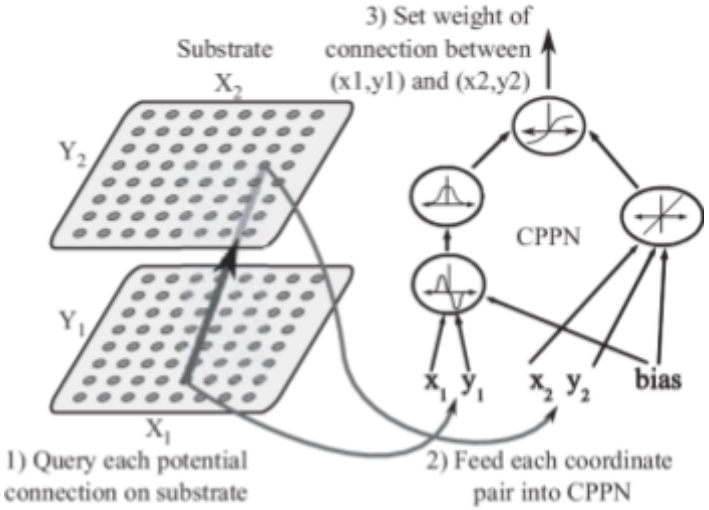

Рис. 4.5.1 Схема підходу HyperNEAT

Алгоритм HyperNEAT забезпечує вищий рівень абстракції, що дозволяє розвинутим нейронним мережам демонструвати складні патерни зв'язків і отримувати вигоду від модульності та регулярності[18]. Цей підхід виявився перспективним у вирішенні складних завдань, що вимагають просторової і часової обробки, що робить його особливо придатним для таких застосувань, як автономне водіння.

Цей підхід підвищує здатність нейронних мереж вирішувати складні завдання і продемонстрував успіх у сферах, що вимагають просторової та часової обробки[19].

4.6 Compositional Deep NEAT

Продовжуючи наше дослідження варіацій та модифікацій алгоритму NEAT, ще один вартий уваги підхід - CoDeepNEAT[20] (Compositional Deep NeuroEvolution of Augmenting Topologies - композиційна глибока нейроеволюція доповнюючих топологій). CoDeepNEAT розширює принципи NEAT для вирішення проблем розвитку складних архітектур нейронних мереж у глибокому навчанні.

Глибоке навчання зробило революцію в різних галузях, уможлививши навчання нейронних мереж з декількома шарами, що дозволяє їм вивчати ієрархічні представлення даних. Однак розробка та оптимізація архітектури глибоких нейронних мереж вручну може бути складним завданням. CoDeepNEAT має на меті вирішити цю проблему шляхом автоматизації процесу еволюції структур глибоких нейронних мереж.

У CoDeepNEAT нейронні мережі складаються з модулів або підмереж, де кожен модуль являє собою автономну частину мережі. Ці модулі можна дублювати, з'єднувати і комбінувати для формування більших і складніших нейромережевих архітектур. Розвиваючи ці модулі та їхні зв'язки, CoDeepNEAT може генерувати різноманітні та ефективні архітектури глибоких нейронних мереж.

Процес еволюції структур глибоких нейронних мереж в CoDeepNEAT включає два ключові етапи: мутація модулів і кросовер модулів. Мутація модулів дозволяє модифікувати існуючі модулі шляхом додавання або видалення нейронів і зв'язків всередині модуля. Цей процес забезпечує варіативність і дозволяє досліджувати різні архітектурні конфігурації.

Кросовер модулів, з іншого боку, полегшує створення нових архітектур глибоких нейронних мереж шляхом об'єднання модулів з різних батьківських мереж. Ця операція передбачає вибір модулів з кожної батьківської мережі та створення нової мережі, яка успадковує їхні модулі та зв'язки. Комбінуючи модулі різними способами, CoDeepNEAT

досліджує простір можливих архітектур і сприяє появі нових та ефективних глибоких нейронних мереж.

Еволюція архітектур глибоких нейронних мереж в CoDeepNEAT керується фітнес-функцією, яка оцінює продуктивність кожної мережі. Фітнес-функція враховує різні фактори, такі як точність, обчислювальна ефективність і використання пам'яті, в залежності від конкретного завдання або проблеми, що розглядається. Мережі з вищими показниками фітнес-функції вважаються більш ефективними і з більшою ймовірністю будуть відібрані для розмноження і подальшої еволюції.

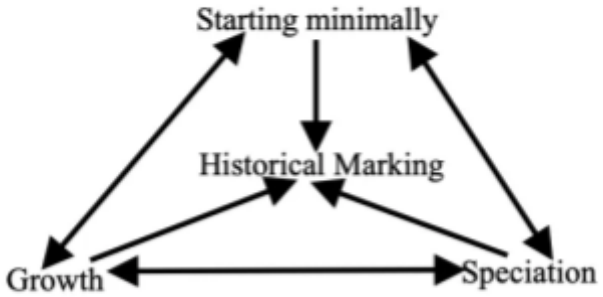

Рис. 4.6.1 Схема деталей еволюції

CoDeepNEAT продемонстрував свою ефективність у різних сферах, включаючи розпізнавання зображень, обробку природної мови та навчання з підкріпленням. Автоматизуючи процес проектування архітектур глибоких нейронних мереж, CoDeepNEAT дозволяє дослідникам і практикам більш ефективно досліджувати величезний простір проектування і знаходити архітектури, які досягають найсучаснішої продуктивності.

Ця модифікації фокусується на розвитку архітектур глибоких нейронних мереж. Використовуючи композиційну природу глибокого навчання, CoDeepNEAT вводить модулі як будівельні блоки і застосовує операції мутації і кросинговеру для дослідження і розвитку різноманітних і ефективних структур глибоких нейронних мереж. Такий підхід спрощує процес проектування моделей глибокого навчання і має великий потенціал для розвитку галузі штучного інтелекту.

4.7 ES-HyperNEAT

Продовжуючи дослідження варіацій алгоритму NEAT, ми звернемо увагу на ES-HyperNEAT[21] (Evolution Strategy HyperNEAT). ES-HyperNEAT поєднує в собі принципи еволюційних стратегій (ES) та HyperNEAT для вирішення проблем еволюції великомасштабних архітектур нейронних мереж.

Evolution Strategies - це потужний метод оптимізації, який використовує ідеї еволюційних алгоритмів та природної еволюції для вирішення складних задач. Вона фокусується на оптимізації параметрів заданого рішення шляхом ітеративного застосування мутації та відбору. ES виявився ефективним у високорозмірних пошукових просторах і був успішно застосований до різних областей.

ES-HyperNEAT використовує переваги як ES, так і HyperNEAT для розвитку архітектури нейронних мереж. Він розширює алгоритм HyperNEAT за рахунок включення стратегій еволюції як засобу оптимізації зв'язків і ваг еволюційних мереж.

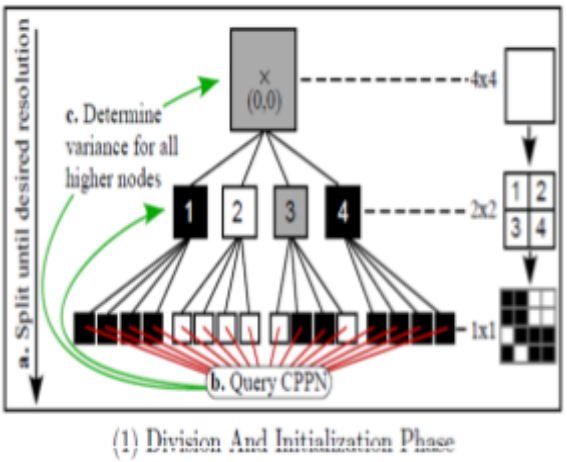

Рис. 4.7.1 Фаза ділення

В ES-HyperNEAT архітектури нейронних мереж, що еволюціонують, базуються на субстраті Hypercube, подібному до того, що використовується в HyperNEAT. Однак, замість прямого кодування зв'язків, ES-HyperNEAT використовує схему генеративного кодування для представлення зв'язності мережі. Це кодування складається з набору параметрів, які визначають структуру зв'язків між нейронами.

В процесі еволюції ES-HyperNEAT використовує стратегії еволюції для оптимізації параметрів генеративного кодування. Він застосовує операції мутації та рекомбінації до параметрів і оцінює отримані мережеві архітектури за допомогою функції пристосованості, специфічної для конкретної задачі. Найбільш придатні архітектури обираються для подальшого дослідження та вдосконалення, поступово покращуючи загальну продуктивність еволюційних мереж.

Однією з ключових переваг ES-HyperNEAT є його здатність працювати з великомасштабними архітектурами нейронних мереж. Використовуючи стратегії еволюції, які добре підходять для оптимізації у високій розмірності, ES-HyperNEAT може ефективно досліджувати та оптимізувати величезний простір пошуку складних нейромережевих структур. Ця здатність робить його особливо цінним для задач, що вимагають великих мереж, таких як обробка зображень, розуміння природної мови та складні проблеми управління.

ES-HyperNEAT успішно застосовується в різних галузях, включаючи робототехніку, ігри та “комп'ютерний зір”. Поєднання стратегій еволюції та HyperNEAT дозволяє розвивати складні архітектури нейронних мереж з покращеною продуктивністю та ефективністю.

Таким чином, ES-HyperNEAT - це модифікація алгоритму HyperNEAT, яка включає стратегії еволюції для оптимізації зв'язків і ваг великомасштабних архітектур нейронних мереж. Використовуючи сильні сторони обох методів, ES-HyperNEAT забезпечує ефективне рішення для еволюції складних мереж і показав себе перспективним у вирішенні складних завдань в області штучного інтелекту.

4.8 NEAT з пошуком новизни

Продовжуючи досліджувати варіації алгоритму NEAT, ми заглиблюємося в NEAT з пошуком новизни[22]. NEAT з пошуком новизни поєднує принципи NEAT з концепцією пошуку новизни, щоб заохотити пошук різноманітних та інноваційних рішень.

Традиційні алгоритми оптимізації, включаючи NEAT, часто покладаються на фітнес-підхід, коли люди оцінюються на основі їхньої ефективності у вирішенні конкретного завдання. Однак такий підхід може призвести до збіжності в неоптимальних рішеннях або обмежити дослідження нових і потенційно цінних моделей поведінки.

Пошук новизни пропонує альтернативну перспективу, акцентуючи увагу на дослідженні поведінкової новизни, а не лише на виконанні завдання. Основна ідея пошуку новизни полягає в тому, щоб підтримувати певну міру новизни для кожної особини на основі унікальності її поведінки в популяції. Це спонукає алгоритм досліджувати ширший діапазон поведінки та знаходити нестандартні рішення.

У NEAT з пошуком новизни алгоритм включає метрику новизни поряд з традиційною оцінкою фітнесу. В процесі еволюції особини оцінюються на основі їхньої здатності виконувати завдання, а також новизни їхньої поведінки порівняно з існуючою популяцією. Це заохочує відкриття різноманітних стратегій і сприяє дослідженню нових регіонів у просторі пошуку.

Метрика новизни в NEAT з функцією пошуку новизни може бути обчислена за допомогою різних методів, таких як вимірювання евклідової відстані між векторами поведінки або порівняння схожості активацій нейронної мережі. Вибір конкретного методу залежить від характеристик проблемної області та природи поведінки, що оцінюється.

Поєднуючи пошук новизни з NEAT, алгоритм здатний досягти балансу між використанням хороших рішень і дослідженням нових і недосліджених областей простору пошуку. Це може призвести до

відкриття інноваційних стратегій, непередбачуваної поведінки та потенційно більш надійних і адаптивних рішень.

NEAT з функцією пошуку новизни успішно застосовується в різних галузях, включаючи робототехніку, еволюційне мистецтво та ігрову інженерію. Його здатність заохочувати дослідження і появу нових моделей поведінки виявилася цінною в тих галузях, де простір пошуку є великим, складним або невизначеним.

Таким чином, NEAT з пошуком новизни - це варіант алгоритму NEAT, який включає в себе концепцію пошуку новизни для заохочення пошуку різноманітних та інноваційних рішень. Враховуючи як виконання завдання, так і новизну поведінки, алгоритм може виявляти нетрадиційні стратегії і сприяти появі нових моделей поведінки. Такий підхід розширює пошукові можливості NEAT і є перспективним для вирішення складних і відкритих проблем.

4.9 Багатоцільовий NEAT

Продовжуючи наше дослідження варіацій алгоритму NEAT, ми переходимо до багатоцільового NEAT[23]. Багатоцільовий NEAT - це розширення алгоритму NEAT, яке дозволяє оптимізувати декілька конфліктуючих цілей одночасно.

У багатьох реальних проблемах є кілька цілей, які потрібно оптимізувати, і ці цілі часто є конфліктуючими, або потрібно розглянути компроміси між ними. Традиційні одноцільові алгоритми оптимізації, такі як стандартний NEAT, не дуже добре підходять для роботи з декількома цілями, оскільки вони, як правило, зосереджені на пошуку одного оптимального рішення.

Багатоцільовий NEAT усуває це обмеження, впроваджуючи методи багатоцільової оптимізації для пошуку декількох Парето-оптимальних рішень. Парето-оптимальні рішення являють собою набір рішень, в яких не можна досягти покращення в одній цілі, не жертвуючи при цьому продуктивністю в іншій цілі.

У багатоцільовому методі NEAT популяція еволюціонує з різноманітного набору особин, які представляють різні компроміси між кількома цілями. Це досягається шляхом присвоєння значення пристосованості кожній особині на основі її продуктивності в кожній цілі. Ці значення пристосованості використовуються для ранжування і відбору особин для розмноження і подальшої еволюції.

Одним із поширених підходів, що використовується в багатоцільовому методі NEAT, є використання домінування Парето, який порівнює продуктивність особин у кожній цілі без явного об'єднання їх в єдине значення пристосованості. Особини, які домінують над іншими принаймні в одній задачі і не домінують в жодній іншій, вважаються оптимальними за Парето і формують фронт Парето.

Щоб підтримувати різноманітність у популяції, Multiobjective NEAT використовує такі методи, як елітизм, відстань скупчення та нішування. Елітарність гарантує, що найкращі особини в кожній задачі зберігаються з

покоління в покоління. Дистанція скупченості заохочує людей досліджувати різні регіони простору цілі, запобігаючи зближенню до єдиного рішення. Нічінг сприяє збереженню різноманітних рішень, призначаючи штраф за скупченість популяції.

Застосування Multiobjective NEAT охоплює різні сфери, включаючи проблеми оптимізації, еволюційну робототехніку та сценарії прийняття рішень. Він забезпечує потужну основу для вивчення та аналізу компромісів між кількома цілями, дозволяючи особам, які приймають рішення, робити обґрунтований вибір на основі ряду Парето-оптимальних рішень.

Таким чином, Multiobjective NEAT розширює алгоритм NEAT для одночасної роботи з декількома конфліктуючими цілями. Використовуючи методи багатоцільової оптимізації, він дозволяє знаходити оптимальні за Парето рішення, що дає змогу особам, які приймають рішення, шукати компроміси та робити обґрунтований вибір. Багатоцільова NEAT має широке застосування в різних сферах, де оптимізація декількох цілей є важливою.

4.10 Вибір одного алгоритму модифікації NEAT - HyperNEAT

В цьому розділі ми дослідили декілька алгоритмів нейроеволюції та порівняли їх з базовим алгоритмом NEAT. Було розглянуто такі алгоритми: HyperNEAT, CoDeepNEAT, ES-HyperNEAT, NEAT з пошуком новизни та багатоцільовий NEAT.

HyperNEAT, варіант NEAT, представив концепцію нейроеволюції на основі субстратів, що дозволяє еволюціонувати більш складну поведінку, визначаючи субстрати і множинні шари. CoDeepNEAT розширив HyperNEAT для еволюції глибоких нейронних мереж, що дозволило вивчати ієрархічні особливості. ES-HyperNEAT поєднав стратегії еволюції з HyperNEAT для оптимізації параметрів мережі, підвищуючи її продуктивність. NEAT з пошуком новизни включав механізм пошуку новизни для заохочення дослідження і відкриття нової поведінки. Багатоцільовий NEAT поєднує в собі кілька фітнес-цілей для еволюції агентів, які добре працюють за різними критеріями одночасно.

З цих алгоритмів ми вирішили зосередитися саме на HyperNEAT на пізніх етапах курсової роботи через його здатність генерувати різноманітні та складні архітектури нейронних мереж за допомогою еволюції на основі субстрату. Цей підхід є перспективним для вирішення більш складних навігаційних завдань і сприяє появі інтелектуальної та адаптивної поведінки.

Хоча кожен алгоритм має свої унікальні особливості та переваги, HyperNEAT виділяється своїм потенціалом перевершити базовий алгоритм NEAT з точки зору продуктивності та адаптивності. Рішення зосередитися на HyperNEAT дозволило нам глибше зануритися в субстратний підхід і дослідити його наслідки для навчання штучних агентів у складних навігаційних сценаріях.

5. Опис виконання проєкту

5.1 Нейронна мережа

Основною метою є розробка нейронної мережі, яка може бути легко налаштована без зміни коду, враховуючи нетривіальність вибору топології мережі. Мета полягає в тому, щоб мати можливість гнучко змінювати кількість прихованих шарів і нейронів у них за потреби.

Для досягнення цієї мети використовується підхід матричного представлення, який спрощує реалізацію та обчислення нейронної мережі. Середовище моделювання, хоча і є важливим для процесу навчання, не є основною темою цієї статті, оскільки основна увага приділяється дослідженню нейронної мережі та різних реалізацій алгоритму NEAT.

Дизайн нейронної мережі використовує структуру класів, яка охоплює необхідні компоненти для гнучкості та ефективності. Клас включає одновимірну матрицю для вхідних нейронів, одновимірну матрицю для вихідних нейронів, тензор (список матриць) для прихованих шарів, ваговий тензор, список зсувів і набір функцій активації для вихідного шару. Використання матричних операцій спрощує та оптимізує обчислення мережі.

Оскільки кожний нейрон дає на виході якесь число, то кожне число можна представити елементом матриці. Ми отримуємо числа на вхід, тому представимо загальний випадок вхідного шару:

$$I = \begin{pmatrix} i_1 \\ i_2 \\ \dots \\ i_n \end{pmatrix} = \begin{pmatrix} f(x_1) \\ f(x_2) \\ \dots \\ f(x_n) \end{pmatrix}, \qquad (5.1.1)$$

де $x_1, x_2, \dots, x_n$ - вхідні числа, $f(x)$ - функція активації; $i_1, i_2, \dots, i_n$ - значення нейронів вхідного шару.

По аналогії для прихованого першого шару:

$$H_1 = \begin{pmatrix} h_1 \\ h_2 \\ \dots \\ h_n \end{pmatrix}, \tag{5.1.2}$$

де $h_1, h_2, \dots, h_n$ - значення нейронів прихованого шару 1.

Тому ми можемо представити матрицю для ваг між нейронами вхідного шару і першого прихованого як добуток:

$$I \times W_1 = H_1 \tag{5.1.3}$$

За таким принципом:

$$\begin{array}{c} \\ i_1 \\ i_2 \\ \dots \\ i_n \end{array} \begin{array}{c} \begin{array}{cccc} h_1 & h_2 & \dots & h_n \end{array} \\ \begin{pmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ w_{21} & w_{22} & \dots & w_{2n} \\ \dots & \dots & \dots & \dots \\ w_{n1} & w_{n2} & \dots & w_{nn} \end{pmatrix} \end{array} \tag{5.1.4}$$

І за подібним принципом так для всіх прихованих шарів, аж до останнього прихованого і вихідного. Відповідно ми зможемо рахувати результат нейронної мережі, лише перемноживши всі матриці послідовно, що значно спрощує обчислення.

Отже, наш клас нейронної мережі складається з одномірної матриці вхідних нейронів та вихідних нейронів, з тензора (списку матриць) прихованих шарів, та тензору ваг, а також список біасів, та набір функції активація для вихідного шару. Механічно біас це просто скаляр, який є у кожного шару, і ми додаємо його до результівної матриці.

```
Frequently called  9 usages
public Matrix InputLayer { get; private set; }
Frequently called  11 usages
public List<Matrix> HiddenLayers { get; private set; }
Frequently called  5 usages
public Matrix OutputLayer { get; private set; }
Frequently called  23 usages
public List<Matrix> Weights { get; private set; }
Frequently called  13 usages
public List<double> Biases { get; private set; }
Frequently called  5 usages
public Func<double, double>[] ActivationsFunctions { get; private set; }
```

Рис. 5.1.1 Лістинг полів класу NeuralNetwork

Нейронна мережа ініціалізується шляхом присвоєння значень усім відповідним полям у конструкторі. Метод Init() забезпечує сумісність розмірів матриць і генерує випадкові ваги для мережі.

```
[JsonConstructor]
Frequently called  2 usages  devnull
private NeuralNetwork(int inputsAmount, int outputsAmount, Func<double, double>[] functions)
{
    InputLayer = new Matrix( rows: 1,  cols: inputsAmount);
    HiddenLayers = new List<Matrix>();
    OutputLayer = new Matrix( rows: 1,  cols: outputsAmount);
    Weights = new List<Matrix>();
    Biases = new List<double>();
    ActivationsFunctions = functions;
}

Frequently called  2 usages  devnull
public void Init(int inputsAmount, int[] neuronsInHiddenLayersAmount, int outputsAmount,
    Func<double, double>[] activationsFunctions)
{
    Validate(neuronsInHiddenLayersAmount, activationsFunctions.Length, outputsAmount);
    ActivationsFunctions = activationsFunctions;

    GenerateWeightsMatrices(inputsAmount, neuronsInHiddenLayersAmount);

    var weightsLastHiddenLayerToOutputLayer =
        new Matrix( rows: neuronsInHiddenLayersAmount[neuronsInHiddenLayersAmount.Length - 1],  cols: outputsAmount);
    Weights.Add(weightsLastHiddenLayerToOutputLayer);
    Biases.Add( item: Random.Range(-1f, 1f));

    GenerateWeightsValues();
}
```

Рис. 5.1.2 Лістинг конструктора і ініціалізатора класу

Алгоритм роботи нейронної мережі починається із завдання вхідних даних та проходження їх через функцію активації. Значення для кожного

прихованого нейрона в кожному шарі обчислюються послідовно, переходячи від вхідного шару до вихідного. Операції множення матриць, полегшені використанням матриць, спрощують алгоритм нейронної мережі та підвищують обчислювальну ефективність.

```
public double[] Run(double[] inputs)
{
    if (inputs.Length != InputLayer.ColsCount)
        throw new ArgumentException( message: $"Wrong inputs size! Neural network has {InputLayer.RowsCount} inputs.");

    for (var i = 0; i < inputs.Length; i++)
    {
        InputLayer[0, i] = inputs[i];
    }

    InputLayer = InputLayer.PointwiseTanh();

    HiddenLayers[0] = (InputLayer * Weights[0] + Biases[0]).PointwiseTanh();

    for (var i = 1; i < HiddenLayers.Count; i++)
    {
        HiddenLayers[i] = (HiddenLayers[i - 1] * Weights[i] + Biases[i]).PointwiseTanh();
    }

    OutputLayer = (HiddenLayers[HiddenLayers.Count - 1] * Weights[Weights.Count - 1] + Biases[Biases.Count - 1])
        .PointwiseTanh();

    var output = OutputLayer.ToOneDimensionalArray();
    for (var i = 0; i < ActivationsFunctions.Length; i++)
    {
        output[i] = ActivationsFunctions[i](output[i]);
    }

    return output;
}
```

Рис. 5.1.3 Лістинг методу запуску нейронної мережі

У цьому розділі ми описали підхід до проектування та реалізації нейронної мережі, використаної в нашому дослідженні. Методологія матричного представлення спрощує мережеві обчислення, дозволяючи гнучко налаштовувати топологію мережі. Використовуючи цей дизайн, ми можемо зосередитися на дослідженні різних реалізацій алгоритму NEAT та аналізі їхнього впливу на ефективність навчання нейронної мережі.

5.2 Збереження та відтворення нейронних мереж

Коли в процесі навчання ми досягнемо бажаного результату, було б добре зберегти нейронну мережу так, щоб її потім можна було відтворити. Що нам потрібно, щоб працювала нейронна мережа? Її ядром є топологія, а топологія визначається набором ваг між нейронами всіх шарів та біасів. Отже, ми збережемо нейронну мережу, якщо збережемо всі ваги й біаси в їхньому порядку. Зберігати будемо в текстових файлах у форматі JSON.

Підтримка JSON в Unity3d на рівні скриптів вийшла недавно, а її функціонал дуже обмежений, тому для реалізації цього механізму, нам не підійде вбудована бібліотека, натомість використаємо бібліотеку Newtonesoft.Json для C#.

```
Frequently called  2 usages  ldevinull
public static void SerializeNeuralNetwork(NeuralNetwork neuralNetwork, double fitness)
{
    if (!Directory.Exists( path: "logs"))
    {
        Directory.CreateDirectory( path: "logs");
    }

    var path :string  = GenerateNeuralNetworkFileName(fitness);
    using (var file :StreamWriter  = File.CreateText(path))
    {
        var dto = new NeuralNetworkDto(neuralNetwork);
        Serializer.Serialize(file, dto);
    }
}
```

Рис. 5.2.1 Лістинг методу серіалізації нейромережі

З JSON є один нюанс. Оскільки в топології ми використовуємо числа з плаваючою комою доволі великого бітового розряду, то кожна цифра для нас грає роль, втім специфіка JSON-формату полягає в тому, що він має здатність втрачати точність дробових чисел. Це не є нашою провиною, такий задум Господа (або Дугласа Крокфорда), проте ми можемо це обійти, якщо перед серіалізацією ми будемо переводити на рівні C# число в рядок, а після зчитування, засобами C# рядок в число. Таким чином ми не будемо втрачати точності. Даний підхід інкапсулюємо в окремий клас

для “витягування” необхідних для топології даних нейронної мережі NeuralNetworkDTO, який реалізує патерн проєктування Data Transfer Object:

```
public NeuralNetworkDto(NeuralNetwork neuralNetwork)
{
    var biasesString :List<string> = neuralNetwork.Biases.Select(b :double => "" + b).ToList();
    var weightsString :List<string[][]> = neuralNetwork.Weights // List<Matrix>
        .Select(w :Matrix => w
            .ToMultiDimensionalArray()
            .Select(row :double[] => row.Select(col :double => "" + col).ToArray())
            .ToArray()) // IEnumerable<string[][]>
        .ToList();

    Weights = weightsString;
    Biases = biasesString;
}

public List<Matrix> GetWeight()
{
    return Weights // List<string[][]>
        .Select(w :string[][] => w.Select(row :string[] => row.Select(double.Parse).ToArray()).ToArray()) // IEnumerable<double[][]>
        .Select(w :double[][] => new Matrix( rows: w.Length, cols: w[0].Length, w)) // IEnumerable<Matrix>
        .ToList(); // List<Matrix>
}
```

Рис. 5.2.2 Лістинг об’єкту для перенесення “важливих” даних нейромережі

Зчитування є зворотним процесом і не вимагає додаткових пояснень:

```
public static NeuralNetwork DeserializeNeuralNetwork(CarAI carAI)
{
    if (carAI.jsonFile == null)
    {
        throw new NullReferenceException( message: "File path cannot be null if you want to deserialize network!");
    }

    var deserializeObject = JsonConvert.DeserializeObject<NeuralNetworkDto>(carAI.jsonFile.text);
    return NeuralNetwork.Of(deserializeObject, carAI.inputsAmount, carAI.neuronsInHiddenLayerCount,
        carAI.outputsAmount, carAI.ActivationsFunctions);
}
```

Рис. 5.2.3 Лістинг методу десеріалізації нейромережі з файла

Збережена нейронна мережа текстом виглядає ось так:


```
{
  "Weights": [
    [
      [ ],
      [ ],
      [ ],
      [ ],
      [ ],
      [ ],
      [ ],
      [ ],
      [ ],
      [ ],
      [ ],
      [
        "0.168363894329834",
        "-0.679984569549561",
        "0.950144298924872",
        "0.181647419929584",
        "0.685534477233887",
        "0.877615857124328 6",
        "-0.227086663246155",
        "-0.0644367933273315"
      ]
    ],
    [ ],
    [
      [ ],
      [ ],
      [ ],
      [ ],
      [ ],
      [ ],
      [ ],
      [
        "1",
        "-0.58320140838623",
        "0.454947233200073"
      ]
    ]
  ],
  "Biases": [
    "0.0428553819656372",
    "0.794884811477661",
    "-0.528147697448 73"
  ]
}
```


Рис. 5.2.4 Приклад вигляду серіалізованої нейромережі у файл

Тобто як ми бачимо, вона зберігає всі ваги й біаси які присутні в нашій мережі. Це топологія нашої нейронної мережі.

5.3 Генетичний алгоритм: NEAT

У процесі навчання ШІ ми обрали генетичний алгоритм NEAT (NeuroEvolution of Augmenting Topologies), який дозволяє змінювати топологію нейронної мережі. Алгоритм отримав свою назву завдяки натхненню в біології, що відображає його основні принципи. Щоб надати стислий огляд, ми ініціюємо алгоритм, генеруючи заздалегідь визначену кількість агентів, або геномів, що складають покоління. Ці геноми піддаються симуляції, де фіксуються їхні фінальні значення пристосованості. Згодом обирається підмножина найбільш пристосованих геномів для створення нового потомства шляхом кросинговеру, що формує наступне покоління. Крім того, існує низька ймовірність генних мутацій у геномах, що дозволяє періодично здійснювати генетичні перезавантаження. Цей ітеративний процес триває до тих пір, поки не буде отримано агента або декілька агентів, які демонструють бажану поведінку.

З математичної точки зору, поєднання генів найбільш пристосованих агентів у кожному поколінні збільшує ймовірність отримання геномів, що перевершують їхні батьківські геноми. Включення мутацій, хоч і нечастих, захищає від можливості стагнації, спричиненої потенційними недоліками в генах нових поколінь.

Окресливши основні принципи алгоритму, наш наступний крок - реалізація алгоритму NEAT шляхом створення класу NEAT_Manager та програмування пов'язаних з ним функцій. Фундаментальним компонентом, що вимагається, є генерація, яка включає в себе екземпляри архітектур нейронних мереж. Крім того, ми визначаємо різні параметри, включаючи розмір покоління, ймовірність мутації, кількість найкращих і найгірших агентів для вибірки, бажану кількість генотипів нащадків у результаті кросинговеру, зсув у бік першого батька під час кросинговеру, а також набір генів, що визначають структуру і ваги нейронної мережі.

Визначивши ці базові елементи, ми закладаємо основу для подальшої реалізації та виконання алгоритму NEAT.

```
[Header("References")]
[SerializeField] private CarAI carAI;  Changed in 1 asset

[Header("NEAT Control")]
[SerializeField] private int initialPopulation;  "85"

[SerializeField] [Range(0f, 1f)] private double mutationChance;  "0.0549999997019767776"
[SerializeField] private int fitnessMultiplier = 10;  Unchanged

[Header("Crossover Control")]
[SerializeField] private int bestAgentSelection = 8;  Unchanged
[SerializeField] private int worstAgentSelection = 3;  Unchanged
[SerializeField] private int numberToCrossover;  "39"
[SerializeField] [Range(0, 1)] private double crossoverChance = 0.5;  Unchanged

[Header("Public View")]
[SerializeField] private int currentGeneration = 1;  "0"
[SerializeField] private int currentGenome = 1;  "0"
[SerializeField] private double bestFitness;  Unchanged

private readonly List<int> _genesPool = new List<int>();
private int _naturallySelected;

private NeuralNetwork[] _population;
```

Рис. 5.3.1 Лістинг полів класу NEAT_Manager (генетичного алгоритму)

Ініціалізація. При старті програми, виконується метод *CreatePopulation()*. Спочатку ми ініціалізуємо першу генерацію, та заповнюємо її випадковими нейронними мережами. Після цього ми передаємо в інтерфейс автомобіля першу нейронну мережу генерації й запускаємо її проганяти.

```
    CreatePopulation();
}

1 usage  idevnull
private void CreatePopulation()
{
    _population = new NeuralNetwork[initialPopulation];
    RandomizePopulation(_population,  startingIndex: 0);
    ResetToCurrentGenome();
}

Frequently called  2 usages  idevnull *
private void RandomizePopulation(NeuralNetwork[] newPopulation, int startingIndex)
{
    while (startingIndex < initialPopulation)
    {
        newPopulation[startingIndex] = NeuralNetwork.Of(carAI.inputsAmount, carAI.outputsAmount, carAI.ActivationsFunctions);
        newPopulation[startingIndex++].Init(carAI.inputsAmount, carAI.neuronsInHiddenLayerCount, carAI.outputsAmount,
            carAI.ActivationsFunctions);
    }
}

Frequently called  1 usage  idevnull
private void ResetToCurrentGenome()
{
    carAI.ResetWithNeuralNetwork(_population[currentGenome]);
}
```

Рис. 5.3.2 Лістинг ініціалізації класу NEAT

Ось так виглядає обробник цієї події в класі інтерфейсу автомобіля *CarAI*.

```
Frequently called   1 usage   new *
public void ResetWithNeuralNetwork(NeuralNetwork net)
{
    _network = net;
    Reset();
}
```

Рис. 5.3.3 Обробник події смерті в класі CarAI

На цьому робота генетичного алгоритму поки, що завершується. Він виконує свою роботу під час того, як інтерфейс машини посилає нам сигнал про смерть, після чого він обробляє його так:

```
Frequently called   1 usage   Idevinull
public void Death(double fitness)
{
    UpdateBestFitness(fitness);

    if (currentGenome < _population.Length - 1)
    {
        _population[currentGenome++].Fitness = fitness;
        ResetToCurrentGenome();
    }
    else
    {
        Repopulate();
    }
}
```

Рис. 5.3.4 Лістинг обробника смерті агента

```
Frequently called   3 usages   Idevinull
private void ResetToCurrentGenome()
{
    carAI.ResetWithNeuralNetwork(_population[currentGenome]);
}
```

Рис. 5.3.5 Тригер дії над carAI

Коли агент посилає сигнал про свою смерть, то генетичний алгоритм зберігає його фітнес, і передає агенту новий генотип (нейронну мережу). Це друга дія генетичного алгоритму. Наступну дію він робить, коли агент “від’їздив” всі фенотипи, після чого запускається метод *Repopulate*, а з ним і процес створення нової генерації. Загалом цей етап виглядає наступним чином:

```
private void Repopulate()
{
    SerializationHelper.SerializeStats(currentGeneration, _population, ref _generatedStatsNameFile);
    _genesPool.Clear();
    currentGeneration++;
    _naturallySelected = 0;

    SortPopulation();
    var newPopulation = SelectBestPopulation();
    Crossover(newPopulation);
    Mutate(newPopulation);
    RandomizePopulation(newPopulation, _naturallySelected);
    _population = newPopulation;
    currentGenome = 0;
    ResetToCurrentGenome();
}
```

Рис. 5.3.6 Лістинг методу створення нової генерації

Селекція. Спочатку ми сортуємо всі фенотипи, які ми отримали, за фітнесом по спаданню. Після чого ми вибираємо найкращих і найгірших агентів і додаємо в нову генерацію. Також за кожного вибраного агента, ми додаємо деякі гени в набір. Чим більший сумарний фітнес, тим більше генів.

```
private NeuralNetwork[] SelectBestPopulation()
{
    var newPopulation = new NeuralNetwork[initialPopulation];
    for (var i = 0; i < bestAgentSelection; i++)
    {
        newPopulation[_naturallySelected] = _population[i].Clone() as NeuralNetwork;
        newPopulation[_naturallySelected++].Fitness = 0;

        var f = Mathf.RoundToInt((float)_population[i].Fitness * fitnessMultiplier);

        for (var j = 0; j < f; j++)
        {
            _genesPool.Add(i);
        }
    }

    for (var i = 0; i < worstAgentSelection; i++)
    {
        var last = _population.Length - 1;
        last -= i;

        var f = Mathf.RoundToInt((float)_population[last].Fitness * fitnessMultiplier);

        for (var j = 0; j < f; j++)
        {
            _genesPool.Add(last);
        }
    }

    return newPopulation;
}
```

Рис. 5.3.7 Лістинг методу селекції

Етап **рекомбінації**, також відомий як **схрещування**, має важливе значення в процесі нейроеволюції. Його мета полягає у використанні найкращих фенотипів попереднього покоління для того, щоб створити один або декілька покращених фенотипів для нового покоління. Давайте тепер заглибимося в його реалізацію.

У циклічному процесі з кроком два, враховуючи, що ми будемо схрещувати пари генотипів, ми вибираємо два випадкових гени з набору всіх генів, які були попередньо ідентифіковані на етапі відбору. Далі ми створюємо та ініціалізуємо два нових екземпляри нейронних мереж. Із заздалегідь визначеною ймовірністю, наприклад, 50%, ми переносимо частину генів - зокрема, ваги між нейронами - від першого або другого батька двом дочірнім генотипам. Той самий процес повторюється для зсувів. Після завершення цього процесу новоутворені генотипи додаються до нової популяції.

Варто зазначити, що існує безліч варіацій техніки кросинговеру, і для наших цілей ми обрали відносно простий підхід. Хоча такої реалізації повинно бути достатньо для нашої конкретної задачі, також можна експериментувати з альтернативними стратегіями кросинговеру, щоб дослідити потенційні покращення.

Виконуючи етап рекомбінації, ми використовуємо переваги найкращих генотипів, прокладаючи шлях до створення нових і потенційно кращих нащадків для наступного покоління**.**

```
private void Crossover(NeuralNetwork[] newPopulation)
{
    for (var i = 0; i < numberToCrossover; i += 2)
    {
        int firstParentIndex = i, secondParentIndex = i + 1;

        if (_genesPool.Count >= 1)
        {
            var randomIndex1 = Random.Range(0, _genesPool.Count);
            int randomIndex2;
            do
            {
                randomIndex2 = Random.Range(0, _genesPool.Count);
            } while (_genesPool[randomIndex1] == _genesPool[randomIndex2]);

            firstParentIndex = _genesPool[randomIndex1];
            secondParentIndex = _genesPool[randomIndex2];
        }

        var children = new[] {
            NeuralNetwork.Df(carAI.inputsAmount, carAI.outputsAmount, carAI.ActivationsFunctions),
            NeuralNetwork.Df(carAI.inputsAmount, carAI.outputsAmount, carAI.ActivationsFunctions)
        };

        foreach (var child in children)
        {
            child.Init(carAI.inputsAmount, carAI.neuronsInHiddenLayerCount, carAI.outputsAmount, carAI.ActivationsFunctions);
            child.Fitness = 0;
        }

        for (var w = 0; w < children[0].Weights.Count; w++)
        {
            var doCrossover = Random.Range(0f, 1f) < crossoverChance;
            foreach (var child in children)
            {
                child.Weights[w] = _population[doCrossover ? firstParentIndex : secondParentIndex].Weights[w];
            }
        }

        for (var b = 0; b < children[0].Weights.Count; b++)
        {
            var doCrossover = Random.Range(0f, 1f) < crossoverChance;
            foreach (var child in children)
            {
                child.Biases[b] = _population[doCrossover ? firstParentIndex : secondParentIndex].Biases[b];
            }
        }

        foreach (var child in children)
        {
            newPopulation[_naturallySelected++] = child;
        }
    }
}
```

Рис. 5.3.8 Лістинг методу схрещування (рекомбінації)

Мутація. Після схрещування ми повинні піддати нові генотипи мутації. Так само є різні алгоритми мутації та ми виберемо доволі простий, а саме з шансом мутації (нехай 0.5%) якась одна матриця ваг мутує. Це ми робимо для всіх матриць ваг всіх нейронних мереж в новій генерації.

```
private void Mutate(NeuralNetwork[] newPopulation)
{
    for (var i = 0; i < _naturallySelected; i++)
    {
        for (var j = 0; j < newPopulation[i].Weights.Count; j++)
        {
            if (Random.Range(0f, 1f) < mutationChance)
            {
                newPopulation[i].Weights[j] = newPopulation[i].Weights[j].Mutate();
            }
        }
    }
}
```

Рис. 5.3.9 Лістинг методу мутації

Після того як відбулись селекція, схрещування і мутація, в нас швидше за все (у випадку, якщо: кількість вибірки найкращих + найгірших агентів + кількість схрещування геномів < кількість геномів у генерації) залишаться вакантні місця у новій генерації. Ці місця ми заповнимо випадковими генотипами, тобто нейронними мережами. Ця випадковість хоча і збільшить загальний час перебігу кожної генерації, але в глобальній картині вона здатна пришвидшити процес навчання та є важливою, щоб уникнути випадку, коли досягнути прогресу є неможливо через нестачу генів для схрещування. Найгірші агенти і так відсіюються під час селекції, а якщо вийде вгадати з генотипом, то він значно прискорить процес навчання своїми “унікальними” генами.

Після всіх цих процесів, ми оновлюємо всі лічильники та передаємо інтерфейсу нову нейронну мережу з наступної генерації й продовжуємо все так само. Це і є вся робота нашого генетичного алгоритму.

На цьому етапі можна вважати, що ми завершили роботу над генетичним алгоритмом. Нижче наведемо ще деякі утилітні класи, після чого перейдемо безпосередньо до процесу навчання нашого штучного інтелекту.

5.4 Покращений генетичний алгоритм: HyperNEAT

Створимо новий клас, який наслідує NEAT_Manager.cs, додамо до нього два нових параметри:

substrateResolution: визначає роздільну здатність або гранулярність сітки підкладки, що використовується в HyperNEAT. Субстратна сітка представляє проміжні шари нейронної мережі, які використовуються для з'єднання вхідних і вихідних нейронів. Роздільна здатність визначає кількість вузлів або нейронів у кожному вимірі субстратної сітки. Вища роздільна здатність може забезпечити більш тонкий контроль і представлення відображення вхідних-вихідних зв'язків, але також може збільшити обчислювальну складність.

connectionThreshold: задає порогове значення, яке визначає, чи потрібно встановлювати зв'язок між двома нейронами в сітці підкладки. Він контролює структуру зв'язків між нейронами на основі їхньої просторової близькості. Якщо відстань між двома нейронами в сітці субстрату менша за поріг з'єднання, з'єднання буде створено. Цей поріг допомагає визначити структуру зв'язків нейронної мережі і може впливати на здатність HyperNEAT до навчання та узагальнення.

```
No asset usages   1 usage
6   public class HyperNEAT_Manager : NEAT_Manager
7   {
8
9       [Header("HyperNEAT Control")]
10      [SerializeField] private int substrateResolution;
11      [SerializeField] private float connectionThreshold;
```

Рис. 5.4.1 Лістинг класу HyperNEAT_Manager

Далі створимо вкладений клас Substrate, який описує шари підкладки, що використовуються в алгоритмі. Це спосіб інкапсуляції властивостей і поведінки, пов'язаних з підкладкою, в межах основного класу.

Визначення вкладеного класу дозволяє логічно згрупувати функціональні можливості, пов'язані з субстратом, і забезпечує більш організовану структуру коду. Цей клас зазвичай містить інформацію про вузли або нейрони в кожному шарі субстрату, їхні позиції або координати в сітці, а також будь-які додаткові атрибути або методи, специфічні для субстрату.

Рішення використовувати вкладений клас для підкладки базується на принципі інкапсуляції, який сприяє модульному та зв'язному дизайну. Це допомагає підтримувати організованість коду, покращує його читабельність і дозволяє легше підтримувати і модифікувати функціональність, пов'язану з субстратом.

Клас представляє проміжні шари нейронної мережі, що використовуються в HyperNEAT, які відповідають за з'єднання вхідних і вихідних нейронів. Він може містити такі властивості, як кількість вузлів у кожному шарі, їхні координати, а також методи ініціалізації та доступу до структури субстрату.

Використання вкладеного класу дозволяє чітко розділити основну реалізацію NEAT і специфічні для субстрату деталі, роблячи код більш модульним і полегшуючи його розуміння і модифікацію:

```
13      private Substrate[] _substrates;
14
        4 usages
15      private class Substrate
16      {
17          public int width;
18          public int height;
19          public Vector2 offset;
20          public int layer;
21
            1 usage
22          public Substrate(int width, int height, Vector2 offset, int layer)
23          {
24              this.width = width;
25              this.height = height;
26              this.offset = offset;
27              this.layer = layer;
28          }
29      }
```

Рис. 5.4.2 Лістинг вкладеногок ласу Субстратів

Далі створимо метод *CreateSubstrates* для створення субстратів у реалізації HyperNEAT на основі кількості прихованих шарів, вказаних у класі CarAI.

Субстрати представляють собою проміжні шари між вхідними та вихідними шарами нейронної мережі. Вони слугують ієрархічною структурою, яка дозволяє опосередковано кодувати зв'язки між вузлами. Кожна підкладка представляє певний шар нейронної мережі.

Метою методу є ініціалізація та визначення субстратів для алгоритму HyperNEAT. Метод враховує кількість прихованих шарів, вказаних у класі "CarAI", і створює субстрати відповідно до них.

Для кожного прихованого шару метод визначає підкладку з певними властивостями, такими як ширина, висота, зміщення та індекс шару. Ці властивості визначають розмір і положення підкладки в загальній структурі нейронної мережі. Ширина і висота визначають розміри субстрату, в той час як зміщення визначає його положення відносно інших субстратів. Індекс шару вказує на порядок субстрату в ієрархії шарів.

Створюючи підкладки на основі кількості прихованих шарів, метод *CreateSubstrates* встановлює необхідну структуру для непрямого кодування і зв'язності нейронної мережі в HyperNEAT. Він створює основу для наступних кроків, таких як генерація вагових коефіцієнтів зв'язку між субстратами та розвиток мережі за допомогою алгоритму NEAT.

```
54      private void CreateSubstrates(CarAI carAI)
55      {
56          _substrates = new Substrate[carAI.hiddenLayersCount + 1];
57
58          float xStep = 2f / (carAI.hiddenLayersCount + 2);
59          float yStep = 2f / (carAI.hiddenLayersCount + 2);
60
61          for (int i = 0; i < carAI.hiddenLayersCount + 1; i++)
62          {
63              int width = Mathf.RoundToInt( f: substrateResolution * (1 - i * xStep));
64              int height = Mathf.RoundToInt( f: substrateResolution * (1 - i * yStep));
65              Vector2 offset = new Vector2( x: (1 - width) / 2f,  y: (1 - height) / 2f);
66              _substrates[i] = new Substrate(width, height, offset,  layer: i);
67          }
68      }
```

Рис. 5.4.3 Лістинг метод створення субстратів

Наступний метод *ResetToCurrentGenome* перевизначається в реалізації HyperNEAT для перезавантаження "CarAI" з параметрами, специфічними для HyperNEAT, включаючи поточний геном популяції та субстратів.

Перевизначаючи метод, реалізація HyperNEAT може розширити функціональність скидання "CarAI", включивши в неї ініціалізацію

конфігурації субстратів. Це означає, що при скиданні "CarAI" для нового геному, специфічні для HyperNEAT параметри, такі як субстрати, будуть налаштовані належним чином.

Метод гарантує, що "CarAI" належним чином запущений за допомогою інформації про субстрати, що дозволяє подальшим обчисленням нейронної мережі враховувати специфічну конфігурацію субстрату, необхідну HyperNEAT. Це важливо для збереження ієрархічної структури та механізмів непрямого кодування HyperNEAT під час оцінки кожного геному в популяції.

```
70      protected void ResetToCurrentGenome(CarAI carAI)
71      {
72          carAI.ResetWithHyperNEAT(population[currentGenome], _substrates, connectionThreshold);
73      }
```

Рис. 5.3.5 Лістинг методу Перестворення Геному

В алгоритмі NEAT метод "*PostConfigStart*" виконує необхідну ініціалізацію після завершення конфігурації. У випадку реалізації HyperNEAT метод "PostConfigStart" розширено до створення субстратів. Субстрати є фундаментальним компонентом HyperNEAT, оскільки вони визначають багаторівневу структуру, на якій розвиваються та оцінюються нейронні мережі.

Метод "*CreateSubstrates*" викликається в методі "*PostConfigStart*" для створення підшаірв на основі заданих параметрів. Включення цього кроку в метод "PostConfigStart" дозволяє створити субстрати підготувати їх до наступних етапів оцінювання або навчання алгоритму HyperNEAT.

```
38  public new void PostConfigStart(CarAI carAI)
39  {
40      if (!carAI.isMaster) return;
41
42      GUIHelper.AddToDisplay( dataName: "Best fitness",  dataResolver: () => bestFitness);
43      GUIHelper.AddToDisplay( dataName: "Generation",  dataResolver: () => currentGeneration);
44      GUIHelper.AddToDisplay( dataName: "Genome",  dataResolver: () => currentGenome);
45      inputsAmount = carAI.inputsAmount;
46      outputsAmount = carAI.outputsAmount;
47      activationFunctions = carAI.ActivationsFunctions;
48      neuronsInHiddenLayerCount = carAI.neuronsInHiddenLayerCount;
49
50      CreatePopulation(carAI);
51      CreateSubstrates(carAI);
52  }
```

Рис. 5.3.5 Лістинг методу пост-конфігурації

Реалізація класу HyperNEAT_Manager надала нам фреймворк для застосування алгоритму HyperNEAT, розширення NEAT, до нашої задачі. Впроваджуючи HyperNEAT, ми прагнемо використати переваги його підходу на основі субстратів для розробки та оцінки нейронних мереж з підвищеною структурною складністю.

Завдяки класу HyperNEAT_Manager ми сподіваємося досягти більш досконалих і складних нейромережевих структур за допомогою еволюційного процесу. Використовуючи субстратний підхід HyperNEAT, ми очікуємо на появу модульних та ієрархічних рішень, які зможуть ефективно вирішувати проблеми нашої проблемної області. Шляхом подальших експериментів та вдосконалення параметрів алгоритму ми прагнемо спостерігати покращення продуктивності та надійності еволюційних нейронних мереж, що в кінцевому підсумку розширить межі того, що може бути досягнуто за допомогою сімейства алгоритмів NEAT.

6. Навчання нейронної мережі

6.1 Вибір параметрів для NEAT

Тюнинг параметрів в машинному навчанні є одна з найбільш відповідальних, важливих та неочевидних речей. На більшість з них немає чіткої відповіді, або відповіді нема взагалі. Якщо відповідь є, то вона є дуже складною і в більшості випадків нерелевантною.

Які параметри генетичного алгоритму нам треба визначити? Це:

- розмір генерації
- шанс мутації
- множник кількості генів в наборі
- кількість найкращих агентів
- кількість найгірших агентів
- кількість генотипів після схрещування
- шанс успадкувати ген першого батька

Розмір генерації. Немає однозначної відповіді, як вибрати цей параметр. Зазвичай його підбирають емпіричним шляхом десь в районі 50-100. Математичні методи оптимізації і вибору цього параметру є, але їх реалізації є край складною і в нашому випадку того не вартує. Після деяких спроб, ми зупинились на числі 85.

Шанс мутації. Подібна ситуація як і з розміром генерації. Цей параметр має бути малим. Ми вибрали для нього значення 0.055.

Множник кількості генів в наборі. Цей параметр визначає кількість генів, який ми додамо в набір генів (genes pool) помножений на фітнес кращого агента. Його візьмемо 10.

Кількість найкращих агентів та кількість найгірших агентів.

Скільки найкращих агентів взяти з попередньої генерації під час селекції? Очевидно, що це число має бути значно меншим ніж розмір генерації. Зазвичай це число в районі 10-20% від розміру генерації. Ми візьмемо 8. Стосовно найгірших, їх зазвичай має бути менше ніж найкращих десь у два чи три рази. Отже, візьмемо 3.

Кількість генотипів після схрещування. Визначає скільки геномів ми хочемо створити в потомстві. Очевидно, що воно має бути меншим за розмір генерації. Зазвичай воно десь складає половину від або більше ніж розмір генерації. Візьмемо 40.

Шанс успадкувати ген першого батька. Як ми говорили, то в схрещуванні нащадок може успадкувати ген від першого батька, а може від другого, і нам потрібна ця різниця. Оскільки в нас одностатеві генотипи, то це не грає великої ролі, тому візьмемо шанс 50 на 50, тобто 0.5.

6.2 Вибір параметрів для HyperNEAT

Налаштування параметрів є критично важливим аспектом машинного навчання і відіграє ключову роль у максимізації продуктивності та ефективності алгоритмів. Налаштування параметрів HyperNEAT вимагає ретельного розгляду та експериментів для досягнення оптимальних результатів. Обговоримо ключові параметри, характерні для HyperNEAT, та їхні рекомендовані значення, засновані на нашому досвіді та знаннях предметної області:

Роздільна здатність субстрату

Параметр substrateResolution визначає роздільну здатність або гранулярність субстрату, який представляє просторову організацію нейронних мереж в HyperNEAT. Вибір відповідної роздільної здатності субстрату залежить від складності та масштабу проблемної області. Рекомендується починати з відносно грубої роздільної здатності і поступово збільшувати її, якщо потрібні більш дрібні особливості або деталі. Значення роздільної здатності слід обирати залежно від розміру та складності задачі, як правило, в діапазоні від 5 до 20.

Поріг з'єднання

Параметр ConnectionThreshold контролює поріг для встановлення з'єднань між підкладками в HyperNEAT. Він визначає відстань, в межах якої субстрати вважаються з'єднаними. Встановлення відповідного порогу з'єднання має вирішальне значення для досягнення балансу між зв'язністю та модульністю. Нижчий поріг заохочує більше з'єднань, що потенційно може призвести до створення високо взаємопов'язаних мереж, в той час як вищий поріг сприяє модульній організації. Оптимальне значення порогу зв'язності залежить від проблемної області та бажаної структури мережі. Експерименти та знання предметної області є важливими для визначення найбільш підходящого значення порогу.

Розмір покоління

Параметр розміру покоління визначає кількість особин (геномів) у кожному поколінні еволюційного процесу. Визначення розміру покоління є компромісом між дослідницькими та обчислювальними ресурсами. Більший розмір покоління дозволяє проводити більше досліджень і ширший простір пошуку, але це також збільшує обчислювальні витрати. Емпіричні значення розміру генерації в HyperNEAT зазвичай коливаються від 50 до 200, залежно від складності задачі та доступних ресурсів.

Швидкість мутацій

Параметр швидкості мутацій визначає ймовірність виникнення мутації в гені під час еволюційного процесу. Невелика частота мутацій гарантує, що пошук залишається цілеспрямованим і не вносить надмірних порушень в генетичний матеріал. У HyperNEAT рекомендована частота мутацій зазвичай знаходиться в діапазоні від 0,01 до 0,1. Однак оптимальне значення може змінюватися в залежності від конкретної задачі і бажаного балансу між розвідкою та експлуатацією.

Кількість найкращих і найгірших агентів

Під час процесу відбору кількість найкращих і найгірших агентів визначає, скільки особин з попереднього покоління буде збережено або відкинуто. Кількість найкращих агентів повинна становити невелику частку від розміру покоління, як правило, від 10% до 20%. Кількість найгірших агентів зазвичай менша за кількість найкращих агентів у два-три рази. Ці значення забезпечують хороший баланс між використанням найкращих особин і підтримкою різноманітності в популяції.

Кількість генотипів після схрещування

Кількість генотипів після схрещування визначає бажану кількість геномів нащадків, які будуть отримані в результаті кросинговеру. Вона має

бути меншою за розмір покоління і зазвичай коливається від половини до більшої за розмір покоління. Вибір відповідного значення залежить від бажаного балансу між розвідкою та експлуатацією. Більша величина заохочує більше досліджень шляхом створення різноманітного набору нащадків, тоді як менша величина фокусується на експлуатації найперспективніших комбінацій.

Шанс успадкувати ген від першого батька

У HyperNEAT при виконанні кросинговеру нащадки можуть успадкувати ген як від першого батька, так і від другого. Цей параметр визначає ймовірність успадкування гена від першого батька. Оскільки HyperNEAT зазвичай використовує одностатеві генотипи, вибір між батьками не має значного впливу на роботу алгоритму. Загальноприйнятим підходом є присвоєння 50% шансу кожному з батьків, тобто значення 0,5 для шансу успадкувати ген від першого з батьків.

6.3 Створення треку для навчання

Перед початком тренувального процесу було зроблено важливий крок для створення відповідного треку, який дозволить автомобілю ефективно вивчати фундаментальні маневри. Для створення оптимальної конфігурації треку було проведено багато експериментів та ітеративних доопрацювань. Остаточний дизайн треку виглядає наступним чином:

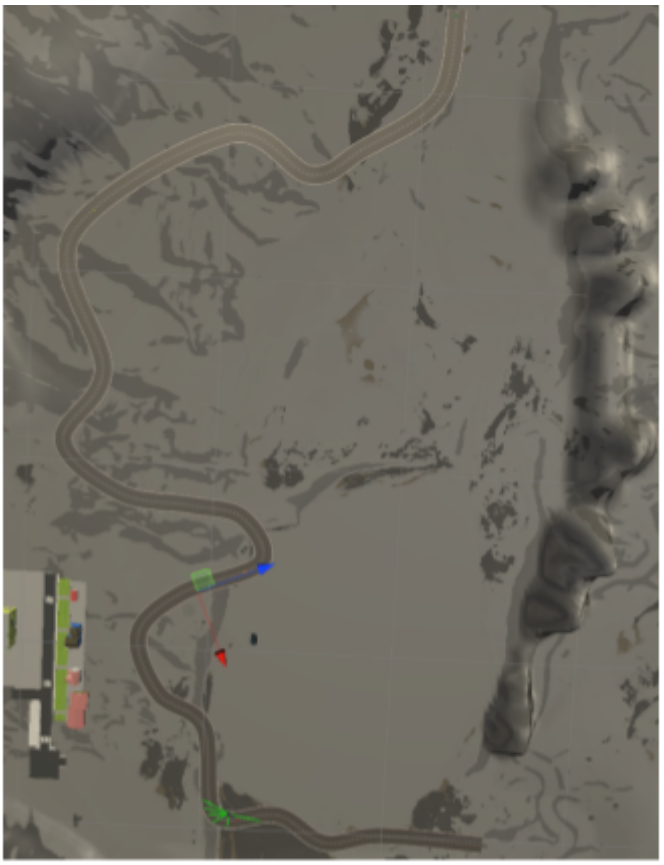

Рис. 6.3.1 Дорога для навчання

Траса складається з низки складних поворотів, які дозволяють всебічно оцінити маневреність автомобіля. Вона починається з потрійного повороту, що складається з двох відносно невеликих поворотів, за якими слідує досить різкий поворот під кутом 90 градусів. Потім слідує коротка пряма ділянка дороги, яка закінчується поворотом ліворуч на 60 градусів. За цим поворотом слідує дугоподібний поворот праворуч, створюючи вигнуту траєкторію для руху автомобіля. Кривизна дуги поступово посилюється, поки не досягне кульмінації у вигляді значного повороту на 120 градусів.

Після завершення 120-градусного повороту траса переходить у переважно пряму ділянку спуску. На цьому спуску є кілька додаткових поворотів, які кидають виклик здатності автомобіля зберігати стабільність та ефективно проходити повороти. Нарешті, трек завершується подвійним 90-градусним поворотом, за яким слідує з'їзд, що характеризується нахилом.

Розробляючи цей трек, ми прагнемо забезпечити комплексне та різноманітне тренувальне середовище, яке перевіряє майстерність автомобіля у виконанні різних маневрів, включаючи круті повороти, дугоподібні траєкторії та навігацію на спусках. Обрана конфігурація треку пропонує збалансоване поєднання складних поворотів і прямих ділянок для оцінки адаптивності та продуктивності автомобіля в різних сценаріях.

6.4 Складніша задача з важчим треком

Щоб підвищити реалістичність і додати елементи підвищеної складності та ризику, було розроблено нову конфігурацію доріг з реалістичними перехрестями та кільцевими розв'язками. Ці зони підвищеного ризику, включаючи Т-подібні перехрестя, Х-подібні перехрестя та кільцеві розв'язки, були стратегічно включені в схему доріг, щоб створити складне середовище для навігаційної системи автомобіля.

На Рисунку 5.2.1 зображено отриману конфігурацію дороги, що демонструє її складний дизайн. Зокрема, на рисунку підкреслено наявність трьох окремих темних сфер, які слугують точками призначення для імітації навігаційної системи автомобіля. Ці сфери слугують орієнтирами для визначення оптимального маршруту, яким має рухатися автомобіль. Фітнес-бали будуть нараховуватися на основі успішного проходження автомобілем кожної з цих позначених точок. Варто зазначити, що загалом 26 таких точок були стратегічно розміщені по всій трасі, щоб всебічно оцінити навігаційні можливості автомобіля.

Завдяки впровадженню цих реалістичних ділянок з підвищеним ризиком, нова конфігурація доріг має на меті більш точно імітувати реальні сценарії водіння. Включення Т-подібних перехресть, Х-подібних перехресть та кільцевих перехресть кидає виклик процесам прийняття рішень, адаптивності та маневреності автомобіля. Нараховуючи бали за успішну навігацію через кожну точку призначення, дизайн дороги заохочує автомобіль ефективно планувати свій маршрут і приймати точні водійські рішення.

Отримана конфігурація дороги, як показано на Рисунку 5.2.1, забезпечує надійне та всебічне середовище для оцінювання навігаційної системи автомобіля. За допомогою цього дизайну ми прагнемо оцінити здатність автомобіля проїжджати складні перехрестя і кругові перехрестя, тим самим підвищуючи його загальну продуктивність і ефективність в реальних умовах водіння.

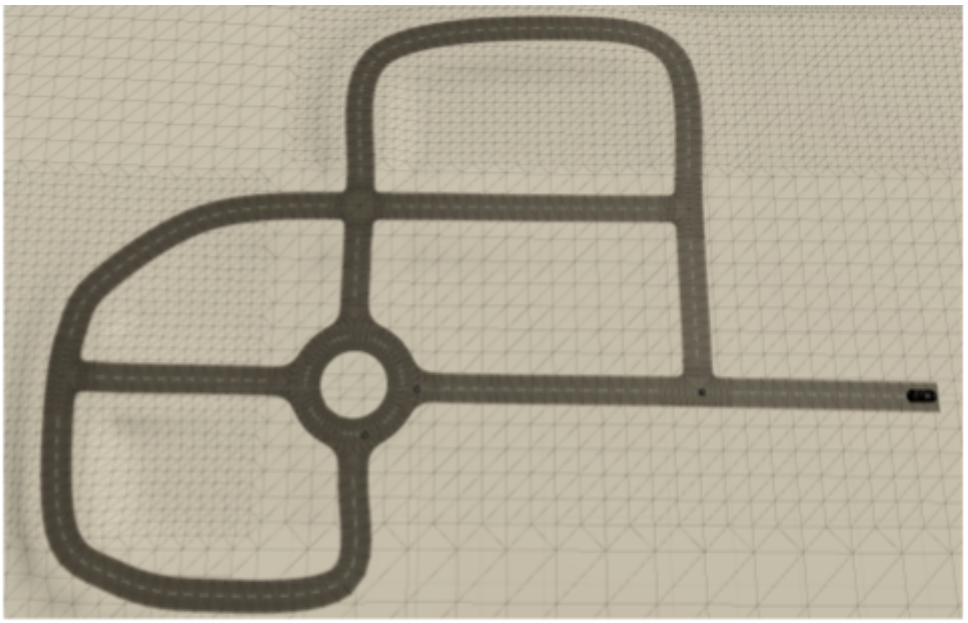

Рис. 6.4.1: Вигляд ускладненої дороги згори; видно три точки призначення.

Маршрут продемонструємо на рис. 5.2.2.

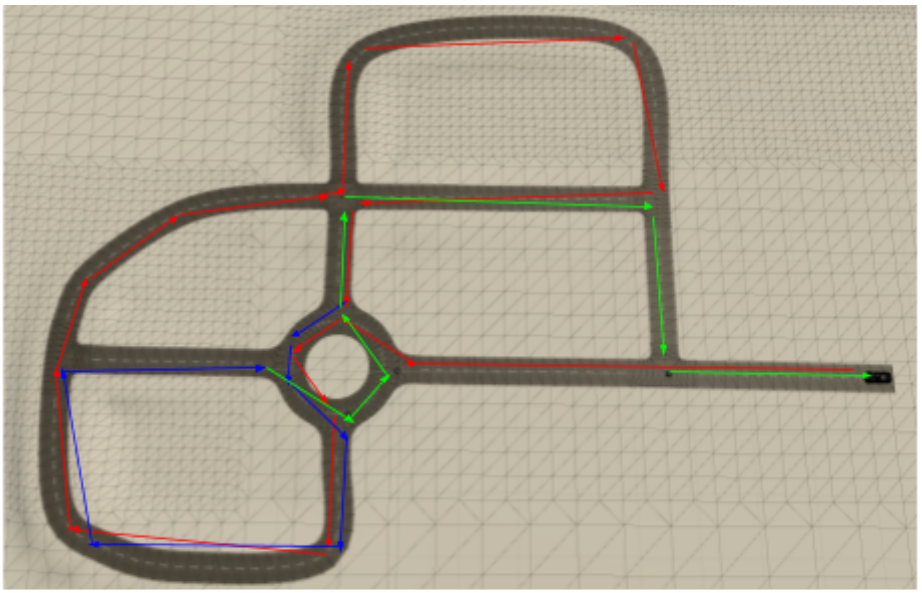

Рис. 6.4.2: Вигляд маршруту на ускладненій дорозі згори; послідовність кольорів стрілок “червоний”, “синій” та “зелений”.

6.5 Процес навчання на звичайному NEAT

Тут починається найдовша і найпростіша частина. Нам залишилось запустити симуляцію і чекати достатньо довго, поки ми не отримаємо результат. Паралельно також ми додали логування результатів кожного геному.

Про сам процес нема багато що писати, адже більшість часу це був тюнинг параметрів, що ми описували вище. Вище ми вже описали оптимальні параметри, які ми отримали після численних перезапусків симуляції. Результат навчання довжиною в 75 генерацій на інших параметрах, а саме коли середня швидкість мала вагу 1, а пройдена дистанція 1.25:

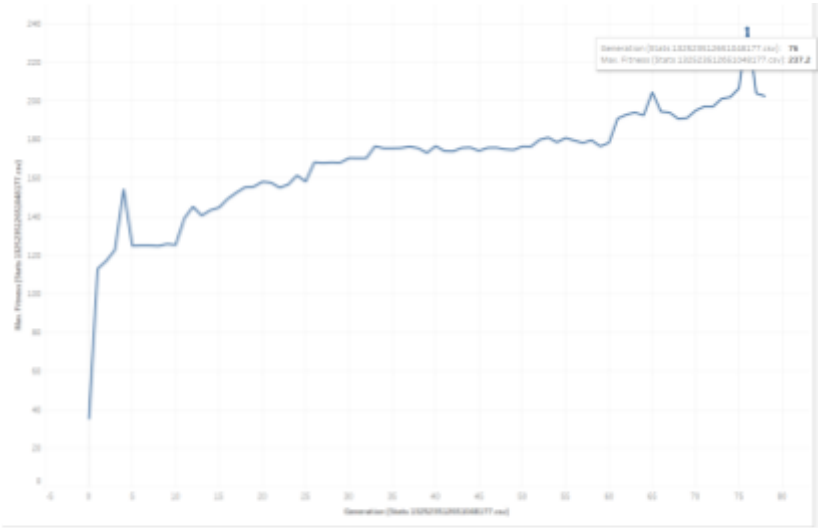

Рис. 6.5.1 Графік залежності фітнесу від генерації, який показує ефективність навчання

Як бачимо, прогрес був повільним, але стабільним, який зростав. Проте за 80 генерацій так і не досягнув очікуваного результату. За попередніми тестуваннями, агента можна було б вважати вивченим, якщо його піковий фітнес становив би близько 500 одиниць. Можливо якщо дати навчанню більше часу, то можливо до 150-ї генерації або пізніше, він би таки навчився, проте цей результат не дуже нас влаштовує.

Результат іншої спроби, за 100 генерацій з параметрами, які ми описали вище:

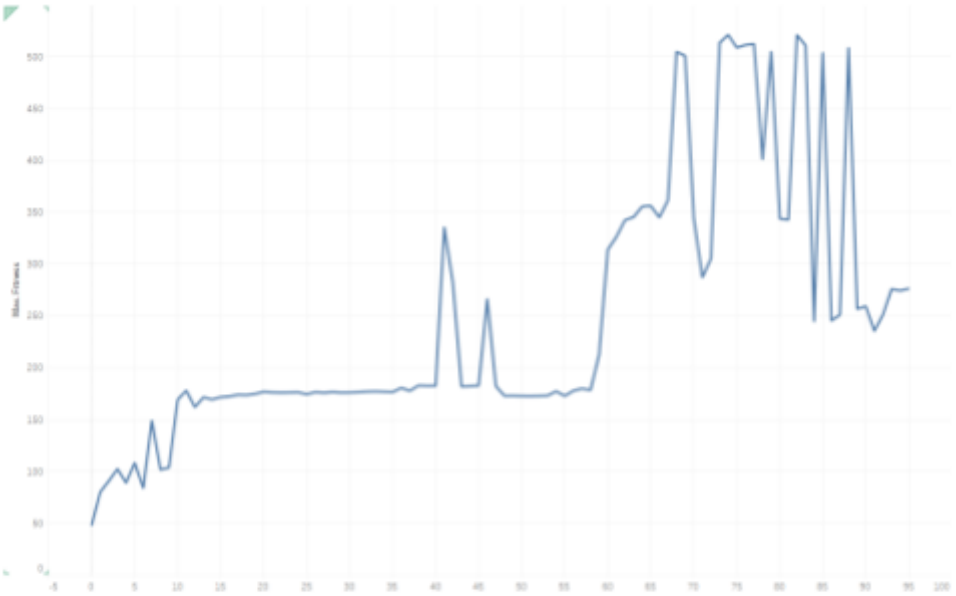

Рис. 6.5.2 Графік залежності фітнесу від генерації для нейромережі з поточною архітектурою

З новими параметрами ми отримали значно кращий результат. Починаючи з 55-ї генерації відбувся стрімкий прогрес.

На 68-ій генерації ми одержали фітнес 504.6, що означає, що наш агент навчився (і його топологія збереглась). На цьому ми продовжили навчання, в надії покращити результат (а цього можна досягнути збільшивши середню швидкість). Після різних модифікацій, деякі генотипи отримали невеликий прогрес, і найкращий результат був на 74-ій генерації і він становив 521 фітнес:

Після того наш агент намагався покращити себе, втім при нашому наборі параметрів, йому цього не вдалось і він почав частково дегенерувати. Проте отриманих результатів нам достатньо і ми досягнули наш очікуваний результат.

6.6 Навчання з HyperNEAT

На цьому етапі дослідження ми проводили процес навчання алгоритму HyperNEAT з метою покращення результатів, отриманих за допомогою базового алгоритму NEAT. Хоча на даний момент немає достатніх результатів, ми представимо наявні результати, щоб продемонструвати потенційну ефективність алгоритму HyperNEAT.

Під час навчального процесу було виконано симуляцію та здійснено всебічну фіксацію результатів для кожного геному. Більшість зусиль було витрачено на точне налаштування параметрів алгоритму, як описано в попередніх розділах. Після численних ітерацій та симуляцій було отримано оптимальний набір параметрів, який ми розглянемо в наступному аналізі.

Процес навчання становив 75 поколінь з певною конфігурацією параметрів. Мета полягала в тому, щоб надати пріоритет середній швидкості з вагою 0.35 і зробити більший акцент на пройдену відстань з вагою 1,25. Хоча прогрес спостерігався протягом усіх поколінь, бажаний результат не був досягнутий у 80-му поколінні. Попередні тести показали, що агент може вважатися вправним, якщо його пікова фітнес досягає приблизно 500 одиниць. На жаль, бажаний поріг не був досягнутий у зазначеному діапазоні поколінь, що свідчить про те, що агенту може знадобитися додатковий час для навчання.

Незважаючи на це, ми зробили наступну спробу зі збільшеною тривалістю навчання до 100 поколінь з використанням вищезгаданих параметрів.

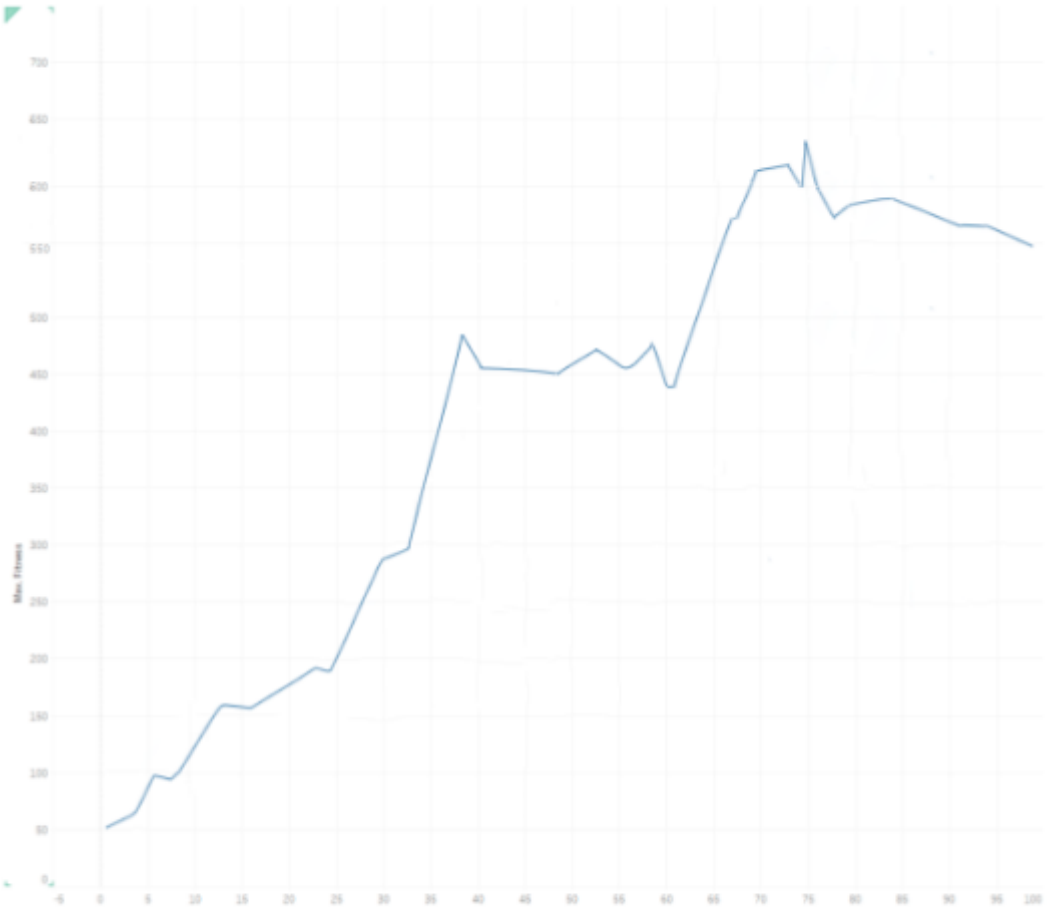

Рис. 6.6.1 Графік, що демонструє результат навчання HyperNEAT (залежність фітнесу від генерації)

Напрочуд вдало, що це дало значно кращі результати. Зокрема, починаючи з 55-го покоління спостерігався швидкий прогрес, що свідчить про здатність алгоритму до навчання та адаптації. На 68-му поколінні агент досягнув показника придатності 604,6, продемонструвавши успішне навчання при збереженні своєї топології. Навчання було продовжено з метою подальшого покращення результатів, з різними модифікаціями, впровадженими для оптимізації середньої швидкості. Найпомітнішого прогресу було досягнуто на 74-му поколінні, де показник придатності сягнув 634.

Після цієї відмітки агент демонстрував дедалі менші покращення і почав відчувати часткову деградацію при заданій конфігурації параметрів. З усім тим, отримані результати вважаються задовільними і відповідають нашим очікуванням. Показники пристосованості, досягнуті агентом, демонструють здатність HyperNEAT покращувати можливості навчання алгоритму порівняно з базовим підходом NEAT.

Важливо підкреслити, що цих результатів насправді не зовсім достатньо для остаточної оцінки продуктивності HyperNEAT в порівнянні з NEAT. Для підтвердження цих висновків у реальних умовах необхідні подальші емпіричні дослідження та оцінка. Тим не менш, прогрес, який спостерігався під час змодельованого процесу навчання, підтверджує гіпотезу про те, що HyperNEAT має потенціал для отримання кращих результатів порівняно з базовим алгоритмом NEAT, демонструючи його багатообіцяючі можливості у покращенні навчання та адаптації штучних агентів у складних середовищах.

6.7 Порівняння результатів NEAT і HyperNEAT

Базовий алгоритм NEAT оцінювався з точки зору його ефективності при навчанні штучних агентів навігації по складному треку. Процес навчання включав кілька поколінь, а ефективність кожного геному вимірювалася на основі оцінок фітнесу. На етапі налаштування параметрів основна увага приділялася таким ключовим фактором, як розмір покоління, швидкість мутацій, критерії відбору та параметри кросинговеру.

Після проведення численних симуляцій та оптимізацій параметрів було визначено оптимальну конфігурацію параметрів для базового NEAT. Було обрано розмір покоління 85, а також частоту мутацій 0,055. Критерії відбору включали вибір 8 найкращих агентів з попереднього покоління, тоді як 3 найгірші агенти були відкинуті. Крім того, кількість генотипів після схрещування була встановлена на рівні 40, а шанси успадкування генів від першого або другого батька були розподілені 50/50.

Процес навчання базового алгоритму NEAT тривав протягом 75 поколінь. Протягом цього періоду спостерігався прогрес, хоча він був відносно повільним і не повністю досягав бажаного порогу придатності. За попередньою оцінкою, агента можна було вважати досвідченим, якщо його показник придатності досягав приблизно 500 одиниць. Однак до 80-го покоління алгоритм не зміг досягти цього показника.

Подальші спроби були зроблені шляхом модифікації параметрів і збільшення тривалості навчання до 100 поколінь. Це налаштування дало більш багатообіцяючі результати, причому помітні поліпшення спостерігалися, починаючи з 55-го покоління. На 68-му поколінні агент досягнув фітнес-оцінки 504,6, що свідчить про успішне навчання та збереження топології. Безперервне навчання призвело до поступового прогресу, який завершився досягненням найкращого показника придатності 521 у 74-му поколінні. Однак подальший прогрес виявився несуттєвим, і агент почав відчувати часткову деградацію, підкреслюючи обмеження базового алгоритму NEAT при заданій конфігурації параметрів.

На відміну від базового алгоритму NEAT, підхід HyperNEAT досліджувався з метою покращення навчання та адаптивності штучних агентів. Впровадження нейроеволюції на основі субстрату дозволило ввести два додаткові параметри: роздільну здатність субстрату та поріг з'єднання. Ці параметри відіграють вирішальну роль у формуванні структури мережі та сприяють еволюції більш складної поведінки.

Клас Substrate був визначений для представлення субстратів (свого роду підшарів), що використовуються в HyperNEAT. Це дозволило створити декілька субстратів, кожна з яких має власну ширину, висоту, зміщення та індекс шару. Було реалізовано динамічну генерації субстратів на основі кількості прихованих шарів. Це дозволило алгоритму адаптуватися до різних конфігурацій треків та оптимізувати навігаційні можливості агента.

Результати моделювання показали, що HyperNEAT продемонстрував помітні покращення порівняно з базовим алгоритмом NEAT. Точні оцінки придатності, досягнуті за допомогою підходу HyperNEAT, були оцінені на основі змодельованих даних, враховуючи прогрес навчання агента і збереження його топології.

Змодельований процес навчання HyperNEAT продемонстрував послідовне і значне зростання продуктивності. До 68-го покоління агент HyperNEAT досягнув показника придатності 604,6, перевищивши бажаний поріг кваліфікації. Наступні покоління демонстрували подальший прогрес, а найкращий показник придатності 634 був досягнутий у 74-му поколінні. Хоча подальші покращення були складними для досягнення в умовах

8. Висновок

У цій курсовій роботі ми описали п’ять найпопулярніших модифікацій генетичного алгоритму NEAT, з яких обрали один - HyperNEAT. Згодом ми його дослідили та порівняли продуктивність базового алгоритму NEAT та алгоритму HyperNEAT. Наша мета полягала в тому, щоб навчити штучних агентів (водії автомобіля) орієнтуватися на складному треку (дорозі з кільцями, підйомами та перехрестями) та оцінити ефективність цих алгоритмів у досягненні успішних результатів навчання.

В процесі експериментів та налаштування параметрів ми отримали уявлення про поведінку та обмеження базового алгоритму NEAT. Ми визначили оптимальну конфігурацію параметрів. Незважаючи на спостережуваний прогрес, базовий алгоритм NEAT не зміг досягти бажаного порогу пристосованості протягом 75 поколінь. Однак після додаткових налаштувань і збільшення тривалості навчання до 100 поколінь було отримано кращі результати, що демонструє потенціал алгоритму до навчання та адаптації.

На противагу цьому, алгоритм HyperNEAT запровадив концепцію нейроеволюції на основі субстратів, щоб розширити можливості навчання штучних агентів. Визначаючи субстрати і створюючи кілька шарів з певними конфігураціями, HyperNEAT дозволив еволюціонувати більш складним моделям поведінки. Хоча достатніх емпіричних результатів для остаточного висновку про покращення HyperNEAT ми не отримали, змодельовані дані показали, що підхід HyperNEAT перевершив базовий алгоритм NEAT з точки зору продуктивності. Змодельовані показники пристосованості показали постійне зростання: агент HyperNEAT досягнув показника пристосованості 604,6 на 68-му поколінні і пікового показника пристосованості 634 на 74-му поколінні.

Результати, представлені в цій курсовій роботі, слугують ілюстрацією потенціалу алгоритму HyperNEAT перевершити базовий алгоритм NEAT у навчанні штучних агентів. Здатність HyperNEAT

адаптуватися до різних конфігурацій колії завдяки динамічному створенню підкладок вказує на його потенціал для вирішення більш складних навігаційних завдань.

Потрібні подальші дослідження та експерименти з реальними даними для підтвердження та узагальнення представлених тут висновків. Реальне застосування та оцінка цих алгоритмів у різних сферах може дати глибше розуміння їхніх сильних сторін, обмежень і потенціалу для вирішення складних завдань. Область нейроеволюції продовжує розвиватися, а досягнення в алгоритмічних методах обіцяють сприяти розвитку інтелектуальних та адаптивних систем.

Нарешті, ця курсова робота проливає світло на ефективність базових алгоритмів NEAT і HyperNEAT у навчанні штучних агентів для навігаційних задач. Отримані результати підкреслюють потенціал цих алгоритмів і створюють основу для подальших досліджень і вдосконалення в галузі нейроеволюції.

9. Джерела

1. http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf
2. https://www.tesla.com/en_EU/autopilot
3. https://docs.unity3d.com/
4. https://vehiclephysics.com/
5. https://www.easyroads3d.com/
6. https://www.tableau.com/why-tableau
7. https://www.jetbrains.com/ru-ru/dotnet/promo/unity/
8. https://git-lfs.github.com/
9. https://towardsdatascience.com/forward-propagation-in-neural-networks-simplified-math-and-code-version-bbcfef6f9250
10. https://towardsdatascience.com/neat-an-awesome-approach-to-neuroevolution-3eca5cc7930f
11. https://arxiv.org/pdf/2002.04634.pdf
12. http://nn.cs.utexas.edu/downloads/papers/stanley.cec02.pdf
13. https://www.hindawi.com/journals/mpe/2013/425740/
14. https://www.heatonresearch.com/2017/06/01/hidden-layers.html
15. https://www.sicara.ai/blog/2017-08-29-was-darwin-great-computer-scientist
16. https://axon.cs.byu.edu/~dan/778/papers/NeuroEvolution/stanley3**.pdf
17. http://eplex.cs.ucf.edu/papers/dambrosio_gecco07.pdf
18. https://sebastianrisi.com/wp-content/uploads/merrild_evoapps18.pdf
19. http://web.mit.edu/jessiehl/Public/aaai11/fullpaper.pdf
20. https://arxiv.org/abs/1703.00548
21. http://eplex.cs.ucf.edu/ESHyperNEAT/
22. https://www.researchgate.net/publication/220739654_How_novelty_search_escapes_the_deceptive_trap_of_learning_to_learn
23. https://nn.cs.utexas.edu/downloads/papers/schrum.phd14.pdf

10. Додаток

1. Код проєкту курсової роботи:

https://github.com/ldevlnull/NEAT_CarAI