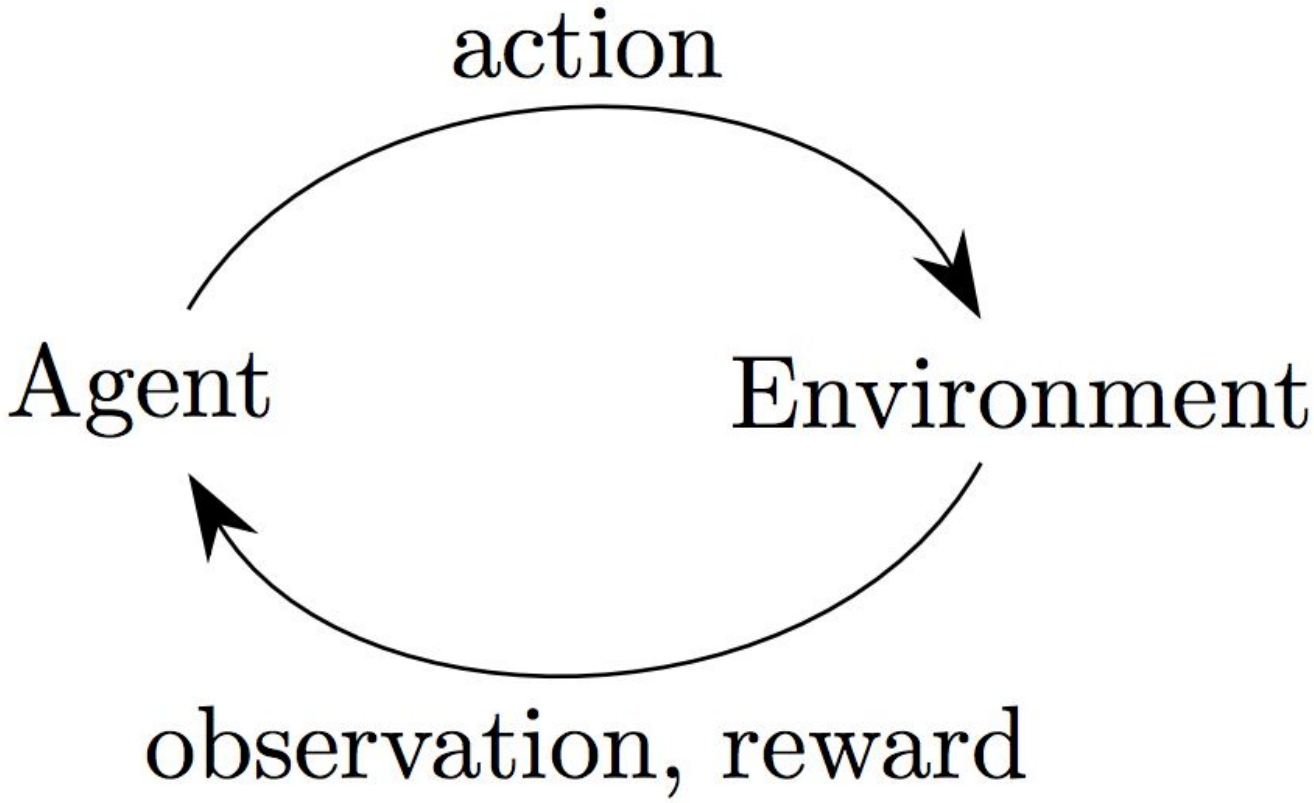
# Uczenie Maszynowe

Wprowadzenie do Reinforcement Learning, 4/20/2017

# Definicja z wikipedii

**uczenie przez wzmacnianie** (ang. Reinforcement Learning) – uczenie przez wzmacnianie to metoda wyznaczania optymalnej polityki sterowania przez agenta w nieznanym mu środowisku, na podstawie interakcji z tym środowiskiem. Jedyną informacją, na której agent się opiera jest sygnał wzmocnienia (poprzez wzorowanie się na pojęciu wzmocnienia z nauk behawioralnych w psychologii), który osiąga wysoką wartość (nagrodę), gdy agent podejmuje poprawne decyzje lub niską (karę) gdy podejmuje decyzje błędnie.


action
Agent
Environment
observation, reward

# Dlaczego to jest ciekawe?

- Metody RL zaczynają działać bardzo dobrze w praktycznych zastosowaniach
- Umożliwiają nam stworzenie stworzenie systemów, których ręczne pisanie jest trudne lub niewykonalne, np. bot do gry Go

# Przykłady - robotyka

# Przykłady - gry komputerowe

# Przykłady - tłumaczenie maszynowe

| *Input sentence:* | *Translation (PBMT):* | *Translation (GNMT):* | *Translation (human):* |
|---|---|---|---|
| 李克強此行將啟動中加總理年度對話機制，與加拿大總理杜魯多舉行兩國總理首次年度對話。 | Li Keqiang premier added this line to start the annual dialogue mechanism with the Canadian Prime Minister Trudeau two prime ministers held its first annual session. | Li Keqiang will start the annual dialogue mechanism with Prime Minister Trudeau of Canada and hold the first annual dialogue between the two premiers. | Li Keqiang will initiate the annual dialogue mechanism between premiers of China and Canada during this visit, and hold the first annual dialogue with Premier Trudeau of Canada. |

# Różnice między Reinforcement Learning (RL) i Supervised Learning (SL)

Supervised Learning:

- Środowisko losuje parę (x, y) ~ $\rho$
- Agent (model) przewiduje y’ = f(x)
- Agent płaci cenę *loss(y, y’)* za swoją predykcję

Środowisko zadaje agentowi pytanie i mówi jaka była poprawna odpowiedź.

# Różnice między Reinforcement Learning (RL) i Supervised Learning (SL)

Reinforcement Learning:

- Środowisko losuje $x_t \sim P(x_t \mid x_{t-1}, y_{t-1})$
  - $x_t$ zależy od poprzednich akcji!
- Agent podejmuje decyzję $y_t = f(x_t)$
- Agent płaci cenę $c_t \sim P(c_t \mid x_t, y_t)$, przy czym agent nie zna rozkładu P

# Różnice między Reinforcement Learning (RL) i Supervised Learning (SL)

W skrócie:

- Nie mamy pełnego dostępu do funkcji, którą chcemy optymalizować. Dowiadujemy się o niej przez interakcje.
- Stany wejściowe i koszty interakcji zależą od decyzji podjętych wcześniej

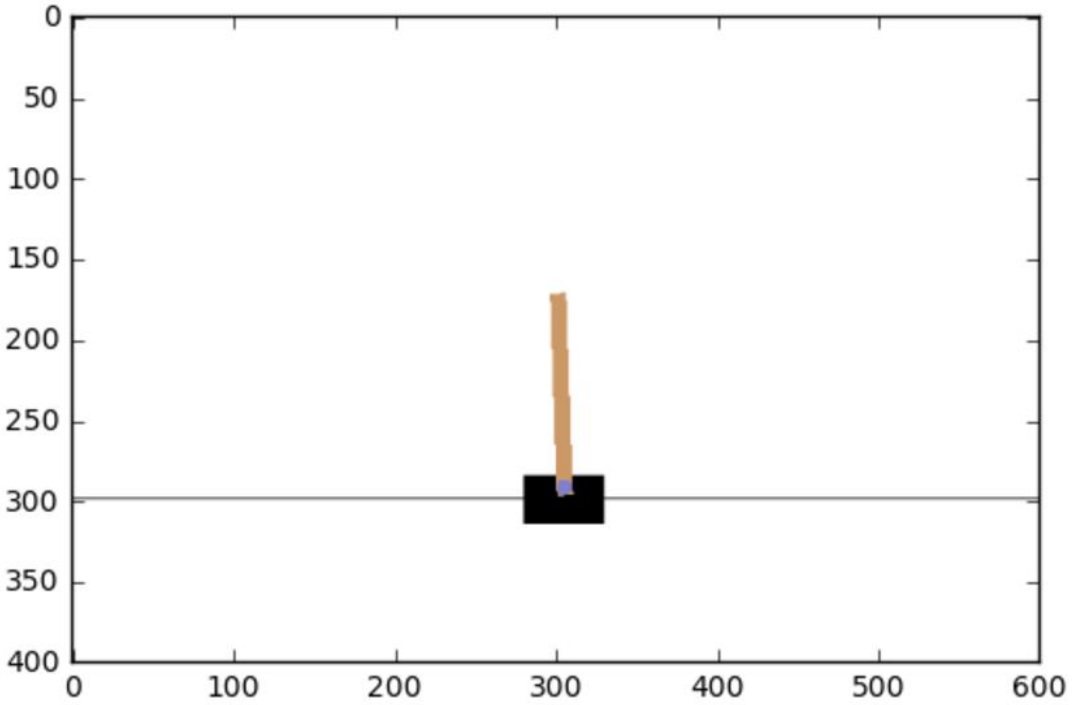
# Pierwsze kroki - OpenAI gym

In [2]:
```
import gym
env = gym.make('CartPole-v1')
env.reset()
```

```
[2017-04-19 21:03:08,649] Making new env: CartPole-v1
```

# Pierwsze kroki - OpenAI gym

```
In [3]: plt.imshow(env.render(mode='rgb_array'))
Out[3]: <matplotlib.image.AxesImage at 0x1131ed2e8>
```



# Pierwsze kroki - random agent

In [6]:
```
def agent(observation):
    return env.action_space.sample()

done = False
observation = env.reset()
while not done:
    action = agent(observation)
    observation, reward, done, _ = env.step(action)
    env.render()
```

# Dostępne środowiska - https://gym.openai.com

MuJoCo

Continuous control tasks, running in a fast physics simulator.


InvertedPendulum-v1
Balance a pole on a cart.

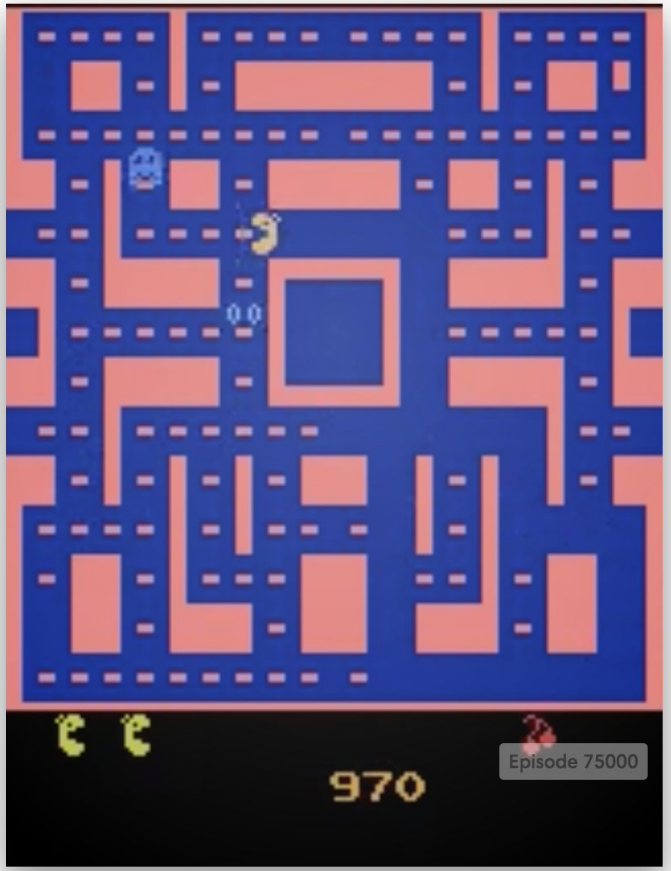

InvertedDoublePendulum-v1
Balance a pole on a pole on a cart.


Reacher-v1
Make a 2D robot reach to a randomly located target.


HalfCheetah-v1
Make a 2D cheetah robot run.


Swimmer-v1
Make a 2D robot swim.


Hopper-v1
Make a 2D robot hop.



# Dostępne środowiska - https://universe.openai.com/

# Rozwiążmy teraz CartPole

- Użyjemy do tego bardzo prostej sieci neuronowej:

```
wejscie = x = tf.placeholder(tf.float32, [None] + list(env.observation_space.shape))
x = tf.nn.relu(linear(x, 'l0', 32))
x = linear(x, 'l1', 1)
predykcja = tf.nn.sigmoid(x)
akcja = tf.greater(predykcja, tf.random_uniform(tf.shape(predykcja)))
```

- W zależności od prawdopodobieństwa zwróconego przez sieć, wykonamy ruch w lewo lub prawo,

# Cross-Entropy Method (CEM)

Bardzo prosty algorytm, który działa zaskakująco dobrze na wielu problemach.

Ogólny schemat:

- Wylosuj 100 różnych wag sieci i uruchom każdą z nich na problemie
- Zbierz 20% najlepszych wyników i uśrednij ich wagi
- Iteruj w nieskończoność

Wagi losujemy początkowo z rozkładu $\theta \sim N(\theta_{init}, std=1.0)$ i będziemy uaktualniać std = np.std(`20% najlepszych wag`) + trochę dodatkowego szumu.

Cały algorytm można zaimplementować w 10 liniach.

# CEM na CartPole

Trening zajmuje zwykle <3 minuty na moim laptopie, żeby osiągnąć perfekcyjny wynik. CEM potrzebuje do tego ok 15 iteracji (* 100 epizodów)

Scores after  1 iterations - deterministic: 9.46 stochastic: 19.65
Scores after  6 iterations - deterministic: 143.62 stochastic: 83.55
Scores after 11 iterations - deterministic: 493.41 stochastic: 444.34
Scores after 16 iterations - deterministic: 500.00 stochastic: 499.29

# CEM

- Warto zwrócić uwagę, że po raz pierwszy wytrenowaliśmy sieć neuronową nie używając pochodnych!
- CEM jest zaskakująco mocny na wielu niskopoziomowych środowiskach
- Uruchomimy niedługo CEM na innych zadaniach, ale potrzebujemy wprowadzić kilka poprawek, żeby radzić sobie z różnymi typami akcji:
  - lewo/prawo
  - naciśnij jeden z N klawiszy
  - Obróć się o X stopni

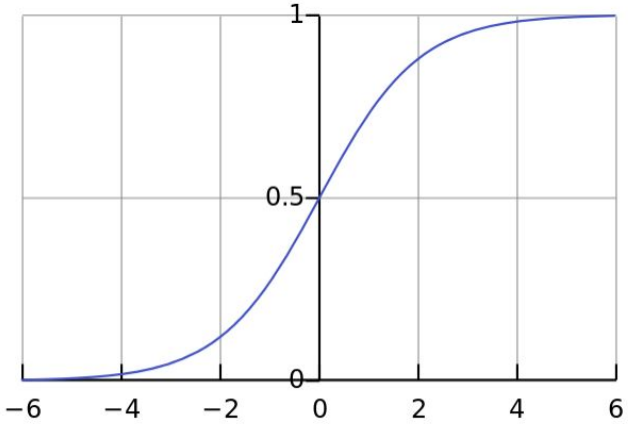
# Action Spaces

- Różne rodzaje akcji można interpretować jako rozkłady podobieństwa
  - Binarne decyzje - rozkład Bernoulliego
  - jeden-z-N - rozkład kategoryczny(?)
  - Numeryczne - rozkład normalny
- Można wybrać inne rozkłady w miarę potrzeb (np. poisson), ale te są najczęściej stosowane

# Rozkład Bernoulliego

- Najczęściej jest parametryzowany przez funkcję sigmoid, która ściąga wartości do przedziału [0, 1]
- Odpowiadająca funkcja kosztu jest zaimplementowana w TF jako: tf.nn.sigmoid_cross_entropy_with_logits()

$$S(t) = \frac{1}{1 + e^{-t}}$$



# Rozkład kategoryczny

- Najczęściej parametryzowany przez funkcje softmax, która normalizuje wektor wejściowy tak aby wartości sumowały się do 1 i były nieujemne
- Odpowiadająca funkcja kosztu jest zaimplementowana w TF jako: tf.nn.sparse_softmax_cross_entropy_with_logits()

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

# Rozkład normalny (ze stałą wariancją)

- Nie wymaga dodatkowej parametryzacji, tzn. wystarczy wziąć wartość warstwy po mnożeniu macierzy
- Odpowiada funkcji kosztu 0.5 (x - y)^2

$$f(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

- log N(y; $\mu$=x, σ=1.0) = ~~0.5 log(2 pi)~~ + 0.5 (y - x)^2

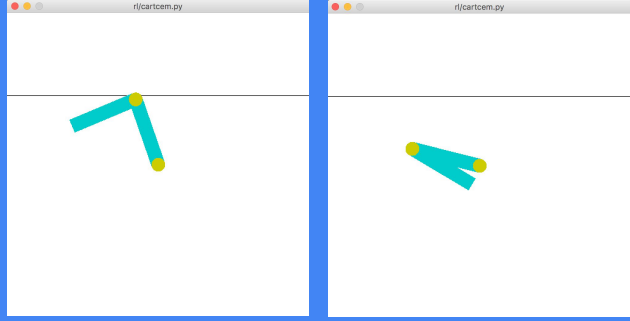
# Action spaces

- Ta interpretacja akcji (i odpowiadające im funkcje kosztu) nie jest niczym odkrywczym, ale pomaga uporządkować notacje:
- Sieć neuronowa zwraca wartości, które parametryzują jakiś rozkład **p**
- Funkcja kosztu do optymalizacji to zwykle **- log p(y|x; θ)**
- Można wyznaczyć dodatkowe statystyki rozkładu jak np:
  - Entropia akcji - przydaje się do wizualizacji/debugu (nie powinna spadać do zera!)
    - Używa się jej często jako dodatkowa nagroda przy RL
  - Moda - używana przy ewaluacji
  - Dywergencja Kullbacka-Leiblera (KL) - używana do szacowania jak bardzo się zmienia rozkład akcji pomiędzy uaktualnieniami wag

# Action spaces

- Mając te intuicje, jesteśmy też w stanie prosto rozwiązać przypadek w którym agent jednocześnie może zwrócić numeryczną wartość A i kategoryczną B (predykcja na AxB)
- Odpowiada to rozkładowi
  **p(a, b | x,θ) = Normal(a|x; θ) * Categorical(b|x; θ)**
- Funkcja kosztu to suma kosztów każdego z tych rozkładów, etc

# CEM na Acrobot-v1



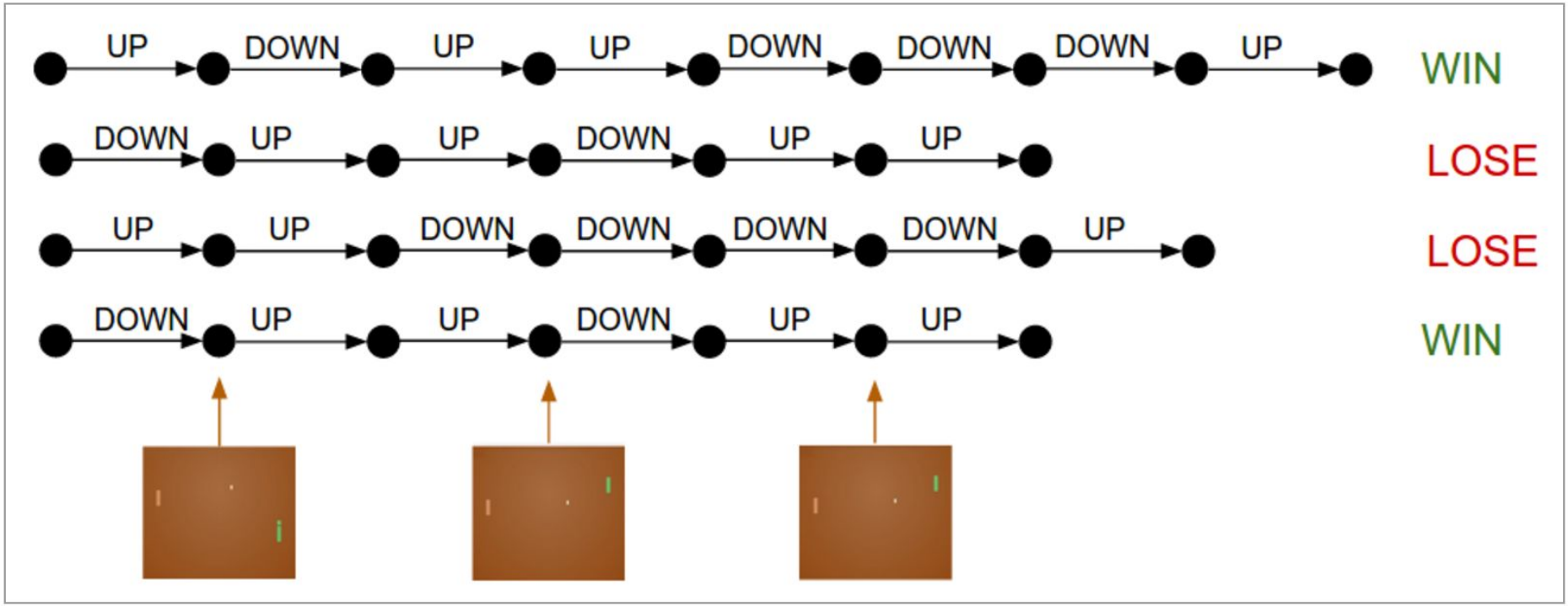

Scores after 1 iterations - deterministic: -290.20 stochastic: -320.25
Scores after 6 iterations - deterministic: -90.83 stochastic: -86.53
Scores after 11 iterations - deterministic: -85.43 stochastic: -88.81
Scores after 16 iterations - deterministic: -86.86 stochastic: -88.80
Scores after 21 iterations - deterministic: -88.53 stochastic: -89.11
Scores after 26 iterations - deterministic: -85.02 stochastic: -87.78

# Policy Gradients

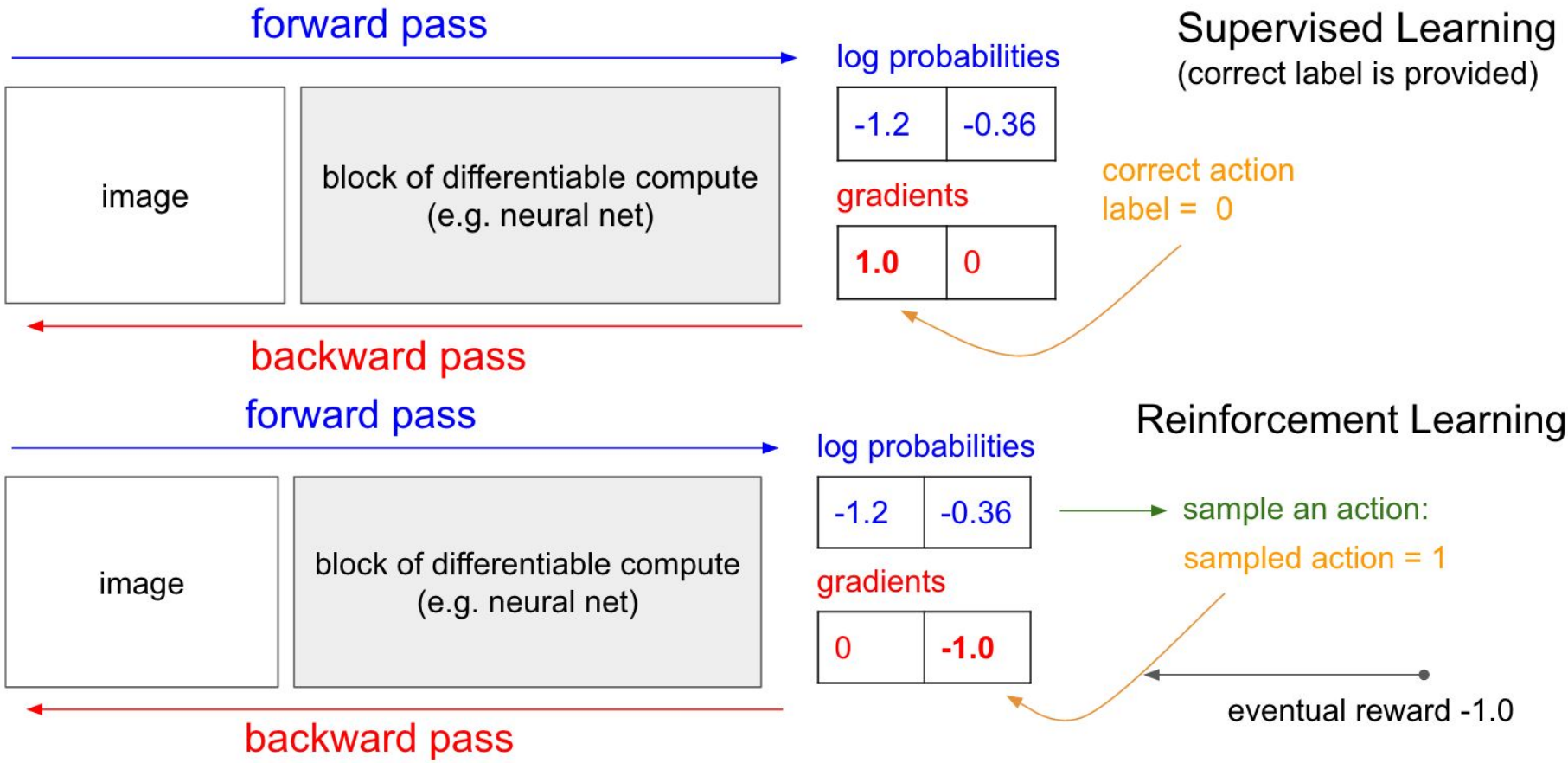
- Do tej pory nie używaliśmy jeszcze pochodnych przy optymalizowaniu sieci neuronowej
- Suma przyszłych nagród będzie najprawdopodobniej nieróżniczkowalną funkcją, ale gdybyśmy tylko znali optymalne akcje, to potrafilibyśmy wyznaczyć pochodne **log p(y|x)** (czyli traktować ten problem tak samo jak przy supervised learning)

# Policy Gradients

*Policy Gradients: Run a policy for a while. See what actions led to high rewards. Increase their probability.*



# Policy Gradients



# Policy Gradients - bardziej formalnie

$$\nabla_\theta E_x[f(x)] = \nabla_\theta \sum_x p(x) f(x) \qquad \text{definition of expectation}$$

$$= \sum_x \nabla_\theta p(x) f(x) \qquad \text{swap sum and gradient}$$

$$= \sum_x p(x) \frac{\nabla_\theta p(x)}{p(x)} f(x) \qquad \text{both multiply and divide by } p(x)$$

$$= \sum_x p(x) \nabla_\theta \log p(x) f(x) \qquad \text{use the fact that } \nabla_\theta \log(z) = \frac{1}{z} \nabla_\theta z$$

$$= E_x[f(x) \nabla_\theta \log p(x)] \qquad \text{definition of expectation}$$

# Policy Gradients - bardziej formalnie

To znaczy, że jesteśmy w stanie wyznaczyć estymator do pochodnej naszej funkcji, który jest *unbiased*

- Te wyprowadzenie jest poprawne nawet gdy sama funkcja f(x) jest nieciągła/nieróżniczkowalna
- Podstawiając $f(.) = \Sigma r_i$ dostajemy metodę na optymalizowanie sumy nagród

# Policy Gradients - algorytm

1. Uruchom sieć neuronową 100 razy na środowisku, losując akcje w każdym kroku (z rozkładu sieci) i zbierz dane (stany, akcje, nagrody)
2. Potraktuj wylosowane akcje jako etykiety i dodaj do siebie pochodne log p(y|x), pomnożone przez sumę nagród w danym epizodzie
    a. Nazwijmy ten czynnik przez **A**, od angielskiego *advantage*
3. Uaktualnij wagi sieci i wróć do 1.

Ta wersja algorytmu nazywa się REINFORCE (1991)

# Problemy z Reinforce

- Jeśli wykonaliśmy bardzo dobry ruch na początku epizodu a potem zrobiliśmy coś bardzo głupiego, to model będzie zniechęcany do wykonywania podobnych akcji w przyszłości
  - Efekt jest pomniejszony z powodu rozgrywania ogromnej liczby gier w trakcie treningu
  - W praktyce używa się *discounting*, tzn. optymalizujemy $\Sigma \gamma^i r_i$ co powoduje, że przyszłe wydarzenia są coraz mniej ważne. $\gamma$=0.99 odpowiada patrzenie na nagrody ~100 kroków do przodu
    - Warto dodać, że ta operacja dodaje *bias*, ale używanie jej jest często konieczne
- Zdarza się używać efektywnego batch_size=$10^6$ i to często nie wystarcza!
  - Wariancja estymatora pochodnych jest ogromna i cała zabawa teraz polega na jej redukcji

# “Bias” jaki i błędy w nagrodach od środowiska powodują często dziwne efekty

# Redukcja wariancji - spostrzeżenia

- Prosty przykład na redukcję wariancji estymatora - *A* może być równe sumie *przyszłych* nagród w danym epizodzie zamiast sumie wszystkich nagród
- Bardziej formalnie, można pokazać, że dowolna funkcja, która *nie* zależy od aktualnej akcji i przyszłych stanów, może być odjęta od A i wciąż pozostaje dobrym estymatorem

$$\nabla_\theta \mathbb{E}_\tau [R] = \mathbb{E}_\tau \left[ \sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t \mid s_t, \theta) \left( \sum_{t'=t}^{T-1} r_{t'} - b(s_t) \right) \right]$$

# Redukcja wariancji - baseline

- Dla dowolnego b, estymator jest *unbiased*
- Oczekiwana suma przyszłych nagród jest bardzo dobre jako *baseline*
  $b(s_t) \approx \mathbb{E}\left[r_t + r_{t+1} + r_{t+2} + \cdots + r_{T-1}\right]$
  (gdybyśmy tylko znali tę wartość przy podejmowaniu decyzji)
- Interpretacja jest taka, że chcemy zwiększyć prawdopodobieństwa akcji, które nas najbardziej pozytywnie *zaskoczyły*

$$\nabla_\theta \mathbb{E}_\tau\left[R\right] = \mathbb{E}_\tau\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t \mid s_t, \theta)\left(\sum_{t'=t}^{T-1} r_{t'} - b(s_t)\right)\right]$$

# Baseline - dowód poprawności

$$
\begin{aligned}
&\mathbb{E}_{\tau}\left[\nabla_{\theta} \log \pi(a_t \mid s_t, \theta) b(s_t)\right] \\
&= \mathbb{E}_{s_{0:t}, a_{0:(t-1)}}\left[\mathbb{E}_{s_{(t+1):T}, a_{t:(T-1)}}\left[\nabla_{\theta} \log \pi(a_t \mid s_t, \theta) b(s_t)\right]\right] && \text{(break up expectation)} \\
&= \mathbb{E}_{s_{0:t}, a_{0:(t-1)}}\left[b(s_t)\mathbb{E}_{s_{(t+1):T}, a_{t:(T-1)}}\left[\nabla_{\theta} \log \pi(a_t \mid s_t, \theta)\right]\right] && \text{(pull baseline term out)} \\
&= \mathbb{E}_{s_{0:t}, a_{0:(t-1)}}\left[b(s_t)\mathbb{E}_{a_t}\left[\nabla_{\theta} \log \pi(a_t \mid s_t, \theta)\right]\right] && \text{(remove irrelevant vars.)} \\
&= \mathbb{E}_{s_{0:t}, a_{0:(t-1)}}\left[b(s_t) \cdot 0\right]
\end{aligned}
$$

Last equality because $0 = \nabla_{\theta}\mathbb{E}_{a_t \sim \pi(\cdot \mid s_t)}[1] = \mathbb{E}_{a_t \sim \pi(\cdot \mid s_t)}\left[\nabla_{\theta} \log \pi_{\theta}(a_t \mid s_t)\right]$

# Baseline - typowe implementacje

- Średnia nagroda otrzymywana od i-tego kroku w poprzednich epizodach
- Liniowy model, który próbuje przewidzieć sumę przyszłych nagród
- Inna sieć neuronowa!

# “Vanilla” Policy Gradients - algorytm

Initialize policy parameter $\theta$, baseline $b$

**for** iteration=$1, 2, \ldots$ **do**

Collect a set of trajectories by executing the current policy

At each timestep in each trajectory, compute

the *return* $R_t = \sum_{t'=t}^{T-1} \gamma^{t'-t} r_{t'}$, and

the *advantage estimate* $\hat{A}_t = R_t - b(s_t)$.

Re-fit the baseline, by minimizing $\|b(s_t) - R_t\|^2$,

summed over all trajectories and timesteps.

Update the policy, using a policy gradient estimate $\hat{g}$,

which is a sum of terms $\nabla_\theta \log \pi(a_t \mid s_t, \theta)\hat{A}_t$.

(Plug $\hat{g}$ into SGD or ADAM)

**end for**

# Więcej trików - Value Function

$$\hat{A}_t^{(1)} = r_t + \gamma V(s_{t+1}) - V(s_t)$$

$$\hat{A}_t^{(2)} = r_t + r_{t+1} + \gamma^2 V(s_{t+2}) - V(s_t)$$

$$\dots$$

Aktualnie używamy tego estymatora

$$\hat{A}_t^{(\infty)} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots - V(s_t)$$

$A^1$ ma duży bias i małą wariancję

$A^\infty$ ma niewielki bias, ale za to bardzo dużą wariancje

Najlepsze modele używają średniej ważonej po nich jako estymatora!

https://arxiv.org/abs/1506.02438

# Czemu te estymatory mogą być lepsze?

- $V(s_{t+1})$ może być mnie losowe od faktycznej sumy wszystkich nagród
  - Główny minus jest taki, że value function musi być dość dobre co zwykle nie jest prawdą na początku treningu
- Pozwalają nam też one na uczenie się online - nie musimy czekać do końca epizodów, żeby uaktualnić wagi sieci
  - Jest to bardzo przydatne w środowiskach, które wymagają wiele kroków
  - Łatwiej jest też taki algorytm zrównoleglić jako że wykonuje stałą liczbę kroków

# A3C - równoległa wersja Policy Gradients

**for** iteration=1, 2, . . . **do**

Agent acts for $T$ timesteps (e.g., $T = 20$),

For each timestep $t$, compute

$$\hat{R}_t = r_t + \gamma r_{t+1} + \cdots + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V(s_t)$$

$$\hat{A}_t = \hat{R}_t - V(s_t)$$

$\hat{R}_t$ is target value function, in regression problem

$\hat{A}_t$ is estimated advantage function

Compute loss gradient $g = \nabla_\theta \sum_{t=1}^{T} \left[ -\log \pi_\theta(a_t \mid s_t) \hat{A}_t + c(V(s) - \hat{R}_t)^2 \right]$

$g$ is plugged into a stochastic gradient descent variant, e.g., Adam.

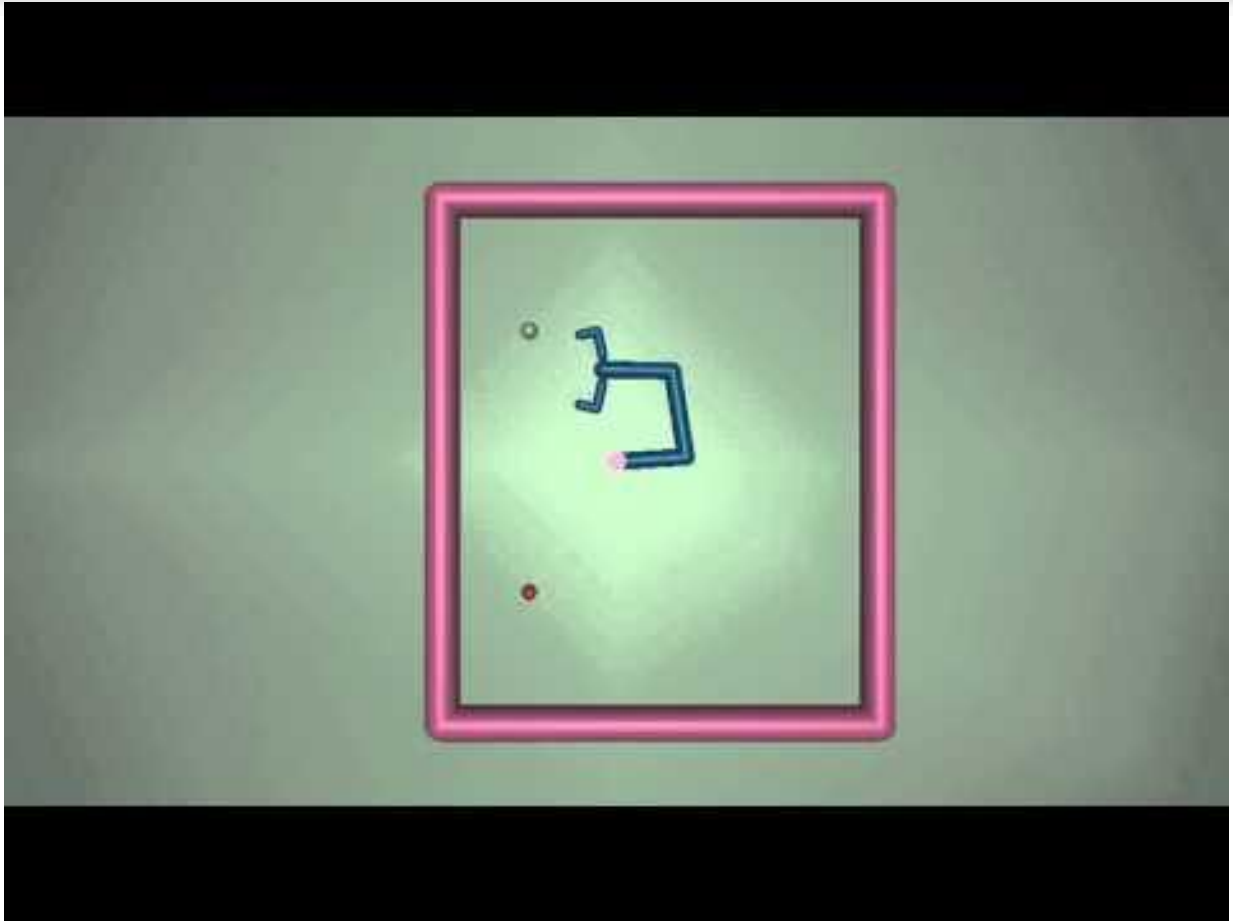
**end for**

# A3C

- Każdą iterację można wykonywać równolegle na wielu wątkach, które wchodzą w interakcje z różnymi instancjami środowiska
- W praktyce dodaje się do funkcji kosztu **stała * entropia(akcja$_t$)**
  - Powoduje ona, że model jest zachęcany do eksploracji nie zwiększając znacząco kosztu obliczeń
- Używając A3C (i kilku wątków) z estymatorem GAE, można rozwiązać Ponga prosto z pikseli w ciągu <2h na laptopie
  - Na wejściu dostajemy zmniejszone obrazki wielkości 42x42x3 piksele
  - Przetwarzamy je kilkoma warstwami sieci konwolucyjnej
  - Wynik przekazujemy do LSTMa (lub innej sieci rekurencyjnej), który przewiduje akcję
- Na większej maszynie (np. m4.16xlarge z Amazonu) i 32 wątkach, agent nie traci żadnego punktu z wbudowaną AI po 10 minutach.

# A3C - videos

# A3C - videos



# A3C - videos

# Inne sposoby na przyspieszenie PG

- Prosty trik, który znacząco przyspiesza algorytmy:
  - W trakcie zbierania danych zapamiętajmy cały wektor prawdopodobieństwa po akcjach
  - Po wykonaniu uaktualnienia wag, wyliczmy na nowo wektory prawdopodobieństw
  - Obliczmy metrykę KL-divergence między poprzednimi a nowymi wektorami
  - Gdy kl > 2 * 0.002, podziel *learning rate* przez 1.5
  - Gdy kl < 0.5 * 0.002, pomnóż *learning rate* przez 1.5
- A3C + ten trik jest bardzo bliski state-of-the-art na wielu problemach

# Problemy z RL

- Agent musi otrzymywać jakieś nagrody ze środowiska, żeby potrafił się nauczyć czegokolwiek
- Jest wiele prac w których ludzie eksperymentowali z różnymi sposobami, które dodają więcej sygnału do nagród
  - Najprostszą z nich jest dodawanie entropii akcji
  - Inną jest nagroda za odwiedzanie nowych stanów (w praktyce nietrywialne jak to zrobić w ogólny sposób i żeby działało)
    - Na małych problemach można zdyskretyzować przestrzeń stanów i dawać nagrodę proporcjonalną do: 1 / sqrt(liczba_razy_kiedy_stan_byl_odwiedzony(s_t))