Autonomiczny samochód

Implementujemy wirtualnego kierowcę przy pomocy sieci neuronowych i algorytmu ewolucyjnego

Czy wam również zdarzyło się użyć jakiegoś sformułowania, które umieszczone w odpowiednim kontekście było całkowicie sensowne, ale wyrwane z tegoż kontekstu brzmiało całkowicie absurdalnie? Moim dotychczasowym faworytem jest komentarz w kodzie: „Metoda niszczy wszystkie światy" (chodziło o usunięcie scen – światów – w autorskim silniku renderującym 3D). Wczoraj do kolekcji dołączył kolejny wpis; zapytany przez żonę, dlaczego siedzę do późna przy komputerze, odpowiedziałem: „Nie mogę zmusić moich zmutowanych kierowców do przejechania całej trasy". Zdanie to, choć brzmi niedorzecznie, jest jednocześnie całkiem niezłym wprowadzeniem do tematyki, którą chcę poruszyć w niniejszym artykule.

AUTONOMICZNY SAMOCHÓD

Można powiedzieć, że historia rozwoju technologicznego w przeciągu kilku ostatnich wieków oparta jest w dużej mierze na ciągłym podważaniu tego, co jest możliwe, a co nie. Z rozbawieniem czyta się autorytatywne wypowiedzi fizyków (bodaj) XVI wieku, którzy twierdzili, że nic cięższego od ptaka nigdy nie wzniesie się w powietrze. Ale przecież nawet kilka lat temu na takie koncepcje, jak choćby komputer malujący obrazy, patrzyliśmy w zasadzie tylko z perspektywy opowieści science-fiction, a tymczasem już teraz w Internecie możemy znaleźć takie programy jak na przykład DALL•E, który potrafi wygenerować zdjęcia i obrazy na podstawie tekstowego opisu, lub też darmowy Nvidia Canvas, który z kilku maźnięć potrafi wygenerować fotorealistyczny pejzaż.

Autonomiczne pojazdy – poruszające się bez bezpośredniego udziału człowieka – przez jakiś czas również były postrzegane z perspektywy hipotetycznej ciekawostki. Następnie pojawiły się wielowirnikowce, które zaczęły latać po wyznaczonych trasach (omijając przy okazji przeszkody), ale to wciąż można było osiągnąć dosyć prostym, deterministycznym algorytmem. Potem jednak pojawiła się Tesla, która jest w stanie samodzielnie nawigować w rzeczywistym ruchu ulicznym, i nagle okazało się, że hipotetyczna ciekawostka przestała być hipotetyczna i w zasadzie – jako rozwiązanie komercyjne – przestała być również ciekawostką.

Mechanizm, który chcę zaprezentować, jest oczywiście zdecydowanie mniej skomplikowany niż te, które zaimplementowane są w popularnych elektrycznych samochodach, ale pomimo swojej prostoty, można z powodzeniem pokusić się o wdrożenie go na przykład w małych, samobieżnych robotach.

ZAŁOŻENIA

Dla uproszczenia przyjmijmy, że poruszamy się tylko w przestrzeni dwuwymiarowej. Dana jest trasa do przejechania, w ramach której otrzymujemy:

- Zbiór odcinków reprezentujących nieprzekraczalne krawędzie trasy; kolizja z takim odcinkiem traktowana będzie jako rozbicie się samochodu.
- Odcinek reprezentujący metę; kolizja z takim odcinkiem traktowana jest jako prawidłowe przejechanie całej trasy.
- Punkt startowy w postaci współrzędnych oraz orientacji, czyli początkowego kąta ruchu samochodu.

Oprócz tego przydatna okaże się również prowadnica; będzie nią krzywa reprezentująca pojedynczy, prawidłowy przejazd. Dzięki niej będziemy w stanie punktować przejazdy naszych wirtualnych samochodów, co za chwilę okaże się konieczne do zbudowania działającego algorytmu automatycznego kierowcy.

Następnym krokiem jest zdefiniowanie fizyki ruchu samochodu.

Naszym celem jest powołanie do życia algorytmu, który mógłby prowadzić samochód w rzeczywistości, więc w naszym interesie leży skonstruowanie modelu, który – w miarę możliwości – najbardziej przypomina prowadzenie prawdziwego samochodu. Aby jednak udało nam się osiągnąć pozytywne rezultaty w możliwie krótkim czasie, konieczne będzie zaakceptowanie pewnej liczby uproszczeń. I tak, oprócz tego, że poruszać się będziemy na płaszczyźnie (pomijamy więc podjazdy, zjazdy, tunele itp.), przyjmujemy również uproszczony model fizyczny, w którym samochód jest punktem, dzięki czemu nie musimy przejmować się też takimi parametrami jak opory toczenia, opór powietrza lub choćby pogoda. Nic nie stoi na przeszkodzie, by wytrenować sieć neuronową, która poradzi sobie również w bardziej złożonych scenariuszach, ale wtedy konieczne jest również zaimplementowanie odpowiednio realistycznego symulatora.

Na początku zdefiniujmy pewien zestaw parametrów, które będą opisywać samochód. Przede wszystkim chcemy umieścić go gdzieś w przestrzeni, więc konieczne będzie zdefiniowanie jego pozycji. Oprócz położenia musimy wiedzieć również, w którą stronę samochód jest zwrócony, bo to definiuje kierunek jego jazdy. Bez straty ogólności możemy przyjąć, że kąt 0^o oznacza jazdę w kierunku (0, 1), czyli pionowo w dół (operujemy we współrzędnych ekranu).

Praca z legacy nie musi być koszmarem!

Dołącz do mailingu

legacyfighter.pl

Samochodem możemy sterować przy pomocy pedałów gazu i hamulca, które wpływają na prędkość jazdy, oraz przy pomocy kierownicy, która definiuje kierunek poruszania się pojazdu. Warto jednak zauważyć, że nie możemy bezpośrednio określić docelowej prędkości jazdy: jesteśmy ograniczeni przez fizyczną konstrukcję pojazdu – jego maksymalne przyspieszenie i prędkość, drogę hamowania czy też maksymalny promień skrętu. Musimy więc przede wszystkim zdefiniować te właśnie parametry. Ponieważ (tymczasowo) poruszamy się po ekranie komputera, jednostką długości są piksele, więc maksymalną prędkość do przodu i do tyłu wyrazimy przy pomocy jednostki `PixelsPerSecond` (piksele na sekundę), zaś maksymalne przyspieszenie oraz opóźnienie – w jednostce `PixelsPerSecondSquared` (piksele na sekundę do kwadratu). Maksymalny promień skrętu zapiszemy jako liczbę stopni na sekundę, czyli `DegreesPerSecond`, zaś maksymalną zmianę promienia skrętu (co przekłada się na to, jak szybko jesteśmy w stanie obrócić kierownicę, by zmienić kierunek jazdy) wyrazimy jako zmianę promienia skrętu, czyli liczbę stopni na sekundę do kwadratu (`DegreesPerSecondSquared`).

Prawdziwy samochód prowadzimy poprzez aplikowanie zmian do jego prędkości oraz promienia skrętu. Nasz mózg, bazując na wszystkich informacjach, które do niego docierają, generuje odpowiednie wartości zmian prędkości (metrów na sekundę do kwadratu) oraz zmian kierunku jazdy (stopni na sekundę do kwadratu), które aplikujemy przez wciśnięcie pedałów gazu, hamulca i obrócenie kierownicy. Teraz musimy „tylko” skonstruować algorytm, który będzie realizował podobne zadanie.

Zauważmy, że problem nie jest wcale zbyt prosty do rozwiązania wprost. Gdybyśmy mogli bezpośrednio i bez opóźnień wpływać na prędkość i kierunek jazdy, możliwe byłoby napisanie prostego algorytmu deterministycznego. Z aplikowaniem zmian jest już niestety gorzej. Zauważmy na przykład, że czasem przed wykonaniem skrętu musimy zredukować prędkość, by zmieścić się w krzywiźnie łuku, potem musimy obrócić kierownicę o odpowiedni kąt, by samochód zaczął skręcać, zaś po wykonaniu skrętu musimy obrócić kierownicę z powrotem, aby samochód na powrót jechał prosto. A żeby tego było mało, jeśli popełnimy błąd (co oczywiście może się zdarzyć), trzeba samochód zatrzymać, wycofać i ponowić manewr. Ubranie tego procesu w serię warunków i wzorów nie wydaje się być najlepszym rozwiązaniem.

Dlatego też podejdziemy do problemu w zupełnie inny sposób.

IMPLEMENTACJA

Zacznijmy powoli pisać trochę kodu. Całość oprogramujemy w języku C# dla .NET 6, dzięki czemu będziemy mogli skorzystać z kilku ciekawych konstrukcji językowych. Dodajmy też od razu, że kompletna implementacja całego algorytmu jest otwarta i dostępna pod adresem: https://gitlab.com/spook/GeneticAlgorithm.

Przede wszystkim, aby uniknąć trudnych do wykrycia błędów wynikających z jednostek, oprogramujmy każdą z nich jako osobną strukturę. Dla przykładu, typ opisujący przyspieszenie może wyglądać następująco:

Listing 1. Struktura opisująca przyspieszenie

```
public record struct PixelsPerSecondSquared(
    double Value)
{
    public PixelsPerSecondSquared Clamp(
        PixelsPerSecondSquared min,
        PixelsPerSecondSquared max)
    {
        if (min.Value > max.Value)
            throw new ArgumentOutOfRangeException(
                nameof(min));

        return new(Math.Min(max.Value,
            Math.Max(min.Value,
                Value)));
    }

    public static PixelsPerSecond operator * (
        PixelsPerSecondSquared acceleration,
        Second time)
    {
        return new(acceleration.Value * time.Value);
    }

    public static PixelsPerSecond operator * (
        Second time,
        PixelsPerSecondSquared acceleration)
    {
        return new(acceleration.Value * time.Value);
    }

    public static Pixels operator * (
        PixelsPerSecondSquared first,
        SecondsSquared second)
    {
        return new(first.Value * second.Value);
    }

    public static Pixels operator *(
        SecondsSquared first,
        PixelsPerSecondSquared second)
    {
        return new(first.Value * second.Value);
    }

    public static PixelsPerSecondSquared operator * (
        PixelsPerSecondSquared first,
        double second)
    {
        return new(first.Value * second);
    }

    public static PixelsPerSecondSquared operator * (
        double first,
        PixelsPerSecondSquared second)
    {
        return new(second.Value * first);
    }

    public static PixelsPerSecondSquared Zero { get; }
        = new(0.0f);
}
```

Przeciążanie operatorów to prawdziwe błogosławieństwo: dzięki niemu wszystkie wzory, z których będziemy korzystać w kodzie, możemy zapisać w bardzo naturalny sposób, na przykład `Speed = (Speed + acceleration * time)` (fragment metody `Car.ApplyMotionChange`). Nie ma też większych szans na pomylenie jednostek – kompilator bardzo szybko będzie protestował.

Teraz możemy przygotować strukturę, która zdefiniuje parametry samochodu – Listinng 2.

Listing 2. Parametry samochodu

```
public record class CarDefinition
{
    public CarDefinition(
        PixelsPerSecondSquared maxAcceleration,
        PixelsPerSecondSquared maxDeceleration,
        PixelsPerSecond maxSpeed,
        DegreesPerSecondSquared maxTurningRateChange,
        DegreesPerSecond maxTurningRate)
    {
        if (maxAcceleration.Value <= 0.0)
            throw new ArgumentOutOfRangeException(
                nameof(maxAcceleration));
        if (maxDeceleration.Value >= 0.0)
```