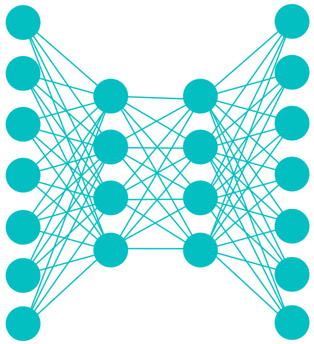
Lecture Notes for

Neural Networks and Machine Learning

Q-Learning

Course Retrospective

“World Models”



Logistics and Agenda

- Logistics
  - Final Paper Due at end of Finals (**May 7**)
  - This is last lecture!!
- Agenda
  - Deep Q-Learning
  - Class Retrospective
  - Any Remaining Time: World Models

Last Time

Frozen Lake

The setup of the lake is as follows: Observations space, integer, based on the square you selec
frozen spaces, and a goal. The reward only happens at the end, otherwise the reward is zero.

```
SFFF
FHFH
FFFH
HFFG
```

To encode this observation space, we will convert the integer value (1-16) into a one hot encod

The action space is defined as moving left(1), right(2), up(3), down(4), which will be the output o

Value Iteration (Value Based)

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

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

With infinite time and exploration, this update will
Converge to Optimal Policy

Value Iteration
Reinforcement Learning

M. Lapan Implementation for
and Frozen Lake

Follow Along:
08a_Basics_Of_Reinforcement_Learning.ipynb

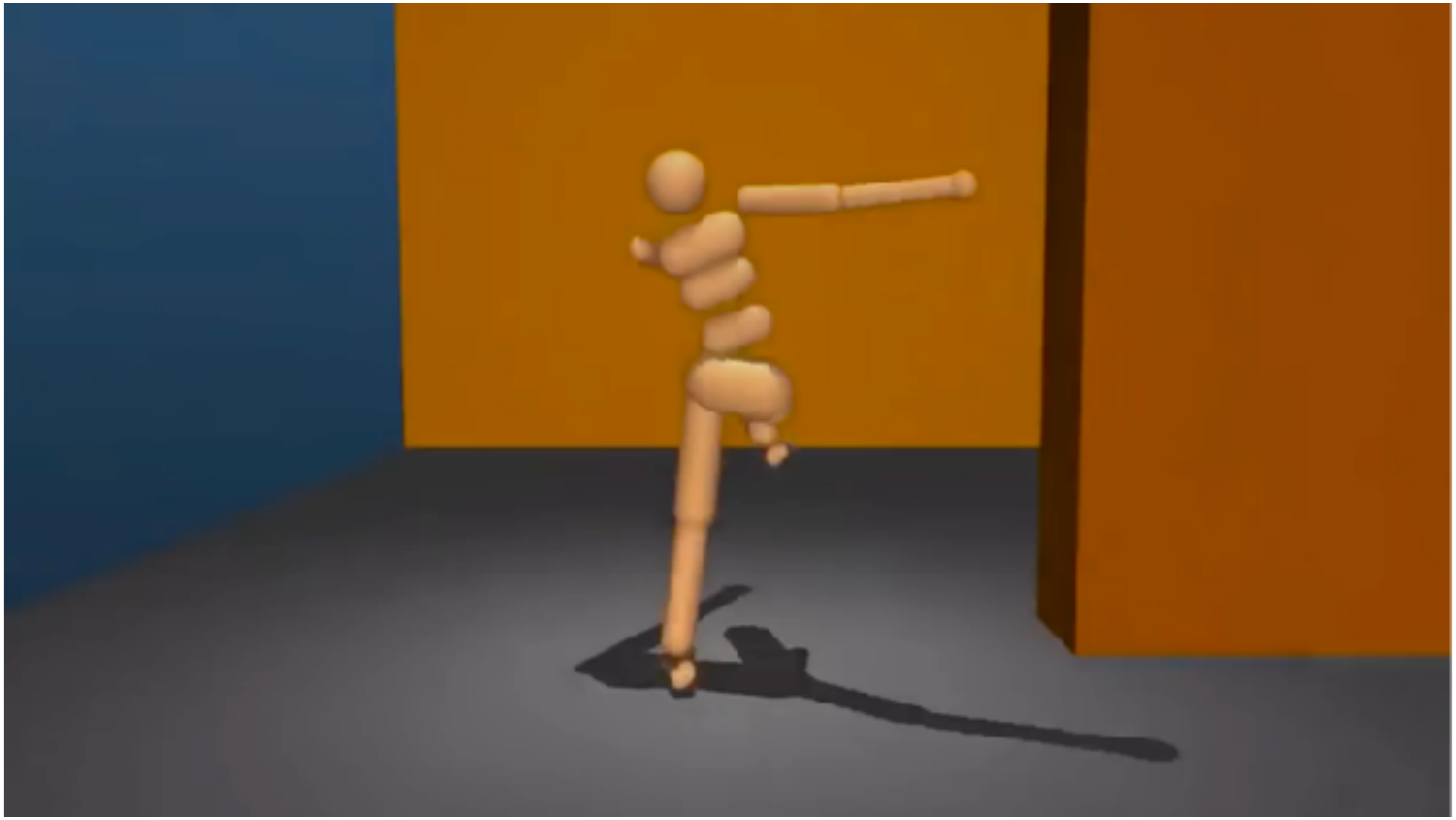
Review: Value Iteration and Q Learning

- **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

Deep Q-Learning



Q-Learning with a Neural Network

- Want to approximate $Q(s,a)$ when the state space is potentially large. Given s_t (could be continuous), we want the network to give us a row of actions from $Q(s,a)$ table that we can choose from:

[… other states…]

→ [$Q(s_t,a_1)$, $Q(s_t,a_2)$, $Q(s_t,a_3)$, … $Q(s_t,a_A)$] ←

[… other states…]

- How to train 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')

$$\mathscr{L} = \Big[\underbrace{Q(s,a)}_{\text{from current network params}} - [r_{s,a} + \gamma \underbrace{\max_{a' \in A} Q^*(s',a')}_{\text{from older network params (better stability)}}]\Big]^2$$

Periodically Update
Params of Q^* from Q

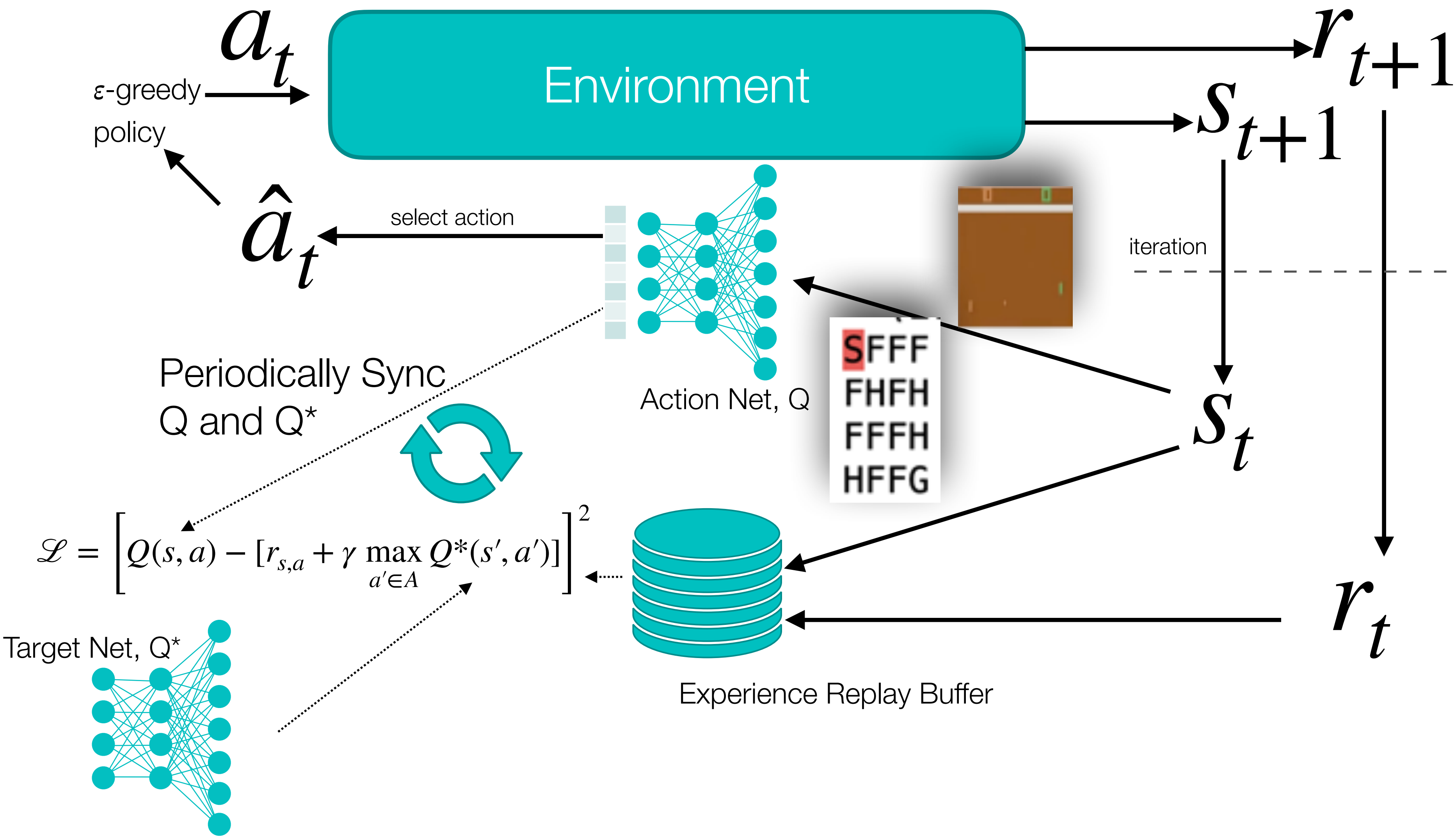
$$\mathscr{L} = \big[Q(s,a) - [r_{s,a}]\big]^2$$

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 again use **experience replay**
- **Update schedule**: make Q and Q^* same every N steps

Deep Q-Learning Overview



Deep Q-Learning Implementation

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)
```

ε-g

po

r_{t+1}

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

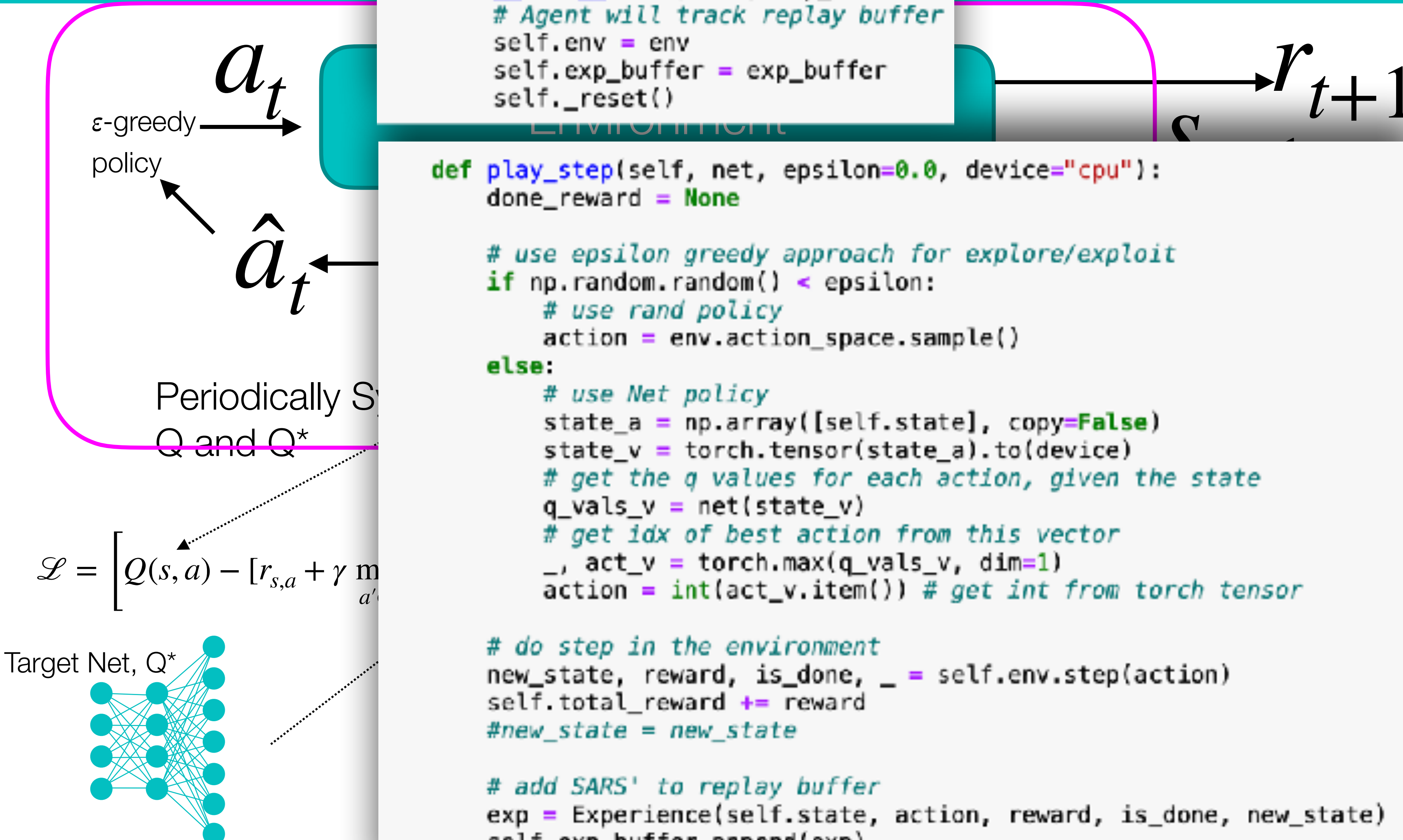
Target Net, Q*

Experience Replay Buffer

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
```

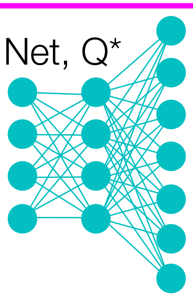
Deep Q-Learning, Implementation Details

```python
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
```

```python
# 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
#  because it is just observations that we want to be true
#  That is, we want to change the expected values output from
#  the net, not the observations calculation
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
```

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

Target Net, Q*



Experience Replay Buffer

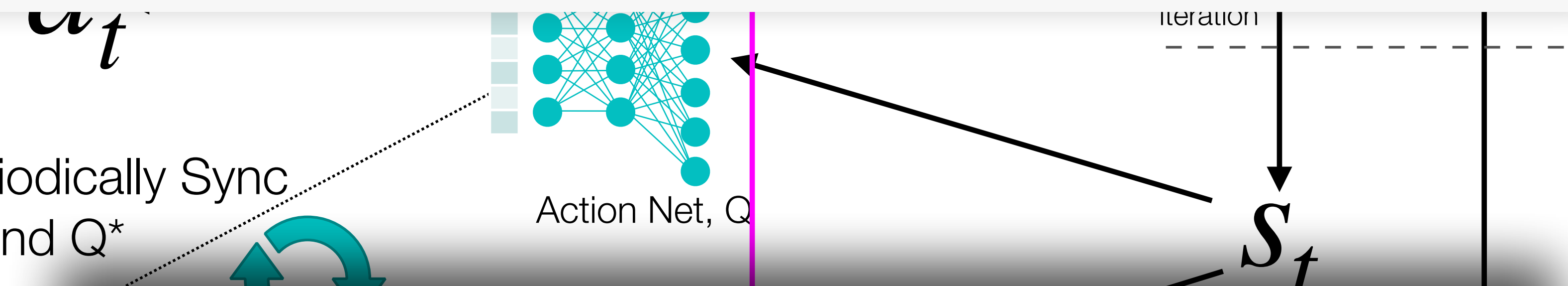
t

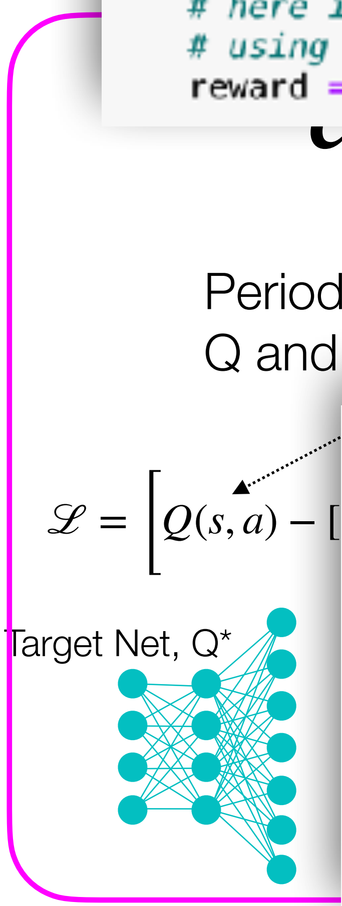
$$\mathscr{L} = \left[Q(s,a) - [r_{s,a}] \right]^2$$

if no next state (env is done)

Deep Q-Learning, Implementation Details

```python
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)
```

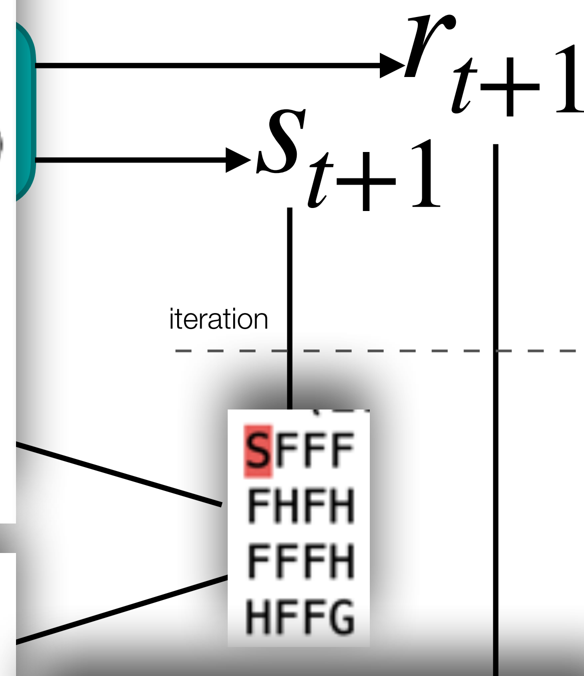





```python
# sync the networks every so often
if frame_idx % SYNC_TARGET_FRAMES == 0:
    # use current state dictionary of values to overwrite tgt_net
    tgt_net.load_state_dict(net.state_dict())

# 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__()
```

r_{t+1}

```
# 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
```

See `q_learning_utils.py`

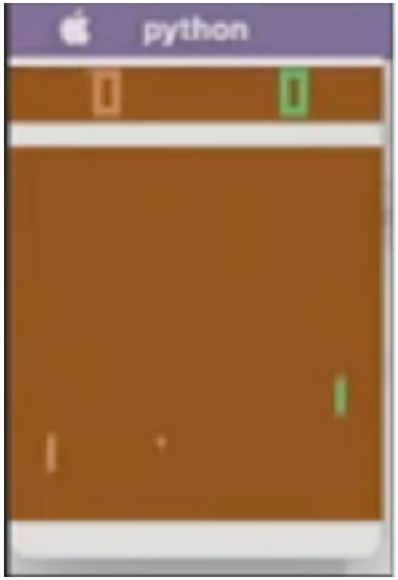
3 Days CPU Training

```
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)

Atari CPU



RL Agent

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.

If Time

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)

Follow Along:
`08a_Basics_Of_Reinforcement_Learning.ipynb`

Course Retrospective

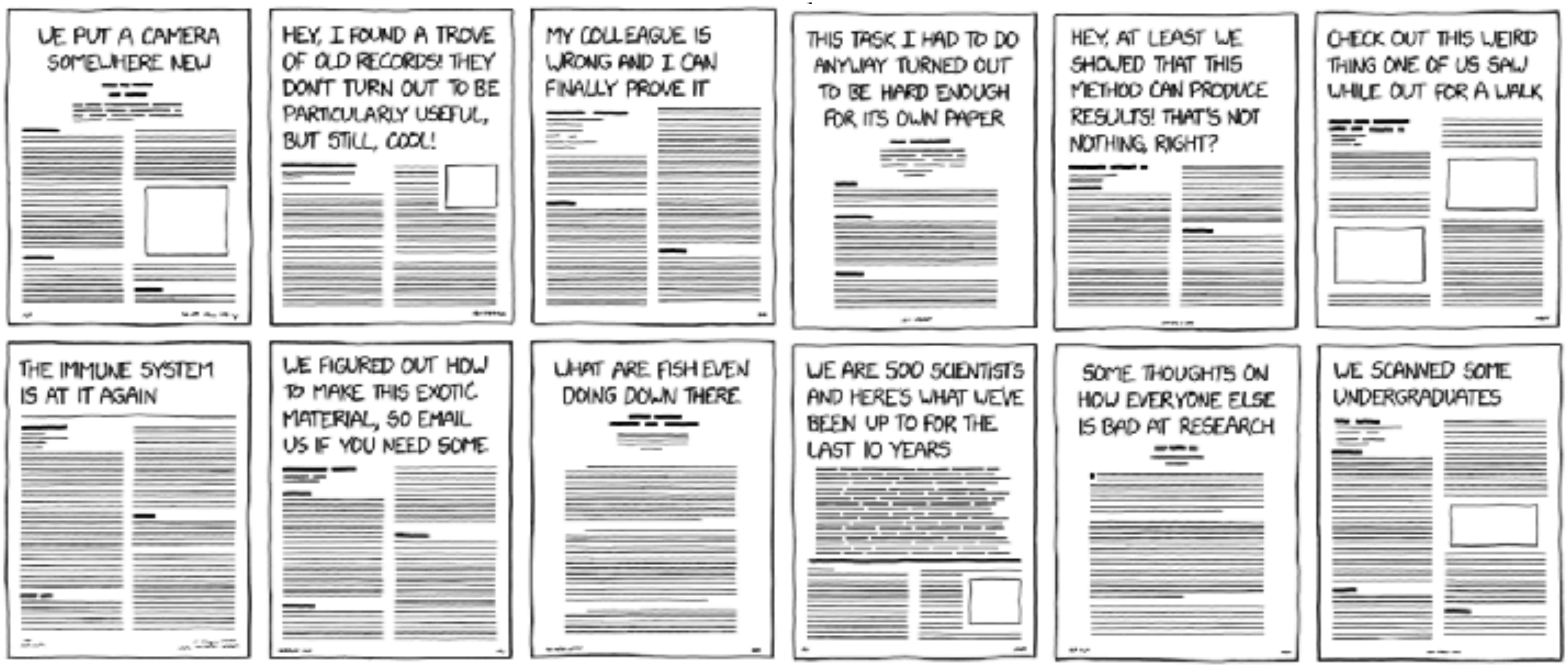
Day 1 of python: How can I learn python ?

Day 3 of python: machine learning engineer positions near me

Course Retrospective

- **Ethics**: Guidelines, ConceptNet NumberBatch
- **Transfer Learning**: Bottleneck Methods
- **X-Formers**: Simple, Large, Vision X-formers
- **Multi-task and Multi-modal**: Self-consistency
- **CNN Visualization**: Heatmaps, Grad-CAM, Circuits
- **CNN Fully Convolutional**: R-CNN, YOLO, Mask-RCNN, YOLACT, **Tracking**: DeepSORT, Trackformer
- **Stable Diffusion 3**
- **RL**: CE, Value Iteration, Q-Learning, Deep Q-Learning
- What was good, **bad**, **ugly**? What could be **changed**?
- Other Topics not covered:
  - **Style Transfer**: Gatys, FastStyle, WCT
  - **GANs**: Vanilla to Wasserstein to BigGAN (and others), AAEs
  - **RL**: Async Advantage Actor Critic, Proximal Policy Opt.

Types of Scientific Papers



Thanks for a great semester!!!

Please fill out the course evaluations!!