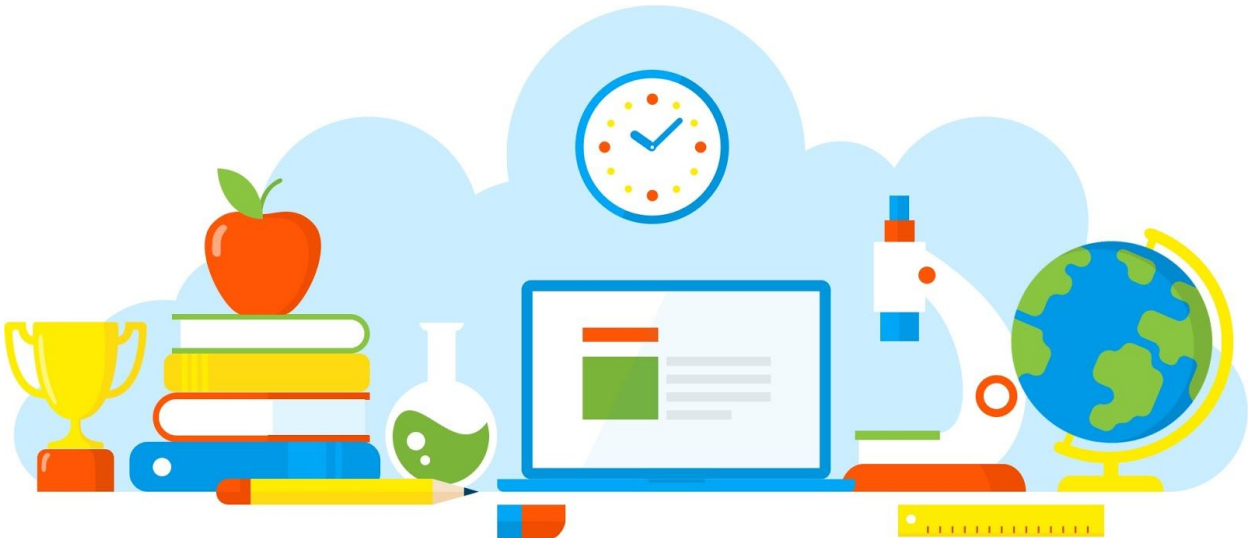


대규모 데이터분석 특강

Final Project Report

Training 'Sonic The Hedgehog Genesis' with Reinforcement Learning using Neural Network and Genetic Algorithm

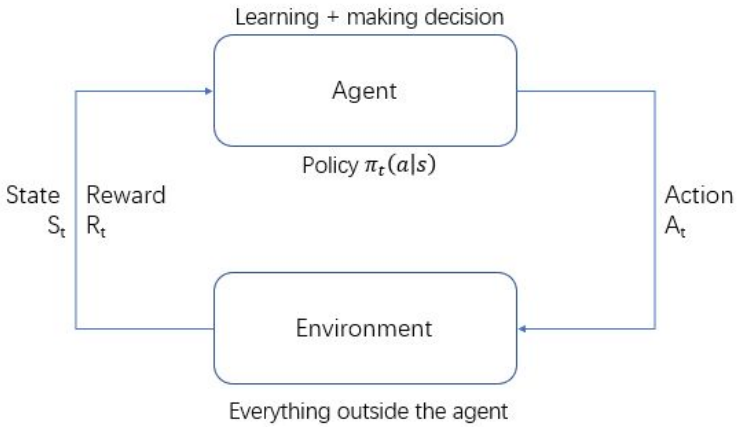
박민우, 박성준, 육예진수

2019년 6월 10일

Introduction

Introduction to Reinforcement Learning

강화 학습(Reinforcement learning)은 기계 학습의 한 영역이다. 어떤 환경(environment) 안에서 정의된 에이전트(agent)가 현재의 상태(state)를 인식하여, 선택 가능한 행동(action)들 중 보상(reward)을 최대화하는 행동 혹은 행동 순서를 선택하는 방법이다.



에이전트(Agent) : 상태를 관찰, 행동을 선택, 목표지향

환경(Environment) : 에이전트를 제외한 나머지 (물리적으로 정의하기 힘듦)

상태(State) : 현재 상황을 나타내는 정보

행동(Action) : 현재 상황에서 에이전트가 하는 것

보상(Reward) : 행동의 좋고 나쁨을 알려주는 정보

심층 인공신경망이 도입되기 전 강화학습은 지속적인 발전에도 불구하고 계산 복잡도, 차원의 저주등의 문제를 겪고 있었다. 이러한 문제를 딥러닝(Deep Learning: DL)의 특성인 경험 재생(experience replay)을 통한 함수 근사(function approximation)를 통해 해결한 연구들이 발표되었다. DL과 강화학습이 결합되어 딥강화학습(Deep Reinforcement Learning :DRL) 이라고 부른다. DRL은 고차원의 물리 기반 캐릭터 애니메이션 환경 에서 에이전트가 물리 기반 제어의 행동을 학습하고 강건하게 재생할 수 있는 가능성을 열어 주었다.

문제 정의

게임 선정

우리 그룹은 본 프로젝트를 통해 강화학습으로 Retro 게임을 학습시키고자 한다. 논문, 프로젝트, 블로그 등으로 이미 많이 다뤄진 대부분의 Atari 게임보다 복잡하지만 높은 하드웨어 사양이나 복잡도를 요구하는 최신게임보다는 단순한 게임을 학습 시키고자 하여 Retro 게임을 학습 대상으로 선정하였다. 첫 프로젝트 제안에서는 TENGAI라는 게임을 선택하였었지만 이를 지원하는 도구인 MAMEToolkit을 사용하는데에 여러 어려움을 직면하여 정보도 쉽게 얻을 수 있고 대기업에서 개발한 오픈소스 도구인 Gym-Retro를 사용하기로 결정하였다. Gym-Retro에서 지원하는 여러 게임중에서도 우리 그룹은 'Sonic The Hedgehog-Genesis'라는 게임을 학습하고자 한다. 우리 그룹은 게임에 대한 지식이 전혀 없는 AI를 강화학습으로 학습시켜서 'Sonic The Hedgehog-Genesis'를 플레이하고 스테이지를 클리어 할 수 있도록 만들고자 한다.

소닉(Sonic) 게임 소개

우리 그룹은 'Sonic The Hedgehog-Genesis'라는 게임을 본 프로젝트에서의 학습대상으로 선정하였다. 본 게임은 1991년 6월23일에 발매되었으며 게임의 이름에서도 알 수 있듯이, 주인공 캐릭터로 게임 회사 세가(SEGA)의 마스코트인 소닉(Sonic)이라는 캐릭터가 나온다. 세가는 전세계적으로 매우 유명한 게임회사이며 당시, 닌텐도의 슈퍼마리오 브라더스를 벤치마킹함과 동시에 세가만의 독창적인 게임을 만들고자 한 프로젝트의 결과물이다. 슈퍼마리오 브라더스와 같은 횡스크롤 게임으로 기본적으로 앞뒤, 그리고 점프를 하며 목적지에 도달하여 스테이지를 클리어해 나아가는 게임이다. 하지만, 슈퍼마리오와 다른 차별점으로 소닉은 조작법이 더 단순하고 급경사, 스프링 그리고 360도 루프와 같은 지형등을 통해 게임플레이에서 스피드감에 중점을 두었다. 이렇게 스피드감을 중요시하는 설정에 걸맞게 소닉이라는 캐릭터 자체가 초음속으로 달리며 몸을 둥글게 말아 공격하여 적을 처치하는 고슴도치라는 설정을 갖고 있다.

게임 목적

'Sonic The Hedgehog-Genesis'는 방향키와 점프버튼(A 버튼)을 이용하여 장애물과 적들을 통과하여 목적지를 통과하여 스테이지를 완료하는 게임이다. 스테이지는 몇 개의 존(Zone)으로 나뉘어 있고, 또 각 존은 몇 개의 액트(Act)로 나뉘어 있다. 각 액트는 저마다의 목적지가 존재한다. 본 프로젝트에서는 강화학습을 통해 첫 번째 스테이지의 첫 번째 액트를 클리어하는 것을 목표로 한다. 그리고 학습이 잘되었는지 다른 액트에서도 학습된 Agent를 테스트해 보았다.

게임 중요 요소

본 게임은 앞서 말했듯이 스피드감이 중요하여 소닉이 일정 속도 이상을 유지했을 때만 통과 할 수 있는 루프, 높은 언덕, 나선형 코스가 많이 존재한다. 또한, 시간에도 제약이 있다. 모든 액트는 10분 이내로 클리어해야 하고 시간이 길어질 수록 목적지에 도달했을 때 얻는 보너스 점수가 적어진다.

또하나의 중요 요소로 링(Ring) 시스템이 있다. 소닉은 맵 곳곳에 존재하는 링을 획득함으로써 적에게 공격을 당해도 생존할 수 있다. 링의 보유 개수와는 상관없이 링이 하나라도 있으면 소닉은 죽지 않는다. 적에게 공격당하면 보유하고 있던 링을 바닥에 떨어뜨리며 빨리 다시 줍지 않으면 금방 사라진다. 링이 하나도 없을 때 적에게 공격당하면 비로소 소닉이 죽으며 목숨(Life Count)이 하나 소모된다. 이러한 링 시스템 덕분에 한번 공격으로 바로 목숨을 잃는 슈퍼마리오에 비해서 게임의 흐름을 유지하기 좋았고 세가입장에서 난이도 조절을 하기에도 용이하였다. 또한, 링을 많이 보유하고 있으면 그 개수에 비례하여 목적지에 도달했을 때 보너스 점수를 받고 링을 100개를 모았을 때는 목숨을 하나 추가로 얻을 수 있다.

따라서, 소닉을 잘 플레이한다는 것은 빠르게 이동하여 짧은 시간안에 목적지에 도달하면서도 그 과정에서 링을 최대한 많이 획득하는 것이다. 우리는 이러한 소닉 게임의 특성을 살리는 방향으로 우리의 Agent를 학습시키고자 한다.

Survey

Existing Approaches to retro games

Retro games를 딥강화학습 방법으로 학습한 성공한 예제는 여러 가지가 있다. 예를 들어 actor-critic 방법이라는 value 함수랑 policy를 동시 optimize하는 학습방법이 있는데, Policy structure는 action을 선택하는데 사용되기 때문에 actor라고 하고, value 함수를 구하기 위한 value estimator는 선택된 action을 평가하는 역할을 하므로 critic이라고 한다. 그래서 Actor-critic 방법은 두 개의 parameter set을 가져야 하는데, 하나는 critic이 update하는 action value function을 위한 parameter ω이고, 다른 하나는 actor가 policy를 위한 parameter θ이다. 이 θ는 critic이 제안하는 방향으로 θ를 update시키게 된다.

또는 DQN (Deep Q-Network)이라는 방법은 고전게임의 raw pixel들을 input으로해서 CNN을 함수 근사치로 이용하여 value 함수을 output으로 내는 학습방법이다. DeepMind의 paper (Playing Atari with Deep Reinforcement Learning)에서는 DQN을 이용하여 고전게임인 Atari 게임에서 성공을 선보이었다. DQN algorithm의 핵심은 Q-learning이고 그러므로 Q value의 학습이 가장 중요하다. Q-value의 update는 optimizer를 사용하여 실행한다. DQN에서 사용하는 Q networks는 사실 2개이다. 하나는 target Q network이고 다른 하나는 Q network이다. Target Q network를 별도로 두는것이 특징이다. 두 networks는 weight parameter만 다르고 완전히 같은 network이다. DQN에서는 수렴을 원활하게 시키기 위해 target network는 계속 update되는 것이 아니라 주기적으로 한번씩 update하도록 되어 있다.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Algorithm from paper: Playing Atari with Deep Reinforcement Learning

하지만 DQN는 많은 저장용량 및 계산량이 요구되며, replay memory로 부터 데이터를 가져오기 때문에 과거의 policy에 의존하여 update를 수행해야 하는 off-policy method일 수 밖에 없는 단점을 가지고 있다. 이러한 배경에서 2016년 초에 발표한 Asynchronous Methods for Deep Reinforcement Learning라는 논문에서 A3C(asynchronous advantage actor-critic)라는 방법을 제출한다. Experience replay대신 고안한 아이디어는 environment를 여러개 구동시켜 비동기적으로 agent를 병렬로 실행시키는 방법이다. 이러한 병렬환경은 agent의 데이터의 상호 연관성을 줄이고, 각 agent가 다양한 state를 경험하게 해주므로 학습과정이 stationary process가 되도록 유도한다. 이와 같은 간단한 아이디어로 인해 Q learning과 같은 off-policy method뿐만아니라 더 다양하게 개발되어 있는 on-policy method (예를 들면, Sarsa, TD, actor-critic)도 적용가능하게 만들어 준다. A3C는 discrete action space뿐만 아니라 continuous space에도 적용 가능하며, feedforward와 recurrent agent도 모두 학습시킬 수 있는 장점도 가지고 있다.

Algorithm 1 Asynchronous one-step Q-learning - pseudocode for each actor-learner thread.

// Assume global shared θ, θ^-, and counter $T = 0$.
Initialize thread step counter $t \leftarrow 0$
Initialize target network weights $\theta^- \leftarrow \theta$
Initialize network gradients $d\theta \leftarrow 0$
Get initial state s
repeat
  Take action a with ϵ-greedy policy based on $Q(s, a; \theta)$
  Receive new state s' and reward r
  $y = \begin{cases} r & \text{for terminal } s' \\ r + \gamma \max_{a'} Q(s', a'; \theta^-) & \text{for non-terminal } s' \end{cases}$
  Accumulate gradients wrt θ: $d\theta \leftarrow d\theta + \frac{\partial (y - Q(s,a;\theta))^2}{\partial \theta}$
  $s = s'$
  $T \leftarrow T + 1$ and $t \leftarrow t + 1$
  if $T \mod I_{target} == 0$ **then**
    Update the target network $\theta^- \leftarrow \theta$
  end if
  if $t \mod I_{AsyncUpdate} == 0$ or s is terminal **then**
    Perform asynchronous update of θ using $d\theta$.
    Clear gradients $d\theta \leftarrow 0$.
  end if
until $T > T_{max}$

Algorithm from paper: Asynchronous methods for deep reinforcement learning

Neuro-Evolution of Augmented Topologies

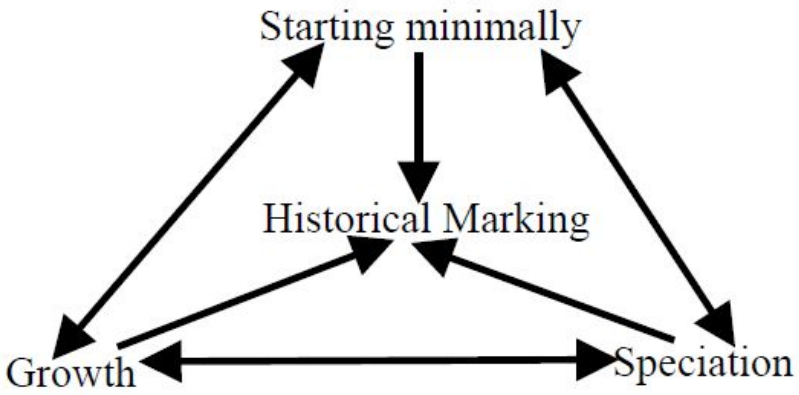
강화학습은 그 이론적 기반이 마르코프 의사결정 프로세스(Markov Decision Process; MDP)에 있으며, value function 또는 action-value function을 정의하고 이것을 학습함으로써 주어진 과제를 달성할 수 있다는 것을 쉽게 보일 수 있다. 하지만 현실에서 우리가 재미있다고 느낄 수준의 문제에서는 이런 방법이 유효하지 않은데, 그 이유는 state가 연속적이거나 매우 많은 경우가 일반적이라 계산이나 메모리 사용을 효율적으로 할 수 없기 때문이다.

그러한 이유로 인해 Q-Table을 이용해 모든 action-value와 reward를 외우는 방식에서 벗어나 다항함수나 인공 신경망을 이용해 value function 또는 policy를 근사하는 방법이 소개되었다. 하지만 이것 또한 한 episode가 종료되어야 reward를 얻는 게임 등 implicit한 환경에서 효율적으로 작동하지 않았으며, Temporal-difference 또는 Monte Carlo 방식의 return sampling 또한 학습에 시간이 많이 걸리는 등 한계가 있다.

한편, Neuro-evolution은 상술한 신경망을 구축하는 데 유전 알고리즘의 방법으로 접근하는 분야이다. 유전 알고리즘은 게임의 데이터로부터 직접 모델을 학습하는 부분이 전혀 없으며, 다양한 모델(Neuro-evolution의 경우 신경망에 해당함)을 생성하고 각 모델을 episode 끝까지 플레이하여 높은 reward를 얻는 모델을 남기는 방식이다. 따라서

데이터와 경험으로부터 어떤 계산을 하지 않아도 되며 적합한 신경망이 알아서 진화하는 장점이 있다.

그러나 유전 알고리즘의 특성상 이런 선순환은 정책의 표현에 선지식이 적절한 형태로 잠재되어 있는 경우에 한해 가능했으며, 신경망의 파라미터를 상속하기 쉬운데 반해 신경망의 구조를 상속하는 것은 까다로웠다. 때문에 신경망의 구조를 정하는 과정에서 많은 시행착오를 필요로 했다.



NEAT는 이런 문제를 해결하고, Neuro-evolution이 파라미터 뿐만 아니라 신경망의 구조를 효과적으로 유전할 수 있도록 제안된 알고리즘 프레임워크이다. NEAT에서는 모델의 표현과 모델이 상속되는 방식의 주요한 부분인 교차 연산(crossover operation)을 효과적으로 정의하기 위해, 유전자에 히스토리 마커를 붙인다. 기존의 그래프 표현의 문제는 competing conventions problem을 가지는데, 이것은 두 모델의 표현이 인코딩된 방식이 달라서 통합이 되지 않을 때 발생하는 문제이다. NEAT에서는 이것을, 모델의 유전자에 해당하는 부분의 기원을 추적함으로써 해결한다.

또한, NEAT에서는 새로운 구조를 가지는 신경망을 도입했을 때 이 신경망이 즉시 낮은 적합도로 인해 도태되지 않게 하기 위해 종(species) 개념을 차용했다. 따라서, 신경망 모델은 한 종 내에서 적합도를 가지고 경쟁하게 되며 다른 종보다 적합도가 떨어진다고 해서 즉시 Population에서 제거되지 않는다. 이 방법으로 NEAT는 잠재적으로 유효한 구조의 신경망을 보존할 수 있었다.

마지막으로 NEAT가 신경망의 구조를 적응적으로 구성하는 장점을 최대화하기 위해, 신경망은 필요한 최소한의(minimal) 구조로부터 출발해 유전자(정점 또는 간선)를 추가하는 방식으로만 이루어진다. 이를 통해 파라미터의 튜닝과 신경망 구조의 진화가 번갈아가며 이루어지며 게임을 플레이하는 적절한 모델이 효과적으로 만들어질 수 있다.

제안기법 : NN, GA(NEAT)

Intuition

1) NN의 RL에서 Advantages (간단 설명)

Reinforcement learning(RL)은 우수한 의사결정 능력으로 인공지능 분야에서 광범위하게 응용하고 있다. 그러나 초기의 강화 학습은 주로 인공적으로 특징을 추출하는 것을 의존하며, 복잡한 고차원적 상태 공간 문제를 잘 처리하지 못한다. Deep learning(DL)의 발전에 따라 알고리즘은 원시적인 고차원데이터에서 직접 특징(features)을 추출할 수 있게 되었다. DL은 감지 능력이 강한데, 의사 결정 능력이 결여되어 있는 반면에, RL은 강한 의사 결정 능력을 가지고 있지만 문제를 감지하는 데는 속수무책이다. 그러므로, 양자를 결합해서 복잡한 상태의 감지, 의사결정 문제에 대한 해결책을 제공할 수 있다.

2015년, DeepMind 팀은 Deep-Q-Network(DQN)를 제안했다. DQN은 게임의 원시 이미지만으로 입력하고 인공적으로 features extraction에 의존하지 않는 학습 방식이다. 경experience replay과 fixed-target Q network를 통해서 DQN는 non-linear Q function approximator의 불안정성과 발산성 문제를 잘 해결하였다.

Asynchronous advantage actor-critic(A3C)방법에서 사용하는 NN는 output layer이외의 다른 layer가 parameter를 공유한다. convolutional layer과 softmax 함수를 통해 policy distribution π를 output한다. 또는 entropy of π를 loss function에서 쓰고 exploration 과정을 격려하므로 model가 local optimum으로 가는 것을 방진하다. A3C 알고리즘은 비동기적으로 학습하는 사상으로 여러 개의 학습 환경에서 sampling한 sample들를 직접 사용해서 학습시킨다. DQN 알고리즘에 비해, A3C 알고리즘은 experience replay을 사용하여 사용했던 samples을 저장할 필요가 없어서 스토리지 공간을 절약하고 데이터를 sampling efficiency를 상형시키므로 학습속도를 높일 수 있다. 그리고, 여러 개의 서로 다른 environments에서 받은 샘플의 분포는 더 균일해서 NN을 학습하기에 도움이 될 수 있다.

2) 유전 알고리즘에 대한 소개

지구상에는 다양한 종의 생물이 존재하며, 설계자 없이도 환경에 고도로 적응한 생태를 보이며 살아간다. 이것은 주어진 환경에서 생존과 생식에 적합한 특성을 갖게 하는 성질이 있다면 그것이 존재하게 되는 원리인 진화에 기인한다. 물질은 두 가지로 나눌 수 있는데 에너지를 많이 보유한 불안정한 물질과 에너지를 이미 방출한 안정한 물질이 그것이다. 이 때, 주변에서는 안정한 물질을 관찰하기가 쉽다. 이것은 불안정한 물질은 그 정의에 의해서 에너지를 방출하여 안정한 상태인 물질로 수렴하려는 경향이 있기 때문이다. 진화는 물질의 안정성의 특수한 경우로, 자기복제가 가능한 생명체의 세계에서 나타나며 넓은 환경과 상호작용한다.

유전 알고리즘이란 복잡한 문제를 푸는 방법을 상술한 생물의 진화에서 착안한 알고리즘이다. 이 접근 방법은 프로그래머가 환경과 답안의 상호작용만을 명시적으로 작성한 뒤, 환경을 시뮬레이션함으로써 주어진 문제를 푸는 적절한 알고리즘을 찾는다. 이를 위해 프로그래머는 답안을 어떻게 표현할지를 설계해야 한다.

어려운 문제의 예를 들어, 그래프를 두 클래스로 나누었을 때(bipartite graph) 서로 다른 클래스 속한 정점에 연결된 간선의 가중치의 합이 최대가 되게 하는 문제를 Max-cut 문제라고 한다. 이것은 문제의 계산이 문제의 크기에 대한 다항 시간 안에 끝나지 않는 어려운 문제이다. 이 문제를 유전 알고리즘으로 푸는 경우, 프로그래머는 답안(solution; 해)을 각 노드가 속한 클래스를 나타내는 이진수 벡터로 나타낼 수 있다. 이로부터 유효한 답안의 정의역이 $\{0,1\}^{|G|}$로 정해지며, 한 답안이 주어졌을 때 이 답안이 기술하는 그래프와 그 때의 상술한 가중치의 합의 값을 유일하게 결정할 수 있다.

문제에 대한 선지식(prior knowledge)를 적용하는 대신, 유전 알고리즘은 임의로 초기화된 답안들의 가중치 합을 계산한다. 문제는 합을 최대화하는 답안을 찾는 것이므로, 임의로 초기화된 답안 가운데 높은 가중치 합을 갖는 목표에 상대적으로 더 근접한 답안이 존재한다. 유전 알고리즘은 생명의 진화를 모방하여 각 답안들을 혼합하고(교차 연산; crossover) 때로는 임의로 변형한다(변이 연산; mutation). 동시에 가중치 합이 낮은 답안들은 덜 선택되어 서서히 도태되며, 가중치 합이 높은 답안들의 재조합과 변형으로 대신한다. 유전 알고리즘은 일련의 과정으로부터, 좋은 답 표현을 중심으로 한 교차와 변이 연산을 이용한 답안 공간의 탐색이 더 좋은 답과 궁극적으로는 전역 최적해를 찾음으로써 어려운 문제를 풀 것을 기대한다.

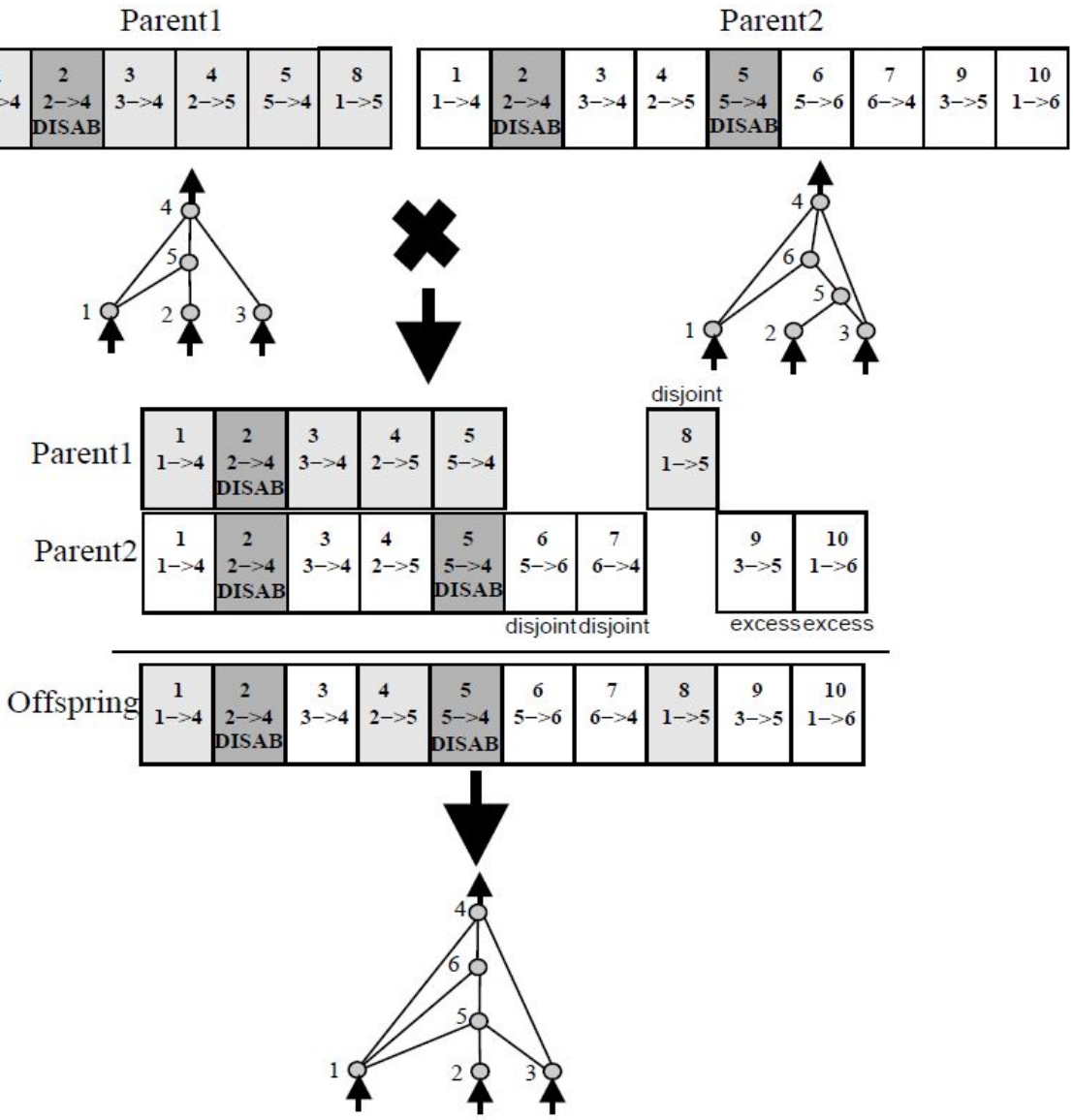
유전 알고리즘에 대한 이와 같은 설명은 언뜻 문제에 대한 선지식을 요구하지 않는다는 오해를 받기 쉽다. 하지만, 유전 알고리즘에서 가정하는 것과 같이 좋은 답안을 섞음으로써 더 좋은 잠재적인 답을 찾을 수 있으려면 답안의 표현과 그에 따라 정의된 교차 연산이 기존의 좋은 답안이 좋은 이유, 즉 위의 예시에서대로라면 높은 가중치의

합을 가질 수 있었던 요인이 보존되어야 한다. 이것은 답안의 표현을 결정하는 과정에서 문제에 대한 선지식이 필요하며, 좋은 결과를 얻으려면 개체(개별 해)이 아닌 실제로 우수한 유전자를 우대해야 하도록 환경을 정의해야 함을 알 수 있다.

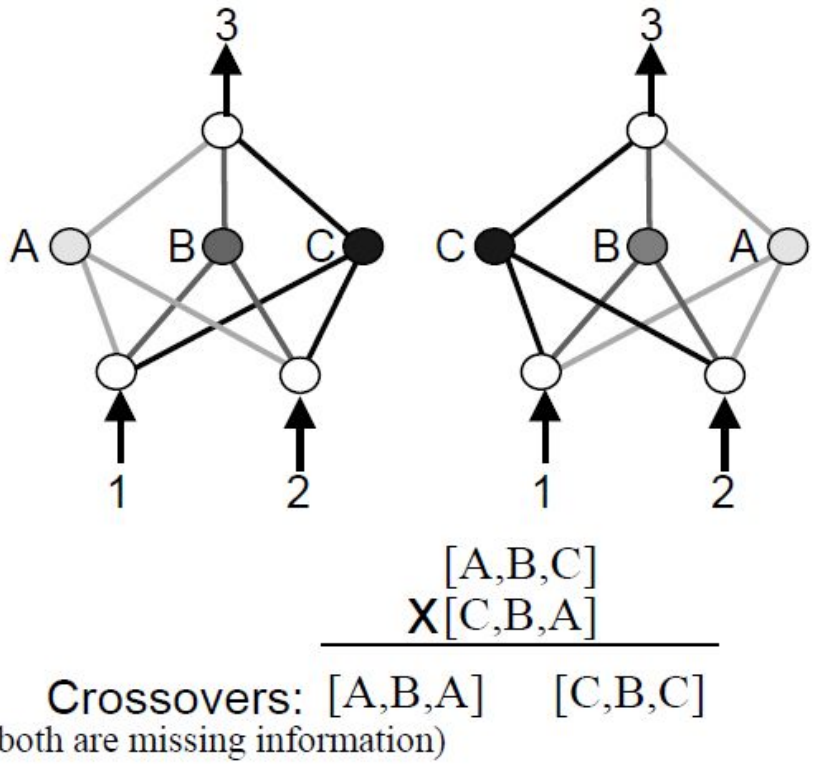
3) NEAT 소개와 NEAT의 advantages & reasoning

유전 알고리즘의 하위 분야로 이것을 이용해 인공 신경망(Neural Network; NN)의 매개변수 또는 구조를 생성하는 분야를 Neuro-evolution이라고 한다. 지도학습으로 인공 신경망을 생성하는 것과 비교했을 때, 지도학습은 참값이 레이블된 데이터가 필요한 데 반해 Neuro-evolution은 생성된 신경망의 적합도를 계산할 방법만 있으면 레이블된 학습 데이터 없이 신경망의 조율이 가능하다는 것이다. 우리의 강화학습 프로젝트에서 사용한 NEAT-Python은 NEAT를 파이썬으로 구현한 패키지이다.

이러한 Neuro-evolution의 성질은 강화학습와 결합하여 사용되기에 대단히 적합하다. 강화학습 역시 전통적인 비지도 학습과 다르게 레이블된 데이터보다는 에이전트의 시뮬레이션 결과 얻어지는 점수를 가지고 에이전트를 학습시킬 수 있기 때문이다. 또한, Neuro-evolution은 state에 대한 평가(value function network)를 학습하려는 것이 아니라 직접적으로 할 행동(policy)을 학습하기 때문에 연속적이고 고차원으로 구성된 state를 가지는 문제를 해결하는 데 적합하다.



NEAT는 Neuro-evolution of Augmenting Topologies의 머리글자로, Ken Stanley에 의해 고안된 알고리즘이다. 기존의 Neuro-evolution 방법론들은 매개변수만의 조율에 그치거나 신경망의 구조를 학습하는 데에 효과적이지 못했다. NEAT는 매개변수와 신경망의 구조를 모두 업데이트하면서, 주어진 과제를 높은 점수로 푸는 해를 생성하면서도 동시에 유전 알고리즘이 가지는 해집단의 다양성을 유지한다. 이를 위해 NEAT는 세가지 기법을 사용하는데, (1) 유효한 교차 연산을 위해 유전자에 히스토리 마커를 붙여 두 신경망 구조의 동일성을 추적하고 (2) 종(species)의 개념을 도입해 잠재적으로 우수한 구조를 풀에 유지하며 (3) 최소한의 구조에서 시작해 신경망을 성장시킨다.



신경망을 해로 가지려면 그래프를 효율적으로 인코딩해야 하는데, 이것은 두 해 사이에 좌위(gene locus)의 정보가 일관되지 않게 하는 부작용이 있다. 예를 들어, 은닉 계층이 A-B-C 인 그래프와 C-B-A 인 그래프는 완전히 동일하지만, 이 두 그래프에 단순한 일점 교차(one-point crossover) 연산을 적용해 얻어진 C-B-C라는 그래프는 기존의 정보 1/3을 잃는다. 자연계에서도 단세포가 복잡한 생명으로 진화하면서 새로운 유전자가 추가되는데, 이에 대한 정렬 없이 교차가 일어난다면 유전정보는 파괴되고 진화는 일어나지 않았을 것이다. 이를 방지하는 장치는 대장균의 RecA 단백질 등에서 관찰된다. NEAT에서는 해를 구성하는 각 신경망의 유전자(이 경우, 정점과 간선)가 동일한 조상에서 기인했는지를 추적해 만일 그렇다면 교차 연산에서 이것을 보전하는 방식을 채택해 의미있는 교차 연산을 구현하였다.

또한, 유효한 신경망 구조라 하더라도 새 구조가 분화한 초기에는 적합도 점수가 낮을 수밖에 없어서 잠재적으로 품질이 좋은 해가 도태되는 경우가 있다. 이것을 방지하기 위해 NEAT에서는 종의 개념을 도입하였다. '유전적으로' 거리가 가까운 해들은 정해진 기준에 따라 하나의 종으로 묶이며, 각 종은 적합도의 차이가 나더라도 어느 한 종이 생태계를 독점하지 않게 조절함으로써 새로운 신경망의 출현이 초기에 제거되는 것을 막는다.

마지막으로, NEAT와 Neuro-evolution의 주요한 강점 가운데 하나는 매개변수 뿐만 아니라 적합한 신경망을 시행착오 없이 학습으로 결정하는 것에 있다. 따라서, 불필요한 구조의 신경망을 탐색하는 것을 줄이기 위해 NEAT에서 사용하는 신경망의 구조는 최소한에서 시작하여 구조적인 변이를 추가하는 방식으로만 신경망을 성장시킨다. 이같은 방법은 거시적으로 현재 해집단에 있는 신경망의 매개변수가 어느정도 튜닝된 이후에 새로운 구조로 확장해나갈 수 있도록 한다.

Description of Method

우리는 NEAT 패키지를 사용해 소닉을 플레이하는 신경망을 만들었다. 먼저 후술할 매개변수들을 설정하고 초기화를 했다. 처음에는 Population의 크기에 해당하는만큼의 해가 존재하고, 각 해는 state 입력과 action 출력을 할 수 있는 최소한의 RNN(Recurrent Neural Network)의 구조를 갖고 파라미터는 임의의 값으로 초기화되어 있다. 각 해는 독립적으로 게임을 플레이할 수 있는 에이전트이고, 플레이한 결과 reward를 받게 된다. NEAT는 세대의 진행을 모방하기 위해, Population에 존재하는 해들을 부모로 하는 자식해를 만들어낸다. 부모해로부터 유전되는 특성은 RNN 간선의 파라미터 값도 물려받지만, 구조에 변이가 생겨 새로운 정점 또는 간선이 추가되기도 한다. 이 때 높은 reward를 기록한 해를 더 편애하여 선택하는 방식으로 게임을 잘 해결하는 특성이 Population에 퍼져나가기를 기대하였으며, 실제 시뮬레이션을 통해 스테이지를 클리어하는 종(species)을 발견하였다.

Experiment

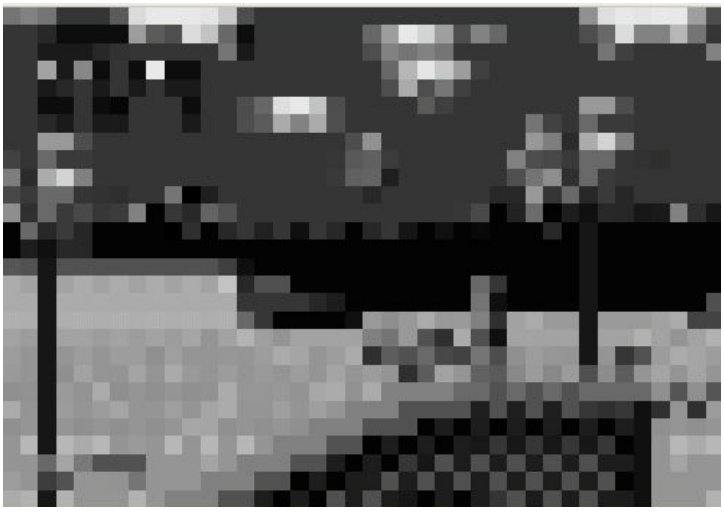
Description of your testbed; list of questions your experiments are designed to answer

1) Neural Network와 비교해서 NEAT의 Performance는?
2) 어떻게 Reward Function 및 다른 Parameter들을 조정해야 학습이 잘될 수 있을지?
3) 어떻게 하면 학습을 효율적으로 할 수 있을지?
4) Generalization은 되는지?

Details of Experiments

1) Input & Observation

본 프로젝트에서는 앞서 설명한 NEAT를 통하여 Evolving Feedforward Network로 Agent가 소닉을 플레이 하도록 하였다. 먼저 네트워크에 들어갈 Input값으로는 게임 화면의 픽셀값들을 사용하였다. 해당 픽셀값들은 Gym-Retro 패키지로 본 게임의 Environment를 구축한 후, 그 Environment에서 사용자들을 위해 생성해주는 변수들 중, Observation 변수에 저장되는 픽셀값들을 이용하였다. 소닉 게임의 해상도는 가로 320 픽셀에 세로 224 픽셀 그리고 RGB 3가지 색상 값으로, 총 215,040개의 값으로 이루어져 있다. 이러한 값들을 그대로 네트워크의 Input으로 사용하게 되면 계산량이 너무 많아지게 될 것이며 결과값인 Action 벡터를 빨리 계산하기 힘들어 질 것이고 결과적으로 플레이 속도가 느려지게 된다. 따라서, 본 프로젝트에서는 계산의 효율을 위해 해상도를 1/8로 낮추고 색상도 Grayscale을 활용하였다. 이렇게 되면 가로 40 픽셀에 세로 28 픽셀로, 총 1,120개의 값이 벡터의 형태로 네트워크에 Feed 될 것이고 이는 플레이 속도도 빠르면서 학습도 원활하게 된다는 것을 확인하였다.



2) Algorithm Parameters

먼저, 초기 Population크기를 30으로 설정하였다. 이는 학습을 시작할 때 30개의 Genome으로 시작한다는 것을 뜻한다. 그리고 여기서 각 Genome은 랜덤하게 생성된 네트워크를 의미한다. 네트워크가 생성될 때 네트워크의 weight, 구조, bias 등의 랜덤의 범위도 모두 설정이 가능하다. 자세한 모든 설정들은 'Software_Run' 압축파일에 포함된 'config-feedforward'라는 파일에 명시되어 있다.

3) Reward

우리 그룹은 본 프로젝트를 위해 NEAT의 변수들을 바꿔보고 다른 액트에서도 학습을 해보는 등 여러번의 학습을 시도해 보았다. 그 중에서도 우리 그룹이 가장 유의미하고 중요하다고 생각한 요소중 하나가 Reward Function을 어떻게 세우는지에 대한 것이었고 그렇기 때문에 서로 다른 여러 Reward Function을 적용시켜 보았다. 처음에는 Naive한 방법을 채택하였다. 소닉의 x좌표가 현재까지의 오른쪽 좌표의 최대치를 넘을 때마다 +1 Fitness 점수를 주는 것이다. 즉, 단순히 오른쪽으로 진행하면 Reward를 받는 방법으로 학습시켜 보았다. 이후에는 하나씩 Reward를 받을 수 있는 요소와 그 크기들을 조정해 나가보았다. 새로운 Ring을 획득 또는 손실 했을 시 + 또는 - Reward를 받게 하였고 소닉이 높은 곳을 갈 수록, 즉 y 좌표가 높아질 수록 Reward를 주기도 하였으며 적을 처치하여 게임 점수를 획득하였을 때 Reward를 주기도 하였다. 그리고, 일정시간동안 새로운 Reward를 받지 못하였을 때, 왼쪽으로 어느정도 진행해도 Reward를 줘서 무조건 오른쪽으로만 가는 것이 아니라 새로운 방향으로의 진행도 시도해 보도록 유도하기도 하였다.

Conclusion

우리는 Gym-Retro를 이용하여 'Sonic The Hedgehog-Genesis'라는 게임을 강화학습으로 학습하고자 하였다. Action을 결정하는 Function Approximator로 Neural Network와 Neural Network에 유전알고리즘을 적용시킨 NEAT 알고리즘을 통해 우리의 Agent를 학습하였고, 그 중에서도 우리는 NEAT 알고리즘에 초점을 맞추었다.

결과적으로 Naive한 방법론, 즉 오른쪽으로 진행하면 Reward를 받도록 하였을 때의 시도부터 여러 요소를 포함하여 정의한 Reward Function을 적용하였을 때까지 모두 Agent가 성공적으로 목표의 액트를 클리어할 수 있었다. Reward를 받을 수 있는 요소를 늘리는것이 항상 좋은 결과를 내는 것은 아니었다. 소닉의 높이에 따라 Reward를 주는 것과 같은 방식은 오히려 좋은 결과를 내는 데에 방해가 되었다. 결국, 단순한 Reward Function을 적용했을 때와 복잡한 Reward Function을 적용했을 때의 Performance는 크게 차이가 나지는 않았다.

그리고 한 액트에 대해 특화되지 않고 일반화가 잘 되었는지 알아보기 위해 다른 액트(Act)에서도 학습된 Agent를 테스트해 보았다. 우리의 목표 액트를 클리어한 Genome들 중에 많은 경우, 다른 액트를 클리어하지 못하였지만 일부는 다른 쉬운 액트를 클리어하기도 하였다. NEAT를 통한 소닉학습은 일반화도 어느정도는 가능하다는 것이다. 앞으로 추가 연구를 통해, 어떻게 하면 좀더 일반화가 잘되도록 학습을 할 수 있는지 연구해보면 좋을 것이다.