Continuous Deep Q-Learning ~~with Model-based Acceleration~~

Sebastian Mueller

June 6, 2018

Motivation

- Many real world tasks are continuous
- Model-free RL: + no feature engineering
  - high sample complexity
- Model-based RL: + more efficient
  - model limits performance of policy
- Goal: Combine both advantages
  - Derive continous variant of Q-Learning
  - Decrease sample complexity

Motivation

Common approaches in continuous domains:

- policy gradient descend
- actor-critic-methods

$\Rightarrow$ high sample complexity
What about Q-Learning?

Motivation

What about Q-Learning?

- off-policy algorithm
- only one optimization goal
- for discrete domains

Normalized Advantage Function (NAF)

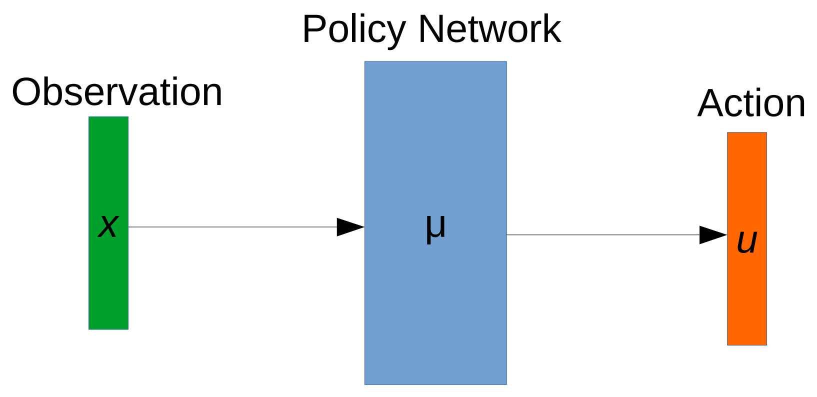
- Classical Q-Learning:
  $Q(x_t, u_t) = Q(x_t, u_t) + \alpha[R_{t+1} + \gamma \max_a Q(x_{t+1}, a) - Q(x_t, u_t)]$
- Decomposing Q (Baird III (1993), Advantage Updating):
  $Q(x_t, u_t) = \underbrace{A(x_t, u_t)}_{advantage-term} + \underbrace{V(x_t)}_{state-value-term}$
  with $A(x_t, u_t) = Q(x_t, u_t) - V(x_t)$
- Normalized Advantage Function:
  $Q(x, u|\theta^Q) = A(x, u|\theta^A) + V(x|\theta^V)$
  $A(x, u|\theta^A) = -\frac{1}{2}(u - \underbrace{\mu(x|\theta^\mu)}_{policy})^T \underbrace{\mathbf{P}(x|\theta^P)}_{positive-definite} (u - \mu(x|\theta^\mu))$

Continuous Q-Learning with NAF

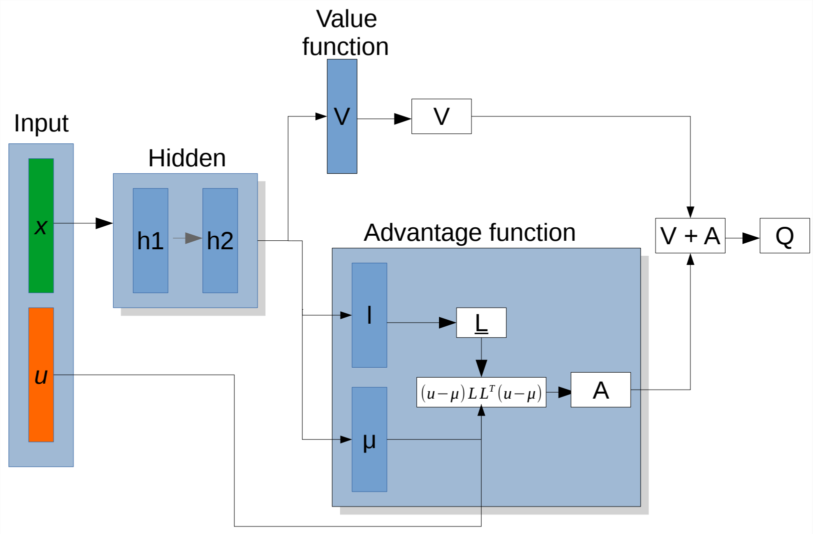
Algorithm 1 Continuous Q-Learning with NAF

Randomly initialize normalized Q network $Q(\boldsymbol{x}, \boldsymbol{u}|\theta^Q)$.
Initialize target network Q' with weight $\theta^{Q'} \leftarrow \theta^Q$.
Initialize replay buffer $R \leftarrow \emptyset$.
for episode=1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state $\boldsymbol{x}_1 \sim p(\boldsymbol{x}_1)$
  for t=1, T **do**
    Select action $\boldsymbol{u}_t = \mu(\boldsymbol{x}_t|\theta^\mu) + \mathcal{N}_t$
    Execute $\boldsymbol{u}_t$ and observe r_t and $\boldsymbol{x}_{t+1}$
    Store transition $(\boldsymbol{x}_t, \boldsymbol{u}_t, r_t, \boldsymbol{x}_{t+1})$ in R
    for iteration=1, I **do**
      Sample a random minibatch of m transitions from R
      Set $y_i = r_i + \gamma V'(\boldsymbol{x}_{i+1}|\theta^{Q'})$
      Update θ^Q by minimizing the loss: $L = \frac{1}{N}\sum_i(y_i - Q(\boldsymbol{x}_i, \boldsymbol{u}_i|\theta^Q))^2$
      Update the target network: $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
    end for
  end for
end for

Architecture: Choosing an action



Architecture: Computing the Q-Value



Constructing the hidden layers [network.py]

```python
with tf.name_scope('hidden'):
    if use_seperate_networks:
        logger.info("Creating seperate networks for v, l, and mu")

        for scope in ['v', 'l', 'mu']:
            with tf.variable_scope(scope):
                if use_batch_norm:
                    h = batch_norm(x, is_training=is_train)
                else:
                    h = x

                for idx, hidden_dim in enumerate(hidden_dims):
                    h = fc(h, hidden_dim, is_train, hidden_w, weight_reg=w_reg,
                           activation_fn=hidden_fn, use_batch_norm=use_batch_norm, scope='hid%d' % idx)
                hid_outs[scope] = h
    else:
        logger.info("Creating shared networks for v, l, and mu")

        if use_batch_norm:
            h = batch_norm(x, is_training=is_train)
        else:
            h = x

        # produces 2 h's, connecting them in tf: x->h0->h1: x->hidden
        for idx, hidden_dim in enumerate(hidden_dims):
            h = fc(h, hidden_dim, is_train, hidden_w, weight_reg=w_reg,
                   activation_fn=hidden_fn, use_batch_norm=use_batch_norm, scope='hid%d' % idx)
            pass
        # V,l, mu get their input from the same object in memory: hidden->{V, l, mu}
        hid_outs['v'], hid_outs['l'], hid_outs['mu'] = h, h, h
```

Computing the Q Value [network.py]

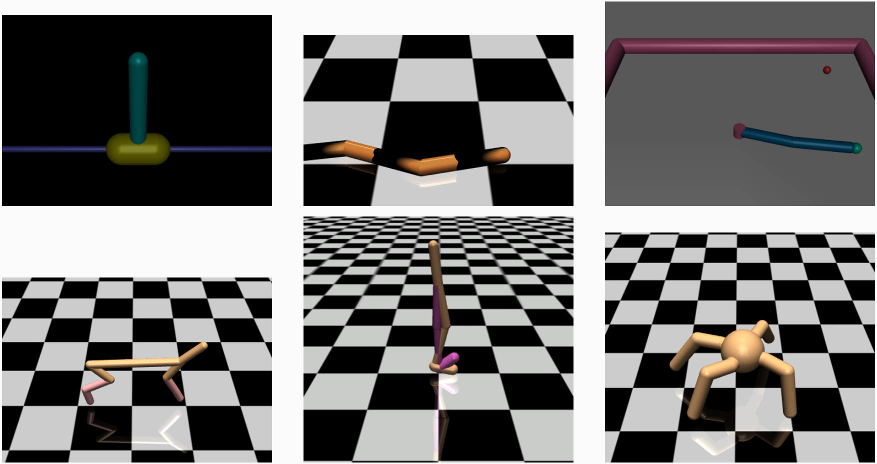
```python
with tf.name_scope('value'):
    V = fc(hid_outs['v'], 1, is_train,
        hidden_w, use_batch_norm=use_batch_norm, scope='V')

with tf.name_scope('advantage'):
    l = fc(hid_outs['l'], (action_size * (action_size + 1))/2, is_train, hidden_w,
        use_batch_norm=use_batch_norm, scope='l')
    mu = fc(hid_outs['mu'], action_size, is_train, action_w,
         activation_fn=action_fn, use_batch_norm=use_batch_norm, scope='mu')

    pivot = 0
    rows = []
    # building L matrix:[paper:] L is a lower triangular matrix generated by linear ouput layer of nn,
    # with the diagonal elements exponentiated
    for idx in xrange(action_size):
        count = action_size - idx
        #tf.exp -> exp(input), tf.slice(tensor, begin, size) -> slices a chunk of size starting at begin from tensor;
        diag_elem = tf.exp(tf.slice(l, (0, pivot), (-1, 1)))
        non_diag_elems = tf.slice(l, (0, pivot+1), (-1, count-1))
        row = tf.pad(tf.concat(1, (diag_elem, non_diag_elems)), ((0, 0), (idx, 0)))
        rows.append(row)

        pivot += count

    L = tf.transpose(tf.pack(rows, axis=1), (0, 2, 1))#tf.pack renamed tf.stack since tf 1.0
    P = tf.batch_matmul(L, tf.transpose(L, (0, 2, 1)))

    tmp = tf.expand_dims(u - mu, -1)
    A = -tf.batch_matmul(tf.transpose(tmp, [0, 2, 1]), tf.batch_matmul(P, tmp))/2
    A = tf.reshape(A, [-1, 1])

with tf.name_scope('Q'):
    Q = A + V
```

Optimization criterion [network.py]

```
with tf.name_scope('optimization'):
    self.target_y = tf.placeholder(tf.float32, [None], name='target_y')
    # as in paper: mean of squared difference of target and q-value
    self.loss = tf.reduce_mean(tf.squared_difference(self.target_y, tf.squeeze(Q)), name='loss')
```

Experiments

Networks were trained on various locomotion tasks in the gym/ mujoco framework



Experiments

Environment	Observation size	Action size
Inverted Pendulum	4	1
Swimmer	7	2
Reacher	10	2
Half Cheetah	16	6
Walker2d	16	6
Ant	110	8