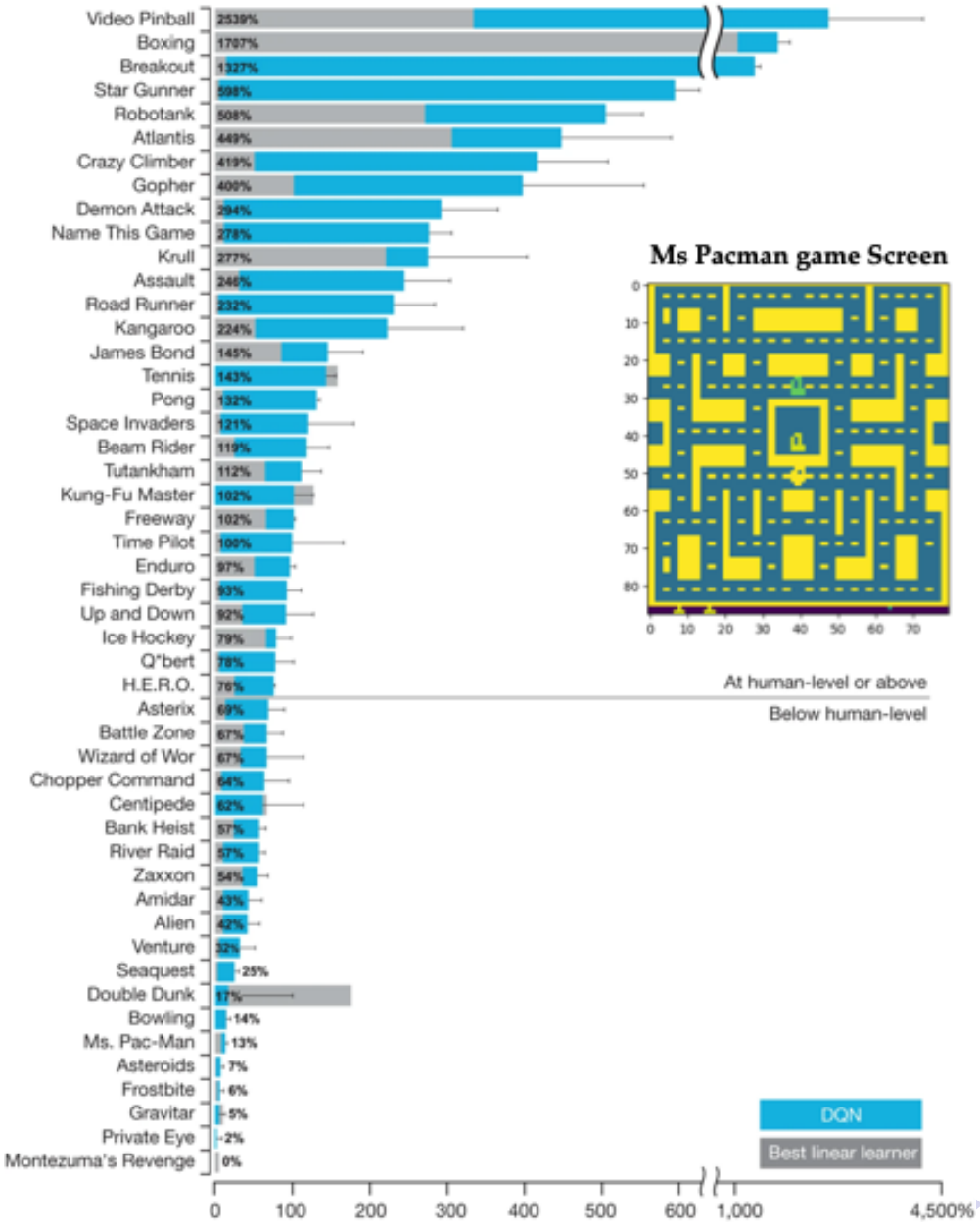
Introduction

- Neural networks were first applied to RL in 1995 for playing backgammon.
- The success of that early work relied on pre-engineered game features that simplified the learning problem.
- In 2013, a deep convolutional network was used to learn how to play the entire suite of Atari video games without any prior feature engineering.
- This model was called deep Q network or DQN agent. It took the video game screen as an input and output the actions to be taken. The model learned Q^* and the policy, π^*, was then determined directly from Q^*.
- DQN outperformed professional human game testers by a wide margin. This demonstration set off a wave of research in deep Reinforcement learning that has been used in a wide range of real-life applications; autonomous driving, datacenter cooling, traffic light control, healthcare, robotics, power management.

Deep Q Reinforcement Learning



Deep Q Reinforcement Learning

- The DQN agent is trained using past agent interactions with the environment that have been saved in a first-in first-out (FIFO) replay buffer.
- Each entry popped onto the buffer is a tuple

$$(s_k, a_k, r_{k+1}, s_{k+1})$$

that represents that environmental reward and state, r_{k+1} and s_{k+1}, returned to the DQN agent for taking action a_k when the environment is in state s_k.

- The DQN agent is then trained with a randomly selected batch of tuples in the replay buffer for a single step of the backpropagation algorithm updating the agent's weights.

Deep Q Reinforcement Learning

- The DQN agent takes the current state, s, and outputs the Q^* value for each action $a \in A$. So we have a regression problem.
- The loss the the MSE between the optimal Q^* and the estimate value, $\widehat{Q}$. The problem is that we don't know the target, Q^*.
- So we use Bellman's equation since

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

- We approximate the expectation by the sample mean evaluated over a minibatch drawn from the replay buffer and then use the estimated value $\widehat{Q}$ in the above equation.
- So the update we actually use is

$$L(w) = \frac{1}{M}\sum_{k=1}^{M}\left(r_{k+1} + \gamma \max_{a' \in A} \widehat{Q}_w(s_{k+1},a) - \widehat{Q}_w(s_k,a_k)\right)^2$$

where w represents the trainable parameters of the neural network

Deep Q Reinforcement Learning

We now show how to build a DQN agent object hat can be used in solving the FrozenLake problem. We initialize the `Agent` class object

```
class Agent:
    def __init__(self, state_size, action_size):
        self.memory = deque(maxlen=2500)
        self.learning_rate = 0.001
        self.epsilon = 1
        self.max_eps = 1
        self.min_eps = 0.01
        self.eps_decay = 0.001/3
        self.gamma = 0.9
        self.state_size = state_size
        self.action_size = action_size
        self.epsilon_lst=[]
        self.model = self.buildmodel()
```

Deep Q Reinforcement Learning

The `buildmodel` methods builds the DQN neural network.

```
def buildmodel(self):
     model = Sequential()
     model.add(Dense(32, input_dim=self.state_size, activation="relu"))
     model.add(Dense(32, activation = "relu")
     model.add(Dense(self.action_size, activation="linear"))
     model.compile(loss = "mse", optimizer = Adam(lr=self.learning_rate))
     return model

def action(self, state):
     if np.random.rand() > self.epsilon:
          return np.random.randint(0,4)
     return np.argmax(self.model.predict(state, verbose=0))

def pred(self, state):
     return np.argmax(self.model.predict(state, verbose=0))
```

Deep Q Reinforcement Learning

The replay buffer is implemented within the DQN Agent object

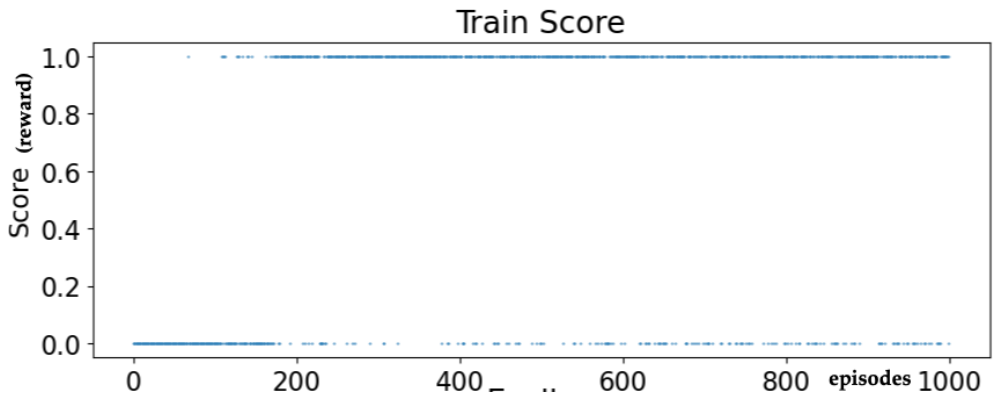
```
def add_memory(self, new_state, reward, done, state, action):
    self.memory.append((new_state, reward, done, state, action))
def replay(self, batch_size):
    minibatch = random.sample(self.memory, batch_size)
    for new_state, reward, done, state, action, in minibatch:
        target = reward
        if not done:
            target = reward +
            self.gamma*np.amax(self.model.predict(new_state,verbose=0))
        target_f = self.model.predict(state,verbose=0)
        target_f[0][action] = target
        self.model.fit(state, target_f, epochs=1, verbose=0)

    if self.epsilon > self.min_eps:
        self.epsilon = (self.max_eps - self.min_eps)*
            np.exp(-self.eps_decay*episode)+ self.min_eps

    self.epsilon_lst.append(self.epsilon)
```

Deep Q Reinforcement Learning

```
reward_lst=[]
for episode in range(train_episodes):
    state= env.reset()[0]
    state_arr=np.zeros(state_size)
    state_arr[state] = 1
    state= np.reshape(state_arr, [1, state_size])
    reward = 0
    done = False
    for t in range(max_steps):
        # env.render()
        action = agent.action(state)
        new_state, reward, done, truncate, info = env.step(action)
        new_state_arr = np.zeros(state_size)
        new_state_arr[new_state] = 1
        new_state = np.reshape(new_state_arr, [1, state_size])
        agent.add_memory(new_state, reward, done, state, action)
        state= new_state
        if done:
            print(f'Episode: {episode:4}/{train_episodes} and step: {t:4}.
                      Eps: {float(agent.epsilon):.2}, reward {reward}')
            break
    reward_lst.append(reward)
    if len(agent.memory)> batch_size:
        agent.replay(batch_size)
```

Deep Q Reinforcement Learning

- We tested the model for 100 episodes and found that the learned policy was successful 100% of the time. This is to be expected here because we used the "deterministic" FrozenLake environment.
- The training score is the "reward" received at the end of each episode. What this shows is that the likelihood of receiving a reward of 1.0 (i.e. getting to the desired destination) becomes more likelihood the longer we train. This graph therefore shows that our training procedure is working well.



Policy Gradient Methods

- All of the preceding RL algorithms (SARSA, Q-learning, DQN) were based on first learning the value function and then determining the optimal policy. These algorithms are therefore referred to as *value gradient* methods.
- Another approach to RL learns the policy directly. In this case the policy is written as $\pi(a : s, \theta)$ where θ is a set of parameters that we need to learn. Reinforcement learning algorithms that learn a model for the policy are called *policy gradient* methods.
- In this case we define a performance measure $J(\theta)$ for the policy model and then use gradient ascent to find those parameters θ that maximize that performance measure

$$\theta_{t+1} = \theta_t + \alpha \widehat{\nabla_\theta J(\theta_t)}$$

where $\widehat{\nabla_{\theta J(\theta_t)}}$ is a stochastic estimate whose expectation approximates the gradient of $J(\theta)$.

Policy Gradient methods

- One advantage that policy-gradient methods have over value-gradient methods is that the approximate policy automatically "explores" the state space.
- Another important advantage of policy gradient methods is that the policy function $\pi(a|s,\theta)$ may be much simpler than the value function model
- The policy function $\pi(a \mid s,\theta)$ can be parameterized in any way we wish, we simply need to make sure it is differentiable with respect to its parameters.
- If the action and state spaces are discrete and not too large, then one can then we can form a set of parameterized numerical preferences, $h(s,a,\theta) \in \mathbb{R}$ for each state-action pair.

Policy Gradient Methods

- The actions with the highest preference in each state are given the highest probabilities of being selected

$$\pi(a \mid s, \theta) = \frac{e^{h(s,a,\theta)}}{\sum_b e^{h(s,b,\theta)}}$$

- These preferences, $h(s, a, \theta)$, can be parameterized in many ways. They many be computed by a deep neural network, or they could simply be linear functions,

$$h(s, a, \theta) = \theta^T \mathbf{x}(s, a)$$

of a predefined set of feature vectors $\mathbf{x}(s, a)$.