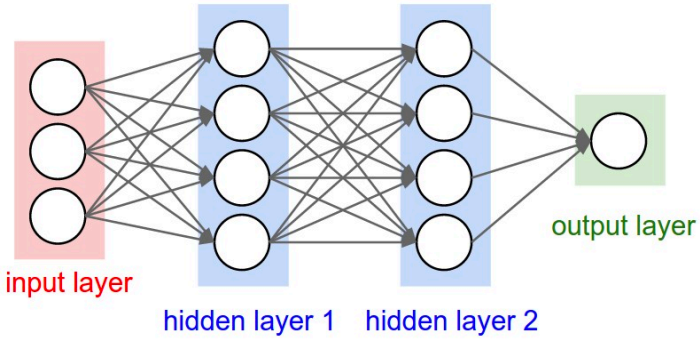
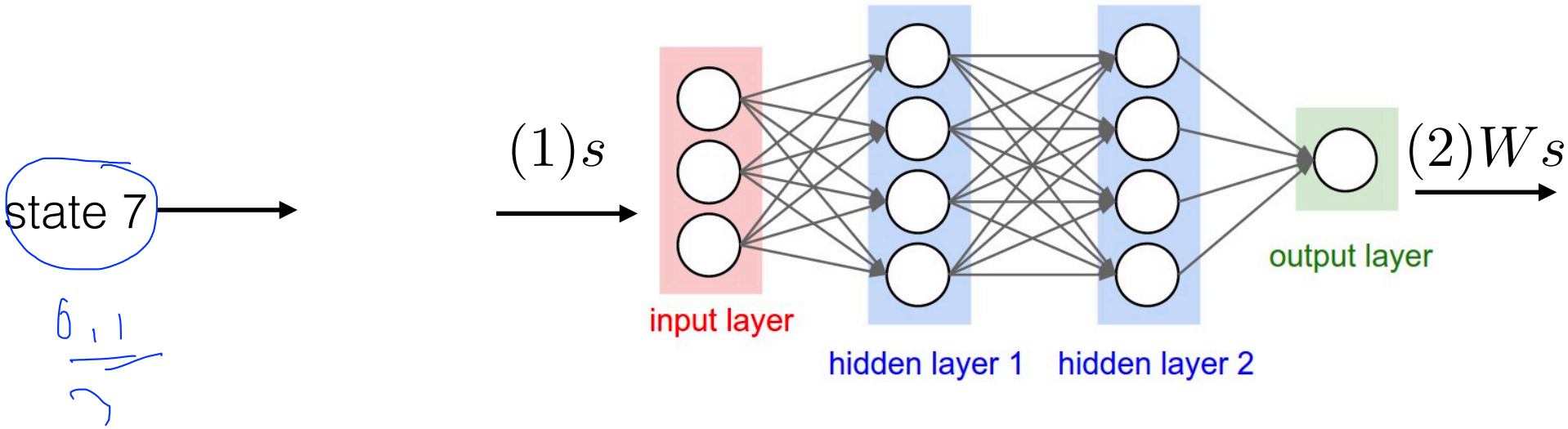


Lab 6-1: Q Network

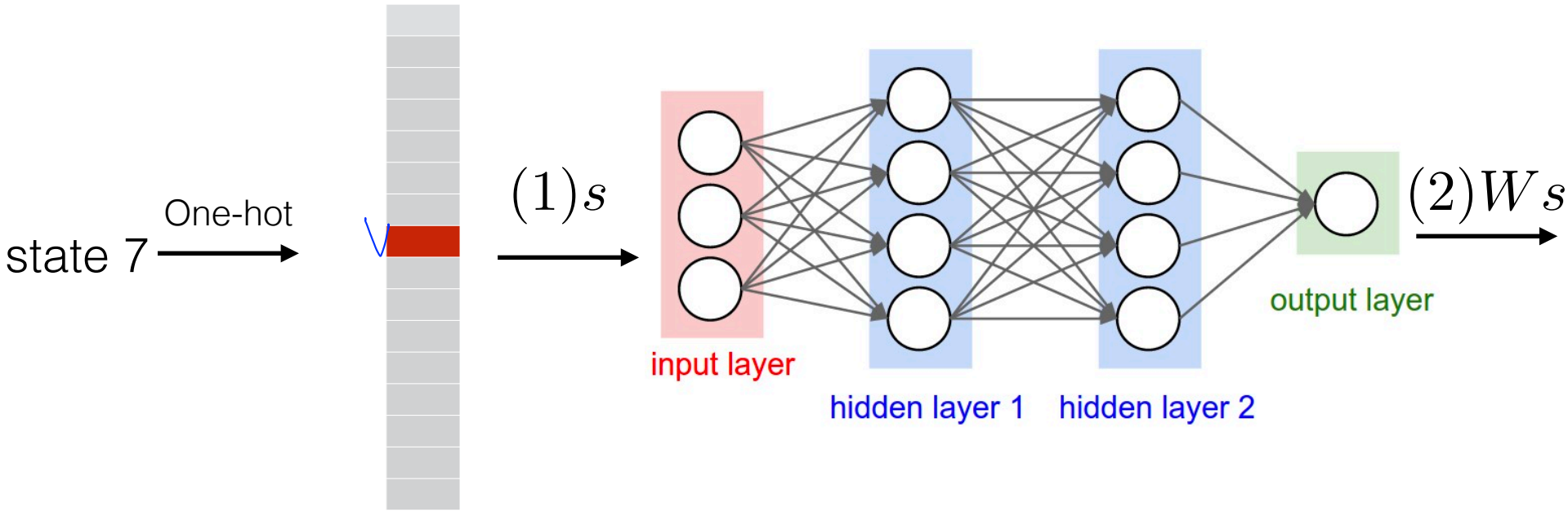
Reinforcement Learning with TensorFlow&OpenAI Gym

Sung Kim <hunkim+ml@gmail.com>

State(0~15) as input



State(0~15) as input

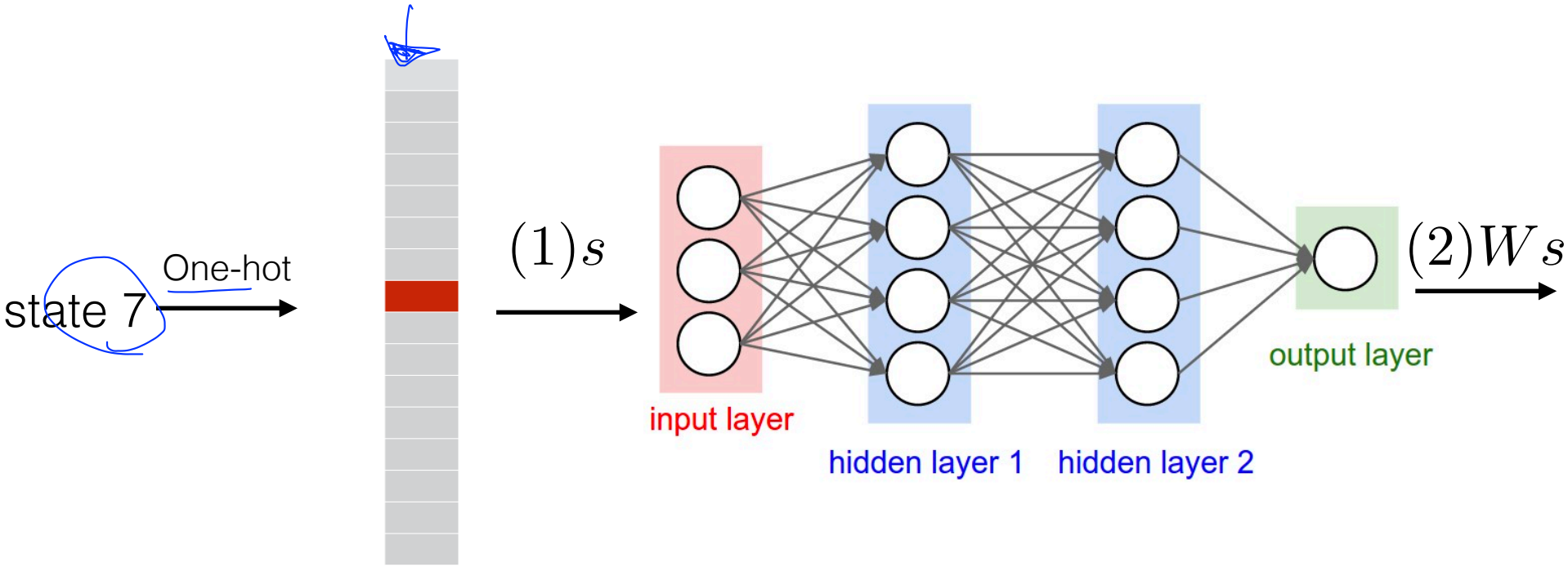


np.identify

state: np.identity(16)[s1:s1 + 1]

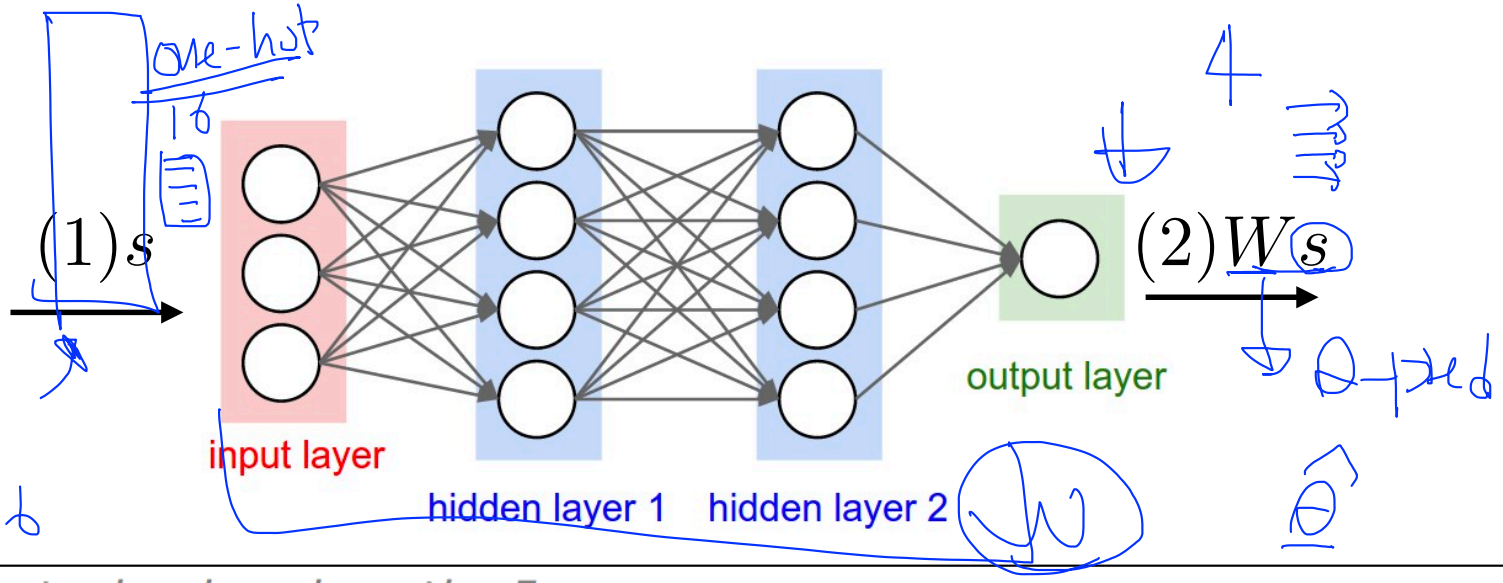
```
In [13]: import numpy as np

In [14]: print(np.identity(16)[0:1])
[[ 1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]]

In [15]: print(np.eye(16)[10:11])
[[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.]]

In [16]: print(np.identity(16))
[[ 1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.]]
```

State (0~15) as input




```
def one_hot(x):
    return np.identity(16)[x:x + 1]
```

Q-Network training (Network construction)

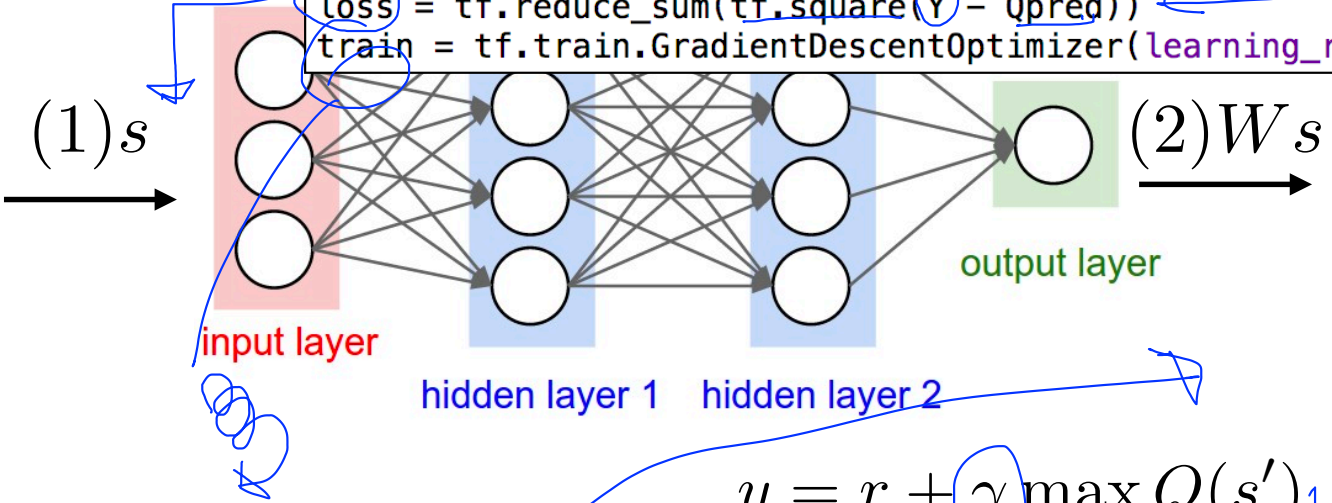



```
# Input and output size based on the Env
input_size = env.observation_space.n
output_size = env.action_space.n

# These lines establish the feed-forward part of the network used to choose actions
X = tf.placeholder(shape=[1,input_size],dtype=tf.float32) # state input
W = tf.Variable(tf.random_uniform([input_size, output_size],0,0.01))  # weight
Qpred = tf.matmul(X, W)  # Out Q prediction
```

Q-Network training (linear regression)

$$cost(W) = (Ws - y)^2$$

```
Qpred = tf.matmul(X, W)  # Out Q prediction
Y = tf.placeholder(shape=[1, output_size],dtype=tf.float32)  # Y label

loss = tf.reduce_sum(tf.square(Y - Qpred))
train = tf.train.GradientDescentOptimizer(learning_rate=learning_rate).minimize(loss)
```



$$y = r + \gamma \max Q(s')$$

```
    Qs[0, a] = reward + dis * np.max(Qs1)

# Train our network using target (Y) and predicted Q (Qpred) values
sess.run(train, feed_dict={X: one_hot(s), Y: Qs})
```

Algorithm

Algorithm 1 Deep Q-learning

```
def one_hot(x):
    return np.identity(16)[x:x + 1]
```

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 ϵ 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 $\phi_{t+1} = \phi(s_{t+1})$

$$\text{Set } y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$$

    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Playing Atari with Deep Reinforcement Learning - University of Toronto by V Mnih et al.

Algorithm

Algorithm 1 Deep Q-learning

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 ϵ 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 $\phi_{t+1} = \phi(s_{t+1})$

```
# Choose an action by greedily (with e chance of random action) from the Q-network
Qs = sess.run(Qpred,feed_dict={X: one_hot(s)})
if np.random.rand(1) < e:
    a = env.action_space.sample()
else:
    a = np.argmax(Qs)
```

Playing Atari with Deep Reinforcement Learning - University of Toronto by V Mnih et al.

Y label and loss function

$$\text{Set } y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$$

Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3

```
if done:
    # Update Q, and no Qs+1, since it's a terminal state
    Qs[0, a] = reward
else:
    # Obtain the Q_s1 values by feeding the new state through our network
    Qs1 = sess.run(Qpred, feed_dict={X: one_hot(s1)})
    # Update Q
    Qs[0, a] = reward + dis * np.max(Qs1)
```

Playing Atari with Deep Reinforcement Learning - University of Toronto by V Mnih et al.

Code: Network and setup

```python
import gym
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

env = gym.make('FrozenLake-v0')

# Input and output size based on the Env
input_size = env.observation_space.n
output_size = env.action_space.n
learning_rate = 0.1

# These lines establish the feed-forward part of the network used to choose actions
X = tf.placeholder(shape=[1,input_size],dtype=tf.float32)  # state input
W = tf.Variable(tf.random_uniform([input_size, output_size],0,0.01))  # weight

Qpred = tf.matmul(X, W)  # Out Q prediction
Y = tf.placeholder(shape=[1, output_size],dtype=tf.float32)  # Y label

loss = tf.reduce_sum(tf.square(Y - Qpred))

train = tf.train.GradientDescentOptimizer(learning_rate=learning_rate).minimize(loss)

# Set Q-learning related parameters
dis = .99
num_episodes = 2000

# Create lists to contain total rewards and steps per episode
rList = []
```

```python
def one_hot(x):
    return np.identity(16)[x:x + 1]
```

gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$

Code: Training

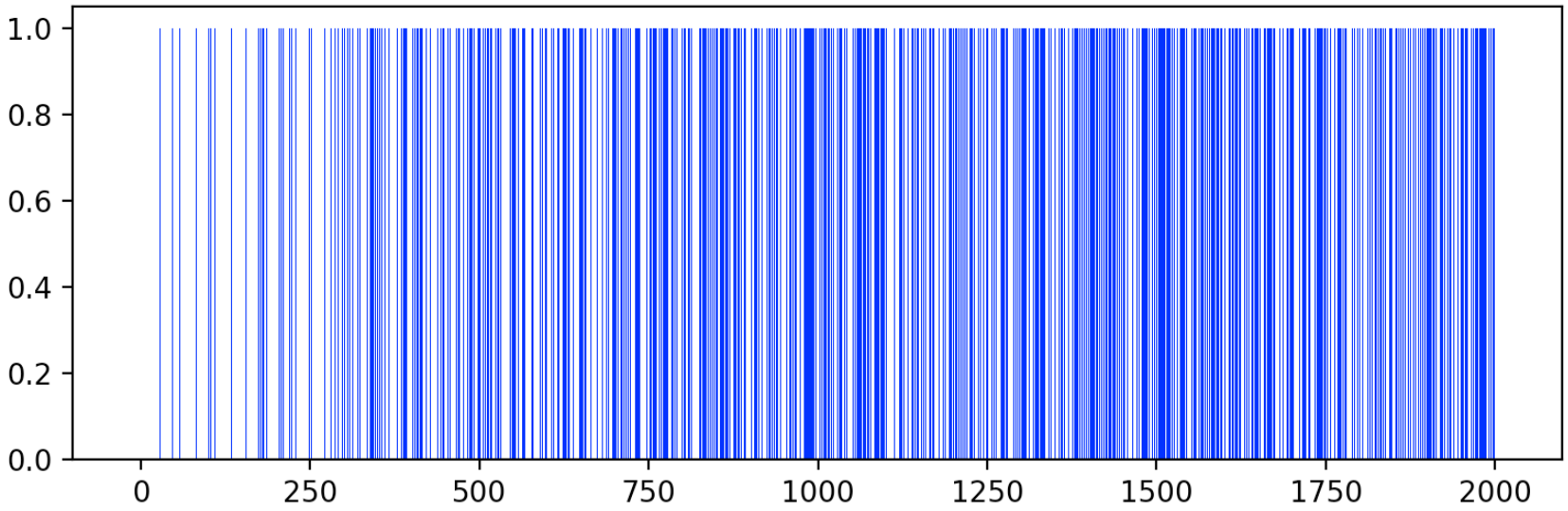
```
with tf.Session() as sess:
    sess.run(init)
    for i in range(num_episodes):
        # Reset environment and get first new observation
        s = env.reset()
        e = 1. / ((i / 50) + 10)
        rAll = 0
        done = False
        local_loss = []

        # The Q-Network training
        while not done:
            # Choose an action by greedily (with e chance of random action) from the Q-network
            Qs = sess.run(Qpred,feed_dict={X: one_hot(s)})
            if np.random.rand(1) < e:
                a = env.action_space.sample()
            else:
                a = np.argmax(Qs)

            # Get new state and reward from environment
            s1, reward, done, _ = env.step(a)
            if done:
                # Update Q, and no Qs+1, since it's a terminal state
                Qs[0, a] = reward
            else:
                # Obtain the Q_s1 values by feeding the new state through our network
                Qs1 = sess.run(Qpred, feed_dict={X: one_hot(s1)})
                # Update Q
                Qs[0, a] = reward + dis * np.max(Qs1)

            # Train our network using target (Y) and predicted Q (Qpred) values
            sess.run(train, feed_dict={X: one_hot(s), Y: Qs})

            rAll += reward
            s = s1
        rList.append(rAll)
```

Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$

Code: results

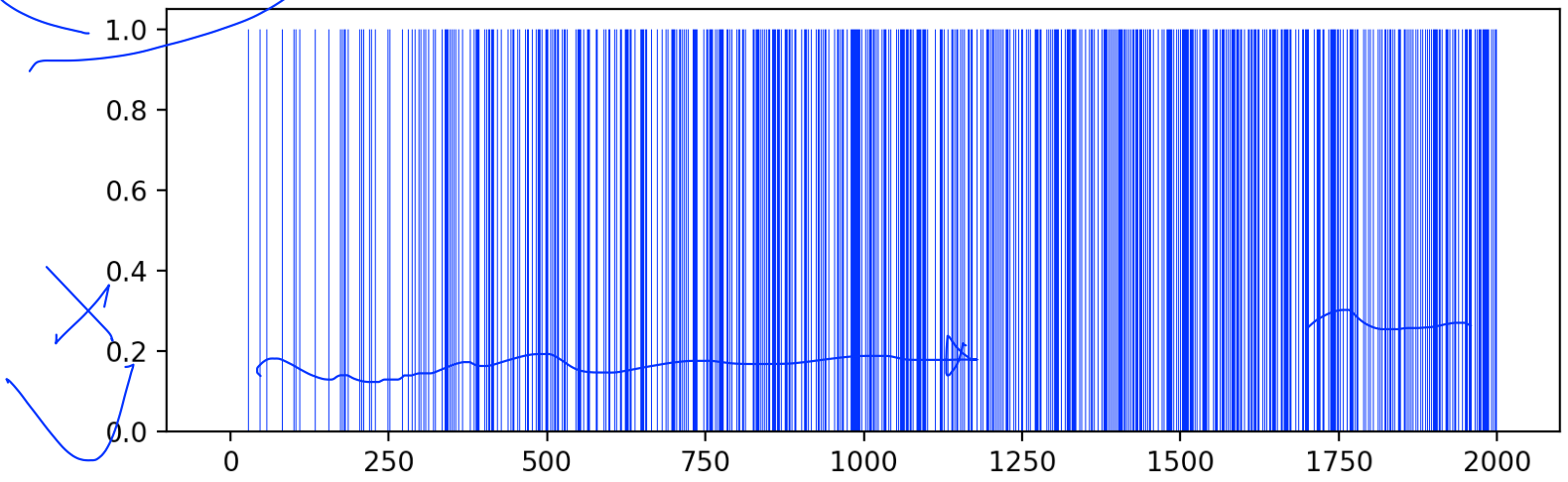
```
print("Percent of successful episodes: " + str(sum(rList)/num_episodes) + "%")
plt.bar(range(len(rList)), rList, color="blue")
plt.show()
```

Percent of successful episodes: 0.5195%

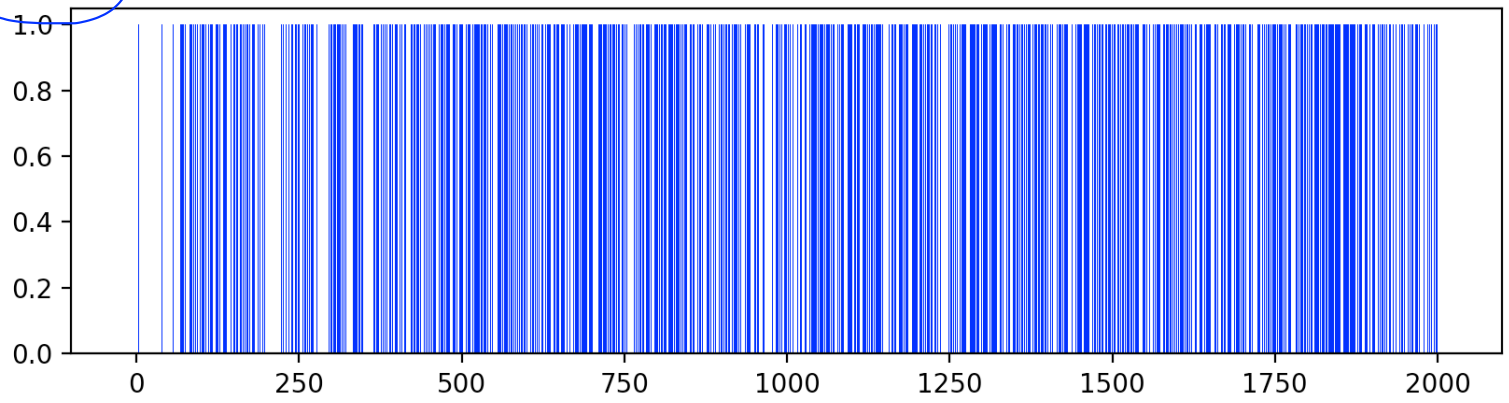


Q-Table VS Network

Q-network: 0.5195%



Q-table: 0.653



Array shape

```python
import gym
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

env = gym.make('FrozenLake-v0')

# Input and output size based on the Env
input_size = env.observation_space.n
output_size = env.action_space.n
learning_rate = 0.1

# These lines establish the feed-forward part of the network used to choose actions
X = tf.placeholder(shape=[1,input_size],dtype=tf.float32) # state input
W = tf.Variable(tf.random_uniform([input_size, output_size],0,0.01))  # weight

Qpred = tf.matmul(X, W)  # Out Q prediction
Y = tf.placeholder(shape=[1, output_size],dtype=tf.float32)  # Y label

loss = tf.reduce_sum(tf.square(Y - Qpred))

train = tf.train.GradientDescentOptimizer(learning_rate=learning_rate).minimize(loss)

# Set Q-learning related parameters
dis = .99
num_episodes = 2000

# Create lists to contain total rewards and steps per episode
rList = []
```

[[0, 2, …]]

1x16

[
[0,1,2,3],
[3,1,2,3],
[0,5,2,3],
]

16x4

$[[a_1,a_2,a_3,a_4]]$

1x4

```
with tf.Session() as sess:
    sess.run(init)
    for i in range(num_episodes):
        # Reset environment and get first new observation
        s = env.reset()
        e = 1. / ((i / 50) + 10)
        rAll = 0
        done = False
        local_loss = []

        # The Q-Network training
        while not done:
            # Choose an action by greedily (with e chance of random action) from the Q-network
            Qs = sess.run(Qpred,feed_dict={X: one_hot(s)})
            if np.random.rand(1) < e:
                a = env.action_space.sample()
            else:
                a = np.argmax(Qs)

            # Get new state and reward from environment
            s1, reward, done, _ = env.step(a)
            if done:
                # Update Q, and no Qs+1, since it's a terminal state
                Qs[0, a] = reward
            else:
                # Obtain the Q_s1 values by feeding the new state through our network
                Qs1 = sess.run(Qpred, feed_dict={X: one_hot(s1)})
                # Update Q
                Qs[0, a] = reward + dis * np.max(Qs1)

            # Train our network using target (Y) and predicted Q (Qpred) values
            sess.run(train, feed_dict={X: one_hot(s), Y: Qs})

            rAll += reward
            s = s1
        rList.append(rAll)
```

Array Shape

Qs['0']

[[a_1,a_2,a_3,a_4]]

1x4

Exercise

- Too slow
  - Minibatch?
- A bit unstable?

Next

Lab: Q-network for cart pole