IA Airstriker-Genesis: Estudo de caso utilizando Algoritmo Genético, NEAT e PPO

Gabriel Candelária Wiltgen Barbosa
gabrielcandelaria@alunos.utfpr.edu.br
Universidade Técnológica Federal - Paraná (UTFPR)
Apucarana, Paraná, Brasil

Camila Costa Durante
camiladurante@alunos.utfpr.edu.br
Universidade Técnológica Federal - Paraná (UTFPR)
Apucarana, Paraná, Brasil

***Abstract*—This case study investigate the application of the Genetic Algorithm, NEAT Algorithm (NeuroEvolution of Augmenting Topologies), and the reinforcement learning algorithm PPO (Proximal Policy Optimization) in the game Airstrike made by Electrokinesis. The main objective was to implement each algorithm to train agents in the same game, in order to analyze the efficiency of each and compare them.**

***Index Terms*—Game, Genetic Algorithm, NEAT Algorithm, Proximal Policy Optimization**

I. Introdução

Na atualidade a busca por soluções utilizando algoritmos de ”Inteligência Artificial” tem se intensificado bastante, desde o surgimento de IAs voltadas a acessibilidade, como o ChatGPT, do aumento e melhora nas responsáveis pelos sistemas de recomendações e propagandas e, até mesmo, na melhora das responsáveis pelo controle de personagens, tanto jogáveis quanto não jogáveis, nos jogos digitais da atualidade [4], algo que levou ao objetivo deste projeto, avaliar diferentes abordagens (algoritmos) de agentes em um mesmo ambiente.

O ambiente escolhido para o projeto foi o jogo Airstriker-Genesis (Fig. 1), feito em 2008 pela companhia Electrokinesis. Esse jogo é um shooter inspirado em jogos de 1980 que foi feito para o console Mega Drive (1988, Japão) ou Sega Genesis (1989, América do Norte).

Fig. 1. Capa do jogo Airstriker

Foram implementados três algoritmos para geração de agentes capazes de jogar no ambiente mencionado, um algoritmo puramente genético, um algoritmo genético de redes neurais artificiais em evolução (NEAT) e um algoritmo de aprendizado por reforço (PPO).

Algoritmos puramente genéticos não são foco de estudo para agentes, uma vez que, grande parte dos algoritmos de agentes utilizam do processo genético para aprimorar sua eficácia [1], como o próprio NEAT (*NeuroEvolution of Augmenting Topologies*) [2], o qual utiliza um sistema genético para analisar entradas geradas por observadores que, por meio de pesos sinápticos evolutivos, definem a saída que corresponde a uma determinada ação dentro de cada indivíduo, compondo uma rede neural artificial que evolui através do *feedback* obtido pelo *fitness* final de cada indivíduo.

Ademais, existem os algoritmos focados em aprendizado por reforço ou RL (*Reinforcement Learning*), os quais consistem em mapear estados ou situações para ações visando maximizar uma recompensa de valor numérico [3], algo que é visto nos algoritmos de *Proximal policy optimization* (PPO) [5], *Proximal policy optimization with Generalized Advantage Estimation* (PPO2) e em algoritmos de *Deep Learning with Long Short-Term Memory* (deep LSTM), como presente na IA *alphaStar* [6], utilizada para vencer jogadores profissionais renomados de *StarCraft II*. O PPO, um dos algoritmos implementados para o projeto, utiliza diretamente de interações recentes do agente com o ambiente para realizar sua aprendizagem, sendo seu agente *on-policy*, aprendendo o valor com base na ação atual derivada da política atual [7], desta forma, se destacando com algoritmo no que diz respeito a geração de agentes para jogos.

Contudo, o algoritmo puramente genético se demonstra útil para agentes em cenários mais simples, no qual o jogo ambiente escolhido (Airstriker) se encaixa, sendo assim, também escolhido para análise.

II. Trabalhos Relacionados

Um exemplo de aplicação de RL com redes neurais (deep LSTM) em contexto de jogos, como já citado, é a IA *alphaStar* criada pelo *Team AlphaStar* [6], a qual utiliza uma *deep neural network* para receber entradas da interface do jogo e gerar saídas que correspondem a uma sequência de instruções que

constituem uma ação em jogo, sendo capaz de vencer até mesmo jogadores profissionais no jogo *StarCraft II*.

Outro trabalho utilizado como referência [8] faz o uso do algoritmo NEAT para gerar agentes que controlam os fantasmas (inimigos) no jogo *PacMan*, sendo os agentes capazes de cooperar como um grupo ao tentar eliminar o jogador em caso de existirem mais de um fantasma no nível, além de gerar adaptações de maneira evolutiva.

III. Metodologia de Desenvolvimento

Sessão dedicada à descrição dos procedimentos metodológicos com os quais foi desenvolvido este trabalho. As sub-seções seguintes detalham segmentos específicos que compõe o trabalho na totalidade, contendo informações importantes para seu desenvolvimento.

A. Ambiente

Para a geração do ambiente utilizado ao testar os algoritmos foi necessário o uso da API (*Application Programming Interface*) da biblioteca *gym-retro* da OpenAI [9], responsável por providenciar a comunicação entre jogo e código, juntamente com a biblioteca *Pygame* para emular graficamente o jogo. A *gym-retro* permite a criação de *environments* (ambientes de simulação) utilizando ROMs de jogos suportados pela biblioteca, como o *Airstriker-Genesis* que, por possuir uma versão não comercial, vem nativo ao instalar a biblioteca.

Para o jogo escolhido, existem 12 possíveis entradas correspondentes aos botões do controle (Fig. 2) do console Sega Genesis, sendo eles, (i) B, (ii) A, (iii) MODE, (iv) START, (v) UP, (vi) DOWN, (vii) LEFT, (viii) RIGHT, (ix) C, (x) Y, (xi) X, (xii) Z. Contudo, apenas 3 entradas são de interesse, (i) B, (vii) LEFT e (viii) RIGHT, uma vez que são as únicas que executam ações dentro do jogo emulado.

Fig. 2. Controle do console Sega Genesis

O processo de configuração do ambiente de simulação necessita da execução de métodos provenientes da biblioteca *gym-retro*, sendo eles, (i) *make*, (ii) *reset*, (iii) *step*, (iv) *close* e, para a simulação visual, métodos da biblioteca *Pygame*, sendo eles, (v) *init*, (vi) *display*, (vii) *make_surface*, (viii) *blit*, (ix) *quit*.

(i) *make* - Utiliza uma ROM suportada como parâmetro para criar um environment.
(ii) *reset* - Reinicia o ambiente para o estado de observação inicial.
(iii) *step* - Utilizado para executar ações no ambiente gerado e retornar informações recorrentes da ação passada como parâmetro.
(iv) *close* - Fecha o ambiente gerado a partir do *make*.
(v) *init* - Inicializa uma instância do Pygame.
(vi) *display* - Inicializa uma tela com tamanho passado por parâmetro do método.
(vii) *make surface* - Converte o frame passado como parâmetro para a superfície gráfica da tela gerada no Pygame.
(viii) *blit* - Redimensiona e desenha o frame formatado através do método *make_surface* na janela do Pygame.
(ix) *quit* - Fecha a instância do Pygame.

Fig. 3. Tela inicial do ambiente simulado

B. Algoritmo Genético

A implementação do algoritmo genético foi feita seguindo as bases de um algoritmo genético padrão, com apenas algumas alterações para possível melhora. Dentro do algoritmo existem métodos para geração da população de indivíduos inicial com *seed* (mantendo a randomização padrão) completa ou com um indivíduo carregado adicionado a ela, para a seleção e torneio de indivíduos, para realizar o crossover com um ponto de corte, para realizar a mutação uniforme e para avaliar o *fitness* de cada indivíduo, além de um método *execute* voltado a estruturar as chamadas de métodos, controlar a criação de gerações e retornar o melhor indivíduo da última população gerada.

Os parâmetros de número de gerações (*numberOfGenerations*), tamanho da população (*populationSize*), taxa de mutação (*mutationRate*), taxa de crossover (*cross_rate*), elitismo (*elitism*) e frequência de torneio (*K*), utilizados para a geração de agentes do algoritmo genético estão descritos na Table 1.

Nome do Parâmetro	Valor do Parâmetro
numberOfGenerations	100
populationSize	50
mutationRate	0.025
cross_rate	0.75
elitism	4
K	10

Table 1: Parâmetros utilizados no AG

C. Algoritmo NEAT

O algoritmo NEAT foi implementado utilizando a biblioteca *neat-python* [10], a qual dispõe de um algoritmo NEAT pré-implementado com métodos e arquivo de configuração para utilizá-lo.

Dentro do arquivo *.ini* de configuração, necessário para o funcionamento do algoritmo, existem diversos parâmetros que podem ser configurados conforme a necessidade, contudo, para o agente NEAT deste trabalho foram alterados apenas alguns parâmetros específicos, sendo eles, *fitness_criterion* (critério de *fitness*) alterado para *max*, *fitness_threshold* (*fitness* limite) para 100000, *pop_size* (tamanho da população) para 50, *elitism* (elitismo) para 3, *species_elitism* (elitismo da espécie) para 3, *num_inputs* (número de entradas) para 150, *num_outputs* (número de saídas) para 4 e *num_hidden* (número de neurônios "escondidos") para 2. Os demais parâmetros foram mantidos como padrão, aqueles que causam mudanças significativas nos indivíduos estão definidos na Table 2.

Nome do Parâmetro	Valor do Parâmetro
reset_on_extinction	*False*
bias_mutate_power	0.5
bias_mutate_rate	0.7
bias_replace_rate	0.1
conn_add_prob	0.5
conn_delete_prob	0.3
weight_mutate_power	0.5
weight_mutate_rate	0.8
weight_replace_rate	0.1
feed_forward	*True*
initial_connection	*full_direct*
max_stagnation	15

Table 2: Parâmetros NEAT

A estruturação do algoritmo que faz o uso do NEAT ficou composta dos métodos (i) *fitness*, (ii) *run_neat*, (iii) *execute*. O método *fitness* é o responsável por carregar a função que avalia o indivíduo, passando o valor dessa avaliação (*fitness*) para *genome.fitness*, sendo *genome* a variável referente a estrutura de como foi gerada a rede neural responsável pelo indivíduo avaliado. Encarregado de rodar o algoritmo NEAT do *neat-python*, o método *run neat* conta com a chamada do arquivo de configuração *.ini*, a geração da população inicial, a recepção das estatísticas relacionados ao treinamento e com o processo de definir e salvar o melhor indivíduo da última população gerada pelo treinamento. Por fim, o *execute* carrega a configuração do *neat-python* para criar a rede neural do melhor indivíduo e convertê-la para um array *NumPy* [11] de valores capazes de serem lidos pelos métodos responsáveis pela simulação gráfica de cada agente.

O algoritmo NEAT utiliza redes neurais compostas de neurônios em múltiplas camadas, os quais são conectados pelos pesos sinápticos (*synaptic weights*) responsáveis por direcionar a entrada em um dos neurônios a entrada de outro neurônio presente na camada seguinte, e assim respectivamente até a camada de saída. Como já mencionado, na configuração feita para o *neat-python* foram definidos 150 *inputs* (entradas), que correspondem a 150 neurônios na primeira camada, 2 *hidden neurons* (neurônios escondidos) que correspondem a 2 neurônios na camada "escondida" seguinte e 4 *outputs* (saídas), que correspondem a 4 neurônios na camada final, sendo um exemplo de rede neural presente nos algoritmos NEAT demonstrado na Fig. 4 [1].

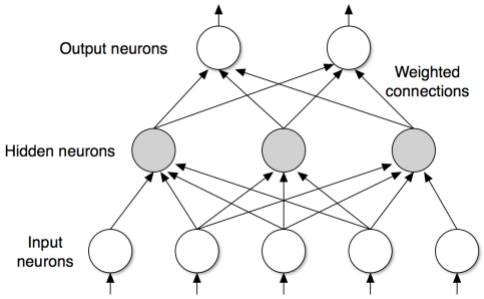

Fig. 4. Arquitetura neural genérica composta de neurônios de entrada (*input neurons*), neurônios escondidos (*hidden neurons*) e neurônios de saída (*output neurons*) conectados por pesos sinápticos.

O código feito utiliza de um redimensionamento do ambiente de simulação para reduzir o número de entradas coletadas do observador pelo algoritmo do *neat-python*, uma vez que, essas entradas aumentam consideravelmente o tempo de execução do treinamento, exigindo uma máquina com poder computacional maior que a utilizada para rodar os testes, sendo assim, foi necessária essa "perda" de potencial.

D. Algoritmo PPO

Para gerar o agente que utiliza *Proximal policy optimization* (PPO) foi necessário fazer o uso da biblioteca *stable_baselines3* [12], a qual possui algoritmos de RL (reinforcement learning) pré-implementados, como no PPO.

O algoritmo PPO treina seu agente mandando ações para o ambiente simulado, o qual, após a execução da ação recebida, retorna os novos valores de estado e recompensa para o agente sendo treinado e, desta forma, ele aprende utilizando desse *feedback*.

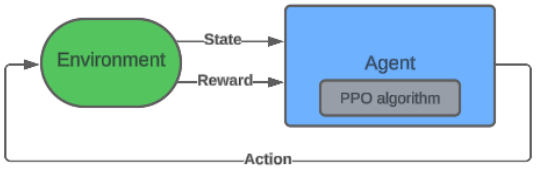

Fig. 5. Diagrama genérico de treinamento de um agente PPO

A implementação do PPO do *stable_baselines3* se deu por meio de um código contendo os métodos (i) *init*, (ii) *train*, (iii) *evaluate*, (iv) *load_model*, (v) *continue_training*, (vi) *get_best_individual*. O método *init* é responsável por inicializar o ambiente de simulação e o algoritmo PPO com a quantidade de *timesteps* desejado, sendo *timesteps* a quantidade de interações que o agente a ser treinado fará

com o ambiente. No método *train* ocorre o treinamento do agente e o processo de salvar o mesmo, o qual pode ser carregado através do método *load_model*, o qual é chamado por *continue_training* que carrega um agente pré-treinado para mais uma seção de treinamento com a quantidade de *timesteps* desejada. *Evaluate* é encarregado de executar um agente já treinado por um determinado número de *episodes*, que correspondem a um conjunto de *timesteps* que compõe um determinado número de ações até uma nova iteração do agente, retornando a recompensa média por *episode* e a quantidade de *timesteps* para a execução de cada *episode*. Por fim, o método *get_best_individual* avalia as ações do último agente gerado (melhor indivíduo), retornando suas ações e a recompensa total do agente.

Os hyperparâmetros e demais configurações relacionadas ao algoritmo PPO não foram alterados, uma vez que o desempenho dos agentes gerados com as configurações padrões se demonstrou o suficiente para a análise.

E. Funções Fitness - AG e NEAT

As funções de *fitness* para o algoritmo genético e para o algoritmo NEAT foram feitas em função da mesma fórmula, a qual, leva em consideração o tempo de vida do agente, o *score* obtido ao eliminar naves inimigas, a ação de passar de nível e a recompensa retornada pelo ambiente de simulação após cada ação do indivíduo.

$$fitness = ((score * 1.2) + point + (reward * 0.4)) * 0.8 + (1/time\_temp) * 0.2$$

(i) *point* - Recompensa por passar de nível
(ii) *reward* - Recompensa por ação
(iii) *time_temp* - Tempo de sobrevivência do agente
(iv) *score* - Recompensa por eliminação de inimigo

Essa equação foi pensada para balancear o *fitness* total entre as possíveis recompensas que tornam o indivíduo mais evoluído, por esse mesmo fator, o peso do *score* foi levemente elevado e o peso de boas ações diminuído, uma vez que os indivíduos estavam priorizando melhores ações e sobrevivência, parando de buscar eliminar inimigos.

F. Função Recompensa - PPO

A função recompensa do algoritmo PPO apenas recolhe a recompensa por ação, provida do ambiente de simulação, após a realização de uma ação do agente, sendo assim, o agente é incentivado apenas ao realizar boas ações.

G. Exigências do Ambiente de Desenvolvimento

Para utilizar o código-fonte é necessário instalar as bibliotecas *gym-retro* versão 0.25.2, *Pygame* versão 2.5.2, *Shimmy* versão 0.2.1, *stable_baselines3* versão 2.3.2, *neat-python* 0.92, *NumPy*, assim como o python instalado em uma de suas versões 3.6, 3.7 ou 3.8, sendo a 3.8.10 recomendada.

Vale ressaltar que os algoritmos NEAT e PPO requerem um poder computacional moderado por serem algoritmos mais robustos, sendo assim, caso esteja rodando em uma máquina de poder computacional baixo a demora para execução do treinamento de agentes pode aumentar consideravelmente. Código-fonte utilizado no projeto disponibilizado via GitHub:

AirstrikerGenesisAI

IV. Resultados e Discussões

Foram realizados um total de 4 treinamentos para agentes do algoritmo genético, 3 treinamentos para o algoritmo PPO e apenas 1 treinamento para o algoritmo NEAT.

- Os treinamentos para o AG foram todos utilizando populações iniciais completas, sem adição de indivíduos pré-treinados, utilizando diferentes *seeds* de criação para variar as populações iniciais, sendo as *seeds* (i) 8, (ii) 27, (iii) 42, (iv) 404.
- Os treinamentos com o PPO foram realizados em partes, foi inicialmente treinado um indivíduo por 100000 *timesteps*, aproximadamente 3 horas de treino, e os agentes seguintes foram gerados por meio de uma continuação de 100000 *timesteps* do agente anterior, resultando em um agente final de 300000 *timesteps*, com treinamento de aproximadamente 9 horas.
- O treinamento do NEAT foi realizado apenas uma única vez, com a configuração já mencionada e para 100 gerações de populações, algo que levou em torno de 17 horas, devido ao poder computacional da máquina utilizada.

A. Agentes Genéticos

Após o treinamento dos agentes genéticos com as diferentes *seeds* foi possível observar que por mais simples que o algoritmo seja em comparação com os outros dois (NEAT e PPO) ainda sim pode ser viável dependendo da complexidade do problema, e, tendo em vista a baixa complexidade do jogo em questão, foi possível obter uma taxa média de *fitness* por geração boa, sendo os agentes gerados capazes de atingir uma boa quantidade de *score* e alcançar o nível 3, embora nenhum tenha conseguido finalizar o jogo. Através do gráfico comparativo das *seeds*(Fig. 6) é possível observar o gradual aumento em desempenho, sendo que o *score* máximo atingido foi de 920 pelo agente da *seed* 404, alcançando apenas o nível 2. Porém, em contrapartida, o agente da *seed* 27 alcançou o nível 3, porém com um *score* de 800.

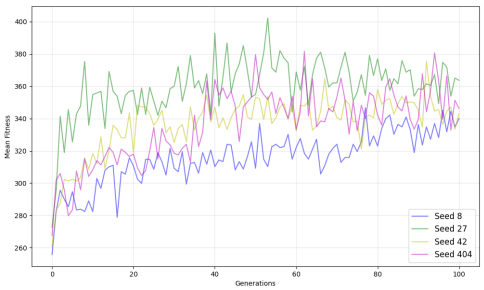

Fig. 6. *Fitness* médio por geração - AG

B. Agente NEAT

Após a geração dos agentes e obtenção do fitness médio por geração dos agentes genético e NEAT foi possível comparar o melhor agente genético com o agente NEAT graficamente (Fig .7).

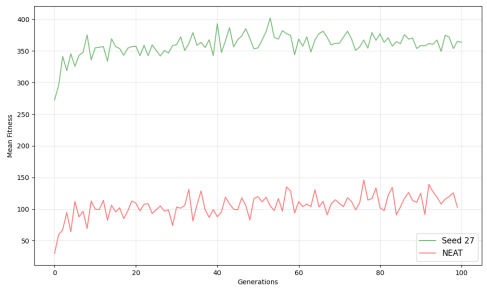

Fig. 7. *Fitness* médio por geração - NEAT x AG

O agente NEAT tomou um rumo bem diferente dos agentes dos outros algoritmos, focando apenas na sobrevivência, mesmo com o ajuste feito na função de *fitness*. Durante a visualização do melhor agente gerado é notável seu comportamento, sendo que ele inicia o jogo correndo para a direita do mapa e lá permanece até sua eliminação, atirando e matando apenas quando existem inimigos nascendo em sua frente, abordagem que afeta drasticamente seu *fitness* médio e *score* final, que dentre os agentes é o mais baixo.

C. Agente PPO

Para os agentes gerado pelo PPO foi feita uma tabela comparativa (Table 3), desta forma é possível avaliar a evolução no desempenho do agente a cada seção de treinamento feita.

Agente em Horas	Pontuação Média Por Episódio	Pontuação Máxima	Nivel Alcançado
3 horas	178	240	1
6 horas	462	700	1
9 horas	772	1220	2

Table 3: Resultados dos agentes PPO

O desempenho dos agentes PPO foi gratificante em questão de *score*, uma vez que, os agentes gerados pareciam estar mais focados em obter pontos, deixando sua sobrevivência a desejar. Algo que refletiu nos resultados de nível atingido pelos agentes, que não passaram do nível 2, assim como o agente NEAT.

Contudo, é notável a diferença na movimentação do agente treinado pelo PPO, sendo ela mais precisa que a dos demais agentes, embora não seja tão eficiente ao desviar de projéteis e asteroides é capaz de eliminar o maior número de inimigos, proporcionalmente, dentre os agentes levando em consideração o pouco progresso de nível que faz, obtendo uma média de X *score* apenas atingindo o segundo nível.

D. Comparações finais

Agente	Pontuação	Nivel Alcançado
Jogador	1660	3
Genético	800	3
NEAT	260	2
PPO	1220	2

Table 4: Relação de resultados

Observando a Table 4 é possível notar a diferença entre os algoritmos, contudo é possível dizer que todos realizaram um bom progresso conforme seu treinamento, considerando o foco tomado por cada agente.

V. Conclusão

Este projeto tinha como objetivo principal realizar uma análise em função dos agentes gerados pelos algoritmos Genético, NEAT e PPO em um mesmo ambiente de simulação, algo que foi realizado com sucesso, em vista que, foi possível comparar todos os agentes desejados e suas eficiências específicas conforme apresentado. Com os testes realizados é possível determinar que, embora existam algoritmos com desempenho melhor, é necessário tempo de treinamento para atingir tal patamar. O algoritmo genético ter se destacado entre o NEAT e o PPO demonstra exatamente isso, uma vez que, o algoritmo genético é mais rápido em questão de soluções a curto prazo, algo que proporcionou ao seu agente um *score* relativamente alto, atingindo o nível 3. Contudo, se os treinamentos dos demais agentes continuassem, algo que demanda bem mais tempo, eles teriam superado o algoritmo genético com facilidade.

Sendo assim, é possível dizer que a curva de aprendizagem dos algoritmos NEAT e PPO começa baixa, contudo ela se propulsiona conforme o nível de aprendizado aumenta, pelo fato de possuírem pesos evolutivos, sendo capazes de realizar adaptações conforme as iterações, algo que não ocorre quando se trata de um algoritmo puramente genético.

Para uma análise mais detalhada demonstrando resultados mais precisos em relação ao agente NEAT e PPO é necessário um maior tempo de treinamento e poder computacional, considerando a demora na execução dos algoritmos.

References

[1] Kearney, William T., "Using Genetic Algorithms to Evolve Artificial Neural Networks" (2016). Honors Theses. Paper 818.

[2] Kenneth O. Stanley, Risto Miikkulainen., "Evolving Neural Networks through Augmenting Topologies", 2002 by the Massachusetts Institute of Technology.

[3] S. Marsland, Machine Learning - An Algorithmic Perspective., ser. Chapman and Hall / CRC machine learning and pattern recognition series. CRC Press, 2009.

[4] Paravantis, Thanos., "Applications of NEAT algorithm for automatic gameplay of agents in deterministic and non-deterministic game environments".

[5] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms", 2017.

[6] Vinyals, O., Babuschkin, I., Chung, J., Mathieu, M., Jaderberg, M., Czarnecki, W., Dudzik, A., Huang, A., Georgiev, P., Powell, R., Ewalds, T., Horgan, D., Kroiss, M., Danihelka, I., Agapiou, J., Oh, J., Dalibard, V., Choi, D., Sifre, L., Sulsky, Y., Vezhnevets, S., Molloy, J., Cai, T., Budden, D., Paine, T., Gulcehre, C., Wang, Z., Pfaff, T., Pohlen, T., Wu, Y., Yogatama, D., Cohen, J., McKinney, K., Smith, O., Schaul, T., Lillicrap, T., Apps, C., Kavukcuoglu, K., Hassabis, D., and Silver, D., "AlphaStar: Mastering the real-time strategy game StarCraft II", 2019.

[7] Equipe DSA., "Deep Learning Book", 2022.

[8] Wittkamp, Mark, Barone, Luigi and Hingston, Philip. (2009). Using NEAT for Continuous Adaptation and Teamwork Formation in Pacman. ECU Publications. 234 - 242. 10.1109/CIG.2008.5035645.

[9] Nichol, Alex and Pfau, Vicki and Hesse, Christopher and Klimov, Oleg and Schulman, John., "Gotta Learn Fast: A New Benchmark for Generalization in RL" (2018). arXiv preprint arXiv:1804.03720.

[10] McIntyre, A., Kallada, M., Miguel, C. G., Feher de Silva, C., and Netto, M. L. "neat-python" [Computer software].

[11] Harris, C.R., Millman, K.J., van der Walt, S.J. et al. "Array programming with NumPy". Nature 585, 357–362 (2020). DOI: 10.1038/s41586-020-2649-2.

[12] Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., and Dormann, N., "Stable-Baselines3: Reliable Reinforcement Learning Implementations" (2021). Journal of Machine Learning Research 268.