Faster Reinforcement Learning After Pretraining Deep Networks to Predict State Dynamics

Chuck Anderson, Minwoo Lee, and Daniel Elliott

Department of Computer Science
Colorado State University
Fort Collins, Colorado  USA

Best Paper Award, International Joint Conference on Neural Networks, July 2015
Killarney, Ireland
Main Idea

- Reinforcement learning problems are multi-step decision problems.
- Common solution is to learn to predict sum of future reinforcements.
- Reinforcement is often sparse and delayed.
- What else can be learned that might speed up learning from reinforcements?
  - State change observed on every step.
  - Features that predict state change might be useful for predicting reinforcement.
Overview

• Cart-pole swing up problem
• Reinforcement learning approach
  – Deep neural network as Q function
• Pretraining neural network to model the dynamics
  – Results (video)
• Summary and next steps
Cart-pole Swing-up Problem

Objective: Balance the pole
Objective: Maximize sum of reinforcement, $r_t$, over time.

$$r_t = \begin{cases} 
+1, & |\theta_t| < 45^\circ \\
-1, & |\theta_t| > 135^\circ \\
0, & \text{otherwise}
\end{cases}$$
Cart-pole Swing-up Problem

- Four state variables: position, velocity, angle, angular velocity
- Three actions: push left, push right, no push
- Elastic collisions at end of track
- Physics implemented with pybox2 (github.com/pybox2d)
- Animation implemented with pygame (www.pygame.org)
Cart-pole Swing-up Problem

Manual control
Overview

- Cart-pole swing up problem
- Reinforcement learning approach
  - Deep neural network as Q function
- Pretraining neural network to model the dynamics
  - Results (video)
- Summary and next steps
Reinforcement Learning Approach

State \( s_t \) contains position, velocity, angle and angular velocity at time \( t \).

Action \( a_t \) is -1, 0, or 1 for a push left, no push, or push right at time \( t \).

Solution requires good approximation, \( Q \), of sum of future reinforcement, given current state \( s_t \) and action \( a_t \).

\[
Q(s_t, a_t) \approx \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}
\]

Select best action by

\[
a_t = \arg\max_{a \in \{-1,0,1\}} Q(s_t, a)
\]
Reinforcement Learning Approach

Approximate $Q$ using a neural network parameterized by vector of weights $w$. 
Reinforcement Learning Approach

Approximate Q using a neural network parameterized by vector of weights $w$.

Update $w$ using gradient of squared error, $E$, between desired value of $Q$ and current estimate.

$$E_t(w) = \left( \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} - Q(s_t, a_t) \right)^2$$

$$E_t(w) = (r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))^2$$

$$\nabla_w E_t(w) = 2(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \nabla_w Q(s_t, a_t)$$

The gradient of $Q$ is calculated using the usual error backpropagation algorithm, where the error being backpropagated is the above temporal-difference error.
Reinforcement Learning Approach

Approximate Q using a neural network parameterized by vector of weights $w$.

Update $w$ using gradient of squared error between desired value of $Q$ and current estimate.

$$E_t(w) = \left( \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} - Q(s_t, a_t) \right)^2$$

$$E_t(w) = (r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))^2$$

$$\nabla_w E_t(w) = 2(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \nabla_w Q(s_t, a_t)$$

The gradient of $Q$ is calculated using the usual error backpropagation algorithm, where the error being backpropagated is the above temporal-difference error.
Reinforcement Learning Approach

A mini-batch of state, action, reinforcement, next state and action, samples are collected.

1) Start cart-pole in state $s_t$.
2) Choose action $a_t$ using $\epsilon$-greedy policy, with $\epsilon = 0.1$
3) Calculate $Q(s_t, a_t)$.
4) Apply $a$ to cart-pole simulation to get next state $s_{t+1}$. Dynamics integrated for 0.25 seconds.
5) Assign reinforcement $r_{t+1}$.
6) Choose next action $a_{t+1}$.
7) Calculate $Q(s_{t+1}, a_{t+1})$.

Repeat Steps 2 – 7 for 2000 steps.

Gradient-descent applied for fixed number of iterations for each mini-batch. We used Möller's Scaled Conjugate-Gradient algorithm.

Train for 100 mini-batches.
Deep Q Network

After training, evaluate by:

• set $\epsilon = 0$
• use trained Q network to choose greedy action
• start from 5 different starting states
• run for 2,000 steps each
• calculate average reinforcement, $R$, over all 10,000 steps.

Repeat train and evaluation phases 50 times with different initial weight values, and for different size networks.

1, 2, 3, 4, or 5 hidden layers
5, 10 or 20 units in each layer
Deep Q Network
Average of 50 Runs
(90 % confidence intervals)
Overview

• Cart-pole swing up problem
• Reinforcement learning approach
  – Deep neural network as Q function
• Pretraining neural network to model the dynamics
  – Results (video)
• Summary and next steps
Deep Q Network

\[ Q(s,a) \]
Deep Q Network - Pretraining

Procedure:

- Collect samples of state, action, and next state, using random actions.

- Train Q network to minimize mean squared error in predicted state change (scaled conjugate gradient) ignoring Q value output.

Now continue on to reinforcement learning procedure, with weights starting at values determined from pretraining.
Deep Q Network – Pretraining

During pretraining

\[ Q(s,a) \]

\[ \Delta s \]
Deep Q Network - Pretraining
Deep Q Network - Pretraining

During reinforcement learning

$Q(s,a)$
Deep Q Network
RL Faster with Pretraining

Tme to achieve 0.3 mean R:

No pretraining
95 batches
211 minutes

With pretraining
45 batches
100 minutes
Twice as fast!

20-20-20 networks
Average of 50 runs
Deep Q Network - Results

Number of Pretraining Samples vs Average Evaluation R

- 5 layers:
  - [5, 5]
  - [5, 5, 5]
  - [5, 5, 5, 5]
- 10 layers:
  - [10, 10]
  - [10, 10, 10]
  - [10, 10, 10, 10]
- 20 layers:
  - [20, 20]
  - [20, 20, 20]
  - [20, 20, 20, 20]
Deep Q Network - Results
Deep Q Network - Training

First minute
Deep Q Network - Training

After 10 minutes
Deep Q Network - Training

After 50 minutes
Deep Q Network - Training

After 100 minutes
Deep Q Network - Training

After 200 minutes
Deep Q Network – Testing
\[ \varepsilon = 0 \]
Deep Q Network – Testing

$\epsilon = 0$
Deep Q Network – Testing

$\varepsilon = 0$
Deep Q Network - Results

It appears that pretraining to predict state dynamics helps. Are there other possible explanations?

Maybe the benefit is caused by starting hidden layer weights at larger values.

One way to test this hypothesis is by pretraining to random state change values.
Deep Q Network - Results

Pretraining does more than just initialize weights to larger values. But, what's going on here?
Overview

- Cart-pole swing up problem
- Reinforcement learning approach
  - Deep neural network as Q function
- Pretraining neural network to model the dynamics
  - Results (video)
- Summary and next steps
Summary

• Deep neural network trained by Scaled Conjugate Gradient solves cart-pole swing up problem.

• Pretraining by learning to predict state changes reduces reinforcement learning time by half.

• Larger neural networks helped more by pretraining.
Next Steps

• Examine learned features in hidden layers.
• Can the two phases, pretraining and reinforcement learning, be combined into one phase?
  – Preliminary results suggest yes
• Model of state dynamics used to develop useful representation in hidden layers. Other uses?
  – Create virtual interactions
  – Look ahead
• Harder problems
  – Visual input to represent state
  – Octopus arm
Thank you

www.cs.colostate.edu/~anderson
Deep Q Network - Results
Qnet Training in “Python”

# StateAction is N x 5    R is N x 1    Qnext is N x 1

Target = R + gamma * Qnext

def objectiveF(w):
    unpack(w)
    OutputByLayer = forwardPass(StateAction)
    return 0.5 * np.mean((Target - OutputByLayer[-1])**2)

def gradF(w):
    unpack(w)
    OutputByLayer = forwardPass(StateAction)
    nSamples = StateAction.shape[0]
    delta = - (Target - OutputByLayer[-1]) / nSamples
    dWs = backwardPass(delta, OutputLayer)
    return pack(dWs)

# Apply your favorite gradient descent algorithm
scg.scg(pack(Ws), objectiveF, gradF, nIterations)