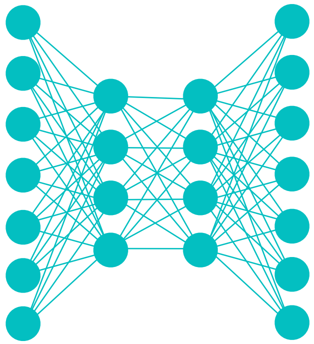
Lecture Notes for

Neural Networks and Machine Learning

Deep Q-Learning



Logistics and Agenda

- Logistics
  - Grading update
  - Sign up for presentation slot (today)!
- Agenda
  - Finish Student Paper Presentation
  - Deep Q-Learning
  - World Models (start today)
  - Any Remaining Time: Class Retrospective

Ajith Mitra Bandlamudi
CS8321 Sect 001 1252

Hrithik Chavva
CS8321 Sect 001 1252

Yonathan Efur
CS8321 Sect 001 1252

Nick Fullerton
CS8321 Sect 001 1252

Nastaran Ghorbani
CS8321 Sect 001 1252

Md Shorif Hossan
CS8321 Sect 001 1252

Tianzuo Huang
CS8321 Sect 001 1252

Md Shamser Ali Javed
CS8321 Sect 001 1252

Sai Sambhu Prasad Kalaga
CS8321 Sect 001 1252

Arman Kamal
CS8321 Sect 001 1252

Tianrui Li
CS8321 Sect 001 1252

Zachary Mitchell
CS8321 Sect 001 1252

Mahesh Molabanti
CS8321 Sect 001 1252

Gaoyang Mou
CS8321 Sect 001 1252

Diego Paredes
CS8321 Sect 001 1252

Travis Peck
CS8321 Sect 001 1252

Michael Perkins
CS8321 Sect 001 1252

Carson Pittman
CS8321 Sect 001 1252

Jeevan Rai
CS8321 Sect 001 1252

Rashedul Islam Seum
CS8321 Sect 001 1252

Yiqing Sha
CS8321 Sect 001 1252

Michael Then
CS8321 Sect 001 1252

Yao Wang
CS8321 Sect 001 1252

Zeeshan Younas
CS8321 Sect 001 1252

Paper Presentation

Reasoning with Latent Thoughts: On the Power of Looped Transformers

Nikunj Saunshi[1], **Nishanth Dikkala**[1], **Zhiyuan Li**[1,2], **Sanjiv Kumar**[1], **Sashank J. Reddi**[1]
{nsaunshi, nishanthd, lizhiyuan, sanjivk, sashank}@google.com
[1]Google Research, [2]Toyota Technological Institute at Chicago

Last Time: Value Iteration and Q Learning

- **Q-Function Value Iteration**:
  - Initialize Q(s,a) to all zeros
  - Take a series of random steps, then follow policy
  - Perform value iteration: $Q(s,a) \leftarrow \sum_{\hat{s} \in S} p_{a,s\rightarrow\hat{s}} \cdot (r_{s,a,\hat{s}} + \gamma \max_{a'} Q(\hat{s}, a'))$
  - Repeat until Q is not changing

Need to estimate $p_{a,s\rightarrow s'}$
Via observed **Transitions**

- **Q-Learning**, (tractable computations, slow convergence):
  - For stability, Bellman approximation with momentum
  - $Q(s,a) \leftarrow (1-\alpha) \cdot Q(s,a) + \alpha \cdot [r_{s,a} + \gamma \max_{a' \in A} Q(s', a')]$
  - Algorithm, start with empty $Q(s,a)$:
    - Sample (with rand) from environment, (s, a, r, s')
    - Make Bellman Update with Momentum
    - Repeat until desired performance

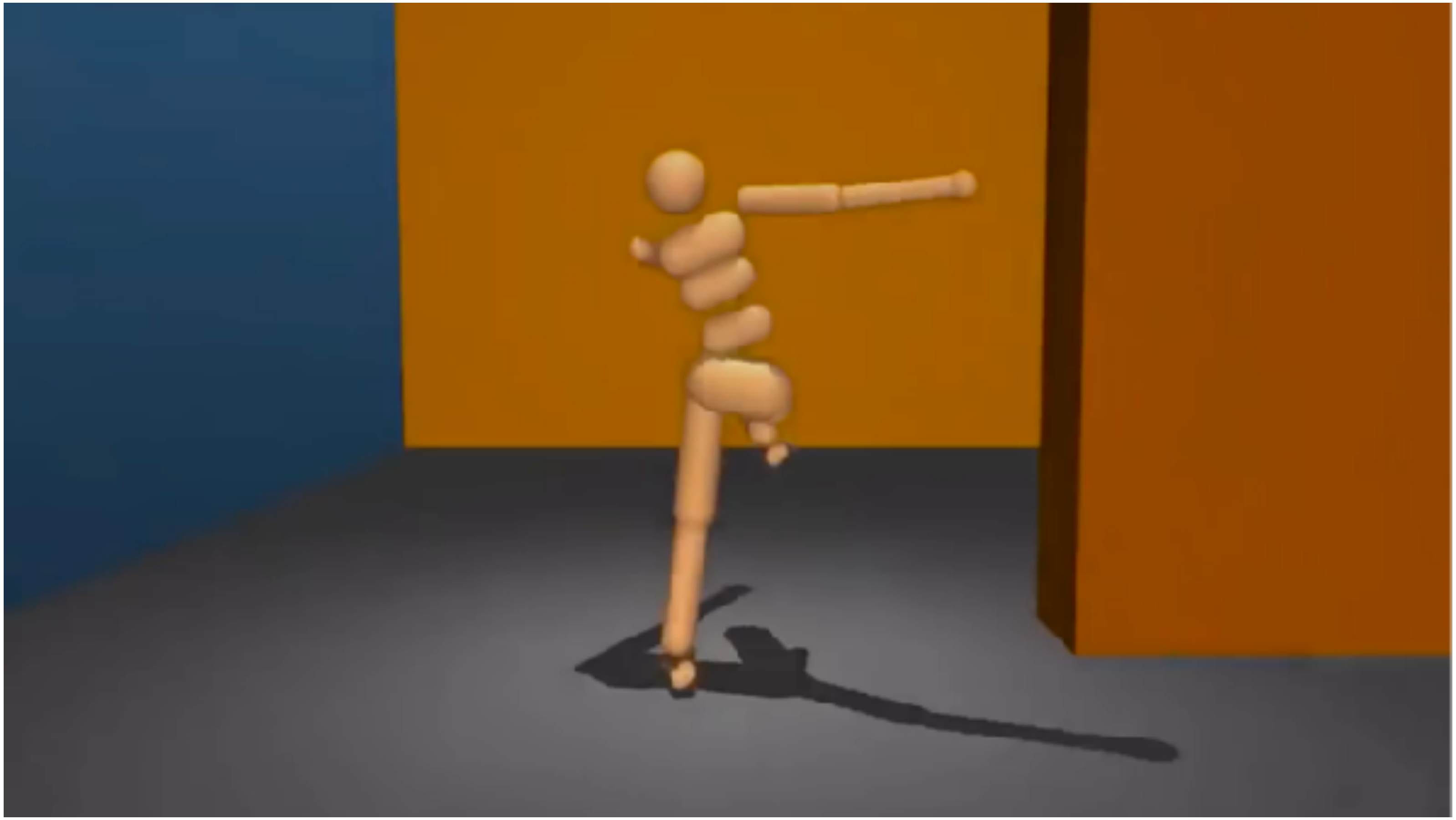
```python
test_env = gym.make("FrozenLake-v0")
train_env = gym.make("FrozenLake-v0")
agent = QLearningAgent(train_env)

iter_no = 0
best_reward = 0.0
while True:
    iter_no += 1
    # sample one step
    s, a, r, next_s = agent.sample_env()

    # update Q
    agent.value_update(s, a, r, next_s)

    # test how well it works
    reward = 0.0
    for _ in range(TEST_EPISODES):
        reward += agent.play_episode(test_env)

    reward /= TEST_EPISODES
    if reward > 0.80:
        print("Solved in %d iterations!" % iter_no)
        break
```

```
Best reward updated 0.000 -> 0.300
Best reward updated 0.300 -> 0.350
Best reward updated 0.350 -> 0.400
Best reward updated 0.400 -> 0.450
Best reward updated 0.450 -> 0.500
Best reward updated 0.500 -> 0.600
Best reward updated 0.600 -> 0.650
Best reward updated 0.650 -> 0.700
Best reward updated 0.700 -> 0.750
Best reward updated 0.750 -> 0.800
Best reward updated 0.800 -> 0.850
Solved in 10103 iterations!
```

Conclusion:

- It still works, but wow it takes much longer to converge!!!
- Placing so much emphasis on the Q-function (to learn all variability) makes the optimization more difficult
- Update to Q noisy (approximation)

Deep Q-Learning



Q-Learning with a Neural Network

- $Q(s,a)$ might be **non-linear** and **state space** is potentially **infinite**. Given s (perhaps continuous), we want the network to give us a row of actions from $Q(s,a)$ table that we can use:

$$[Q(s=s_1, a_1), Q(s=s_1, a_2), Q(s=s_1, a_3), \ldots Q(s=s_1, a_A)\ ]$$
$$\rightarrow [\ Q(s=s_2, a_1), Q(s=s_2, a_2), Q(s=s_2, a_3), \ldots Q(s=s_2, a_A)\ ] \leftarrow$$
$$[\qquad \ldots \text{ other states} \ldots \qquad]$$

- How to train a neural network to be Q?
- Make a loss function which incentives the actual Q-function behavior we desire from a sampled tuple (s, a, r, s')

$$\mathscr{L} = \left[Q(s,a) - [r_{s,a} + \gamma \max_{a' \in A} Q^*(s',a')]\right]^2$$

from current network params

from older network params (better stability)

Periodically Update
Params of Q^* from Q

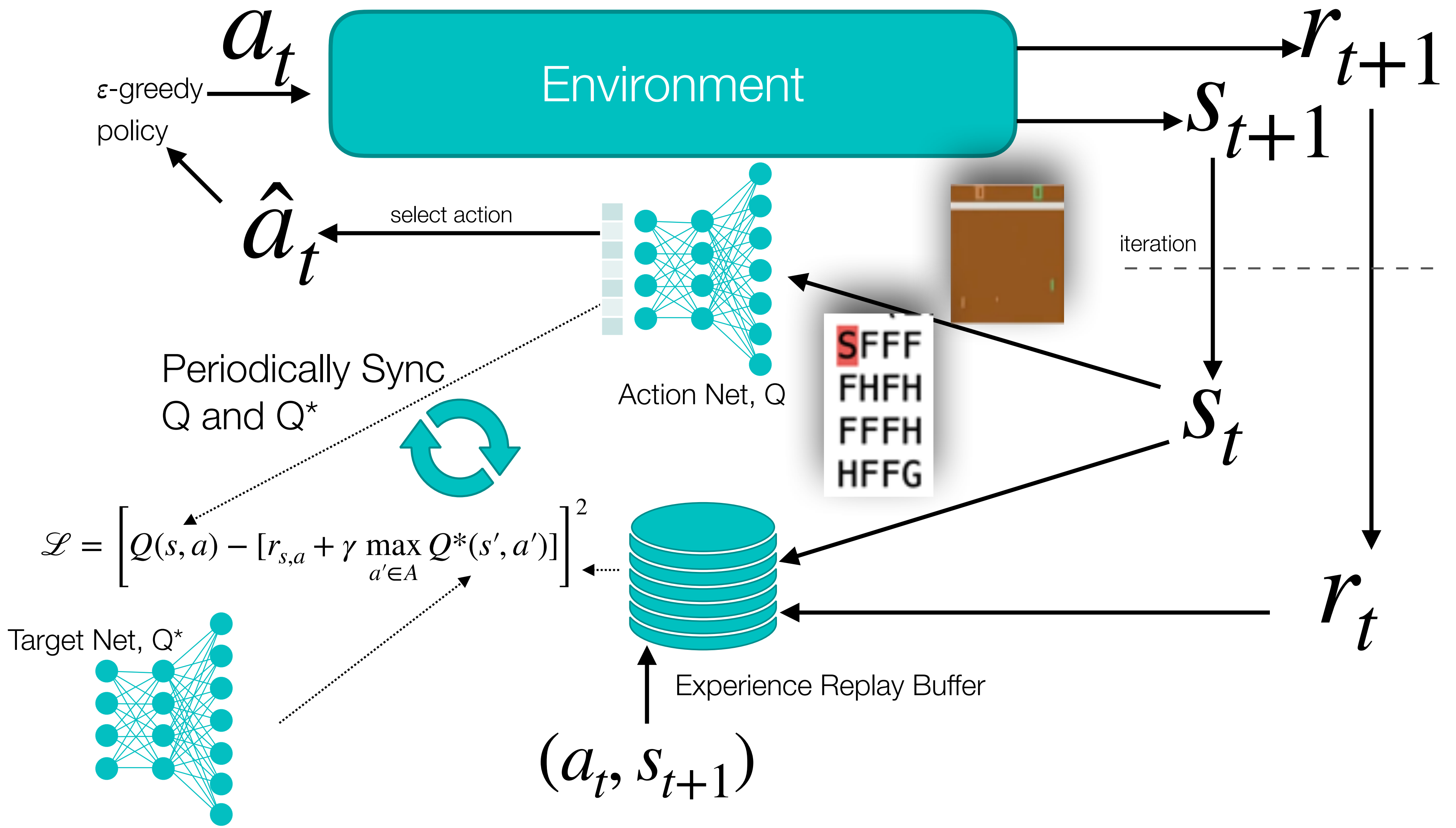
$$\mathscr{L} = \left[Q(s,a) - [r_{s,a}]\right]^2$$

if no next state (env is done)

But we need more power!

- We need to do some **random actions** before following the policy or else we won't learn
- Also, we need to follow the policy more and more during training to get to better places in the environment
  - **Epsilon-Greedy** Approach:
    - Start randomly doing actions with prob epsilon
    - Slowly make epsilon smaller as training progresses
- And also we need to have larger amounts of uncorrelated training batches so we will use **experience replay**
- **Update schedule**: make Q and $Q*$ same every N steps

Deep Q-Learning Overview

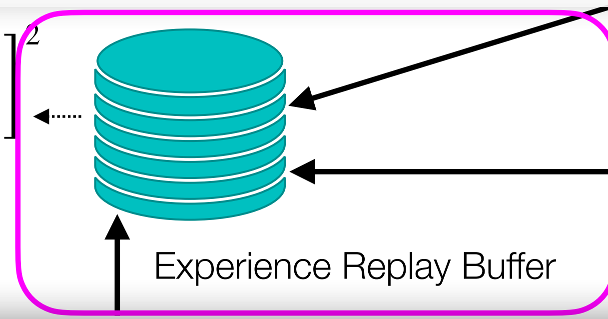


Deep Q-Learning, Implementation Details

```python
class ExperienceBuffer:
    def __init__(self, capacity):
        # this collection will keep track of observed SARS'
        self.buffer = collections.deque(maxlen=capacity)

    def __len__(self):
        return len(self.buffer)

    def append(self, experience):
        # buffer is a queue, so its first in first out
        self.buffer.append(experience)

    def sample(self, batch_size):
        # return a random sample of the experience buffer
        # output will be a numpy array of SARS' and "is done"
        indices = np.random.choice(len(self.buffer), batch_size, replace=False)
        states, actions, rewards, dones, next_states = zip(*[self.buffer[idx] for idx in in
        return np.array(states), np.array(actions), np.array(rewards, dtype=np.float32), \
               np.array(dones, dtype=np.uint8), np.array(next_states)
```

r_{t+1}

$$\mathscr{L} = \left[Q(s,a) - [r_{s,a} + \gamma \max_{a' \in A} Q^*(s',a')] \right]^2$$



r_t

```python
Experience = collections.namedtuple('Experience',
                                    field_names=['state', 'action', 'reward', 'done', 'new_state'])
```

Deep Q-Learning, Implementation Details

```python
class Agent:
    def __init__(self, env, exp_buffer):
        # Agent will track replay buffer
        self.env = env
        self.exp_buffer = exp_buffer
        self._reset()
```

```python
    def play_step(self, net, epsilon=0.0, device="cpu"):
        done_reward = None

        # use epsilon greedy approach for explore/exploit
        if np.random.random() < epsilon:
            # use rand policy
            action = env.action_space.sample()
        else:
            # use Net policy
            state_a = np.array([self.state], copy=False)
            state_v = torch.tensor(state_a).to(device)
            # get the q values for each action, given the state
            q_vals_v = net(state_v)
            # get idx of best action from this vector
            _, act_v = torch.max(q_vals_v, dim=1)
            action = int(act_v.item()) # get int from torch tensor

        # do step in the environment
        new_state, reward, is_done, _ = self.env.step(action)
        self.total_reward += reward
        #new_state = new_state

        # add SARS' to replay buffer
        exp = Experience(self.state, action, reward, is_done, new_state)
        self.exp_buffer.append(exp)
        self.state = new_state
```

ε-greedy policy $\xrightarrow{a_t}$

$\hat{a}_t \leftarrow$

Environment

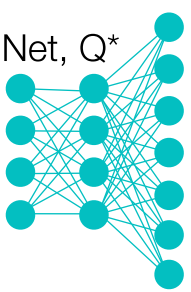
r_{t+1}

s

Periodically Sy
Q and Q*

$\mathscr{L} = \Big[Q(s, a) - [r_{s,a} + \gamma \; \mathrm{m}$ a'

Target Net, Q*



Deep Q-Learning, Implementation Details

```
def calc_loss(batch, net, tgt_net, device
    # batch: set of SARS' from replay buf
    # net: the network we are updating
    # tgt_net: the reference network we u

    # get the observed SARS' from the rep
    states, actions, rewards, dones, next

    # Two networks are passed in, one we
    #  and another that is a previous ver
    #  we use the previous network to obs

    # send the observed states to Net, SA
    states_v = torch.tensor(states).to(de
    next_states_v = torch.tensor(next_sta
    actions_v = torch.tensor(actions).to(
    rewards_v = torch.tensor(rewards).to(
    done_mask = torch.ByteTensor(dones).t
```

```
# get the Network actions for given states
state_action_values = net(states_v).gather(1, actions_v.unsqueeze(-1)
# Q(s,a)

# and the next resulting state
#  but only for states that did not end in a 'done' state
# \max_{a' \in A}Q^*(s',a')
next_state_values = tgt_net(next_states_v).max(1)[0]
next_state_values[done_mask] = 0.0 # ensures these are only rewards

# detach the calculation we just made from computation graph
#  we don't want to back-propagate through this calculation
#  bec                                      to be true
#  Tha                                     s output from
#  the
next_state_values = next_state_values.detach() # because from target

# calc the Q function behavior we want (bellman update)
#       r_{s,a}+\gamma \max_{a' \in A} Q^*(s',a')
expected_state_action_values = rewards_v + next_state_values * GAMMA

# compare what we have to what we want, will update this via back prop
# L=[ Q(s,a)-[r_{s,a}+\gamma \max_{a' \in A}Q^*(s',a')] ]^2
return nn.MSELoss()(state_action_values, expected_state_action_values
```

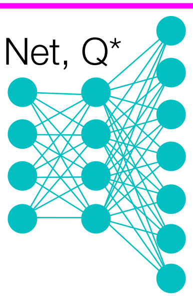
$Q(s,a)$

$\max_{a'} Q^*(s',a')$

Only update `net` not `tgt_net`

$$\mathscr{L} = \left[Q(s,a) - [r_{s,a} + \gamma \max_{a' \in A} Q^*(s',a')] \right]^2$$

Target Net, Q^*



t

Experience Replay Buffer

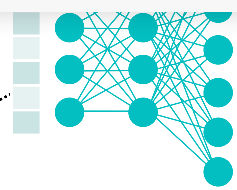
$$\mathscr{L} = \left[Q(s,a) - [r_{s,a}] \right]^2$$

if no next state (env is done)

Deep Q-Learning, Implementation Details

```
while True:
    # track epsilon and cool it down
    frame_idx += 1
    epsilon = max(EPSILON_FINAL, EPSILON_START - frame_idx / EPSILON_DECAY_LAST_FRAME)

    # play step and add to experience buffer
    # here is where we populate the buffer according to a mix of random play and
    # using the policy
    reward = agent.play_step(net, epsilon, device=device)
```

a_t



iteration

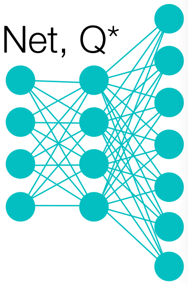
Action Net, Q

s_t

Periodically Sync
Q and Q*

$\mathscr{L} = \Big[Q(s,a) - [$

Target Net, Q*

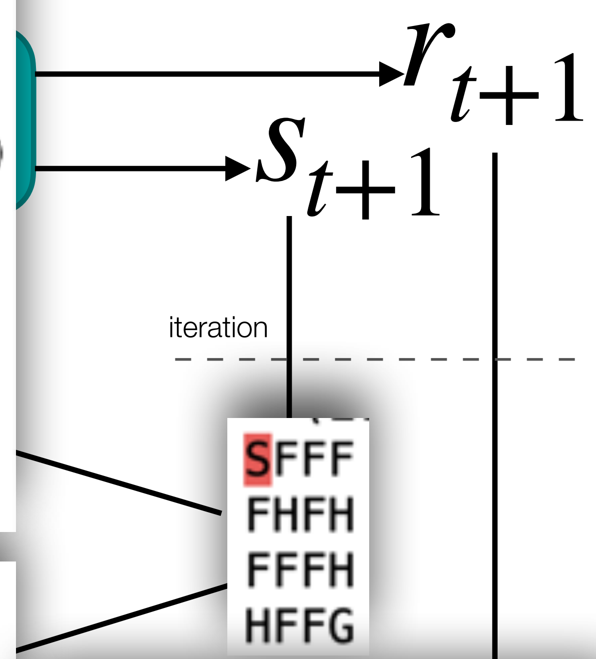


```
# sync the networks every so often
if frame_idx % SYNC_TARGET_FRAMES == 0:
    # use current state dictionary of values to overwrite tgt_net
    tgt_net.load_state_dict(net.state_dict())

# use experience buffer and two networks to get loss
optimizer.zero_grad()
batch = buffer.sample(BATCH_SIZE) # grab some examples from buffer
loss_t = calc_loss(batch, net, tgt_net, device=device)
loss_t.backward()
optimizer.step()
```

Deep Q-Learning, Frozen Lake

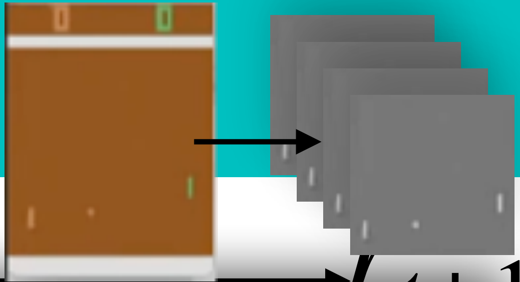
```
Net(
  (net): Sequential(
    (0): Linear(in_features=16, out_features=256, bias=True)
    (1): ReLU()
    (2): Linear(in_features=256, out_features=128, bias=True)
    (3): ReLU()
    (4): Linear(in_features=128, out_features=4, bias=True)
  )
)
16 4
Best mean reward updated 0.000 -> 0.077, model saved
300: done 35 iterations, mean reward 0.029, eps 1.00
400: done 51 iterations, mean reward 0.039, eps 1.00
3000: done 392 iterations, mean reward 0.030, eps 0.97
```

```
125300: done 9454 iterations, mean reward 0.650, eps 0.00
126500: done 9479 iterations, mean reward 0.630, eps 0.00
130100: done 9561 iterations, mean reward 0.730, eps 0.00
Best mean reward updated 0.740 -> 0.750, model saved
Best mean reward updated 0.750 -> 0.760, model saved
Best mean reward updated 0.760 -> 0.770, model saved
131000: done 9585 iterations, mean reward 0.770, eps 0.00
Best mean reward updated 0.770 -> 0.780, model saved
Best mean reward updated 0.780 -> 0.790, model saved
Best mean reward updated 0.790 -> 0.800, model saved
Best mean reward updated 0.800 -> 0.810, model saved
Solved in 132361 frames!
```




```
EPSILON_DECAY_LAST_FRAME = 10**5
EPSILON_START = 1.0
EPSILON_FINAL = 0.0

MEAN_REWARD_BOUND = 0.8
SYNC_TARGET_FRAMES = 50
BATCH_SIZE = 16
REPLAY_SIZE = 500
REPLAY_START_SIZE = 500
LEARNING_RATE = 1e-4
```

Deep Q-Learning, Atari Pong



```
class DQN(nn.Module):
    def __init__(self, input_shape, n_actions):
        super(DQN, self).__init__()
```

See `q_learning_utils.py`

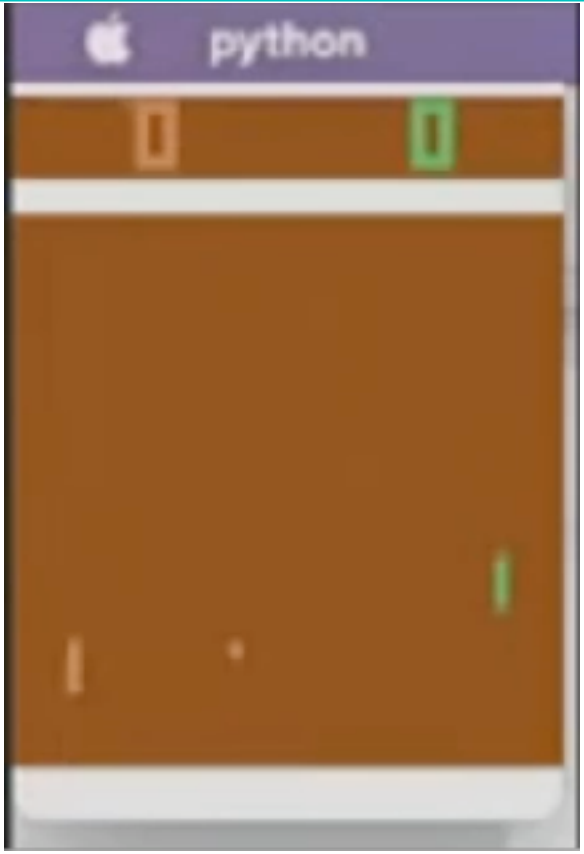
```
# load our own custom environment
env = make_env(DEFAULT_ENV_NAME)
# this has lots of tricks in it, including:
# 1. press fire to start game in atari
# 2. Max pool across frames (max across four frames, keeping last two)
# 3. Resize, gray scale, and crop atari images (get rid of score and other unneeded pixels)
# 4. PyTorch Image conversion
# 5. Image scaling (input 0 to 1, rather than 0-255)
# 6. Use last four buffer of previous observations

# load up a simple convolutional network
# 3 layers of strided conv and two fc layers
```

3 Days CPU Training

```
Best mean reward updated 16.670 -> 16.690, model saved
521198: done 287 games, mean reward 16.700, eps 0.02, speed 5.62 f/s
Best mean reward updated 16.690 -> 16.700, model saved
523572: done 288 games, mean reward 16.610, eps 0.02, speed 5.73 f/s
525237: done 289 games, mean reward 16.610, eps 0.02, speed 5.84 f/s
527041: done 290 games, mean reward 16.630, eps 0.02, speed 5.82 f/s
528859: done 291 games, mean reward 16.640, eps 0.02, speed 5.78 f/s
530865: done 292 games, mean reward 16.630, eps 0.02, speed 5.44 f/s
532944: done 293 games, mean reward 16.620, eps 0.02, speed 5.19 f/s
```

```
EPSILON_DECAY_LAST_FRAME = 10**5
EPSILON_START = 1.0
EPSILON_FINAL = 0.02
```

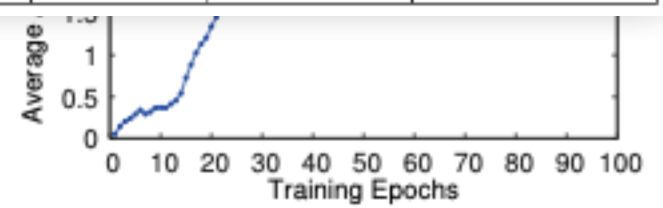
The Trained System (on my laptop)



Strategy: After winning one point, the RL Agent serves and the game is over from there. It will move to the bottom while banking the ball such that the CPU always overshoots the bounce.

Breakout, From Original Atari Deep-Q

	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



Deep Q-Learning Reinforcement Learning

M. Lapan Implementation for Frozen Lake and Atari!

$$\mathscr{L} = \left[Q(s,a) - [r_{s,a} + \gamma \max_{a' \in A} Q^*(s',a')] \right]^2$$

from current network params

from older network params (better stability)

$$\mathscr{L} = \left[Q(s,a) - [r_{s,a}] \right]^2$$

if no next state (env is done)

Look through at your leisure:

`08a_Basics_Of_Reinforcement_Learning.ipynb`

World Models

The Problem

World Models

Can agents learn inside of their own dreams?

DAVID HA Google Brain Tokyo, Japan	JÜRGEN SCHMIDHUBER NNAISENSE Swiss AI Lab, IDSIA (USI & SUPSI)	March 27 2018	NIPS 2018 Paper	YouTube Talk	Download PDF

https://worldmodels.github.io

A Motivation

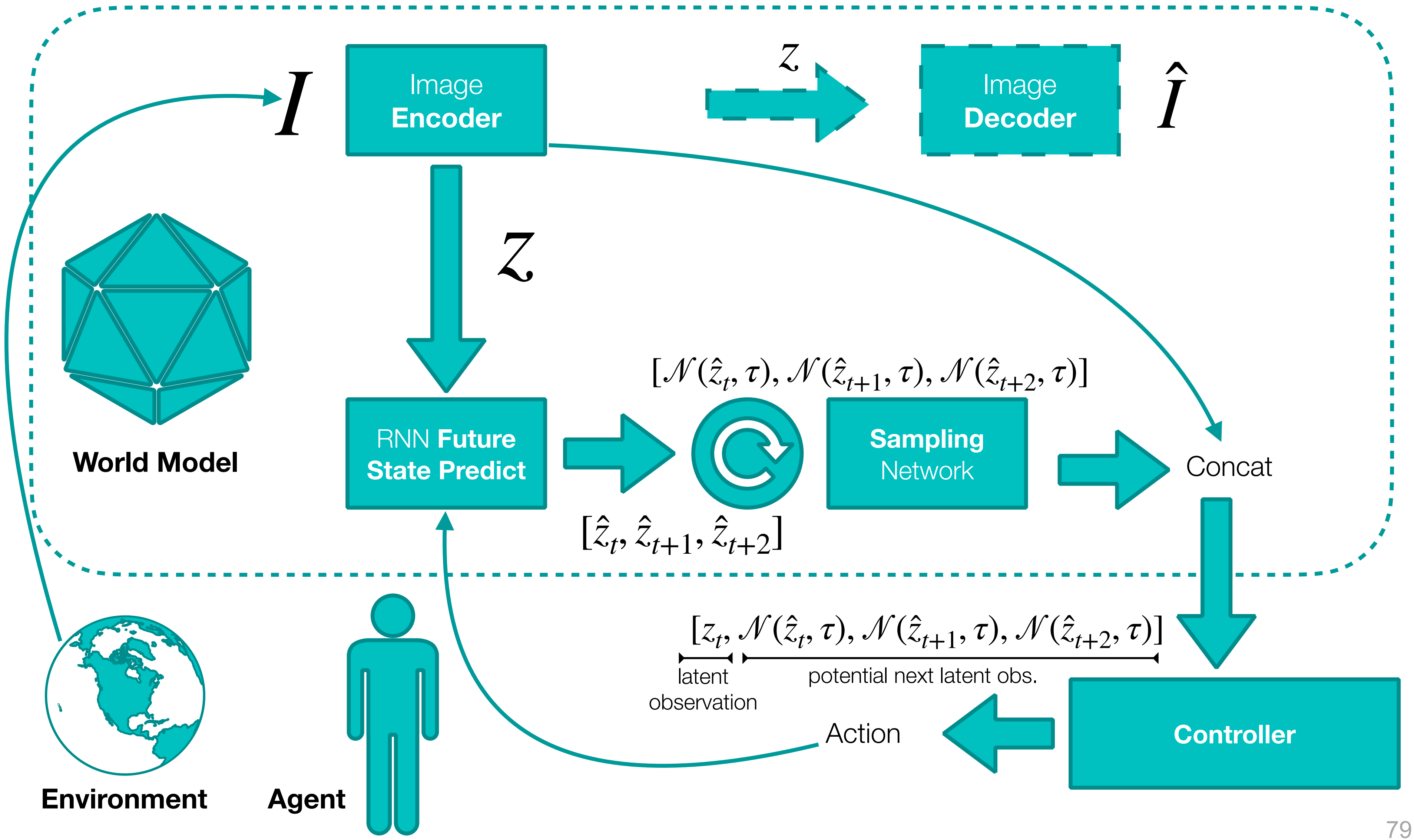
Agents can dream!

And academia can dream about driving the hype train!

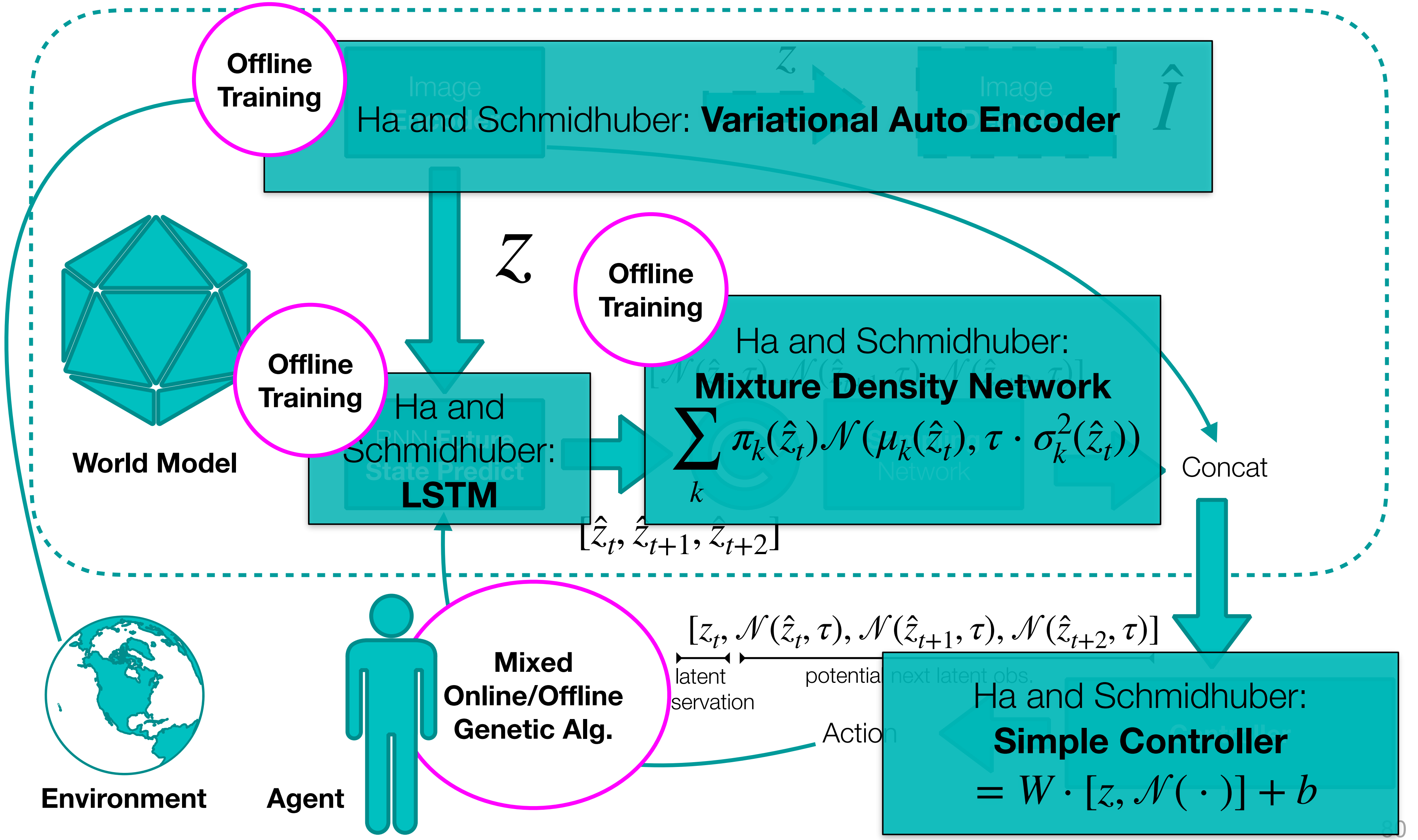
Reminder:

Maybe we should be more careful about the way we describe what an agent does… because they don't dream. That's fluff.

The Main Idea



Implementation

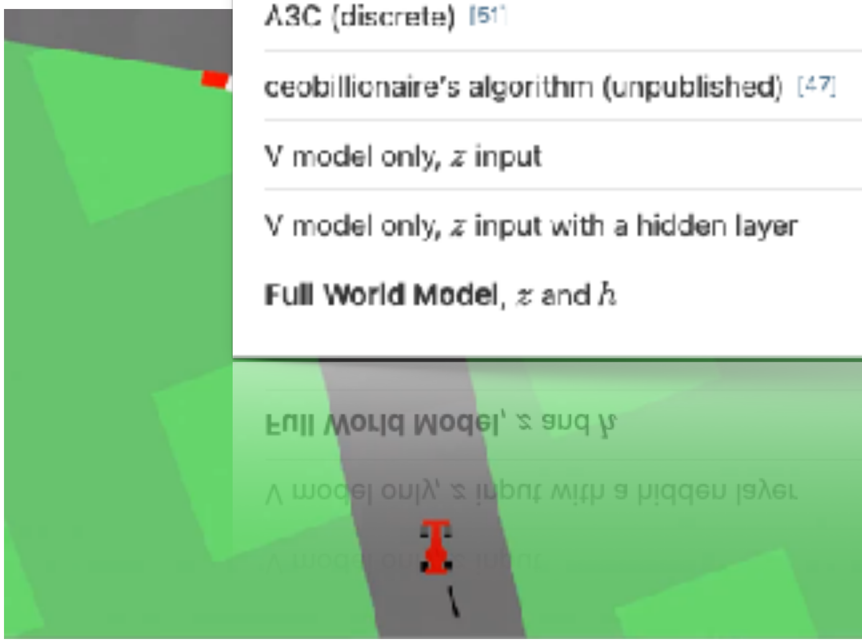


An Example, Racing

- Schmidhuber and Ha Methods:
  - Collect 10,000 rollouts from a random policy.
  - Train VAE (V) to encode each frame
  - Train
  - Evol
    cum

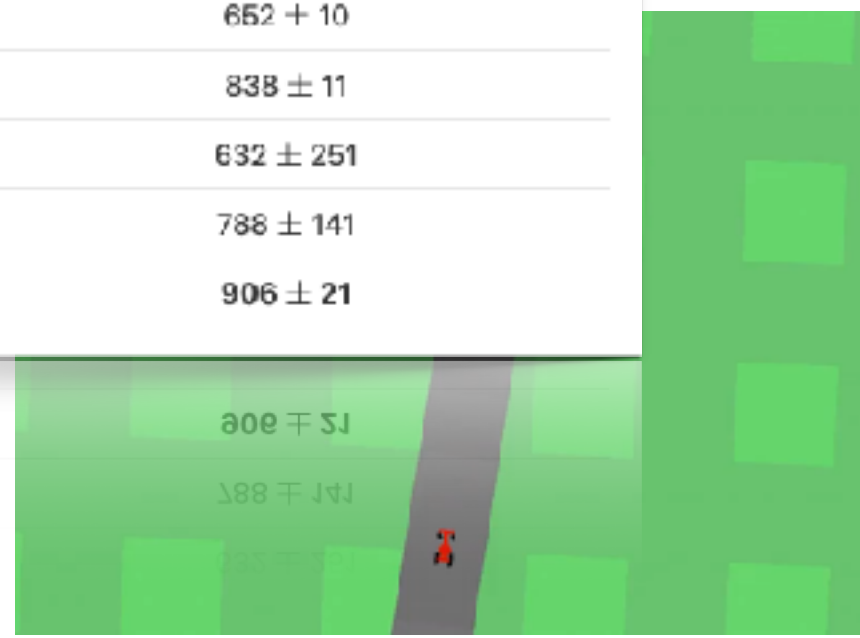
Model	Parameter Count
VAE	4,348,547
	422,368
	867

Method	Average Score over 100 Random Tracks
DQN [53]	343 ± 18
A3C (continuous) [52]	591 ± 45
A3C (discrete) [51]	652 ± 10
ceobillionaire's algorithm (unpublished) [47]	838 ± 11
V model only, z input	632 ± 251
V model only, z input with a hidden layer	788 ± 141
Full World Model, z and h	906 ± 21



Only use VAE Encoding

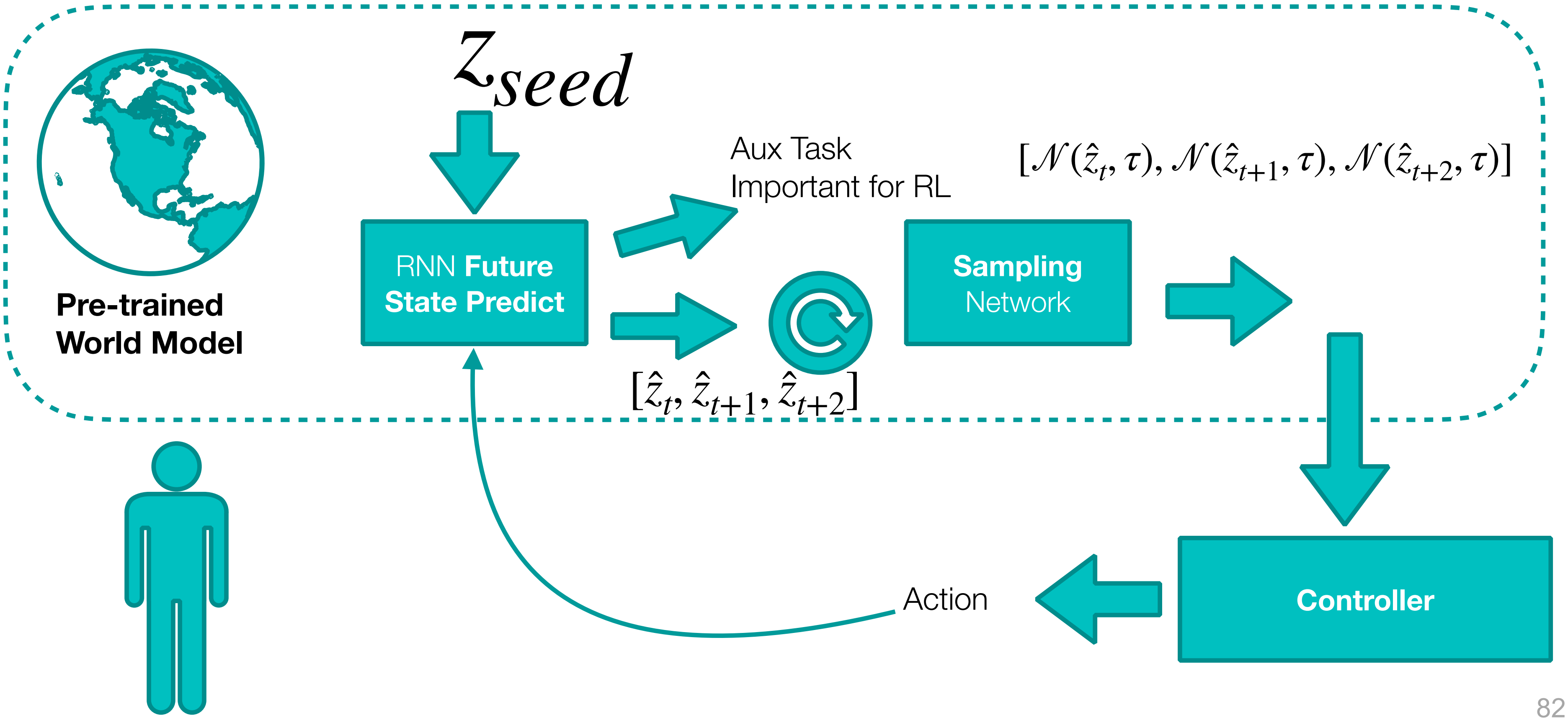
https://worldmodels.github.io



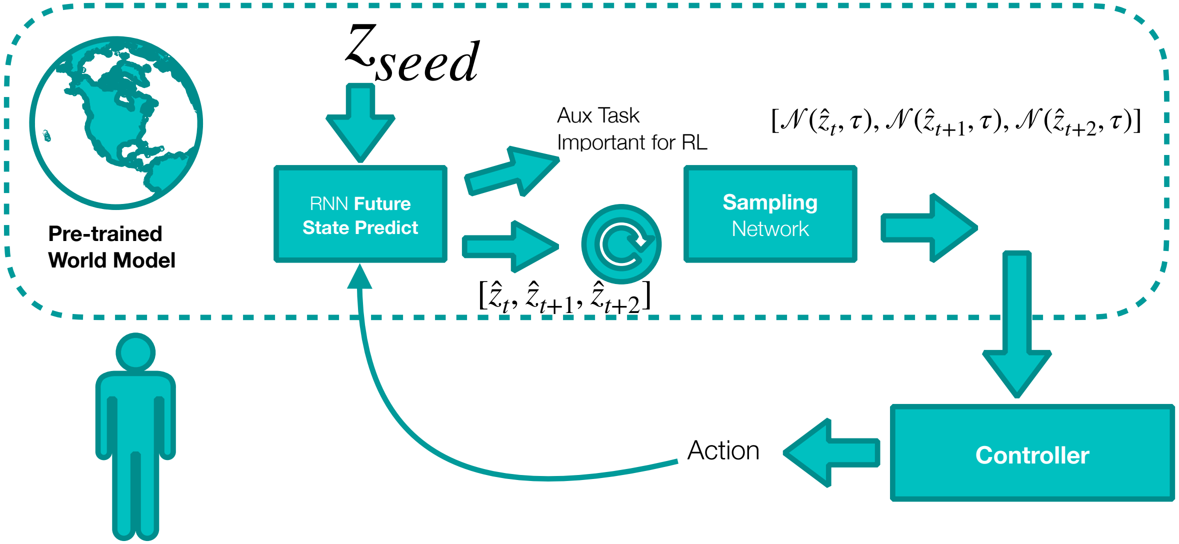
Full World Model

Can we learn without the environment?

- What if we sample from the world model to train our controller?



VizDoom Training Example



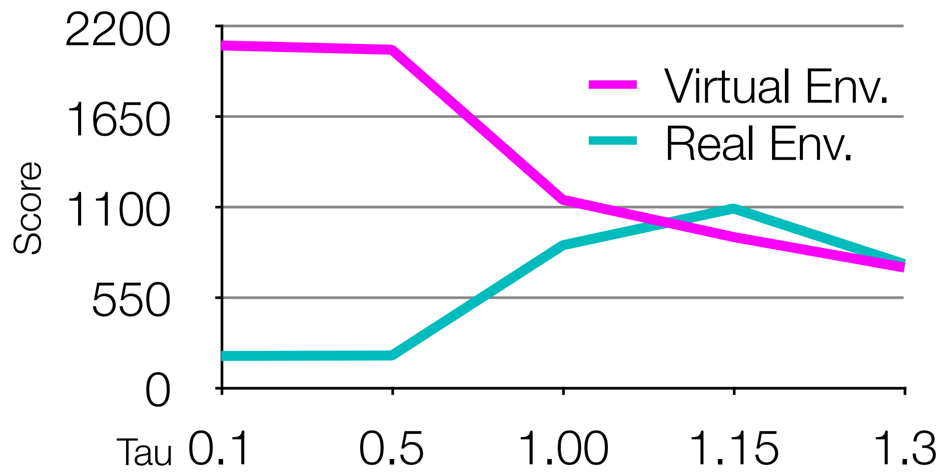
Model	Parameter Count
VAE	4,446,915
MDN-RNN	1,678,785
Controller	1,088

- Collect 10,000 rollouts from a random policy
- Train VAE (V) to encode each frame
- Train MDN-RNN to predict z and “if survived” in next frame
- Evolve Controller (C) to maximize the expected survival time inside the virtual environment.
- Continue training learned policy on actual environment (Gym)
- Call it training inside a “dream” (get attention of journalists)

Learned Policy

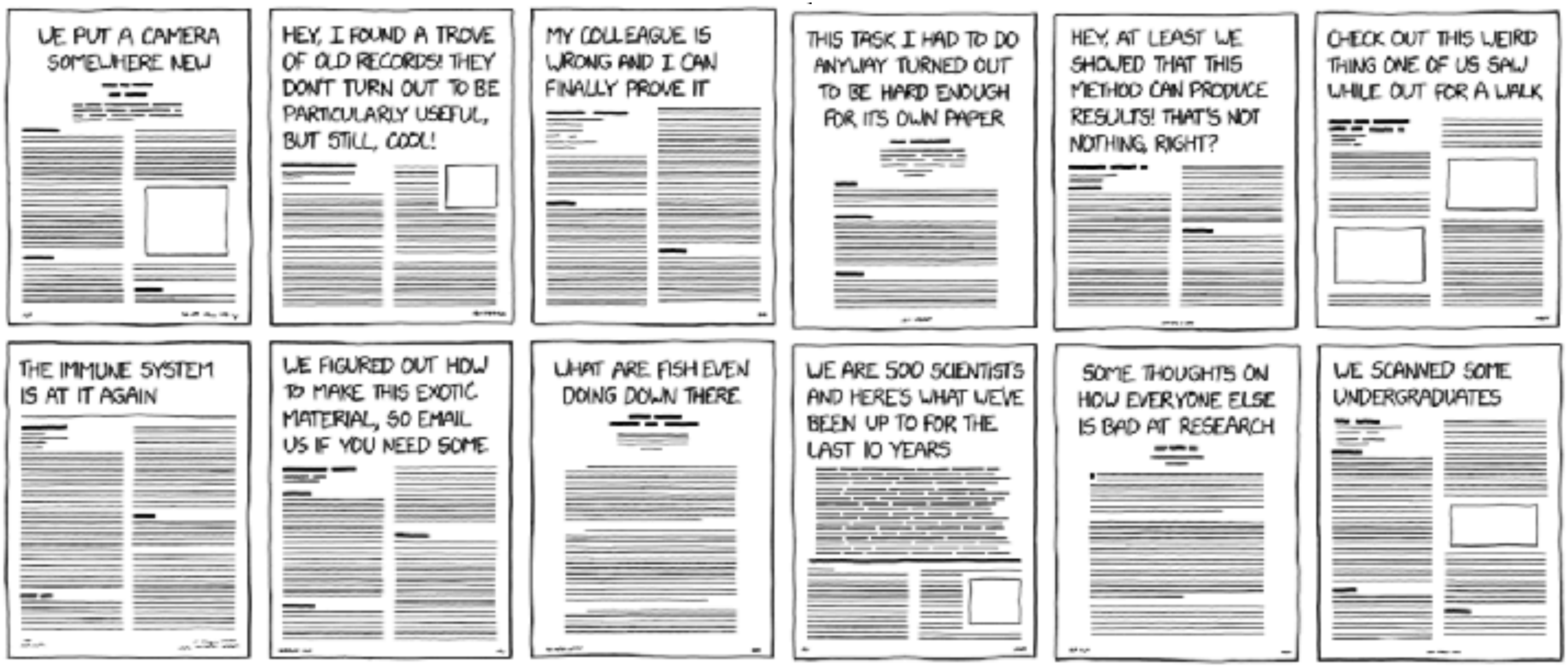
- Important to optimize the temperature control of the MDN

$$\sum_k \pi_k(\hat{z})\mathcal{N}(\mu_k(\hat{z}), \tau \cdot \sigma_k^2(\hat{z}))$$



Temperature	Score in Virtual Environment
0.10	2086 ± 140
0.50	2060 ± 277
1.00	1145 ± 690
1.15	918 ± 546
1.30	732 ± 269
Random Policy Baseline	N/A
Gym Leaderboard [34]	N/A

Types of Scientific Papers



Thanks for a great semester!!!

Please fill out the course evaluations!! (Now?)

Lecture Notes for

Neural Networks and Machine Learning

World Models and Course Retrospective

Next Time:

None!

Reading: Nope