IEOR 8100: Reinforcement learning

Lecture 3: Large-scale Q-learning

By Shipra Agrawal

1 Q-learning with function approximation

The tabular Q-learning does not scale with increase in the size of state space. In most real applications, there are too many states too keep visit, and keep track of. For scalability, we want to *generalize*, i.e., use what we have learned about already visited (relatively small number of) states, and generalize it to new, similar states. A fundamental idea is to use 'function approximation', i.e., use a lower dimensional feature representation of the state-action pair s, a and learn a parametric approximation $Q_\theta(s,a)$. For example, the function $Q_\theta(s,a)$ could simply be a linear function in θ and features $Q_\theta(s,a) = \theta_0 f_0(s,a) + \theta_1 f_1(s,a) + \ldots + \theta_n f_n(s,a)$, or a deep neural net. Given parameter θ, the Q-function can be computed for unseen s, a. Instead of learning the $|S| \times |A|$ dimensional Q-table, the Q-learning algorithm will learn the parameter θ. Here, on observing sample transition to s' from s on playing action a, instead of updating the estimate of $Q(s,a)$ in the Q-table, the algorithm updates the estimate of θ. Intuitively, we are trying to find a θ such that for every s, a, the Bellman equation,

$$Q_\theta(s,a) = \mathbb{E}_{s' \sim P(\cdot|s,a)}[R(s,a,s') + \gamma \max_{a'} Q_\theta(s',a')]$$

can be approximated well for all s, a. Or, in other words, we are trying to minimize some loss function like squares loss:

$$\ell_\theta(s,a) = \mathbb{E}_{s' \sim P(\cdot|s,a)}(Q_\theta(s,a) - R(s,a,s') - \gamma \max_{a'} Q_\theta(s',a'))^2 =: \mathbb{E}_{s' \sim P(s,a,\cdot)}\left[\ell_\theta(s,a,s')\right]$$

where

$$\ell_\theta(s,a,s') = \left(Q_\theta(s,a) - target(s')\right)^2$$
$$target(s') = R(s,a,s') + \gamma \max_{a'} Q_\theta(s',a')$$

Fundamentally, Q-learning uses gradient descent to optimize this loss function given sample (target) observations at s'.

1.1 Q-learning Algorithm overview.

Start with initial state $s = s_0$. In iteration $k = 1, 2, ...,$

- Take an action a.
- Observe reward r, transition to state $s' \sim P(\cdot|s,a)$.
- $\theta_{k+1} \leftarrow \theta_k + \alpha_k \nabla_{\theta_k} \ell_{\theta_k}(s,a,s')$, where

$$\nabla_\theta \ell_{\theta_k}(s,a,s') = \delta \nabla_{\theta_k} Q_{\theta_k}(s,a)$$
$$\delta = r + \gamma \max_{a'} Q_{\theta_k}(s',a') - Q_{\theta_k}(s,a)$$

- $s \leftarrow s'$,

If s' reached at some point is a terminal state, s is reset to starting state.

Effectively, in the tabular Q-learning method discussed in the previous lecture, after taking action a_t and observing s_{t+1}, replace Step 8 and Step 9 by:

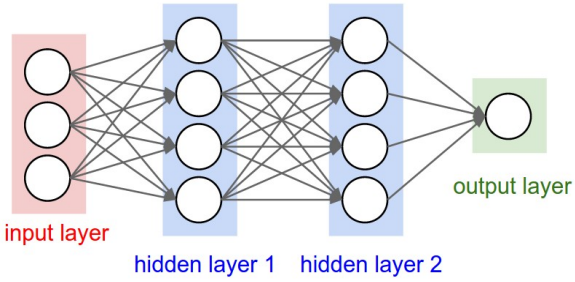
8: Let $\delta_t := r_t + \gamma \max_{a'} Q_{\hat{\theta}}(s_{t+1}, a') - Q_{\hat{\theta}}(s,a)$.

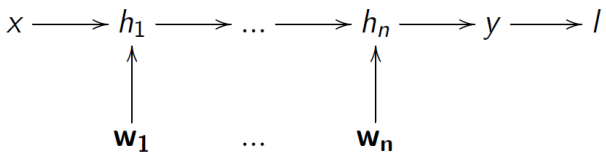
9: Update $\hat{\theta} \leftarrow \hat{\theta} + \alpha_t \delta_t \nabla_{\hat{\theta}} Q_{\hat{\theta}}(s,a)$.

2 Deep Q-learning Networks (DQN)

Deep Q-learning Networks (DQN) use deep neural network for function approximation, with θ being the parameters of the neural network.

Architecture A deep representation is composed of many functions, typically linear transformations alternated by non-linear activation functions: $h_1 = W_1 x, h_2 = \sigma(h_1), \ldots, h_{k+1} = W_k h_k, h_{k+2} = \sigma(h_{k+1}),$, where $\sigma(\cdot)$ is a nonlinear function. Two popular functions are sigmoid and Relu.





The parameters pf this network are $\theta = (\mathbf{W}_1, \ldots, \mathbf{W}_n)$. The parameter θ maps any given input x to output $f(h_n)$, where $f(\cdot)$ is a 'scoring function'. The scoring function could yield a scalar prediction $\hat{y} = f(h_n)$, or scores/probabilities over different possible predictions/labels. For example, softmax scoring function maps $h_n \in \mathbb{R}^k$ to probabilities $\hat{p} = f(h_n) = [p_1, \ldots, p_k]$, where $\hat{p}_i = \frac{e^{h_n[i]}}{(e^{h_n[1]} + \ldots + e^{h_n[k]})}$. Then, this score is evaluated against target prediction y^* using a loss function $\ell(\cdot, y^*)$. Examples are squared loss function, common in regression $\ell(\hat{y}, y^*) = (\hat{y} - y^*)^2$ for evaluating a prediction against a given target, and log-likelihood for evaluating probability scores $\ell(\hat{p}, y^*) = -\log(\hat{p}_{y^*})$ common in multi-label classification. A neural network is trained using labeled data (x_i, y_i) to learn parameters $\theta = (\mathbf{W}_1, \ldots, \mathbf{W}_n)$ that minimize loss $\sum_i \ell(f(\theta), y_i)$.

In DQN, the input x is formed by the feature vector for (s_t, a_t) pairs, and for any given θ the output $f_\theta(h_n)$ is a prediction for $Q_\theta(s_t, a_t)$. However, to evaluate these prediction we do not directly have a target prediction/label y^*. Instead, we bootstrap to use current Q-network (i.e., current value of parameters θ) to generate a target as: $r_t + \gamma \max_a Q_\theta(s_{t+1}, a)$. Then, to update θ we use a gradient descent step for squared loss function.

The second form of output (probability scores) will be more relevant in policy gradient methods, discussed later in the course. There, we will use deep neural networks to learn "policies", i.e., probability scores for different actions in a given state.

Backpropagation for gradient computation The independent-layer structure of deep neural network allows the gradient computations efficiently through **backpropagation**. (In the reinforcement learning context) Backpropagation refers to simply a computational implementation of the chain rule : an algorithm that determines the relevant partial derivatives by means of the backward pass.

Suppose $Q_\theta(s, a)$ is given by a deep neural network with $K - 1$ hidden layers and weights $\theta = (W_1, \ldots, W_K)$. Its input $\mathbf{x}$ is representation of a state action pair s, a, and output

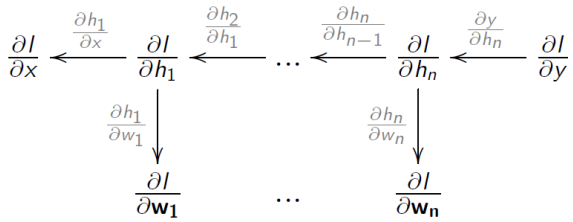
$$\hat{y} = f(h^K) = f(W^K(\sigma(\cdots W^3 \sigma(W^2 \cdot \sigma(W^1 \mathbf{x})))))$$

is a representation of action Q-values $Q_\theta(s,a)$. Given, a target observation $y = target(s')$, we are interested in updating θ by computing gradient of loss function

$$\ell_\theta(\hat{y}, y) = (\hat{y} - y)^2 = (f(h^K) - y)^2$$

where

$$\begin{aligned} h^k = \sigma(z^k), z^k = W^k h^{k-1} \\ h^{k-1} = \sigma(z^{k-1}), z^{k-1} = W^{k-1} h^{k-2}, \\ \dots, \\ h^1 = \sigma(z^1), z^1 = W^1 \mathbf{x} \end{aligned}$$



Now, gradient with respect of ℓ_θ with respect to parameter W^ℓ_{ab} is:

$$\frac{\partial \ell_\theta(\cdot)}{\partial W^\ell_{ab}} = 2(f(h^k) - y)\nabla f(h_n) \left[\frac{\partial h^K_i}{\partial W^\ell_{ab}}\right]_i$$

Then, basic observation is as follows. For a neuron r in layer k, $\ell < k$:

$$\begin{aligned} \frac{\partial h^k_r}{\partial W^\ell_{ab}} &= \sigma'(z^k_i)\frac{\partial z^k_r}{\partial W^\ell_{ab}} \\ &= \sigma'(z^k_r)\frac{\partial W^k_r h^{k-1}}{\partial W^\ell_{ab}} \\ &= \sigma'(z^k_r)\sum_{i \in \text{parents}(r)} W^k_{ri}\frac{\partial h^{k-1}_i}{\partial W^\ell_{ab}} \end{aligned}$$

Thus, the gradients can be back propagated over the network, until we reach layer ℓ, so that

$$\frac{\partial h^\ell_i}{\partial W^\ell_{ab}} = \begin{cases} \sigma'(z^\ell_a)h^{\ell-1}_b, & \text{if } i = a \\ 0 & \text{otherwise} \end{cases}$$

This follows from layer structure of the neural network so that:

$$\begin{aligned} \frac{\partial z^\ell_a}{\partial W^\ell_{ab}} &= \frac{\partial \sum_i W^\ell_{ai} h^{\ell-1}_i}{\partial W^{\ell-1}_{ab}} \\ &= h^{\ell-1}_b + \sum_i W^\ell_{ai}\frac{\partial h^{\ell-1}_i}{\partial W^\ell_{ab}} \\ &= h^{\ell-1}_b \end{aligned}$$

as layer $h^{\ell-1}$ is independent of layer ℓ parameters.

3 Challenges

Exploration. Note that we are trying to minimize many loss functions (one for each s, a) simultaneously. At any point, the number of samples available for different state and action pairs are different, and depend on how much we explore a given state and that action. Therefore, depending on the both the transition dynamics and our exploration scheme, at any point, we will have different level of accuracies for different state and actions. Note that not all s, a need to have high accuracy in order to find optimal policy. For example, if some states are rare and have low value, in those states loss function does not need to be optimized. Further, if two states/action pairs have similar features, then both don't need to be explored in order to learn a good parameter θ. This points to significance of an adaptive exploration scheme which manages appropriate accuracy levels across state and actions in order to quickly converge to a good θ.

A simple exploration scheme often utilized is ϵ-greedy strategy, possibly with decreasing ϵ to adapt to less exploration requirement in the later part of execution. We will study more advanced adaptive exploration methods later in the course.

Stabilizing. When comparing to supervised learning, we may view each sample s' as a training data row, with the loss in s, a for parameter θ for this row being $(Q_\theta(s, a) - target(s'))^2$. However, there are several challenges compared to supervised learning. Firstly, this is different from minimizing loss compared to 'true' labels in the training data, as in supervised learning. Here, the **target** $target(s') = \mathbb{E}_{s' \sim P(s'|s,a)}[R(s, a, s') + \gamma \max_{a'} Q_\theta(s', a')]$ that plays the role of labels is in fact also an estimate. This can make the learning unstable, as the target label may change too quickly as θ changes.

There are two main ideas utilized to make the Q-learning stable.

- Batch learning (experience replay): Experience Replay (introduced in Lin [1992]) stores experiences including state transitions, rewards and actions, the sample observations – the necessary data to perform Q learning updates. Then, these experiences are used in mini-batches to update neural networks. This increases speed of update due to batch updates, as well reduces correlation reduces correlation between samples used to update the parameters with decisions made from those updates.

- Lazy update of target network: The target function is changed frequently with DNN update. Unstable target function makes training difficult. So lazy update fixes the parameters of target function and replaces them with the latest network occassionally, every thousands steps.

References

Long-Ji Lin. *Reinforcement Learning for Robots Using Neural Networks.* PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX93-22750.