# *Playing Atari with Deep Reinforcement Learning*
# Projeto
# Inspeção do comportamento do agente baseado no número de parâmetros da rede

Gabriel Buginga

PESC

# Sumário

1 Introdução
2 Contexto
3 Trabalhos Relacionados
4 *Deep Reinforcement Learning*
5 Experimentos e Método
6 Conclusão
Referência

Gabriel Buginga
PESC

# 1 Introdução

- Lembrem a apresentação sobre o paper "*Playing Atari with Deep Reinforcement Learning*" (de qualquer maneira, será feito uma breve recordação utilizando os recursos anteriores).
- O presente projeto refaz aquele mesmo algoritmo com o jogo Pong. Reproduzindo os resultados básicos para esse jogo.

- Porém agora os **objetivos** não são apenas atingir o desempenho de maneira satisfatória – atingir as recompensas que "ganham" o jogo. E sim testar diferentes vieses indutivos de expressividade, ou seja, no nosso caso o **número de parâmetros** da rede ou a **quantidade relativa de camadas**.
  - Pergunta: se eu diminuir a expressividade da rede o que acontece com meu agente?
  - Fica num planalto de desempenho e não cresce mais?
  - Pouca mudança até um comportamento catastrófico em função dos parâmetros?
  - Não é claro o que acontecerá com o agente.

# 1 Introdução

- Aprendizado por reforço é um subconjunto de aprendizado de máquina que modela um agente e um ambiente onde os dois interagem. O agente precisa ser controlado de tal forma que as suas recompensas sejam as maiores possíveis nessa tarefa.
- Até o artigo de 2013, se quiséssemos aplicar RL para inputs sensoriais de alta dimensão (imagens) usaríamos atributos feitos a mão e funções de representação lineares.
- O aprendizado profundo é exatamente esse conjunto de métodos montados para extrair atributos dessas fontes de alta dimensão. Lembrem das redes convolucionais.
- Porém não se pode aplicar o aprendizado profundo diretamente para o caso RL, dado algumas razões:
  - Deve aprender baseado num sinal de recompensa escalar o qual é frequentemente esparso, ruidoso e atrasado.
  - Atraso entre ações e recompensas.
  - Os pontos dos dados não são identicamente distruibuídos e a distribuição muda de acordo com o aprendizado de novos comportamentos.

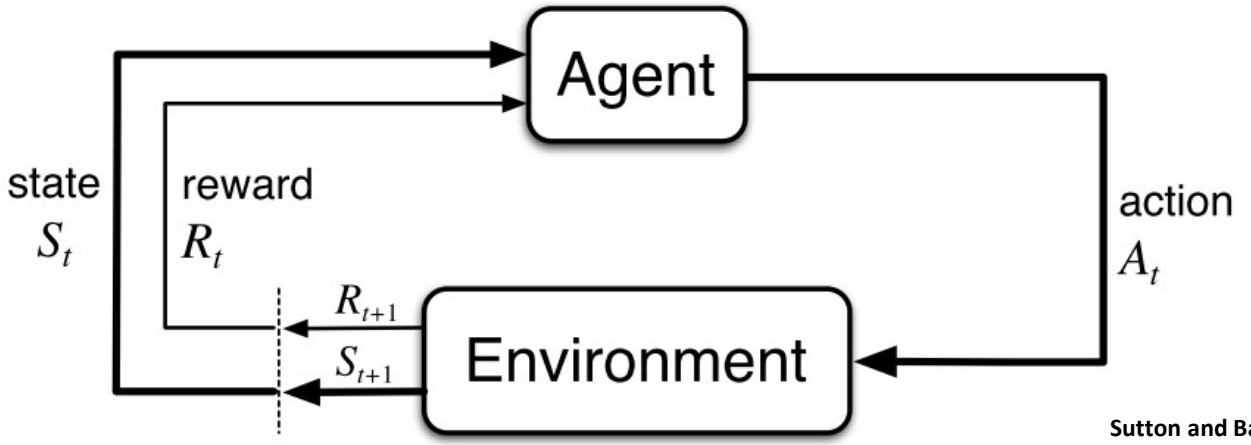
# 1 Introdução

- O artigo mostra que as CNNs podem aprender a controlar agentes baseado apenas no input de vídeo em ambientes complexos. A rede é treinada utilizando uma variação do *Q-Learning* com a ajuda do gradiente descendente estocástico, e um mecanismo chamado *replay buffer* – maiores descrições avante.
- Os Ambientes são sete jogos Atari 2600: input são vídeos 210x160 RGB com 60Hz.
- O objetivo é obter um agente CNN sem informação específica de cada jogo ou atributos específicos que aprenda a jogar apenas com as **imagens**, **recompensas**, **sinais de término** e **conjunto de possíveis ações**. De fato foi obtido: melhorou o desempenho de todos os algoritmos RL passados além de ultrapassar especialistas humanos em 3 dos jogos.


Figure 1: Screen shots from five Atari 2600 Games: (*Left-to-right*) Pong, Breakout, Space Invaders, Seaquest, Beam Rider

# 2 Contexto

- O agente interage com um ambiente $\mathcal{E}$.
- Em cada passo no tempo o agente seleciona uma ação $a_t$ de um conjunto $\mathcal{A} = \{1, \ldots, K\}$ e recebe uma imagem $x_t \in \mathbb{R}^d$ e uma recompensa escalar $r_t$.
- O estado na verdade é $s_t = x_1, a_1, x_2, \ldots, a_{t-1}, x_t$, pois por causa do *perceptual aliasing* é impossível entender a situação observando apenas $x_t$.
- Portanto tem-se uma Markov Decision Process (MDP) tradicional.




Sutton and Barto (2018).

**Figure 3.1:** The agent–environment interaction in a Markov decision process.

# 2 Contexto

- O objetivo é maximizar as soma das recompensas submetidas à um fator de desconto $\gamma$. Sendo T o tempo de término, define-se o retorno descontado como:

$$R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$$

- E a *optimal action-value function* como sendo o valor máximo esperado do retorno atingido por seguir a *policy* ótima (modo de escolher ações) depois que foi observado um estado s e uma ação foi tomada:

$$Q^*(s,a) = \max_\pi \mathbb{E}\left[R_t | s_t = s, a_t = a, \pi\right]$$

Q ótima respeita as
**equações de Bellman**

*Policy é uma função:* $S \rightarrow A$

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s',a') \middle| s, a\right]$$

# 2 Contexto

- A ideia básica é utilizar a equação de Bellman como uma atualização iterativa (caso tabular, essa família de algoritmos chama-se *value iteration* e converge para a função Q ótima). Ou seja:

$$Q_{i+1}(s,a) \quad = \mathbb{E}\left[r + \gamma \max_{a'} Q_i(s',a') | s,a\right]$$

- Como o espaço de estados é enorme a função Q será modelada como uma rede neural Q.
- Portanto em cada iteração resulta um treinamento utilizando uma *loss function* de regressão:

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[\left(y_i - Q(s,a;\theta_i)\right)^2\right]; \; y_i \; = \; \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) | s,a\right]$$

- Note que os parâmetros para $i-1$ são mantidos fixos na otimização.
- O que se faz quando se quer otimar uma *loss function*?

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right]$$

- Porém, utiliza-se o gradiente descendente estocástico – não se precisa calcular esse valor esperado. O *Q-learning* tradicional faz as atualizações dos pesos a cada passo no tempo (o que não se fará no artigo, ver-se-á o porquê).
- É *off-policy* e *model-free*.

# 3 Trabalhos relacionados

- TD-gammon: aprendizado por reforço *model-free* utilizando um algoritmo similar ao *Q-learning*, onde a função de estimativa foi modelo com um MLP com apenas uma camada escondida. Feito em 1995.
- Porém essa mesma abordagem não obteve o mesmo sucesso com xadrez, Go ou damas.
- Além disso, há resultados teóricos que mostram que algoritmos *model-free* e *off-policy* podem causar divergência na rede.
- Redes profundas já foram usadas para estimar o ambiente, Restricted Boltzmann Machines para a *value function* ou a *policy*. E o problema da divergência foi parcialmente tratado via métodos de *gradient temporal-difference.*
- Os trabalhos mais parecidos com o do presente projeto:
  - Utilizam o *neural fitted Q-learning (NFQ)*, otimizando aquela sequência de *loss functions* utilizando o RPROP para atualizar os parâmetros. Porém utiliza uma atualização que escala com o tamanho do dataset, e implementação em tarefas reais precisou de um autoencoder profundo para aprender uma representação com dimensões menores.
- O presente trabalho aplica aprendizado por reforço fim-a-fim.

# 4 *Deep Reinforcement Learning*

- O objetivo é, inspirados pelo trabalho do TD-gammon, utilizar um rede neural profunda para aprender a jogar nos ambientes do Atari 2600. Produzindo um novo algoritmo chamado: *deep Q-learning*.
- Utiliza-se uma técnica chamada *experience replay* onde se guarda as experiências do agente em cada passo de tempo, $e_t = (s_t, a_t, r_t, s_{t+1})$ dentro de um dataset $\mathcal{D} = e_1, ..., e_N$ agrupados sobre vários episódios dentro de uma *replay memory.*
- As atualizações da *Q-function* são realizadas em amostras da *replay memory:* $e \sim \mathcal{D}$.
- Além disso cada estado é passado por uma função $\phi$ que realiza as seguintes transformações:
  - As imagens são tranformadas de 210x160 RGB para escala cinza sub-amostradas até 110x84; mais um corte para uma região quadrada de 84x84.
  - Aplica esse processamento nos últimos **4 frames e os empilha** e passa para a *Q*.

**Algorithm 1** Deep Q-learning with Experience Replay

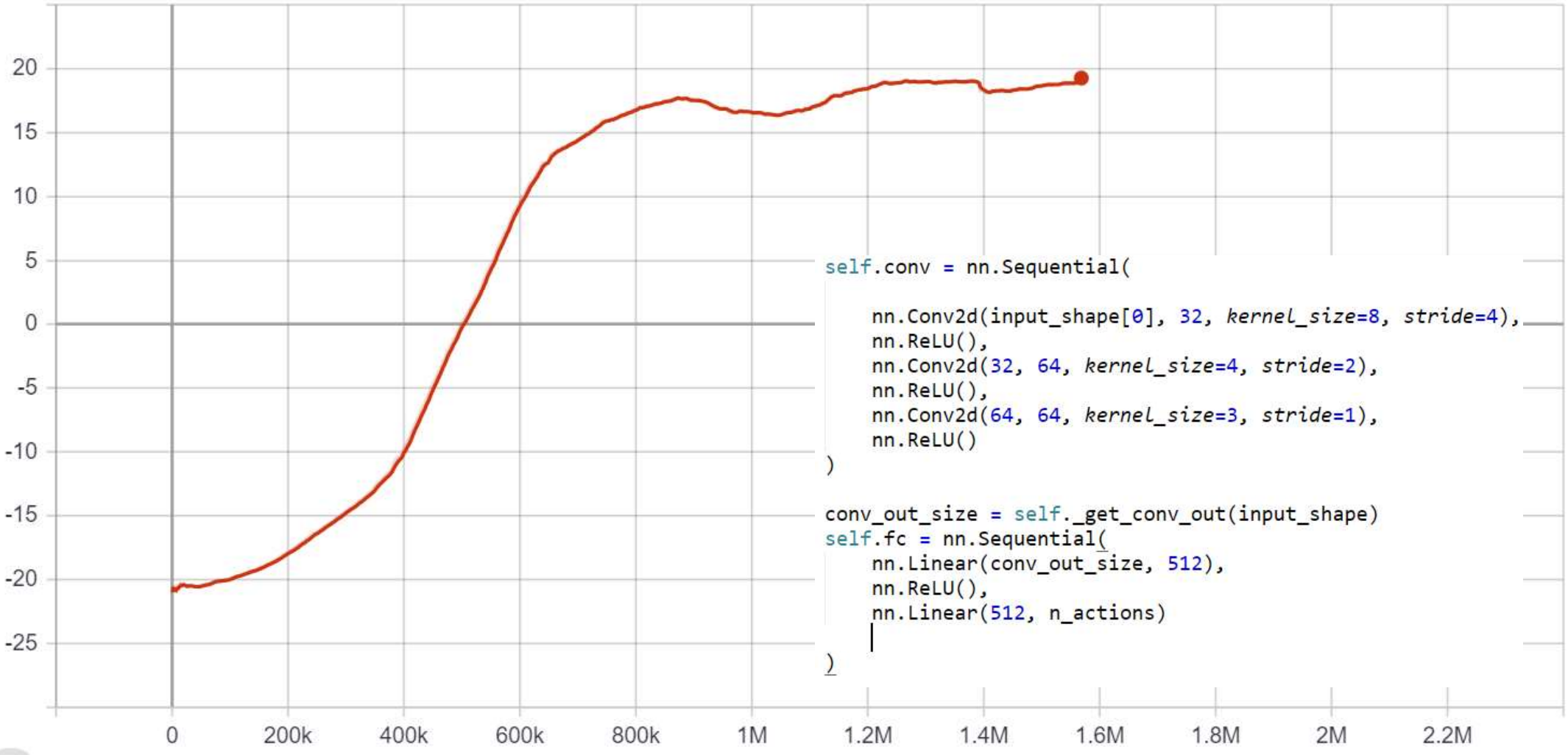
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
**for** episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  **for** $t = 1, T$ **do**
    With probability $\epsilon$ select a random 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**

# 5 Experimentos e Método

- A única diferença do código no presente projeto é o uso de duas redes neurais, o chamado Double Q-learning, para diminuir o problema do regressão com o alvo se movendo constantemente.
- A métrica continua sendo o A*verage total reward*: recompensa total recolhida pelo agente em um episódio, tirando a média de 100 tentativas (métrica exatamente igual à usada pelo artigo original). Além do formato das curvas da recompensa média pela número de amostras recolhidas ser passível de interpretação.
- Será dividido em 5 experimentos, onde cada aplicação do algoritmo é realizada com uma rede progressivamente mais fraca. Sendo que o *baseline* é uma rede

| Name | Smoothed | Value | Step | Time | Relative |
|---|---|---|---|---|---|
| Arquitetura Completa | 19.27 | 19.6 | 1.568M | Mon Dec 9, 07:57:06 | 14h 36m 52s |

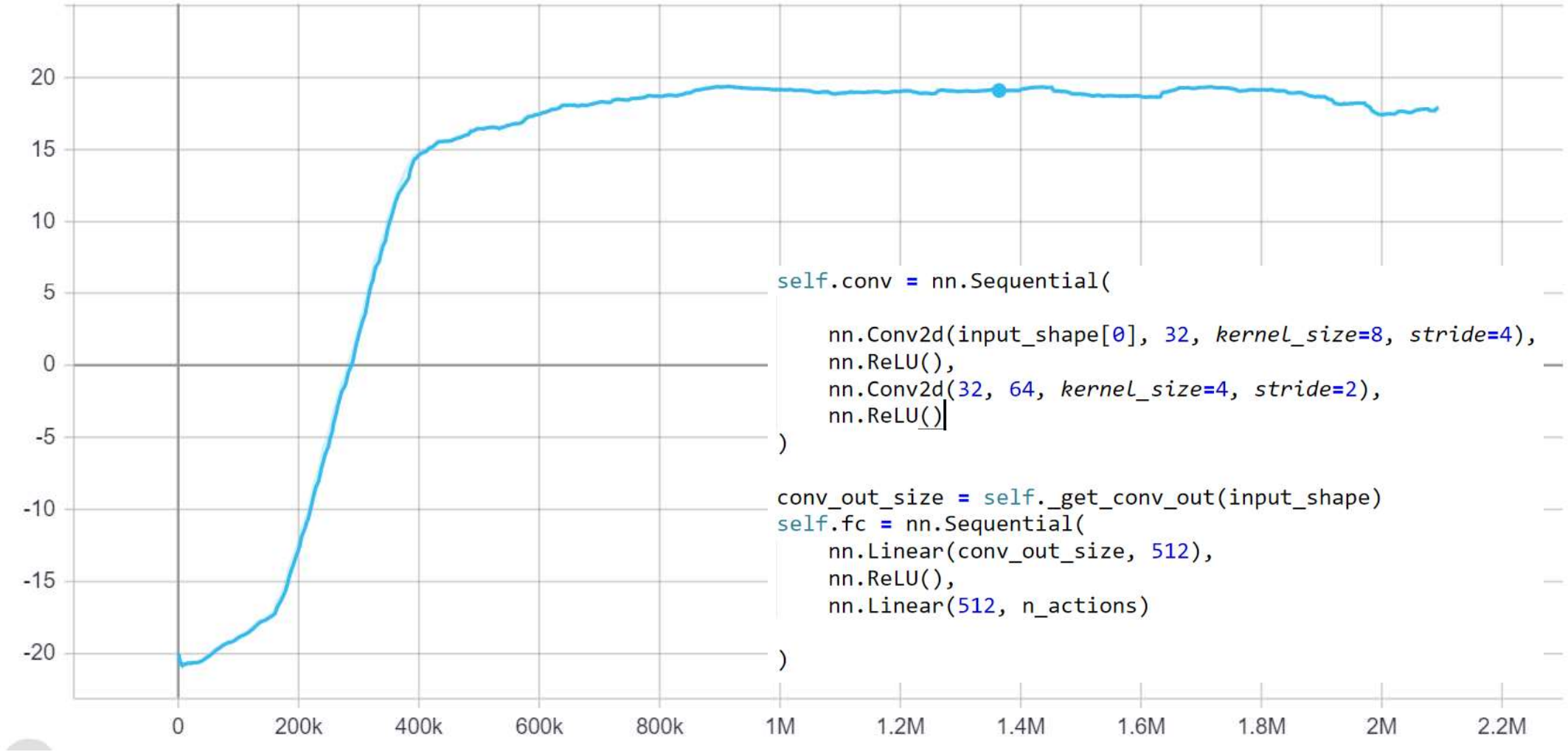
reward_100




```
self.conv = nn.Sequential(

    nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
    nn.ReLU(),
    nn.Conv2d(32, 64, kernel_size=4, stride=2),
    nn.ReLU(),
    nn.Conv2d(64, 64, kernel_size=3, stride=1),
    nn.ReLU()
)

conv_out_size = self._get_conv_out(input_shape)
self.fc = nn.Sequential(
    nn.Linear(conv_out_size, 512),
    nn.ReLU(),
    nn.Linear(512, n_actions)

)
```

| Name | Smoothed | Value | Step | Time | Relative |
|---|---|---|---|---|---|
| Primeiro Experimento | 19.11 | 19.09 | 1.364M | Sat Dec 14, 12:54:36 | 10h 51m 18s |

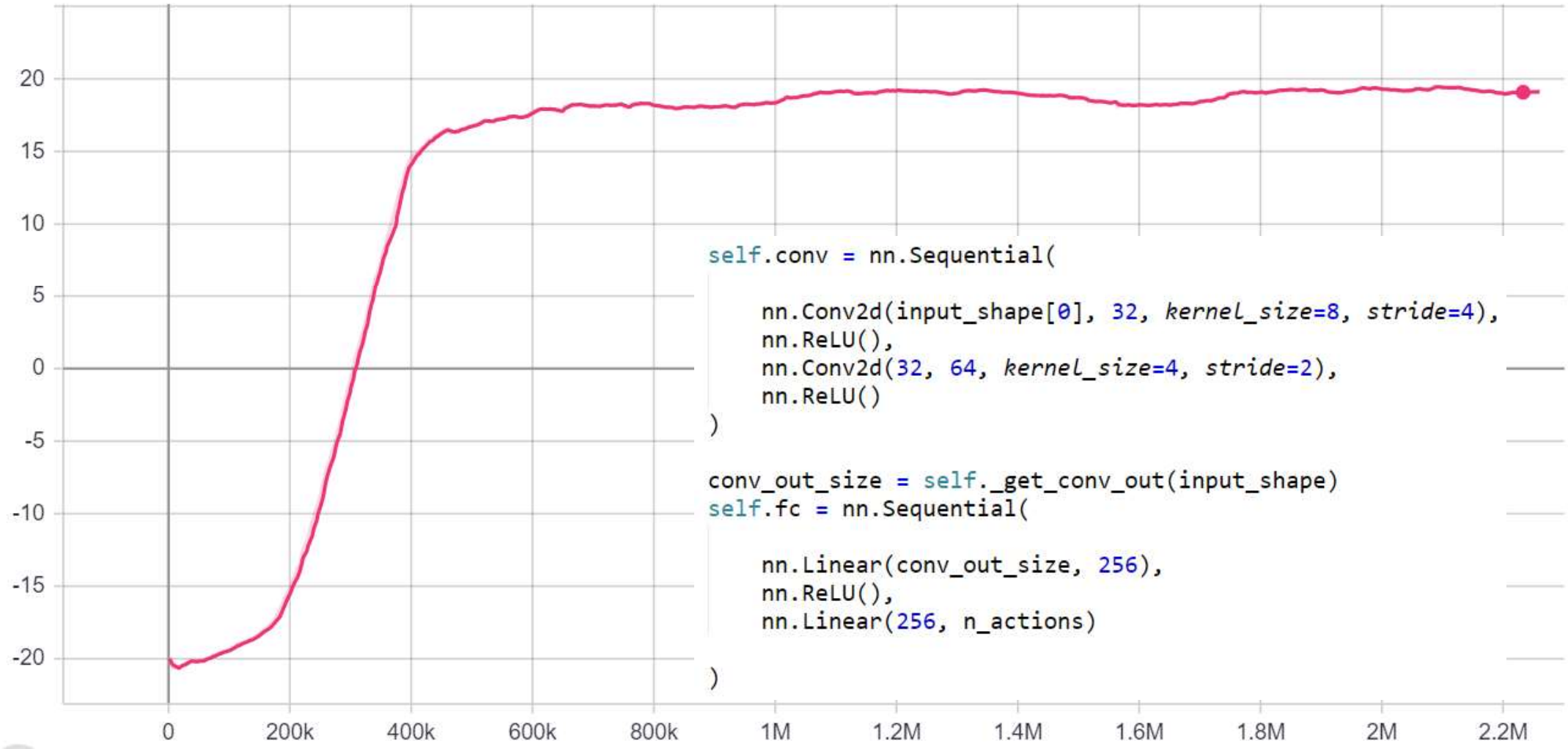
reward_100




```
self.conv = nn.Sequential(

    nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
    nn.ReLU(),
    nn.Conv2d(32, 64, kernel_size=4, stride=2),
    nn.ReLU()
)

conv_out_size = self._get_conv_out(input_shape)
self.fc = nn.Sequential(
    nn.Linear(conv_out_size, 512),
    nn.ReLU(),
    nn.Linear(512, n_actions)

)
```

| Name | Smoothed | Value | Step | Time | Relative |
|---|---|---|---|---|---|
| Segundo Experimento | 19.07 | 19.08 | 2.233M | Sun Dec 15, 12:50:22 | 17h 20m 59s |

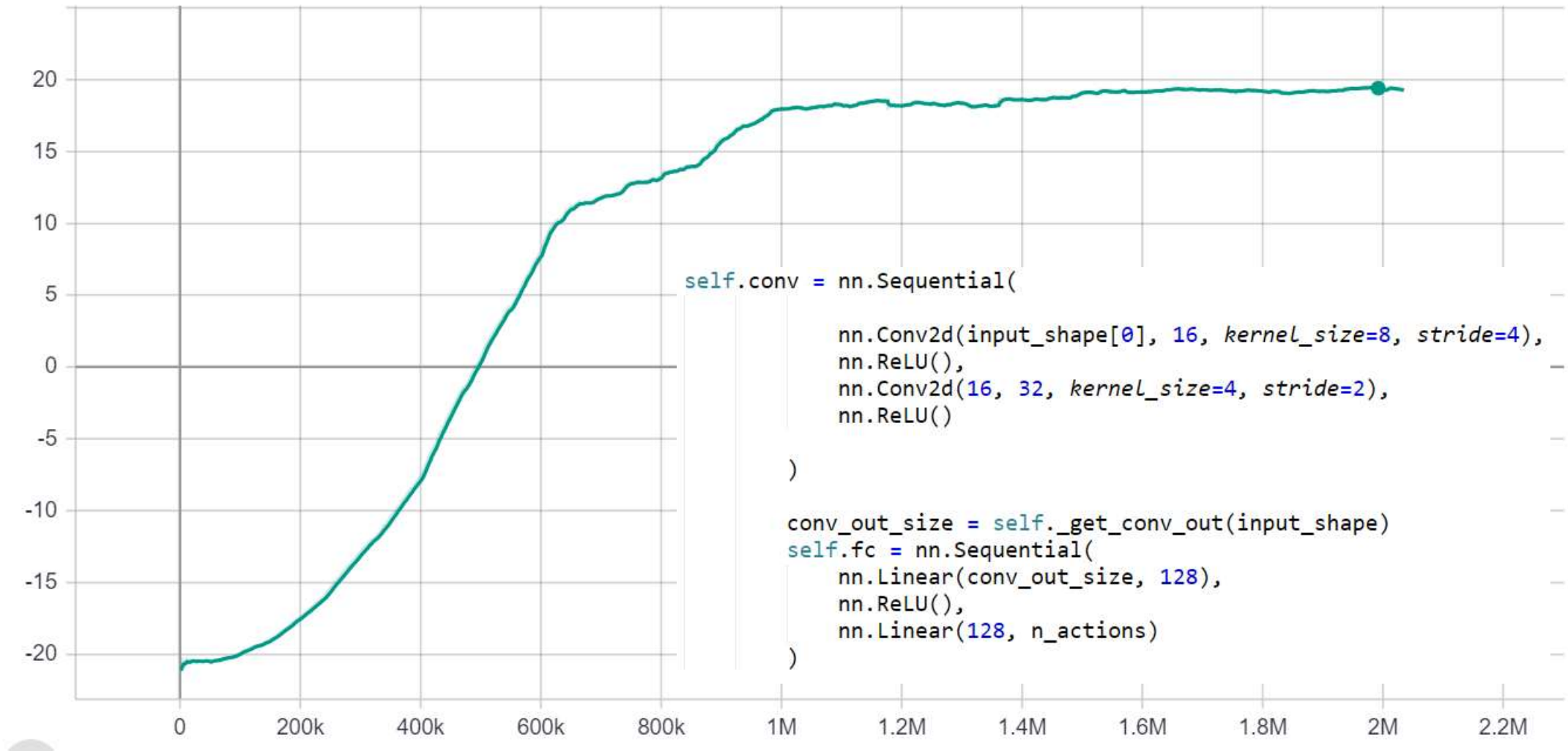
reward_100




```
self.conv = nn.Sequential(

    nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
    nn.ReLU(),
    nn.Conv2d(32, 64, kernel_size=4, stride=2),
    nn.ReLU()
)

conv_out_size = self._get_conv_out(input_shape)
self.fc = nn.Sequential(

    nn.Linear(conv_out_size, 256),
    nn.ReLU(),
    nn.Linear(256, n_actions)

)
```

| Name | Smoothed | Value | Step | Time | Relative |
|---|---|---|---|---|---|
| Terceiro Experimento | 19.41 | 19.37 | 1.992M | Mon Dec 16, 06:21:41 | 17h 10m 22s |

reward_100

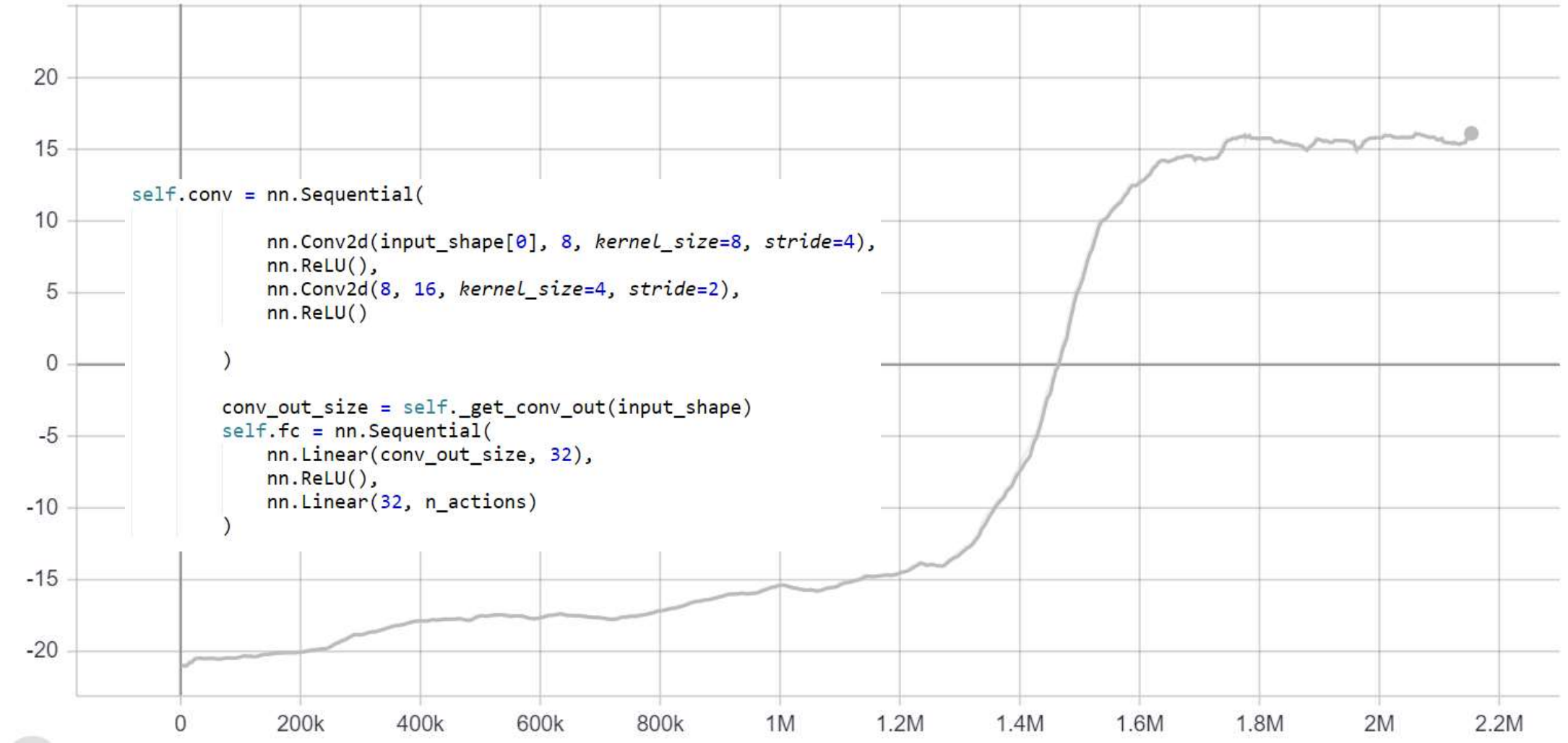



```
self.conv = nn.Sequential(

            nn.Conv2d(input_shape[0], 16, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(16, 32, kernel_size=4, stride=2),
            nn.ReLU()

        )

        conv_out_size = self._get_conv_out(input_shape)
        self.fc = nn.Sequential(
            nn.Linear(conv_out_size, 128),
            nn.ReLU(),
            nn.Linear(128, n_actions)
        )
```

| Name | Smoothed | Value | Step | Time | Relative |
|---|---|---|---|---|---|
| Quarto Experimento | 16.11 | 16.17 | 2.154M | Mon Dec 16, 22:45:02 | 16h 1m 37s |

reward_100

1

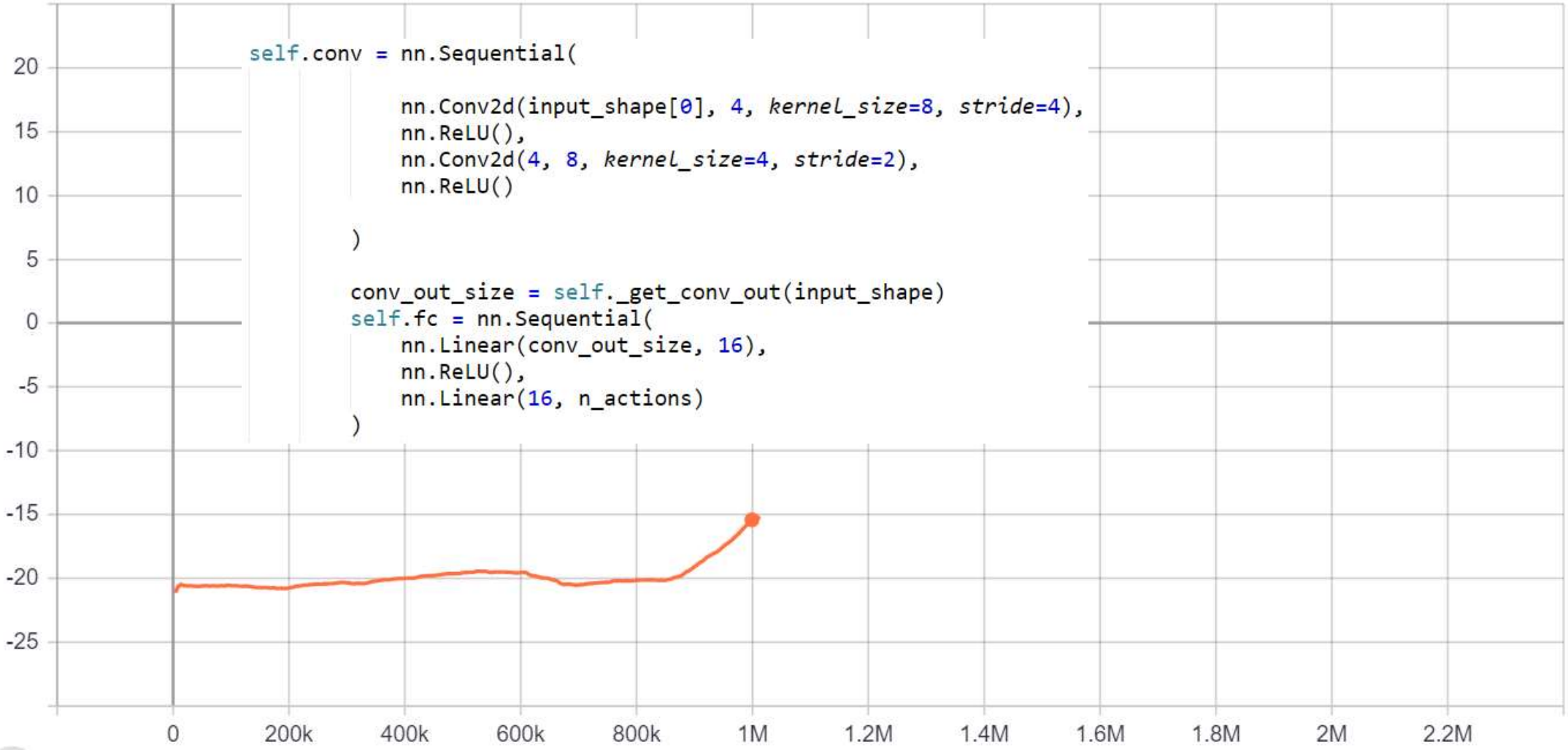



```
self.conv = nn.Sequential(

            nn.Conv2d(input_shape[0], 8, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(8, 16, kernel_size=4, stride=2),
            nn.ReLU()

        )

        conv_out_size = self._get_conv_out(input_shape)
        self.fc = nn.Sequential(
            nn.Linear(conv_out_size, 32),
            nn.ReLU(),
            nn.Linear(32, n_actions)
        )
```

| Name | Smoothed | Value | Step | Time | Relative |
|---|---|---|---|---|---|
| Quinto Experimento | -15.42 | -15.32 | 999.1k | Tue Dec 17, 06:02:21 | 7h 1m 46s |

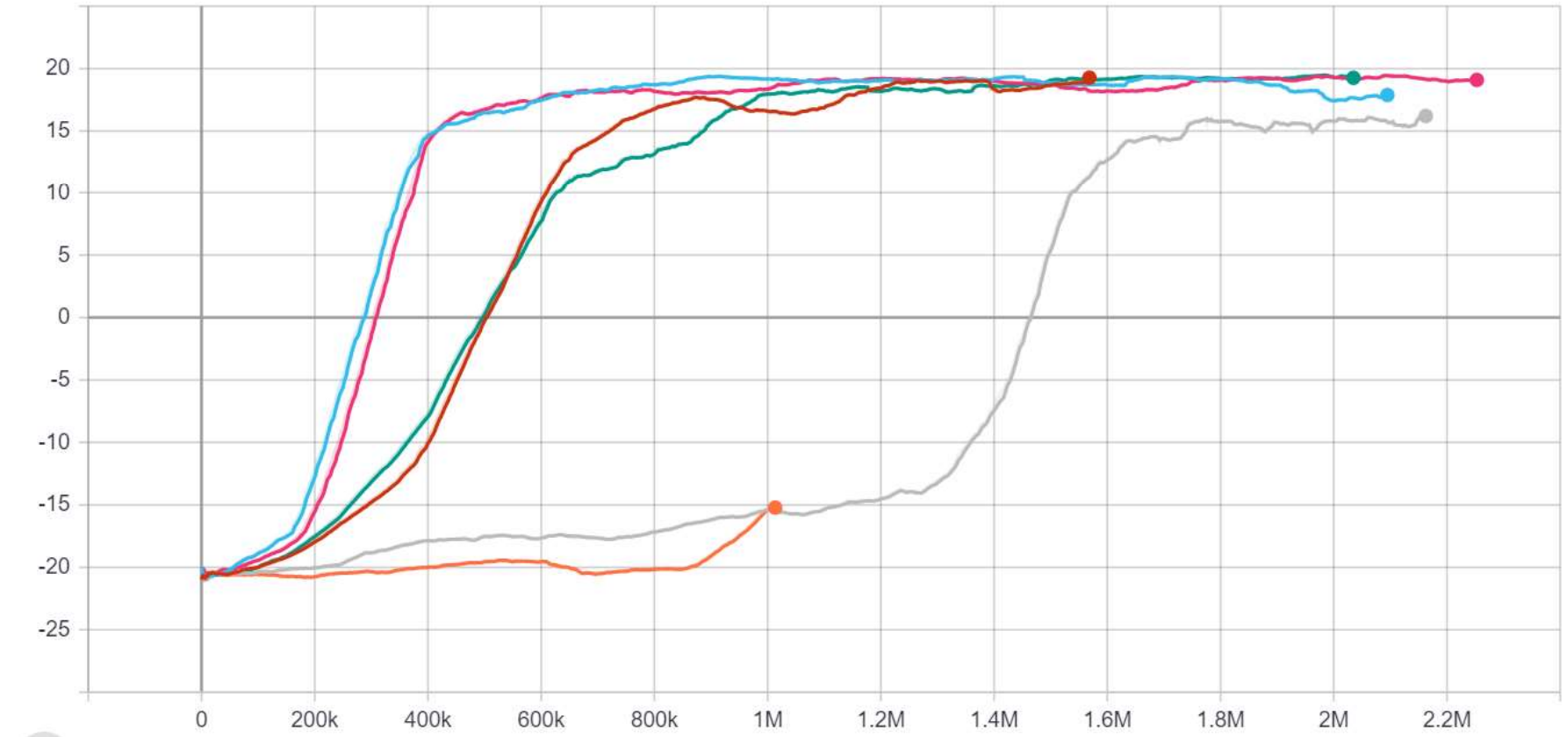
reward_100

```
self.conv = nn.Sequential(

            nn.Conv2d(input_shape[0], 4, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(4, 8, kernel_size=4, stride=2),
            nn.ReLU()

        )

        conv_out_size = self._get_conv_out(input_shape)
        self.fc = nn.Sequential(
            nn.Linear(conv_out_size, 16),
            nn.ReLU(),
            nn.Linear(16, n_actions)
        )
```



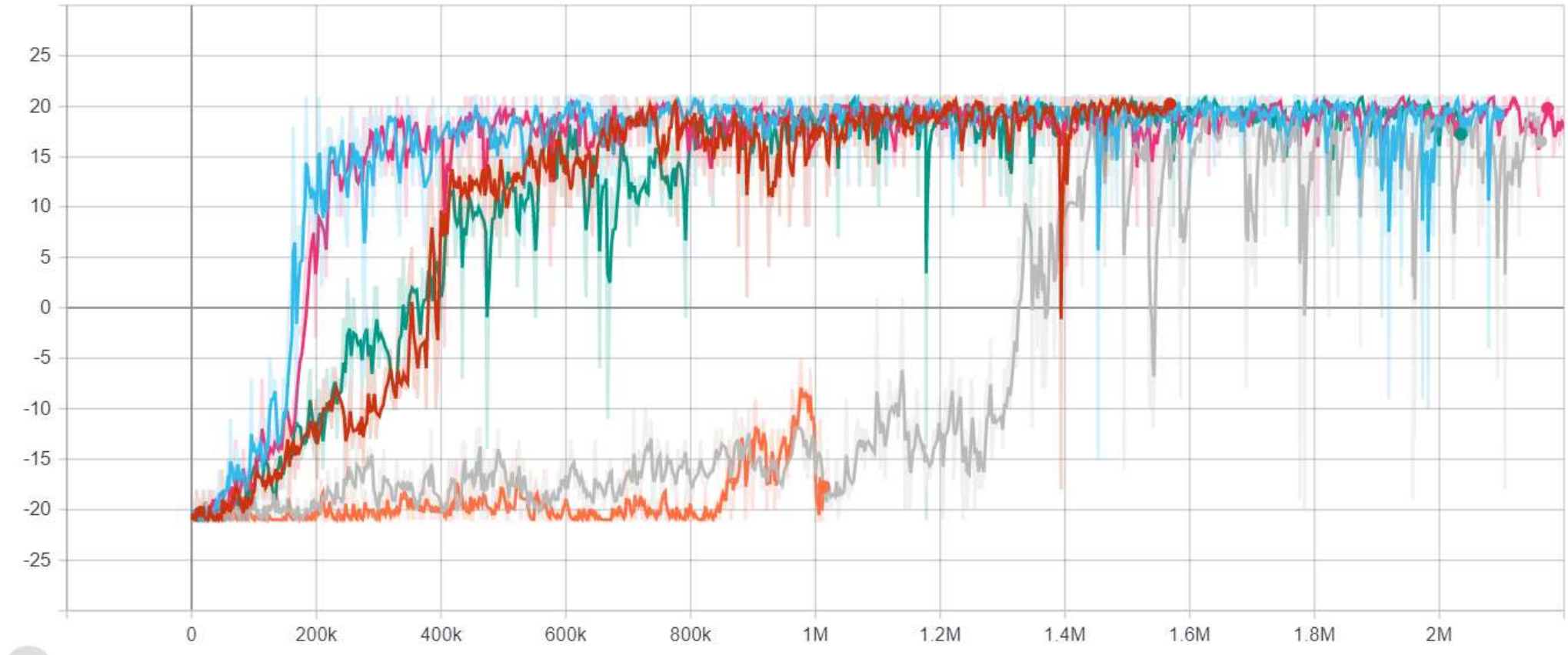
| Name | Smoothed | Value | Step | Time | Relative |
|---|---|---|---|---|---|
| Arquitetura Completa | 19.27 | 19.6 | 1.568M | Mon Dec 9, 07:57:06 | 14h 36m 52s |
| Primeiro Experimento | 17.89 | 17.94 | 2.095M | Sat Dec 14, 19:24:42 | 17h 21m 24s |
| Quarto Experimento | 16.2 | 16.2 | 2.163M | Mon Dec 16, 22:48:55 | 16h 5m 30s |
| Quinto Experimento | -15.22 | -15.19 | 1.013M | Tue Dec 17, 06:09:15 | 7h 8m 41s |
| Segundo Experimento | 19.09 | 19.09 | 2.253M | Sun Dec 15, 12:59:09 | 17h 29m 46s |
| Terceiro Experimento | 19.26 | 19.22 | 2.035M | Mon Dec 16, 06:40:13 | 17h 28m 53s |

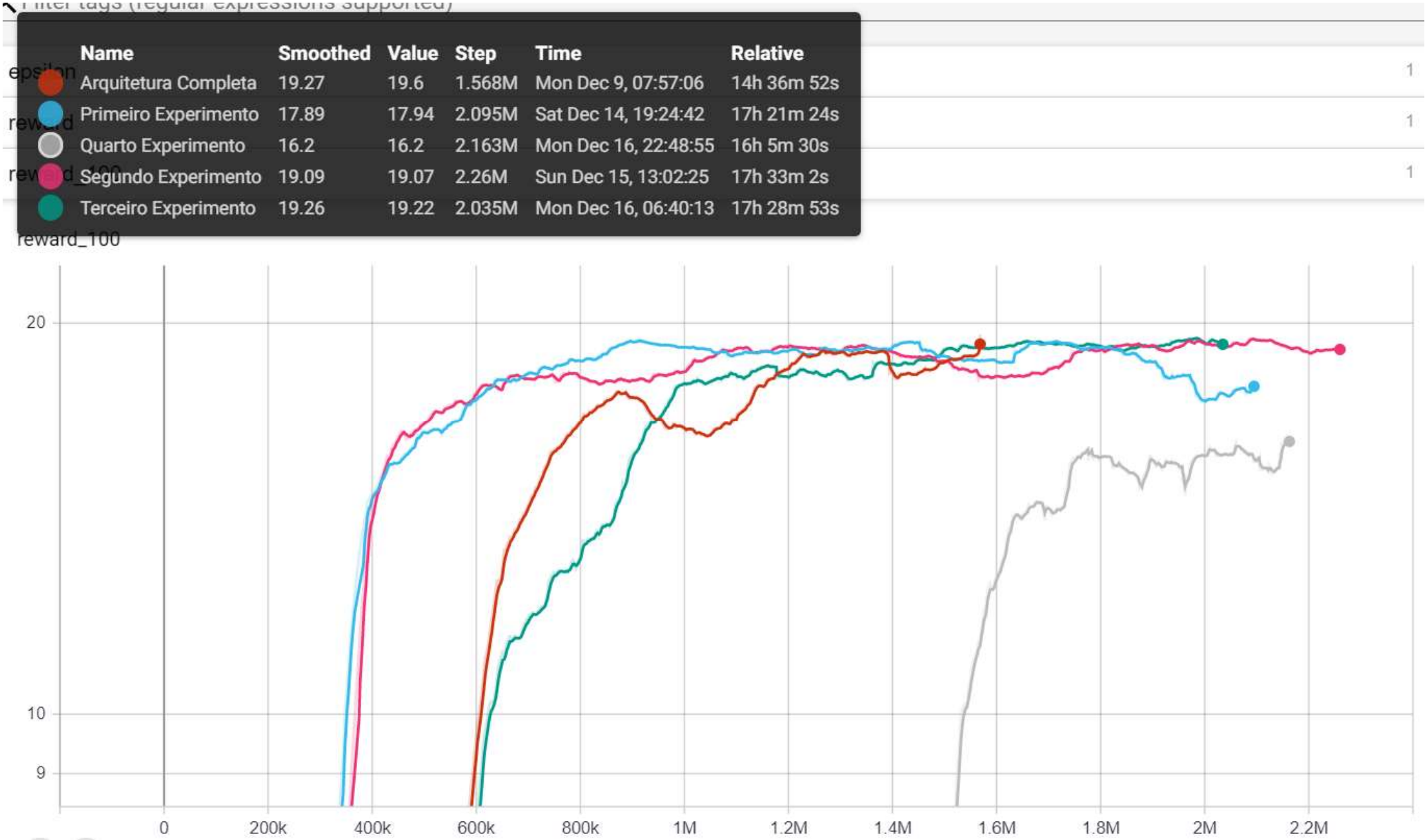
reward_100



| Name | Smoothed | Value | Step | Time | Relative |
|---|---|---|---|---|---|
| Arquitetura Completa | 20.23 | 21 | 1.568M | Mon Dec 9, 07:57:06 | 14h 36m 52s |
| Primeiro Experimento | 19.23 | 18 | 2.095M | Sat Dec 14, 19:24:42 | 17h 21m 24s |
| Quarto Experimento | 16.56 | 17 | 2.163M | Mon Dec 16, 22:48:55 | 16h 5m 30s |
| Quinto Experimento | -17.73 | -16 | 1.013M | Tue Dec 17, 06:09:15 | 7h 8m 41s |
| Segundo Experimento | 19.82 | 21 | 2.174M | Sun Dec 15, 12:24:20 | 16h 54m 57s |
| Terceiro Experimento | 17.27 | 16 | 2.035M | Mon Dec 16, 06:40:13 | 17h 28m 53s |

reward




Filter tags (regular expressions supported)
epsilon
reward
reward_100
1
1
1
Name Smoothed Value Step Time Relative
Arquitetura Completa 19.27 19.6 1.568M Mon Dec 9, 07:57:06 14h 36m 52s
Primeiro Experimento 17.89 17.94 2.095M Sat Dec 14, 19:24:42 17h 21m 24s
Quarto Experimento 16.2 16.2 2.163M Mon Dec 16, 22:48:55 16h 5m 30s
Segundo Experimento 19.09 19.07 2.26M Sun Dec 15, 13:02:25 17h 33m 2s
Terceiro Experimento 19.26 19.22 2.035M Mon Dec 16, 06:40:13 17h 28m 53s
reward_100
20
10
9
0
200k
400k
600k
800k
1M
1.2M
1.4M
1.6M
1.8M
2M
2.2M

# 5 Experimentos

- Tabela de resultados do artigo “Playing Atari with Deep Reinforcement Learning”:

| | B. Rider | Breakout | Enduro | Pong | Q*bert | Seaquest | S. Invaders |
|---|---|---|---|---|---|---|---|
| **Random** | 354 | 1.2 | 0 | −20.4 | 157 | 110 | 179 |
| **Sarsa** [3] | 996 | 5.2 | 129 | −19 | 614 | 665 | 271 |
| **Contingency** [4] | 1743 | 6 | 159 | −17 | 960 | 723 | 268 |
| **DQN** | **4092** | **168** | **470** | **20** | **1952** | **1705** | **581** |
| **Human** | 7456 | 31 | 368 | −3 | 18900 | 28010 | 3690 |
| **HNeat Best** [8] | 3616 | 52 | 106 | 19 | 1800 | 920 | **1720** |
| **HNeat Pixel** [8] | 1332 | 4 | 91 | −16 | 1325 | 800 | 1145 |
| **DQN Best** | **5184** | **225** | **661** | **21** | **4500** | **1740** | 1075 |

Table 1: The upper table compares average total reward for various learning methods by running an $\epsilon$-greedy policy with $\epsilon = 0.05$ for a fixed number of steps. The lower table reports results of the single best performing episode for HNeat and DQN. HNeat produces deterministic policies that always get the same score while DQN used an $\epsilon$-greedy policy with $\epsilon = 0.05$.

# 5 Experimentos

- Resultados do artigo “Playing Atari with Deep Reinforcement Learning”:

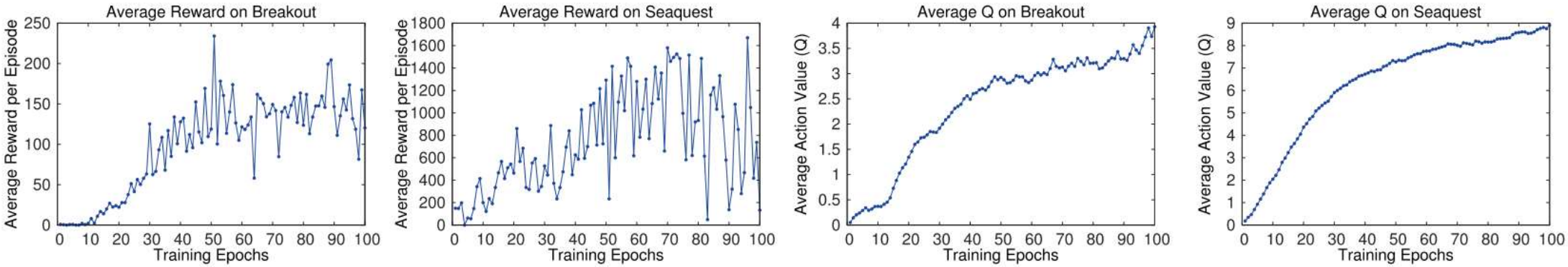



Figure 2: The two plots on the left show average reward per episode on Breakout and Seaquest respectively during training. The statistics were computed by running an $\epsilon$-greedy policy with $\epsilon = 0.05$ for 10000 steps. The two plots on the right show the average maximum predicted action-value of a held out set of states on Breakout and Seaquest respectively. One epoch corresponds to 50000 minibatch weight updates or roughly 30 minutes of training time.

# 6 Conclusão

- O algoritmo é robusto mesmo com uma alta diminuição dos parâmetros.
- Idealmente cada experimento deveria ser repetido diversas vezes e retirar a média, por exemplo, 10 ou até 100 vezes. Todavia a intensidade computacional torna difícil essa computação.
- Diferentes números de parâmetros da rede podem ocasionar comportamentos diferentes de uma maneira não diretamente interpretável.
- Não necessariamente há comportamento catastrófico na diminuição drástica dos paramêtros, vide o “planalto” atingido por um dos experimentos.
- Atingir recompensas próximas do máximo de 20 não parece proibitivo ou frágil.
- Foram tirados vídeos de todos os agentes do experimento.
  - Mostremos alguns.

# Referência:

- *Playing Atari with Deep Reinforcement Learning*, (Mnih, 2013).