# Final Report of DM887 Course 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 **Learning to play Atari games with Bayesian Deep Expected SARSA**, we used (2) implemented according to (3) as a theoretical baseline. The $Q$ used to determine the policy for each action are calculated as the matrix product of the output of a feature extractor in the form of a convolutional neural network, and a randomly generated Gaussian matrix representing the policy Q-weights to calculate the $Q$ from extracted features per permissible action. The feature extractor is updated by gradient descent, while the policy Q-weights are updated by a Bayesian posterior update based on a large number of sampled tuples representing the explored states, actions, next actions, and rewards. The large number of tuples sampled from the replay memory are used to simulate the observations in the posterior probability. In our algorithm, we use the expected $Q$ of the next states to calculate the Bellman values, instead of the maximum $Q$ per action at a $Q$ table during the gradient descent and update of policy Q-weights, which is the case in standard Q-learning.


## 1 List of symbols and constants used as default values during training:

1. **Global variables:**
   (a) $env$ : An Atari Game environment using RGB images to represent its states, which an agent interacts with, e.g., $env1$: ***ALE/Breakout-v5***, and $env2$: ***ALE/Pong-v5*** from Gymnasium (1). We assign the set of all the aforementioned Atari Game environments as $ENV$.
   (b) $reset$ : A statement setting the current $env \in ENV$ to its starting state, represented by a frame $fr$.
   (c) $A$ : The action space of $env \in ENV$ with $A = \{0, 1, 2, \ldots, |A-1|\}$ in $env \in ENV$ which we tested.
   (d) $fr$ : A raw frame generated by $env \in ENV$ with a shape (210, 160, 3), representing an RGB image in $env \in ENV$ which we tested.
   (e) $tfr$ : A transformed frame to the desired shape during pre-processing with a shape (84, 84), representing a grayscale image.
   (f) $|A|$ : Size of $A$ in $env \in ENV$ which we tested.
   (g) $FR_{skipped} = 4$ : Number of frames skipped (we step with the same action for this many frames).
   (h) $t_{total}$ : Count of time steps passed during the algorithm at $env \in ENV$.
2. **Neural Network parameters and $Q$ calculation:**
3. (a) $4$ : Size of the input channels to the first convolutional layer, i.e., the number of stacked frames at the state tensor $s$.
   (b) $512$ : Length of the output layer of the last fully connected linear layer, i.e., the number of features at the output tensor of $\phi(s)$.
   (c) $\phi$ : Policy Network.
   (d) $\theta$ : Weights of $\phi$.
   (e) $\phi_{target}$ : Target Network.
   (f) $\theta_{target}$ : Weights of $\phi_{target}$.
   (g) $M$ : A matrix of mean vectors for generating from the multivariate normal distribution with shape $(|A|, 512)$.
   (h) $M_a$ : A mean vector with shape $(512)$, which is an element of $M$ representing $a \in A$.
   (i) $\Sigma$ : A tensor containing covariance matrices for generating the multivariate normal distribution with shape $(|A|, 512, 512)$.
   (j) $\Sigma_a$ : A covariance matrix with shape $(512, 512)$, which is an element of $Cov$ representing $a \in A$.
   (k) $L$ : A tensor containing matrices of coefficients to generate from the normal distribution, calculated from $\Sigma$.

(l) $L_a$ : A matrix of coefficients to generate from the normal distribution, $L$ representing $a \in A$, calculated from $\Sigma_a$.

(m) $W$ : A matrix of random vectors generated from the multivariate normal distribution with shape $(|A|, 512)$.

(n) $W_a$ : A vector of $W$ with shape $(512)$, representing $a \in A$.

(o) $\phi\phi^T$ and $\phi_Y$ : Assistant variables used during a posterior update.

4. **Replay Memory elements:**

(a) $RM$ : Replay Memory, we store ($s$, $a$, $r$, $s'$, $a'$) tuples each representing a time step $t$ here.

(b) $B$ : A Batch of tuples from the replay memory

(c) $|B|$ : Batch size

(d) $s$: A state tensor with shape $(4, 84, 84)$, concatenated by $tfr_1, tfr_2, tfr_3, tfr_4$, which are pre-processed frames from four adjacent time steps. $(1, 4, 84, 84)$ is the shape of which is fed into $\phi$ or $\phi_{target}$.

(e) $s'$: A tensor representing the next state, represents state which $env \in ENV$ moves to. $s'$ is constructed and shaped in an identical manner as $s$.

(f) $a$ : An action $a \in A$ represented by an index in integer defined by $A$ in $env \in ENV$)

(g) $r$ or $reward$: A reward represented by an integer or a float in $env \in ENV$.

(h) $\sum r'$ : The cumulative reward collected by skipping frames performed during the frame-skipping option.

(i) $a'$ : A boolean value indicating whether $t$ terminates the episode $e$ which $t$ belongs to after $s'$.

(j) $B_s$ : A batch of states with shape $(|B|, 4, 84, 84)$.

(k) $B_{s'}$ : A batch of next states with shape $(|B|, 4, 84, 84)$.

(l) $B_a$ : A batch of actions with shape $(|B|)$.

(m) $B_r$ : A batch of rewards with shape $(|B|)$.

(n) $B_{a'}$ : A batch of done values with shape $(|B|)$.

5. **Constants(unchanged throughout the algorithm):**

(a) $|RM| = 200000$ : Size of the Replay Memory.

(b) $F_1 = 1000$ : Thompson sampling frequency .

(c) $F_2 = 4$ : Training frequency.

(d) $F_3 = 10000$ : Target update frequency.

(e) $F_4 = 10$ : Posterior update frequency beyond $F_3$.

(f) $B_{gd} = 32$ : Training update batch size .

(g) $B_{post}$ : Posterior update batch size.

(h) $LR = 0.0025$ : Learning rate for the optimizer.

(i) $\gamma = 0.99$ : Learning rate for the Bellman equation

(j) $N = 10000$: Maximum number of episodes to run the algorithm. An episode takes place between the initial state and the final state of the environment.

(k) $\sigma = 0.001$ Prior variance

(l) $\sigma_n = 1$ Noise variance

(m) $I$ : Identity matrix

6. **Functions:**

(a) $MSE(\cdot, \cdot)$ : The mean squared error (MSE) function used for loss calculation taking two vectors as inputs.

(b) $Adam(\cdot, LR, \theta)$ : The optimizer used for updating the weights of a neural network's weights by gradient descent. The input parameters represent the loss by a scalar tensor with gradient information, the learning rate, and weights of the $\phi$ respectively.

(c) $Cholesky(\cdot)$ : The Cholesky factorization decomposing a symmetric, positive-definite matrix $X$ which suffices $X = YY^T$, returns $Cholesky(X) = Y$. It is used for calculating the coefficient vectors in the multivariate normal distribution from its covariance matrix.

(d) $Softmax(\cdot)$ : Calculates a vector of probabilities (positive numbers summing to 1) from a vector of numbers, using the softmax function.

7. **Other:**

(a) $\mathcal{N}(M_a, \Sigma_a)$ Multivariate normal distribution with means $M_a$, covariance matrix $\Sigma_a$

## 2 Bayesian Q-factor network design

We used a convolutional neural network structure to extract the features from the frames created in the Gymnasium Atari environments (1). Using the same structure as explained below, we will define a Policy Network as $\phi$ and a Target Network as $\phi_{target}$ for feature extraction. However, the $Q$ used in the action selection (policy) are going to be given by

$$W_a \sim \mathcal{N}(M_a, \Sigma_a) : \ a = 0, 1, 2, \ldots, |A|$$

$$Q(s) = W^T \phi(s).$$

$$Q(s, a) = W_a^T \phi(s).$$

Making our randomly generated $W_a$ vectors a sort of last Bayesian Layer. The parameters $M_a, \ \Sigma_a$ are going to be updated by Bayesian inference based on the explored states during the Posterior Update step.

### 2.1 Input Structure:

The frames $fr$ come in the form of $(210, 160, 3)$ colored images, however, during pre-processing we turn them into size $(84, 84)$ grayscale images $tfr$. A singular input for the network will be referred to as a $s$, with shape $(4, 84, 84)$, that consists of $4$ concatenated, pre-processed $fr$ frames of shape $(84, 84)$. In case we want to have a batch of states as input, it will be of shape $(|B|, 4, 84, 84)$, consisting of $|B| = B_{gd}$ number of $s$.

### 2.2 Network Structure:

The number of in-channels of the first layer will be the number of frames in the frame stack which we refer to as $state$, therefore $4$ in our case.
The length of the output of the last layer (fully connected linear layer), will be a chosen number for the feature vector length, therefore, $512$ in our case.
These and the other channels are as follows:
**First convolutional layer (input):** $(4, 32)$ with a kernel size $8$ and stride $4$
Batch Normalization for the output of layer $1$
*ReLU* activation
**Second convolutional layer (hidden):** $(32, 64)$ with a kernel size $4$ and stride $2$
Batch Normalization for the output of layer $2$
*ReLU* activation
**Third convolutional layer (hidden):** $(64, 64)$ with a kernel size $3$ and stride $1$
Batch Normalization for the output of layer $3$
*ReLU* activation
**Fully-connected linear layer (output):** $(7 \cdot 7 \cdot 32 \cdot 2, 512)$
*ReLU* activation

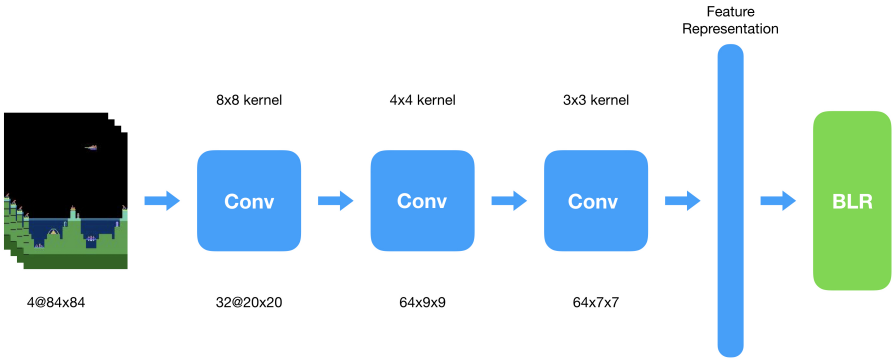



Figure 1: Visual representation of our network design, from (2)

### 2.3 Output Structure:

The output of the neural network for a singular $s$ input will be a feature vector $\phi(s)$ of shape $(512)$. For batch input $B_s$ the output will of shape $(|B|, 512)$, a matrix composed of $|B| = B_{gd}$ number of $\phi(s)$ vectors for the $s$ in the $B$ batch.

## 3 Implementation of the Deep Expected Sarsa Variant

1. We use a $Softmax$ policy instead of $\epsilon$-greedy or greedy policy based on the $argmax$ actions of a $Q$ table or a $Q$ function.

   **See: Assistant functions: Policy**

2. When training the policy network we approximate the expected value of the $Q_{target}(s')$ Q-values at next state by taking the dot product of $M^T\phi_{target}(s')$ with the $Softmax$ function of the next state policy $Q$ values, $Softmax(W^T\phi(s'))$. This value will be used to minimize the Bellman loss, instead of taking $M^T\phi_{target}(s')$ at the coordinate $a = argmax(W^T\phi_{s'})$ as in (3), which was more of a standard $Q$-learning approach.

   **See: Training Functions: TrainModel: lines 12-18**

3. In the case of the posterior update, we take $Softmax(M^T\phi_{target}(s')) \cdot M^T\phi_{target}(s')$ as our expected value estimate, as a replacement for the simpler way of taking the maximum of the $M^T\phi_{target}(s')$ value.

   **See: Training Functions: PosteriorUpdate: lines 30-36**

   (note that there may be a better way to approximate the expected $Q$-values during both the gradient descent updates of $\phi$ and the posterior updates of $W$)

## 4 Bayesian Deep Expected Sarsa Algorithm:

**Main loop**
Initialize: $\theta$, $\theta_{target}$, $W$, $M$, $\Sigma$, $L$, $\phi\phi^T$, $\phi_Y$
$RM = \{\}$
**for** $1, \ldots, N$ **do**
  $fr \leftarrow reset\ environment$
  $s \leftarrow$ Preprocess($fr$)
  **repeat**
  $a \leftarrow$ Policy($s$, $W$)
  $s',\ a,\ r,\ d \leftarrow$ Act($a$, $s$)
  record $(s,\ a,\ s',\ r,\ d)$ to $RM$
  $s \leftarrow s'$
  $t_{total}$ += 1
  **if** $t_{total}$ mod $F_1 = 0$ **then**
    $W \leftarrow$ ThompsonSample()
  **end if**
  **if** $t_{total}$ mod $F_2 = 0$ **then**
    TrainModel()
  **end if**
  **if** $t_{total}$ mod $F_3 = 0$ **then**
    UpdateTargetNetwork()
    **if** $t_{total}$ mod $F_4 = 0$ **then**
      $M$, $\Sigma \leftarrow$ PosteriorUpdate()
      **for** a = 0,1,2,. . . ,|A|-1 **do**
        $L_a = Cholesky(\frac{\Sigma_a + \Sigma_a^T}{2})$
      **end for**
    **end if**
  **end if**
  **until episode end**
**end for**

[1]

**Assistant Functions:**

Policy($s$,$W$):
$Q(s) \leftarrow W^T \phi(s)$
$a \sim$ Softmax($Q(s)$)
Return $a$

Preprocess($s$,$fr$):
$tfr \leftarrow$ transform $fr$ to $84 \times 84$
**if** $fr$ is the initial frame **then**
    Return $[tfr,\ tfr,\ tfr,\ tfr]$
**else**
    Return $[s[2], s[3], s[4], tfr]$
**end if**

Act($a$, $s$):
$Cr \leftarrow 0$
**if** Frame skipping **then**
    **for** 1,...,$FR_{skipped}\ +\ 1$ **do**
        step in the environment with $a$
        observe $fr,\ a,\ r,\ d$ after step
        $\sum r'$ += $r$
    **end for**
**else if** No frame skipping **then**
    step from $s$ with $a$
    observe $fr,\ a,\ r,\ d$ after step
    $\sum r'$ += $r$
**end if**

$s \leftarrow$ Preprocess($s$,$fr$)
Return $s$, $a$, $\sum r'$, $a'$

ThompsonSample():
$Z \sim \mathcal{N}(0, 1, 512)$ vector of 512 independent samples from $\mathcal{N}(0, 1)$
$W = M + LZ$
Return $W$

[2]

**Training Functions**

TrainModel():
$B \leftarrow$ sample $|B| = B_{post}$ from $RM$

$B_s,\ B_{s'},\ B_a,\ Br,\ Bd = B$

$Q(B_s, B_a) \leftarrow W_{B_a}^T \phi(B_s)$

$Q(B_{s'}) \leftarrow M^T \phi_{target}(B_{s'})$

$Probs \leftarrow Softmax(W^T \phi(B_{s'}))$

$EQ(B_{s'}) \leftarrow Q(B_{s'})^T \cdot Probs$

$Loss = MSE(Q(B_s, B_a), Br + (1 - Bd) \cdot \gamma \cdot EQ(B_{s'}))$

$\theta \leftarrow Adam(Loss, LR, \theta)$ as a single gradient descent step

UpdateTargetNetwork():
$\theta_{target} \leftarrow \theta$

PosteriorUpdate():
**for** 1,. . . ,PosteriorBatchSize **do**
    $B \leftarrow$ sample $|B| = 1$ from $RM$
    $s,\ s',\ a,\ r\ d = B$

    $\phi\phi_a^T \leftarrow \phi\phi_a^T + \phi(s)\phi(s)^T$

    $Q(s') \leftarrow M^T \phi_{target}(s')$

    $EQ(s') \leftarrow Q(s')^T \cdot Softmax(Q(s'))$

    $\phi_{Y_a} \leftarrow \phi_{Y_a} + \phi(s)^T (r + (1 - d) \cdot \gamma \cdot EQ(s'))$
**end for**
**for** a = 0,. . . ,|A|-1 **do**
    $inv \leftarrow (\frac{\phi\phi_a^T}{\sigma_n} + \frac{1}{\sigma} \cdot I)^{-1}$

    $M_a \leftarrow \frac{inv \cdot \phi_{Y_a}}{\sigma_n}$

    $\Sigma_a \leftarrow \sigma \cdot inv$
**end for**
Return $M$, $\Sigma$
[3]

## 5 Limitations

1. The feature extraction convolutional neural network $\phi$ might be oversimplified.
2. Image downscaling and grayscaling, automatic frameskipping of four frames between two adjacent time steps in the v5 Atari Game environments, and the absence of upsampling layers may cause loss of important information.
3. Replay memory and posterior update batch size requires a lot of memory.
4. Sparse and computationally expensive posterior updates.
5. Hyper parameters such as the four frequencies $F_1,\ F_2,\ F_3,\ F_4$, or the learning rates $\gamma,\ LR$ are likely to be heavily dependent on $env$.
6. Due to the insufficiency of our budget, we are unfortunately unable to harness the computing power of state-of-the-art GPUs to perform the exhaustive and time-consuming experiments.

## 6 Results:

1. We host our code implementation, *.csv* files describing the experiment results, and the plots at `https://github.com/Orthologues/DM887-ReinforcementLearning/tree/main/final_project/zhao_BDQN`. Our source code is implemented in a modulized manner.
2. The $config.py$ files for each of the learning curves shown at **Figure 2** and **Figure 3** are available publicly at `https://github.com/Orthologues/DM887-ReinforcementLearning/blob/main/final_project/zhao_BDQN/test_results/pong-v5_test1/config.py`, `https://github.com/Orthologues/DM887-ReinforcementLearning/blob/main/final_project/zhao_BDQN/test_results/pong-v5_test2/config.py`, `https://github.com/Orthologues/DM887-ReinforcementLearning/blob/main/final_project/zhao_BDQN/test_results/breakout-v5_test1/config.py`, and `https://github.com/Orthologues/DM887-ReinforcementLearning/blob/main/final_project/zhao_BDQN/test_results/breakout-v5_test2/config.py`.
3. In three out of the four learning curves as shown at **Figure 2** and **Figure 3**, learning was only slightly achieved after a large number of gradient descent time steps (see section "The state of the art"). Meanwhile, a learning curve using a lower-than-default frequency of gradient descent of $\phi$, and a higher-than-default frequency of target and posterior update at $env2$ failed to learn, which is indicated by the negative slope of the linear regression line.
4. As shown at **Figure 2**, increasing $LR$ (default = 0.0025) of the gradient descent update of $\phi$ to 0.01 has non-observable effects on the learning curve.
5. As shown at **Figure 3**, using a higher-than-default frequency for $W_{target}$ update (default = $F_3$) and posterior update (default = $F_3 \times F_4$) culminates in a negative learning curve.
6. As shown at **Figure 3**, using a lower-than-default frequency for $\phi$ update (default = $F_2$) culminates in a negative learning curve.

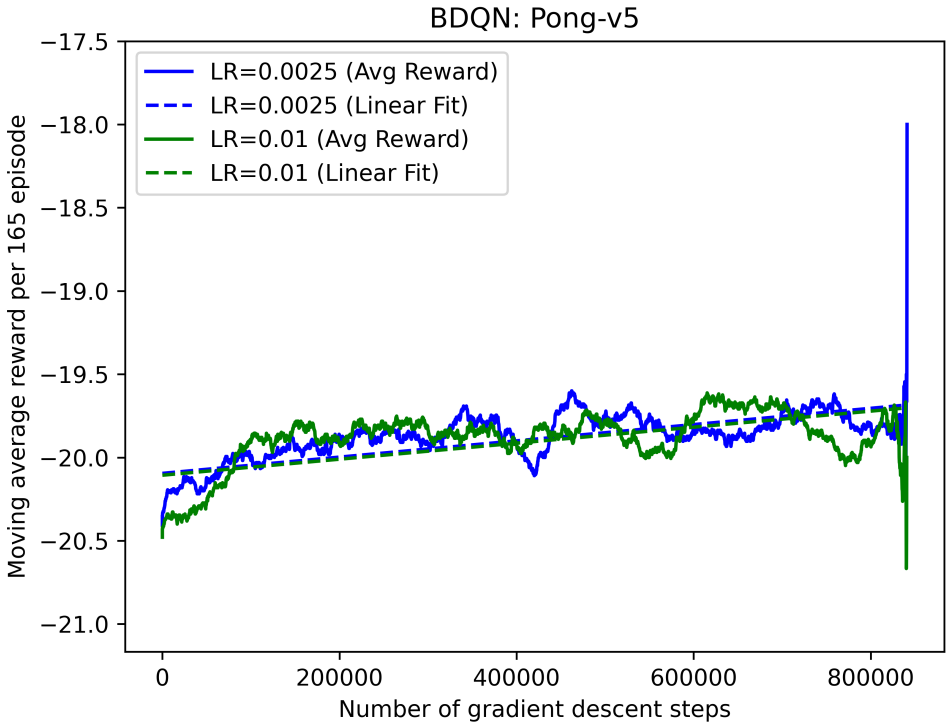



Figure 2: Learning at ***env1***

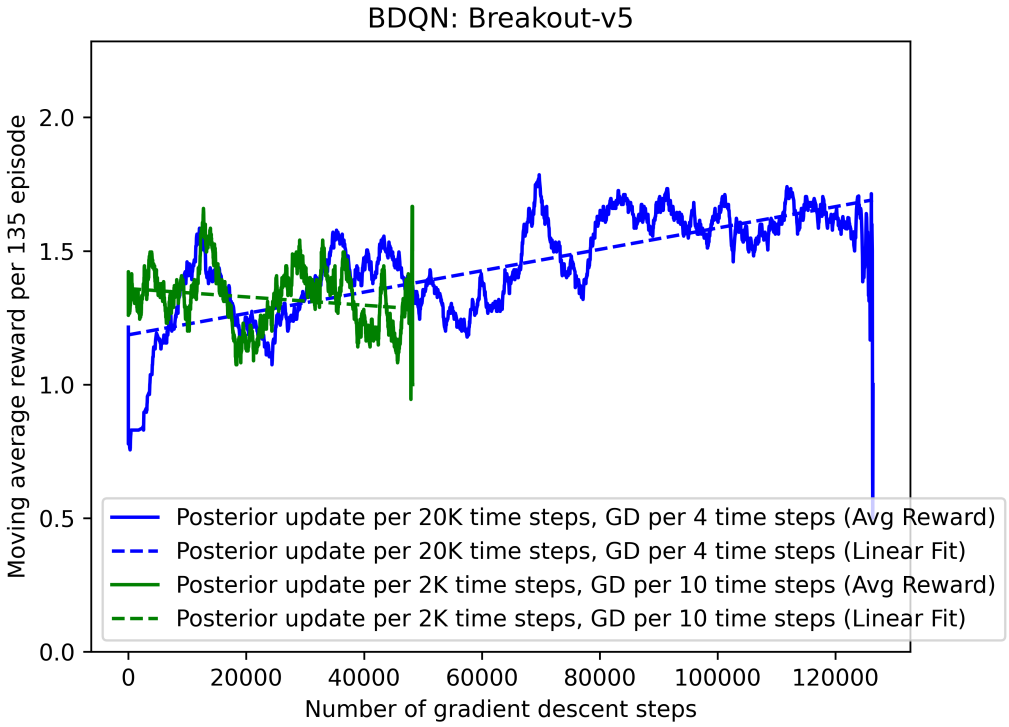
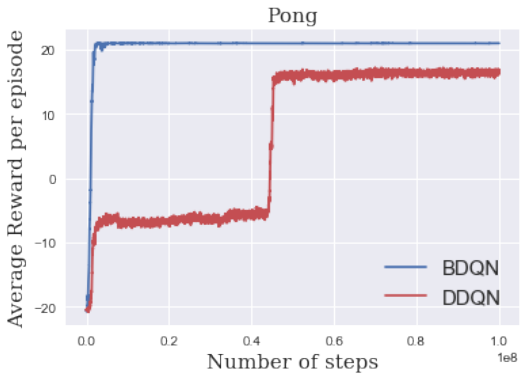



Figure 3: Learning at ***env2***

## 7 The state of the art

One of the main reasons we decided to follow (2) and (3) was that they performed well documented training sessions. In these experiments they achieved very good results, although only after training for far more episodes than we were able to.


(a) Results from (2), on the Breakout environment, plotted with our averaging algorithm.




(b) Results from (2), on the Pong environment.

Figure 4: Their model was able to produce fairly good results after around 1 million steps in both environments, and really good ones after 10 and 100 million steps. We have only ran our algorithms for at most a few million steps, however we couldn't come close to even an average reward of 20. We don't have any information about the exact hyperparameters used for these tests, which could be bring our algorithm to similar learning speeds as well.

In the (2) project, they compared the BDQN algorithm to DDQN, as it can be observed in their plots. They have found the BDQN to work better on Atari game environments. We have also found sources trying with the DQN algorithm (4).

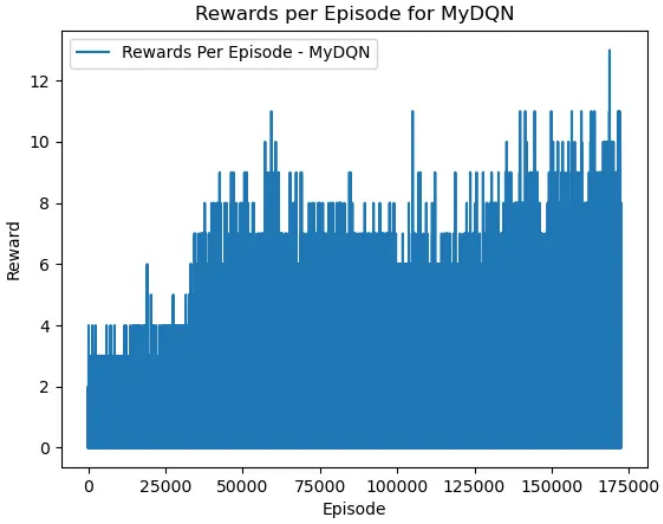



Figure 5: Results of (4) on the Breakout environment. Original caption: "MyDQN on Breakout after 10 million steps which corresponds roughly to 175000 episodes".

The DQN approach seem to have resulted in much less success than the more advanced techniques. This result is the most comparable to our own, as after 10 million steps of training, at around 5 times

more than what we were able to train, their maximal reward doesn't exceed what we observed in our experiments.

We have found trials with a much more complex neural network structures as well, which interestingly also use 4 stacked frames as an input.

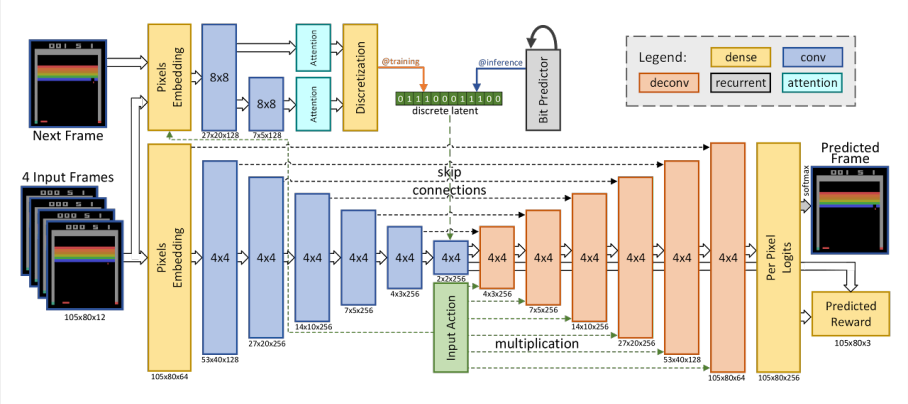



Figure 6: The neural network structure implemented by (5). They are using a Simulated Policy Learning algorithm and train their policy using the PPO algorithm.

Many papers refer to the Rainbow algorithm (6) as the "state of the art" when it comes to reinforcement learning on Atari game environments. As it combines many techniques in the field to achieve consistent and fast learning. Interestingly, the Breakout environment proves difficult for even this algorithm.

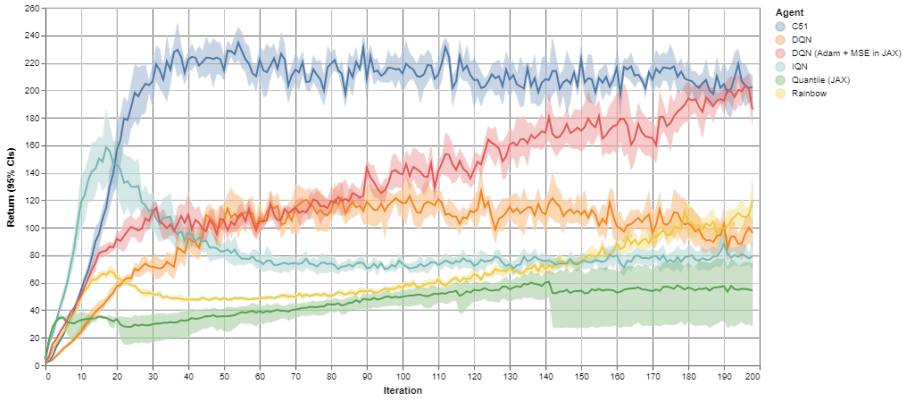



Figure 7: Plot from (7). Results on the Breakout environment. Each iteration stands for 250000 steps.

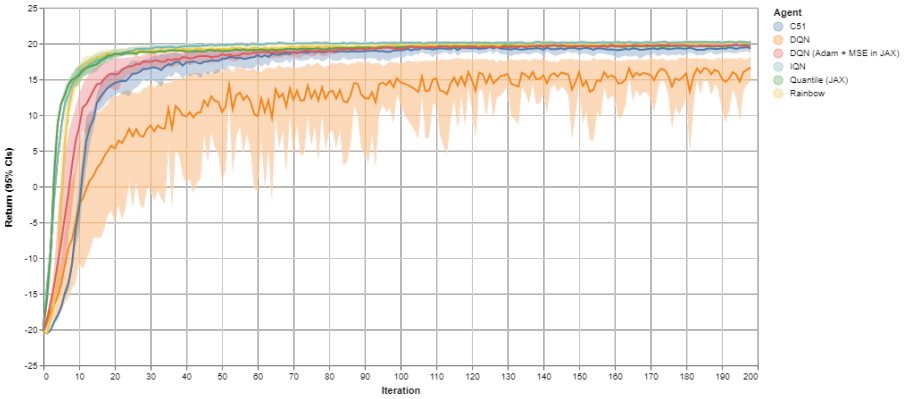



Figure 8: Plot from (7). Results on the Pong environment. Each iteration stands for 250000 steps.

These plots from (7) show the performance of several reinforcement learning algorithms on the Breakout (top) and Pong (bottom) environments. It is clear that for even the best algorithms, many millions of steps are required to achieve high rewards.

## References


[1] *Gymnasium Atari Environments* https://gymnasium.farama.org/environments/atari/

[2] *BDQN Repository* https://github.com/kazizzad/BDQN-MxNet-Gluon/blob/master/BDQN.ipynb

[3] *Efficient Exploration through Bayesian Deep Q-Networks* https://arxiv.org/pdf/1802.04412

[4] *Playing Atari games with Deep Reinforcement Learning and Attention* https://medium.com/@mboungoucolombe/playing-atari-games-with-deep-reinforcement-learning-and-attention-d83312fe4f29

[5] *Model-Based Reinforcement Learning for Atari* https://sites.google.com/view/modelbasedrlatari/home

[6] *Rainbow: Combining Improvements in Deep Reinforcement Learning* https://arxiv.org/pdf/1710.02298

[7] *Baseline plots* https://google.github.io/dopamine/baselines/atari/plots.html