Policy Gradient: Supplement

N.J. Mathai

1 Recap and Forward

Previously we employed function approximation systems (most generally, neural networks) to represent a key object of interest in RL: the value function approximation (Q).

Since Q is a mapping from $S \times A \longrightarrow R$, a neural network with a final linear regression stage was most appropriate. Recall that in this configuration, the portion of the neural network prior to the output-side linear regressor is performing a general transformation of the input data variable to some transformed space. The transformed variable is then input to the final linear regression stage to yield the mapping from domain to range.

Gradient descent was employed to train the neural network. Since in RL we do not have access to the data set in bulk at training time, full-batch gradient descent is out of the question. Instead, we must employ stochastic gradient descent, training the neural network on elements of our dataset as they come from the agent-environment interaction.

More glaringly, however, is the lack of target data (recall: neural networks for supervised learning require datasets with domain and range tuples; the range component is called the *target*)! However, the concepts of the earlier chapters of the text come to our rescue and we may employ:

- Monte Carlo-based sampling of the agent-environment interaction to yield the stochastic gradient descent techniques for RL
- TD(0) and TD(n) approximations for $E[G_t]$; because they employ bootstrapping (i.e., the utilization of the learned Q object to form the above approximation) these are termed semi-gradient descent techniques

In this note, we discuss the utilization of function approximation (i.e., neural networks) to represent the policy object of RL, π. The very notion that we can represent the policy by a differentiable, tunable structure (a neural network) yields the promise of optimizing the policy as an object, without resorting to the value function. This is something that was not possible with the tabular data structure representation of a policy.

2 Neural Networks

Fundamentally, there are two classes of neural networks: ones that regression, and ones that classify. The distinction is that regression neural networks map some domain space to a real vector space, while classification neural networks map some domain space to a discrete space. The distinction is quite critical. Before proceeding we will note that we consider below the application of neural networks to represent policies where the action space is discrete.

With regression networks, the output stage is a standard linear regression layer where the squared error per-prediction error function is employed (recall, the squared error PPE function has the theoretical advantage of being differentiable everywhere).

With classification networks, the output stage must be modified to either a logistic regression stage or a softmax regression stage. If our function approximator is from some domain to a 2-class output, then logistic regression is apt (logistic regression being the fundamental linear classifier stage for the 2-class case, i.e., linear binary classifiers). If our function approximator is from some domain to a K-class output, then softmax regression is applicable.

What is the principle of these classifier stages? First, some setup: linear regression and classification both employ the adaptive linear combiner. For linear binary classification, one must compress the output of this ALC, so as to yield a soft-binary output in $(0, 1)$; the logistic function is employed (for reasons outside our scope). With this, one can not use the square error PPE function of regression (the error is lobsided when applied to the problem). Instead one uses cross entropy PPE. The whole composition from ALC to logistic function to cross entropy PPE is differentiable, and this forms the basis of tuning a neural network with a logistic regression output stage.

For K-class classification, one must adapt the structure as follows. Instead of producing a single real variable output from the ALC and compressing it via the logistic function, one employs a K-output ALC:

$$\mathbf{z} = W\mathbf{u} + \mathbf{b} \tag{1}$$

and then applies the softmax function:

$$y_i = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{2}$$

It is trivial to see that the y_i components are all in $(0, 1)$, and their sum is unity. Thus the various components represent the neural network's normalized choice of the i-th class.

3 Towards Policies

It should be clear that the use of the above K-class classifer is directly aligned to the problem of representing a policy object. First, for a discrete action space, A, with $|A|$ actions, we can see $K = |A|$. Next, the fact that softmax components sum to unity is directly aligned with the probabilistic nature of a policy function.

Now that we've settled on the function approximation method (i.e., a neural network with a K-class softmax classifier), we just need to specify how to train the neural network. As with the use of neural networks for the learning of the value function, we are confronted with the fact that we do not have the targets for our dataset!

4 Training

Recall that the basis of standard neural network training is the use of optimization processes to find an optimal setting for the parameters of our neural network. Gradient descent is employed to bring us down a performance surface.

In the absence of a target for value function neural network approaches, we approximated these targets. For policy function approximation we take a different approach, but one still founded on optimmization.

Equation (13.4) shows the objective function that will form the basis of our optimizer. Note: the objective is a function of the value, and hence we want to ascend (not descend) the performance surface defined by this optimizing objective.

(13.5) gives us an expression for the gradient, per the Policy Gradient Theorem. This yields update law (13.7), which using Monte Carlo sampling becomes (13.8) and the REINFORCE algorithm of p.328.