Introduction to Deep Reinforcement Learning

Trevor Barron

September 26, 2017

Outline

1. Q-Learning Review
2. Why Deep RL?
3. Deep Q-Learning
4. Other facets of Deep RL

Q-Learning Review

Q-Functions

A Q function represents the expected long-term value of taking action, a, in state, s, under policy, π, with discount γ,

$$Q^{\pi}(s, a) = \mathbb{E}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots | s, a].$$

Using the Bellman equation we can write this recursively as,

$$Q^{\pi}(s, a) = \mathbb{E}_{s', a'}[r + \gamma Q^{\pi}(s', a') | s, a].$$

Optimal Q-Functions

In general we wish to find the optimal Q function, denoted Q^*,

$$\begin{aligned} Q^*(s,a) &= \underbrace{\max_\pi Q^\pi(s,a)}_{\text{policy that maximizes the } Q \text{ value}} \\ &= \underbrace{Q^{\pi^*}(s,a)}_{Q \text{ value under optimal policy}} \end{aligned}$$

Optimal Q-Functions

In general we wish to find the optimal Q function, denoted Q^*,

$$\begin{aligned} Q^*(s,a) &= \underbrace{\max_\pi Q^\pi(s,a)}_{\text{policy that maximizes the } Q \text{ value}} \\ &= \underbrace{Q^{\pi^*}(s,a)}_{Q \text{ value under optimal policy}} \end{aligned}$$

After finding Q^* we can behave optimally by selecting actions with highest expected value,

$$\pi^* = \arg\max_a Q^*(s,a).$$

Optimal Q-Functions

In general we wish to find the optimal Q function, denoted Q^*,

$$
\begin{aligned}
Q^*(s,a) &= \underbrace{\max_{\pi} Q^{\pi}(s,a)}_{\text{policy that maximizes the } Q \text{ value}} \\
&= \underbrace{Q^{\pi^*}(s,a)}_{Q \text{ value under optimal policy}}
\end{aligned}
$$

After finding Q^* we can behave optimally by selecting actions with highest expected value,

$$\pi^* = \arg\max_{a} Q^*(s,a).$$

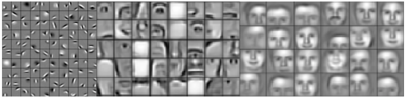
Note that Q^* takes a maximum at every step,

$$Q^{\pi}(s,a) = \mathbb{E}_{s'}[r + \gamma \max_{a'} Q^{\pi}(s',a')|s,a].$$

Why Deep RL?

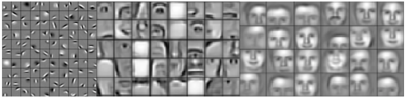
The best of both worlds

Neural Networks: Feature learning & non-linear approximation



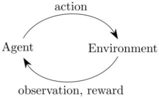
The best of both worlds

Neural Networks: Feature learning & non-linear approximation



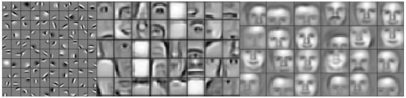
+

Reinforcement Learning: Temporal decision making



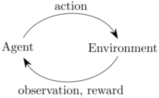
The best of both worlds

Neural Networks: Feature learning & non-linear approximation



+

Reinforcement Learning: Temporal decision making



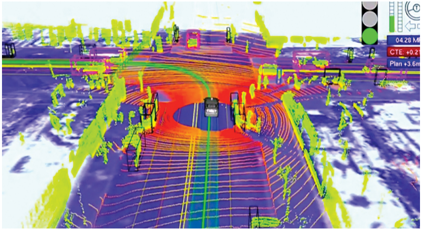
=

Artificial General Intelligence [1]

[1] According to DeepMind

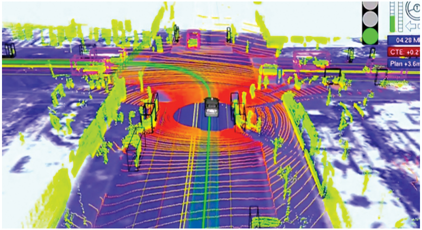
Motivation for Deep RL

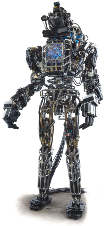
Standard RL methods work well for problems with reasonably small state spaces but real world problems tend to be high-dimensional.



Motivation for Deep RL

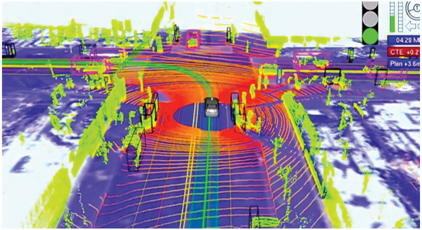
Standard RL methods work well for problems with reasonably small state spaces but real world problems tend to be high-dimensional.

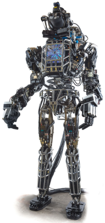


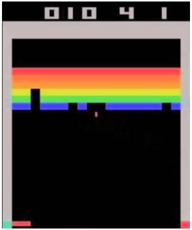


Motivation for Deep RL

Standard RL methods work well for problems with reasonably small state spaces but real world problems tend to be high-dimensional.



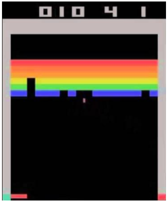




Tabular Q-Learning with Atari?

Let's try tabular Q-Learning with raw image data. What is the size of the table?

- What is a state? Assume states are 8-bit grayscale images of size (48, 48) and there are 2 actions.
- It's hard to enumerate valid states so let's just count all of them. That gives,



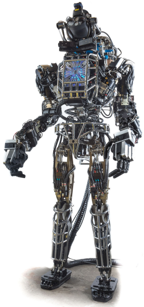
$$\text{table entries} = \underbrace{256^{48^2}}_{\text{states}} \cdot \underbrace{2}_{\text{actions}}$$
$$\approx 7.6 \times 10^{5548}$$

- Without incorporating domain knowledge, the space is enormous.

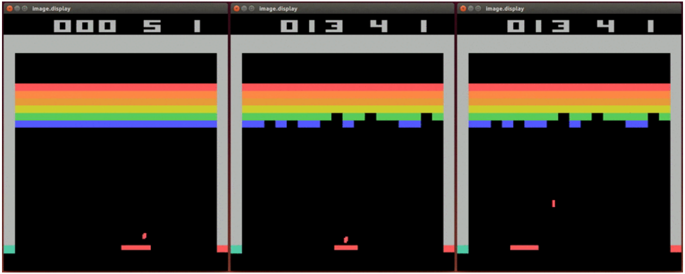
Still not convinced?

What if the state space is continuous? Atlas is a 28-DOF robot from Boston Dynamics.

Then, without function approximation,
the best we can do is discretize the space.



Can we make assumptions about any states?



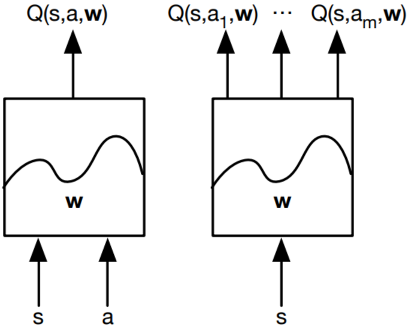
Are changes in the Q-estimates propagated between similar states.
In the tabular case? In the approximate case?

Deep Q-Learning

Use a neural network to approximate Q-values

Goal is to approximate the optimal Q function,

$$Q(s, a; \mathbf{w}) \approx Q^*(s, a)$$



The objective function

$$\mathcal{L} = \left(\overbrace{\underbrace{r + \gamma \max_{a'} Q(s', a'; \mathbf{w})}_{\text{next step estimate}} - \underbrace{Q(s, a; \mathbf{w})}_{\text{current step estimate}}}^{\text{temporal difference error}} \right)^2$$

Intuitively, the estimate of the optimal Q-value at time t should be equal to the reward received at that step plus the optimal value at $t + 1$.

$$\ldots, \underbrace{s_t, a_t}_{Q(s_t, a_t; \mathbf{w})}, r_{t+1}, \underbrace{s_{t+1}}_{\max_{a'} Q(s_{t+1}, a'; \mathbf{w})}, \ldots$$

Let's see this in practice

First, design the policy...

```
import tensorflow as tf
import tensorflow.contrib.layers as layers
```

Let’s see this in practice

First, design the policy...

```
import tensorflow as tf
import tensorflow.contrib.layers as layers

observation = tf.placeholder(tf.float32, [None,
 ↪  *env.observation_space.shape])
```

Let's see this in practice

First, design the policy...

```
import tensorflow as tf
import tensorflow.contrib.layers as layers

observation = tf.placeholder(tf.float32, [None,
 ↪   *env.observation_space.shape])

# add some layers of your choosing...
x = layers.fully_connected(x, 64, activation_fn=tf.nn.relu)
x = layers.fully_connected(x, 32, activation_fn=tf.nn.relu)
q_vals = layers.fully_connected(x, env.action_space.n,
 ↪   activation_fn=None])
```

Let's see this in practice

First, design the policy...

```
import tensorflow as tf
import tensorflow.contrib.layers as layers

observation = tf.placeholder(tf.float32, [None,
 ↪   *env.observation_space.shape])

# add some layers of your choosing...
x = layers.fully_connected(x, 64, activation_fn=tf.nn.relu)
x = layers.fully_connected(x, 32, activation_fn=tf.nn.relu)
q_vals = layers.fully_connected(x, env.action_space.n,
 ↪   activation_fn=None])

def q_values(obs):
  # assume TensorFlow session is available
  return sess.run(q_vals, feed_dict={observation: obs})
```

Let’s see this in practice

Second, define the update step...

```
action_input = tf.placeholder(tf.float32, [None, env.action_space.n])
# element wise multiplication between q values and actions
# to get q_value of action taken. Then sum the rows so we
# have a column vector s.t. each row is a q value for an action.
action_q_val = tf.reduce_sum(tf.multiply(q_vals, action_input),
 ↪  reduction_indices=1)
target_q_val = tf.placeholder(tf.float32, [None])
```

Let’s see this in practice

Second, define the update step...

```
action_input = tf.placeholder(tf.float32, [None, env.action_space.n])
# element wise multiplication between q values and actions
# to get q_value of action taken. Then sum the rows so we
# have a column vector s.t. each row is a q value for an action.
action_q_val = tf.reduce_sum(tf.multiply(q_vals, action_input),
 ↪  reduction_indices=1)
target_q_val = tf.placeholder(tf.float32, [None])

# mean squared error of q estimates
q_val_error = tf.reduce_mean(tf.squared_difference(target_q_val,
 ↪  action_q_val))
update_op = tf.train.RMSPropOptimizer(0.001).minimize(q_val_error)
```

Let's see this in practice

Then, let the agent interact with the environment...

```
import gym
import numpy as np
env = gym.make('Pong-v0')
done = False
obs = env.reset()
```

Let's see this in practice

Then, let the agent interact with the environment...

```
import gym
import numpy as np
env = gym.make('Pong-v0')
done = False
obs = env.reset()

while not done:
  # choose epsilon greedy action
  if np.random.random() < epsilon:
    act = env.action_space.sample()
  else:
    act = np.argmax(q_values(obs))
  next_obs, rew, done, info = env.step(act)
```

Let's see this in practice

Then, let the agent interact with the environment...

```
import gym
import numpy as np
env = gym.make('Pong-v0')
done = False
obs = env.reset()

while not done:
  # choose epsilon greedy action
  if np.random.random() < epsilon:
    act = env.action_space.sample()
  else:
    act = np.argmax(q_values(obs))
  next_obs, rew, done, info = env.step(act)

  # At this point we have everything we need to do an update!
  target = rew if done else rew + gamma * np.max(q_values(next_obs))
  sess.run(update_op, feed_dict={observation: obs, action_input: act,
   ↪  target_q_val: target})
```

Wait, not so fast!

It turns out that doesn't actually work very well. Why?

- High temporal correlation in updates. Remember we wish to approximate expected future reward, not reward over a given trajectory.

$$\ldots \underbrace{s_1, a_1, r_1, s_2}_{\text{update 1}}; \underbrace{s_2, a_2, r_2, s_3}_{\text{update 2}}; \underbrace{s_3, a_3, r_3, s_4}_{\text{update 3}}; \ldots$$

- Non-stationary target values. The estimated $Q(s, a; \mathbf{w})$ is changing during training so doesn't provide a consistent signal.

Tricks of the trade a.k.a. things you have to do to get good results

- Experience replay Store a buffer of previous experience and train on samples from the buffer.
- Target network Maintain two Q-networks where one is used only to estimated targets and is updated to match the main network periodically.

Experience Replay

Update Q network with uniformly drawn samples of past transitions avoiding temporal correlation.

Memory
s_1, a_1, r_1, s_2, t_1
s_2, a_2, r_2, s_3, t_2
$\dots$
$s_n, a_n, r_n, s_{n+1}, t_n$

Experience Replay

Update Q network with uniformly drawn samples of past transitions avoiding temporal correlation.

Memory

s_1, a_1, r_1, s_2, t_1
s_2, a_2, r_2, s_3, t_2
$\dots$
$s_n, a_n, r_n, s_{n+1}, t_n$

Should past experience be sampled randomly from the memory?

Experience Replay

Update Q network with uniformly drawn samples of past transitions avoiding temporal correlation.

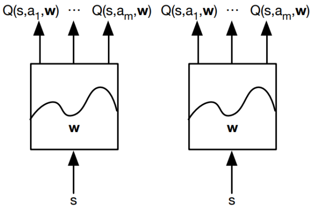
Memory
s_1, a_1, r_1, s_2, t_1
s_2, a_2, r_2, s_3, t_2
$\cdots$
$s_n, a_n, r_n, s_{n+1}, t_n$

Should past experience be sampled randomly from the memory?

Use TD-error to prioritize sampling from memory.

Target Networks

Maintain two copies of the Q-network...



Use target network to approximate target Q-values.

$$\mathcal{L} = \left(r + \gamma \max_{a'} \underbrace{Q^{-}(s', a'; \boldsymbol{w}^{-})}_{\text{target net estimate}} - \underbrace{Q(s, a; \boldsymbol{w})}_{\text{train net estimate}} \right)^2$$

Every n steps ($n \approx 5000$), copy the parameters w to w^{-}.

DQN Demo!

DQN vid.

Other facets of Deep RL

- On- versus off-policy learning
- Policy gradient methods

On- and off-policy RL

- Off-policy The data used to update the policy may come from samples generated by actions not taken from the policy. Example: Q-learning and ϵ-greedy.
- On-policy The data used to update the algorithm is generated solely from the policy. Example: policy gradients.

On- and off-policy RL

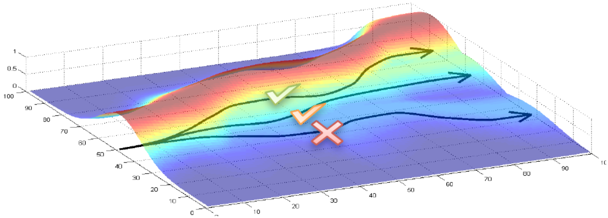
- Off-policy The data used to update the policy may come from samples generated by actions not taken from the policy. Example: Q-learning and ϵ-greedy.
- On-policy The data used to update the algorithm is generated solely from the policy. Example: policy gradients.

What would be pros and cons of each?

Policy Gradient Methods

Instead of estimating state values, directly estimate a distribution over actions to be taken.

$$a \sim \pi(a|s; \boldsymbol{w})$$



Climb the gradient of expected reward!

$$\mathcal{L} \approx -\frac{1}{n}\sum_{i=1}^{n} \underbrace{\nabla_{\boldsymbol{w}} \log P(\tau; \boldsymbol{w})}_{\text{gradient}} \qquad \underbrace{R(\tau)}_{\text{measure of “goodness”}}$$

Likelihood Ratio Policy Gradient

Derivation of policy gradient objective,

$$\mathcal{L} = -\sum_{\tau} P(\tau; \mathbf{w}) R(\tau)$$

Likelihood Ratio Policy Gradient

Derivation of policy gradient objective,

$$\mathcal{L} = -\sum_{\tau} P(\tau; \mathbf{w}) R(\tau)$$
$$\nabla \mathcal{L} = -\nabla_{\mathbf{w}} \sum_{\tau} P(\tau; \mathbf{w}) R(\tau)$$

Likelihood Ratio Policy Gradient

Derivation of policy gradient objective,

$$
\begin{aligned}
\mathcal{L} &= -\sum_{\tau} P(\tau; \mathbf{w}) R(\tau) \\
\nabla \mathcal{L} &= -\nabla_{\mathbf{w}} \sum_{\tau} P(\tau; \mathbf{w}) R(\tau) \\
&= -\sum_{\tau} \nabla_{\mathbf{w}} P(\tau; \mathbf{w}) R(\tau)
\end{aligned}
$$

Likelihood Ratio Policy Gradient

Derivation of policy gradient objective,

$$
\begin{aligned}
\mathcal{L} &= -\sum_{\tau} P(\tau; \mathbf{w}) R(\tau) \\
\nabla \mathcal{L} &= -\nabla_{\mathbf{w}} \sum_{\tau} P(\tau; \mathbf{w}) R(\tau) \\
&= -\sum_{\tau} \nabla_{\mathbf{w}} P(\tau; \mathbf{w}) R(\tau) \\
&= -\sum_{\tau} \frac{P(\tau; \mathbf{w})}{P(\tau; \mathbf{w})} \nabla_{\mathbf{w}} P(\tau; \mathbf{w}) R(\tau)
\end{aligned}
$$

Likelihood Ratio Policy Gradient

Derivation of policy gradient objective,

$$
\begin{aligned}
\mathcal{L} &= -\sum_{\tau} P(\tau; \mathbf{w}) R(\tau) \\
\nabla \mathcal{L} &= -\nabla_{\mathbf{w}} \sum_{\tau} P(\tau; \mathbf{w}) R(\tau) \\
&= -\sum_{\tau} \nabla_{\mathbf{w}} P(\tau; \mathbf{w}) R(\tau) \\
&= -\sum_{\tau} \frac{P(\tau; \mathbf{w})}{P(\tau; \mathbf{w})} \nabla_{\mathbf{w}} P(\tau; \mathbf{w}) R(\tau) \\
&= -\sum_{\tau} P(\tau; \mathbf{w}) \nabla_{\mathbf{w}} \log P(\tau; \mathbf{w}) R(\tau)
\end{aligned}
$$

Likelihood Ratio Policy Gradient

Derivation of policy gradient objective,

$$\begin{aligned}
\mathcal{L} &= -\sum_{\tau} P(\tau; \mathbf{w}) R(\tau) \\
\nabla \mathcal{L} &= -\nabla_{\mathbf{w}} \sum_{\tau} P(\tau; \mathbf{w}) R(\tau) \\
&= -\sum_{\tau} \nabla_{\mathbf{w}} P(\tau; \mathbf{w}) R(\tau) \\
&= -\sum_{\tau} \frac{P(\tau; \mathbf{w})}{P(\tau; \mathbf{w})} \nabla_{\mathbf{w}} P(\tau; \mathbf{w}) R(\tau) \\
&= -\sum_{\tau} P(\tau; \mathbf{w}) \nabla_{\mathbf{w}} \log P(\tau; \mathbf{w}) R(\tau) \\
&= -\mathbb{E}_{\tau} \nabla_{\mathbf{w}} \log P(\tau; \mathbf{w}) R(\tau)
\end{aligned}$$

Let's see this in practice

First, design a stochastic policy...

```
import tensorflow as tf
import tensorflow.contrib.layers as layers
observation = tf.placeholder(tf.float32, [None,
 ↪   *env.observation_space.shape])
# add some layers of your choosing...
x = layers.fully_connected(x, 64, activation_fn=tf.nn.relu)
x = layers.fully_connected(x, 32, activation_fn=None)
probs = tf.nn.softmax(x)
act = tf.multinomial(probs)
```

Let’s see this in practice

First, design a stochastic policy...

```
import tensorflow as tf
import tensorflow.contrib.layers as layers
observation = tf.placeholder(tf.float32, [None,
 ↪  *env.observation_space.shape])
# add some layers of your choosing...
x = layers.fully_connected(x, 64, activation_fn=tf.nn.relu)
x = layers.fully_connected(x, 32, activation_fn=None)
probs = tf.nn.softmax(x)
act = tf.multinomial(probs)

def action_probs(obs):
  # assume TensorFlow session is available
  return sess.run(probs, feed_dict={observation: obs})
```

Let's see this in practice

First, design a stochastic policy...

```
import tensorflow as tf
import tensorflow.contrib.layers as layers
observation = tf.placeholder(tf.float32, [None,
 ↪  *env.observation_space.shape])
# add some layers of your choosing...
x = layers.fully_connected(x, 64, activation_fn=tf.nn.relu)
x = layers.fully_connected(x, 32, activation_fn=None)
probs = tf.nn.softmax(x)
act = tf.multinomial(probs)

def action_probs(obs):
  # assume TensorFlow session is available
  return sess.run(probs, feed_dict={observation: obs})

def select_action(obs):
  # assume TensorFlow session is available
  return sess.run(act, feed_dict={observation: obs}
```

Let's see this in practice

Second, define the update step...

```
actions = tf.placeholder(tf.float32, shape=(None,
 ↪  env.action_space.n))
rewards = tf.placeholder(tf.float32, shape=(None))
```

Let's see this in practice

Second, define the update step...

```
actions = tf.placeholder(tf.float32, shape=(None,
  ↪  env.action_space.n))
rewards = tf.placeholder(tf.float32, shape=(None))

# As with Q-update, do element-wise multiplication
# to get per-action probability
action_probs = tf.reduce_sum(tf.multiply(probs, actions),
  ↪  reduction_indices=[1])
logprobs = tf.log(action_probs)
```

Let's see this in practice

Second, define the update step...

```
actions = tf.placeholder(tf.float32, shape=(None,
 ↪  env.action_space.n))
rewards = tf.placeholder(tf.float32, shape=(None))

# As with Q-update, do element-wise multiplication
# to get per-action probability
action_probs = tf.reduce_sum(tf.multiply(probs, actions),
 ↪  reduction_indices=[1])
logprobs = tf.log(action_probs)

# Note the negative since we are maximizing
L = -tf.reduce_sum(tf.multiply(logprobs, rewards))
update_op = tf.train.AdamOptimizer(learning_rate=0.01).minimize(L)
```

Let’s see this in practice

Then, let the agent interact with the environment...

```
import gym
import numpy as np
env = gym.make('Pong-v0')
done = False
obs = env.reset()
ep_states, ep_actions, ep_rewards = [], [], []
```

Let's see this in practice

Then, let the agent interact with the environment...

```
import gym
import numpy as np
env = gym.make('Pong-v0')
done = False
obs = env.reset()
ep_states, ep_actions, ep_rewards = [], [], []

while not done:
  ep_states.append(obs)
  action = sess.run(act, feed_dict={states: obs})
  obs, reward, done, info = env.step(action)
  ep_rewards.append(reward)
  ep_actions.append(action)
```

Let's see this in practice

Then, let the agent interact with the environment...

```
import gym
import numpy as np
env = gym.make('Pong-v0')
done = False
obs = env.reset()
ep_states, ep_actions, ep_rewards = [], [], []

while not done:
  ep_states.append(obs)
  action = sess.run(act, feed_dict={states: obs})
  obs, reward, done, info = env.step(action)
  ep_rewards.append(reward)
  ep_actions.append(action)

ep_rewards = discount_rewards(ep_rewards, gamma)
ep_actions = make_one_hot(ep_actions, env.action_space.n)
# after each episode, run update
sess.run(update, feed_dict={actions: ep_actions,
                            states: ep_states,
                            rewards: ep_rewards})
```

Some variations of policy gradient methods

- Asynchronous Advantage Actor-Critic Use an advantage estimate $A(s, a) = Q(s, a) - V(s, a)$ as proxy for "goodness" of action. A3C vid.
- Trust Region Policy Optimization Add a KL-divergence constraint on difference between old and new policies. TRPO vid.
- Proximal Policy Optimization Relax KL-divergence constraint with penalty clipped loss acting as a penalty. PPO vid.

Pointers to some recent work

- Combining on- and off-policy learning
  Combining policy gradient and Q-learning. O'Donoghue, et. al.
  Equivalence Between Policy Gradients and Soft Q-Learning. Schulman, et. al.
- Exploration strategies
  VIME: Variational Information Maximizing Exploration. Houthooft, et.al.
  Unifying Count-Based Exploration and Intrinsic Motivation. Bellemare, et. al.
- Meta RL
  Learning to reinforcement learn. Wang, et. al.
  RL^2: Fast Reinforcement Learning via Slow Reinforcement Learning. Duan, et. al.
- RL safety & adversarial methods
  Adversarial Attacks on Neural Network Policies. Huang, et. al.
  A Comprehensive Survey on Safe Reinforcement Learning, Garcia, Fernandez

Questions?