Midway report of DM887 Final Project

Jiawei Zhao, Kristóf Péter
Department of Mathematics and Computer Science
University of Southern Denmark
Moseskovvej, 5230 Odense
jizha22@student.sdu.dk
krpet24@student.sdu.dk

Abstract

To implement the project in **learning to play Atari games with Bayesian Deep Expected SARSA**, we have attempted multiple baseline algorithms on Atari and non-Atari game environments at Gymnasium. Currently, we are unable to improve the learning curve characterized by number of gradient descent (GD) time steps t_{GD} at the X-axis and the total episode rewards G at the y-axis at the experimented Atari environments.

1 Summary of the experiments

We experimented with several different types of neural networks and environments:

1. Only fully connected linear layers
   (a) Testing algorithms on the "CartPole-v1" environment.
   (b) Testing algorithms on the "ram" versions of Atari environments, for example "ALE/Breakout-ram-v5". Here the states are 128 dimensional vectors.
2. Convolutional Neural Network: For Atari environments with "rgb" state output. For example "ALE/Breakout-v5".
   (a) Using the original picture state output, which is an array with size $(210, 160, 3)$, where the first two numbers represent the dimensions of the picture, the last number represents the 3 different colors (red, green, blue) we have values for.
   (b) Transforming the original picture state output into a grayscale picture with dimensions $(210, 160)$ in hopes of simplifying the task and reducing training time.
   (c) Further transforming the grayscale picture to a (50, 50) in hopes further reducing training time.
3. Fully connected linear layers are used as parameters in a Bayesian approach. There are two output layers instead of one. One is used as the mean, one as the standard deviation of the action values generated from the normal distribution.
   (a) In lack of sufficient computational resources, we tested it only on the "CartPole-v1" environment.

We also tried a few different versions of the Expected Sarsa algorithm:

1. Softmax policy: Actions are taken according to a discrete distribution with the softmax function of the Q-values as the probabilities.
2. Epsilon greedy policy: Actions are taken randomly with ϵ probability, and according to the index of the largest Q-value with $1 - \epsilon$ probability. Where ϵ decreases over time, as the agent explores the environment.

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

3. We are using two neural networks with the same architecture. A "policy" network for interacting with the environment and calculating the Q-values for the current state. This is the network updated using GD. The second network is the "target" network, which is used for calculating the expected Q-values for the next state, these values are used in the loss calculation alongside with the Q-values for the current state.

2 Implementation of the algorithms

We experimented on three baseline neural networks each taking the number of observations N_{state} as the dimension of input, and the number of possible discrete actions N_{action} as the dimension of output.
At the **Atari** mode in *CleanVersion.py*, the neural network Q_1 is used to train RGB or greyscale input states e.g. *ALE/Breakout-v5* or *ALE/Tennis-v5*. Q_1 consists of three convolutional layers each followed by a $ReLU$ activation function, one fully-connected linear hidden layer followed by a $ReLU$ activation function, and a fully-connected linear output layer.
The implementation of Q_1 is elaborated at the first section of **Algorithm 1**.

At the non-**Atari** mode in *CleanVersion.py*, the neural network Q_2 is used to train vectorized input states e.g. *ALE/Breakout-v5* or *ALE/Tennis-ram-v5*. Q_2 consists of two fully-connected linear layers each followed by a $ReLU$ activation function, and a fully-connected linear output layer.
The implementation of Q_2 is elaborated at the second section of **Algorithm 1**.

In *BayesianVersion.py*, a simplistic Bayesian neural network Q_3 is used. Q_3 consists of two fully-connected linear layers each followed by a $ReLU$ activation function, followed by two parallel fully-connected linear output layers to calculate the mean value and the standard deviation of the output the preceding layers.
The implementation of Q_3 is elaborated at the third section of **Algorithm 1**. Subsequently, the major pseudo code blocks are stated at **Algorithm 2**.
Pseudo code for the main loop of the current version of the Deep expected SARSA network is stated at **Algorithm 3**.

3 To-do list and a time plan of the remaining tasks

1. Perfect Bayesian Q-factor design for Atari games (in the coming two-three weeks)
   (a) Designing a more advanced Bayesian network. (The next two weeks)
   (b) Deciding on the best approach: Convolution networks with picture states or standard linear layers with vector states. (The next two weeks)
   (c) Designing a more advanced algorithm with more theoretical background for a Bayesian network as the Q-function. (The next two weeks)
   (d) Try our code in more environments and with other hyperparameters. (The next two-three weeks depending on our success)
   (e) Trials using more computing power. (Starting from next week.)
   (f) Writing pseudo-code for the finished algorithm using a Bayesian network (last weeks before project submission)
2. Trying to figure out any possible issues with the current basic Expeceted Sarsa algorithm and finding the reason why it performs so poorly. (from now until the end of the project)
   (a) Extending and possibly correcting the current pseudo code. (from now until the end of the project)
3. Comparing our code's performance to the state of the art.
   (a) Comparing to our own versions of DQN, DDQN, Soft Q-learning etc. (2-3 weeks from now, possibly sooner depending on our success with part 1)
   (b) Comparing to other performances in research papers dealing with reinforcement learning applied to Atari games. (from now until the end of the project, but writing the documentation at the end)
4. Trying different model update intervals. (From now until the end, as we experiment with different models.)

5. Flesh out limitations of our final code and propose further improvement opportunities. (last weeks before project submission)

Algorithm 1 Deep expected SARSA Q-network Q_1, Q_2, and Q_3

Implementation of Q_1:
Assign the size of the preprocessed flattened input tensor as N_{input}
Assign $N_{hidden} = 512$
Assign the number of possible discrete actions at the given Atari game environment as N_{action}

First convolutional layer (input): (N_{input}, *32*) with a kernel size $N_{k1} = 8$ and stride $N_{stride1} = 4$
ReLU activation
Second convolutional layer (hidden): (*32, 64*) with a kernel size $N_{k2} = 4$ and stride $N_{stride2} = 2$
ReLU activation
Third convolutional layer (hidden): (*64, 128*) with a kernel size $N_{k3} = 3$ and stride $N_{stride3} = 1$
ReLU activation
First fully-connected linear layer (hidden): ($128 \times N_{input}, N_{hidden}$)
ReLU activation
Second fully-connected linear layer (output): (N_{hidden}, N_{action})

Implementation of Q_2:
Flatten the input state into a one-dimensional vector V, then assign the length of V as N_{input}
Assign the number of possible discrete actions at the given Atari game environment as N_{action}

First fully-connected linear layer (input): ($N_{input}, 128$)
Second fully-connected linear layer (hidden): ($128, 128$)
Third fully-connected linear layer (output): ($128, N_{action}$)

Implementation of Q_3:
Flatten the input state into a one-dimensional vector V, then assign the length of V as N_{input}
Assign $N_{hidden} = 1024$
Assign the number of possible discrete actions at the given Atari game environment as N_{action}
Assign the sampling batch size $\mid D' \mid = 16$

First fully-connected linear layer (input): (N_{input}, N_{hidden})
Second fully-connected linear layer (hidden): (N_{hidden}, N_{hidden})
A fully-connected linear layer calculating the mean $\mu_{D'}$ of the replay batch and yielding the first element of the output: (N_{hidden}, N_{action})
A fully-connected linear layer calculating the stanard deviation $\sigma_{D'}$ of the replay batch and yielding the second element of the output: (N_{hidden}, N_{action})

Algorithm 2 Deep Expected Sarsa Methods

Create the Neural Network:
Parameters from the environment: $number\ of\ actions$, $state\ dimension$
$action\ space$ = $\{0, 1, \ldots, number\ of\ actions\}$ for all environments we tested
Create a *ReplayMemory*:
The observed $(states,\ action,\ next\ state,\ reward)$ tuples are stored in the *ReplayMemory*. Batches are sampled randomly from here.
Specify parameters:
Episode Number, Batch size, Learning rate for optimizer = γ, Learning rate for Expected Sarsa = α, Loss function (Mean Squared Error or Huber Loss), Optimizer (Adam), Softmax function, epsilon decay values (epsilon, decay $\in\ (0, 1)$), Constant for Polyak averaging = τ, Policy Network = Q, Target Network = Q', Policy Network Weights= θ, Target Network Weights= θ'

Define Policy Method($state$):
if Using Softmax policy **then**
    action probabilities = Softmax($Q(state)$)
    return action $\sim$ DiscreteDistribution($action\ space$, action probabilities)
else if Using Epsilon Greedy Policy **then** Generate U $\sim$ Uniform$(0, 1)$
    if U > epsilon **then**
        return $action$ = Argmax($Q(state)$)
    else
        return a uniformly random $action$ from $action\ space$
    end if
end if

Define method for Preprocessing States($state$):
if Using RGB **then**
    $state$ from $(210,\ 160,\ 3)$ transformed to $(3,\ 210,\ 160)$ where the pixel values stay the same, only the structure of the array changes.
else if Using Greyscale **then**
    $state$ from $(210,\ 160,\ 3)$ transformed to $(1,\ 210,\ 160)$ where the pixel values for different colors are lost.
    if Using further dimension reduction **then**:
        $state$ from $(1,\ 210,\ 160)$ transformed to $(1,\ 50,\ 50)$ where the pixel values change according to the transformation.
    end if
else
    $state$ is unchanged
end if
if Testing on Atari games **then**
    $state$ = $state/255$
end if
return $state$

Define Optimization method and GD:
if *ReplayMemory* has too few observations **then**
    do not do anything
end if
$Batch\ size$ amount of $(state,\ action,\ nextstate,\ reward)$ tuples are sampled randomly from *Replaymemory*.
We create $states,\ actions,\ nextstates,\ rewards$ vectors from the batch with matching coordinates.
StateQvalues = $Q(states)[actions]$, so we take the Q values for the actions in the batch at the matching coordinates
NextStateQvalues = $\underline{0}$
for action in $action\ space$ **do**
    NextStateQvalues +=
    += $Q'(nextstates)[action]$ $\cdot$ Softmax($Q'(nextstates)$)$[action]$
end for
ExpectedValues = α*NextStateQvalues + $rewards$
Loss = Loss function(StateQvalues, ExpectedValues)
We use Optimizer(Loss, γ) to update the weights of Q using GD.
[1]

Algorithm 3 Deep Expected Sarsa Main Loop

Main Loop:
for $1, \ldots, EpisodeNumber$ **do**
  $state \leftarrow reset\ environment$
  repeat
  $action \leftarrow$ Policy Method($state$)
  step from $state$ with $action$
  $state \leftarrow$ Preprocessing States($state$)
  observe $nextstate,\ action,\ reward$ after step
  $nextstate \leftarrow$ Preprocessing States($nextstate$)
  record $(state,\ action,\ nextstate,\ reward)$ to *ReplayMemory*
  $state \leftarrow nextstate$
  Perform Optimization method and GD
  Soft update the weights of Q' with Polyak averaging:
  $\theta' \leftarrow \tau$*θ + $(1 - \tau)$*θ'
  until episode end
  if Using Epsilon Greedy Policy Method **then**
    epsilon $\leftarrow$ epsilon $\cdot$ decay
  end if
end for

[1]