Read Me

Nouman Khan

March 23, 2024

Note

The contents in this document up to (and including) Section 4 are not my own but rather a collection of selected excerpts mostly from [1] (slightly altered for more clarity).

1 Background

The gym environment `platform-v0` has three actions: run, hop, and leap - each with a continuous action parameter to control horizontal displacement. The agent has to hop over enemies and leap across gaps between platforms to reach the goal state. The agent dies if it touches an enemy or falls into a gap. A 9-dimensional state space gives the position and velocity of the agent and local enemy along with features of the current platform such as length. The environment fits well into the framework of *Parametrized Action Markov Decision Processes (PAMDPs).* A PAMDP is specified by the tuple $M = (\mathcal{S}, \mathcal{A}, P, R, \gamma)$. Here, $\mathcal{S}$ is the set of all states; $\mathcal{A}$ is the parameterised action space

$$\mathcal{A} = \bigcup_{k \in [K]} \left\{a_k = (k, x_k) \mid x_k \in \mathcal{X}_k\right\},$$

where each k has a corresponding continuous action-parameter $x_k \in \mathcal{X}_k \subseteq \mathbb{R}^{m_k}$ with dimensionality m_k; $P\left(s' \mid s, k, x_k\right)$ is the Markov state transition probability function; $R\left(s, k, x_k, s'\right)$ is the reward function; and $\gamma \in [0, 1)$ is the future reward discount factor. An action policy $\pi : \mathcal{S} \to \mathcal{A}$ maps states to actions, typically with the aim of maximising Q-values $Q(s, a)$, which give the expected discounted return of executing action a in state s and following the current policy thereafter.

2 Parameterised Deep Q-Networks

[2] has introduced the *P-DQN algorithm* which achieves state-of-the-art performance using a *Q-network* to approximate Q-values (used for discrete action selection), in addition to providing critic gradients for an *actor-network* that determines the continuous action-parameter values for all actions. The Bellman equation that needs to be solved for a PAMDP $M = (\mathcal{S}, \mathcal{A}, P, R, \gamma)$ is given by

$$Q\left(s, k, x_k\right) = \underset{r, s'}{\mathbb{E}}\left[r + \gamma \max_{k'} \sup_{x_{k'} \in \mathcal{X}_{k'}} Q\left(s', k', x_{k'}\right) \mid s, k, x_k\right]. \tag{1}$$

To solve (1), one must be able to quickly compute $\sup_{x_{k'} \in \mathcal{X}_{k'}} Q\left(s', k', x_{k'}\right)$. [2] notes that when the Q function is fixed, one can view $\operatorname{argsup}_{x_k \in \mathcal{X}_k} Q\left(s, k, x_k\right)$ as a function $x_k^Q : S \to \mathcal{X}_k$ for any state $s \in S$ and $k \in [K]$. This allows the Bellman equation to be rewritten as:

$$Q\left(s, k, x_k\right) = \underset{r, s'}{\mathbb{E}}\left[r + \gamma \max_{k'} Q\left(s', k', x_{k'}^Q\left(s'\right)\right) \mid s, k, x_k\right]. \tag{2}$$

To solve (2), P-DQN uses a deep *Q-network* with parameters θ_Q to represent $Q\left(s, k, x_k; \theta_Q\right)$ (an approximation of $Q\left(s, k, x_k\right)$), and a second deterministic *actor-network* with parameters θ_x to represent the action-parameter policy $x_k\left(s; \theta_x\right) : S \to \mathcal{X}_k$ (an approximation of $x_k^Q(s)$).

2.1 Loss Function for Q-network

With this formulation it is easy to apply the standard DQN approach of minimising the mean-squared Bellman error to update the Q-network using minibatches sampled from a replay memory D:

$$L_Q\left(\theta_Q\right) = \mathop{\mathbb{E}}_{(s,k,x_k,r,s')\sim D}\left[\frac{1}{2}\left(y - Q\left(s,k,x_k;\theta_Q\right)\right)^2\right], \tag{3}$$

where $y = r + \gamma \max_{k'\in[K]} Q\left(s',k',x_{k'}\left(s';\theta_x^-\right);\theta_Q^-\right)$ is the update target.

2.2 Loss Function for Actor-Network

The loss for the actor-network in P-DQN is given by the negative sum of Q-values:

$$L_x\left(\theta_x\right) = \mathop{\mathbb{E}}_{s\sim D}\left[-\sum_{k=1}^{K} Q\left(s,k,x_k\left(s;\theta_x\right);\theta_Q\right)\right]. \tag{4}$$

3 Problems with Joint Action-Parameters

The P-DQN architecture proposed by [2] inputs the joint action-parameter vector over all actions to the Q-network, as illustrated in Figure 1. This was pointed out by [2] but not discussed further. While this may seem

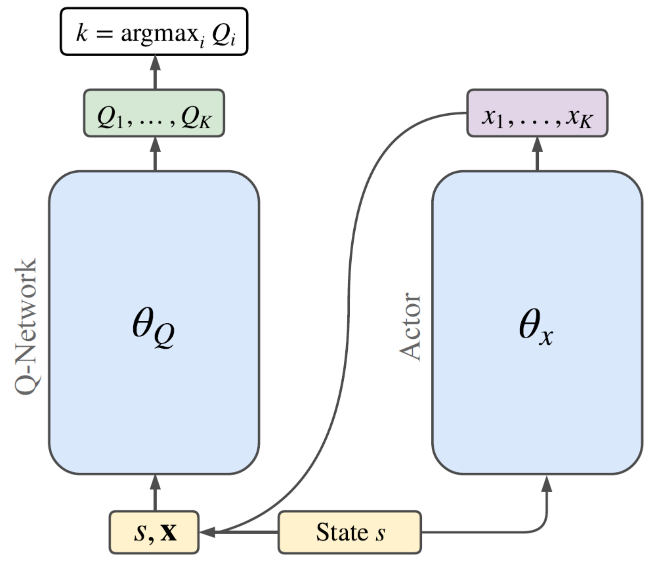

Figure 1: The P-DQN network architecture [Xiong et al., 2018]. Note that the joint action-parameter vector $\mathbf{x}$ is fed into the Q-network.

like an inconsequential implementation detail, it changes the formulation of the original Bellman equation ((2)) since each Q-value is a function of the joint action-parameter vector $\mathbf{x} = (x_1, \ldots, x_K)$, rather than only the action-parameter x_k corresponding to the associated action:

$$Q(s,k,\mathbf{x}) = \mathop{\mathbb{E}}_{r,s'}\left[r + \gamma \max_{k'} Q\left(s',k',\mathbf{x}^Q\left(s'\right)\right) \mid s,k,\mathbf{x}\right].$$

This in turn affects both the updates to the Q-values and the action-parameters.

3.1 False-Gradients

Firstly, we consider the effect on the action-parameters. Consider for demonstration purposes the action-parameter loss (Equation (4)) over a single sample with state s :

$$L_x(\theta_x) = -\sum_{k=1}^{K} Q(s, k, \mathbf{x}(s;\theta_x); \theta_Q).$$

The gradient for *actor-network* is then given by:

$$\begin{aligned}
\nabla_{\theta_x} L_x(\theta_x) &= -\sum_{k=1}^{K} \nabla_{\mathbf{x}} \underbrace{Q(s, k, \mathbf{x}(s;\theta_x); \theta_Q)}_{\triangleq Q_k} \nabla_{\theta_x} \mathbf{x}(s;\theta_x) \\
&= -\sum_{k=1}^{K} \begin{pmatrix} \frac{\partial Q_k}{\partial x_1} & \frac{\partial Q_k}{\partial x_2} & \cdots & \frac{\partial Q_k}{\partial x_K} \end{pmatrix} \begin{pmatrix} \nabla_{\theta_x} x_1(s;\theta_x) \\ \nabla_{\theta_x} x_2(s;\theta_x) \\ \vdots \\ \nabla_{\theta_x} x_K(s;\theta_x) \end{pmatrix} \\
&= -\sum_{k=1}^{K}\sum_{j=1}^{K} \frac{\partial Q_k}{\partial x_j} \nabla_{\theta_x} x_j(s;\theta_x). \qquad (5)
\end{aligned}$$

Theoretically, if each Q_k were a function of just x_k (as the P-DQN formulation intended), then $\frac{\partial Q_k}{\partial x_j} = 0$ for all $k, j \in [K], j \neq k$. So, $\nabla_{\theta_x} L_x(\theta_x)$ simplifies to:

$$\nabla_{\theta_x} L_x(\theta_x) = -\sum_{k=1}^{K} \frac{\partial Q_k}{\partial x_k} \nabla_{\theta_x} x_k(s;\theta_x).$$

However this is not the case in P-DQN (see (5)). This is a problem because each Q-value (Q_k for a given s) is supposed to be updated only when its corresponding action is sampled, and thus has no information on what effect other action-parameters $x_j, j \neq k$ have on transitions or how they should be updated to maximise the expected return. Thus, the gradients for θ_x computed by P-DQN contain *false-gradients.*[1]

3.2 Sub-optimal Greedy Action Selection

The dependence of Q-values on all action-parameters can also negatively affect the discrete action policy. Specifically, with state fixed to s, the Q-value for a given discrete action k should depend only on its own parameter x_k, but the Q-network of P-DQN inputs the entire joint action-parameter $\mathbf{x}_k$. Therefore, updating $x_k(s;\theta_x)$ also perturbs the Q-values of other actions (i.e., $\{Q_{k'} : k' \neq k\}$) This can lead to changes in the relative ordering of the actions' Q-values which in turn can result in *sub-optimal greedy action selection.* See Figure 2 in [1].

4 Multi-Pass Q-Networks

The naive solution to the problem of joint action-parameter inputs in P-DQN would be to split the Q-network into separate networks for each discrete action. Then, one can input only the state and relevant action-parameter x_k to the network corresponding to Q_k. However, this would drastically increase the computational and space complexity of the algorithm due to the duplication of network parameters for each action.

[1] therefore considers an alternative approach that does not involve architectural changes to the network structure of P-DQN. While separating the action-parameters in a single forward pass of a single Q-network with fully connected layers is impossible, we can do so with multiple passes.

[1]This effect may be somewhat mitigated by the summation over all Q-values in the action-parameter loss, since the gradients from each Q-value are summed and averaged over a mini-batch.

1. We perform a forward pass once per action k with the state s and action-parameter vector $\mathbf{x} \star \mathbf{e_k}$ as input, where $\mathbf{e}_k$ is the standard basis vector for dimension k ($\star$ indicates element-wise multiplication). Thus $\mathbf{x} \star \mathbf{e_k} = (0, \ldots, 0, x_k, 0, \ldots, 0)$ is the joint action-parameter vector where each $x_j, j \neq k$ is set to zero. This causes all false gradients to be zero, $\frac{\partial Q_k}{\partial x_j} = 0$, making Q_k only depend on x_k. That is,

$$Q\left(s, k, \mathbf{x} \star \mathbf{e_k}\right) = Q\left(s, k, x_k\right).$$

   Both problems are therefore addressed without introducing any additional neural network parameters. We refer to this as the multi-pass Q-network method, or MP-DQN.

2. A total of K forward passes are required to predict all Q-values instead of one. However, we can make use of the parallel mini-batch processing (provided by libraries such as PyTorch and Tensorflow), to perform this in a single parallel pass, or multi-pass. A multi-pass with K actions is processed in the same manner as a mini-batch of size K:

$$\begin{pmatrix} Q\left(s, \cdot, \mathbf{x} \star \mathbf{e_1}; \theta_Q\right) \\ \vdots \\ Q\left(s, \cdot, \mathbf{x} \star \mathbf{e_K}; \theta_Q\right) \end{pmatrix} = \begin{pmatrix} Q_{11} & Q_{12} & \cdots & Q_{1K} \\ \vdots & \vdots & \ddots & \vdots \\ Q_{K1} & Q_{K2} & \cdots & Q_{KK} \end{pmatrix},$$

   where Q_{ij} is the Q-value for action j generated on the i^{th} pass where x_i is non-zero. Only the diagonal elements Q_{ii} are valid and used in the final output $Q_i \leftarrow Q_{ii}$. This process is illustrated in Figure 2.

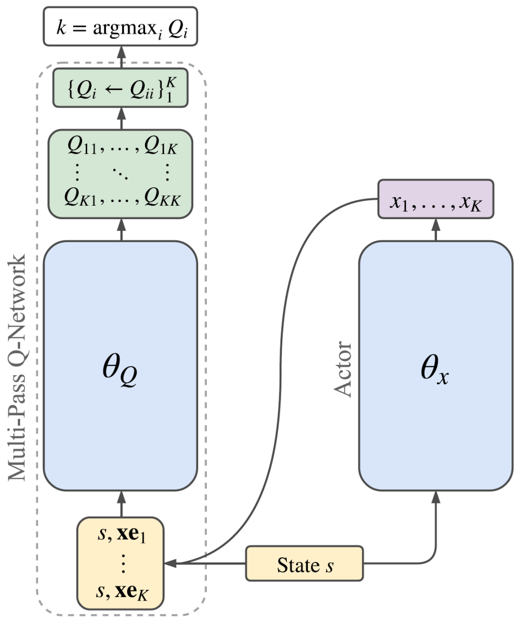
$k = \text{argmax}_i\, Q_i$
$\{Q_i \leftarrow Q_{ii}\}_1^K$
$Q_{11}, \ldots, Q_{1K}$ ⋮ ⋱ ⋮ $Q_{K1}, \ldots, Q_{KK}$
$x_1, \ldots, x_K$
Multi-Pass Q-Network
θ_Q
Actor
θ_x
$s, \mathbf{xe}_1$ ⋮ $s, \mathbf{xe}_K$
State s

Figure 2: Illustration of the multi-pass Q-network architecture.

5 Experiments

The source code for implementation of P-DQN and MP-DQN (and few other algorithms) was already available online at `https://github.com.cycraig/MP-DQN`. [1] has presented their results on evaluation of these algorithms in Table 1 of their paper, where P-DQN and MP-DQN performed exceptionally well compared to other algorithms.[2]. The source code was made use of and refactored to keep it in line with the presentation given in this document. Docstrings have been added to make the code's exposition clearer for anyone reading it. Also,

[2]Reason behind me implementing these two algorithms

Table 1: Mean (Undiscounted) Return of Trained Agents over 10 simulation runs (each with a different seed and consisting of 1000 episodes).

Agent	Mean (Undiscounted) Return
P-DQN	0.9812
MP-DQN	0.9996

importantly, the deep neural-network architecture of the actor-network has been changed to a MMoE (multi-gate mixture of experts) model (see [3]). I have implemented MMoE before a couple of times and have so far seen it to perform much better than other multi-task learning architectures.

Below are a few highlights of the implementation.

1. **Target Networks**: As standard practice in DQN, target-networks were added for both the Q-network and the actor-network. The two target networks were used to create a stable target for the Q-network.
2. **Soft Polyak averaging**: The target networks used soft polyak averaging.
3. **Adam Optimizers**: Adam Optimizers were used instead of SGD; in my experience they perform much better than SGD.
4. **Leaky Rectified Linear Units (Leaky ReLU)**: Leaky ReLUs were used (instead of ReLU) to avoid the *dead ReLU problem.*
5. **Inverting Gradients Approach**: Inverting Gradients approach as prescribed in [4] was used to ensure that action-parameters lie in their permissible ranges.
6. **Scaled Action-parameters**: Action parameters were scaled to $[-1, 1]$ for better learning.
7. **Ornstein Noise**: Ornstein-Uhlenback noise was added to (continuous) action-parameters for exploration. Parameters were $\mu = 0., \theta = 0.15, \sigma = 0.0001$.

   $$X(t+1) \leftarrow X(t) + \theta(\mu - X(t)) + \sigma W(t+1)$$

   where each $W(\cdot)$ has entries drawn from $N(0, 1)$ distribution.
8. **Gradient Clipping**: Gradients were clipped to avoid *exploding gradients.* Threshold set to 10.
9. **L_1-Smooth loss**: L_1 smooth loss was used instead of mean-squared-error loss for the Q-network. [1] mentions it was found to perform well for this problem setting.

5.1 Results

Similar to [1], I trained P-DQN and MP-DQN agents on this domain for 80,000 episodes. The two agents were then run against 10 simulations - each with a different seed and consisting of 1000 episodes. The mean returns from all the simulations were then averaged to get an estimate of the agent's performance.

The learning curves during training for the two agents are shown in Figure3 and their mean (undiscounted) returns evaluated over the 10 simulation-runs are listed in Table 1

5.2 Running Shared Code

I will assume the reader is in the `platform` directory of the shared source code. A virtual environment (for Windows operating system) is present in `platform_venv` folder and can be used to run the shared code. The folder `platform_script` contains batch-scripts, `train_agent_bat.sh` and `test_agent_bat.sh` which can be used to train and evaluate an RL agent. High level guidelines for training and evaluating an agent are given below:

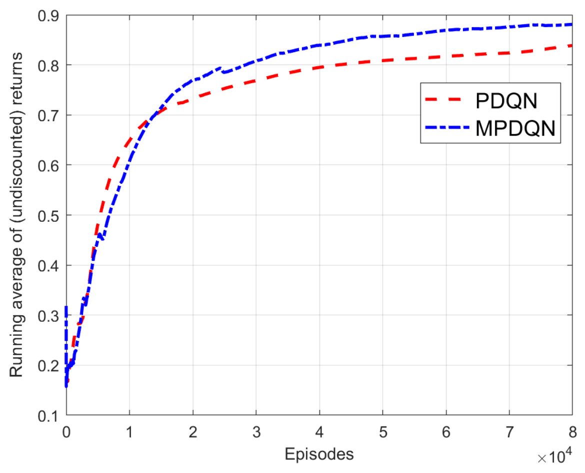

Figure 3: Learning curves of Agents.

5.2.1 Training the Agent

1. Go to `platform\script` directory and open `train_agent_bat.bat` in an editor (Notepad++ for example). Change the training parameters and options to the ones you desire. Ensure that `check_point` variable is set to "None".

2. Open Windows Power Shell and make sure the current working directory is `platform\script`. Then run the command `.\train_agent_bat.bat`.

   - The script will call the virtual environment in `platform\venv` and then run the python file `run_platform_padqn.py`. The code in `run_platform_padqn.py` will train an agent for the set number of episodes and then evaluate it on the set number of evaluation-episodes. During evaluation, `epsilon` will be set to zero and no noise will be used for the continuous action-parameters.
   - Model files for the trained agent will be generated inside the results directory (see code for exact location).

5.2.2 Evaluating a Trained Agent

1. Open Windows Power Shell and make sure the current working directory is `platform\script`. Ensure that the `check_point` variable in `.\test_agent_bat.bat` points to the model file you wish to evaluate and the `episodes` variable is set to 0. Then run the command `.\test_agent_bat.bat`.

2. The script will call the virtual environment in `platform_venv` and then run the python file `run_platform_padqn.py` (this time for evaluation only).

5.3 Implementation Details

Here, I am listing the arguments used in calling the python file `run_platform_padqn.py` and a short important part of the code in the form of a pseudocode.

1. `evaluation episodes`:
2. `episodes`:
3. `batch_size`:
4. `gamma`:
5. `replay_memory_size`:

6. `initial_memory_threshold`:
7. `epsilon_initial`:
8. `epsilon_steps`:
9. `epsilon_final`:
10. `alpha_actor`:
11. `alpha_param_actor`:
12. `tau_actor`:
13. `tau_param_actor`:
14. `loss`:
15. `use_ornstein_noise`:
16. `scale_actions`:
17. `clip_grad`:

Algorithm 1: Outline of Training Workflow.

```
foreach episode $i$ do
    $s \leftarrow$ env.reset().
    $a, x_k, \mathbf{x} \leftarrow$ agent.act(s).
    foreach step $j$ do
        $s', \_, r, terminal, \_ \leftarrow$ env.step($a, \mathbf{x} \star \mathbf{e_k}$).
        $a', x'_k, \mathbf{x}' \leftarrow$ agent.act($s'$).
        agent.step($s, (a, \mathbf{x}), r, s', (a', \mathbf{x}'), terminal, \_$).
        $s \leftarrow s'$.
        $a, x_k, \mathbf{x} \leftarrow a', x'_k, \mathbf{x}'$.
```

Algorithm 2: Initial Instantiations of RL Agent.

```
Initialize initial memory object $D$ with next_actions set to False.
Initialize weights in $\theta_Q$ and $\theta_x$. Ensure weights of output layer of $\theta_x$ are sufficiently small.
$\theta_Q^- \leftarrow \theta_Q$.
$\theta_x^- \leftarrow \theta_x$.
Keep $\theta_Q^-$ and $\theta_x^-$ in eval mode.
$A_Q$ = AdamOptimizer($\theta_Q, \alpha_Q$).
$A_x$ = AdamOptimizer($\theta_x, \alpha_x$).
```

Algorithm 3: Pseudocode for `agent.act(s)`.

```
with torch.no_grad()
    $\mathbf{x} \leftarrow \mathbf{x}(s; \theta_x)$.
    Sample $y \sim Unif(0, 1)$.
    if $y < \epsilon$ then
        $a \leftarrow$ random.choice({0, 1, 2}).
    else
        $Q \leftarrow Q(s, \mathbf{x}; \theta_Q)$ or $Q \leftarrow [Q(s, \mathbf{x} * \mathbf{e}_k; \theta_Q)]_{k=0}^{2}$.
        $a \leftarrow \arg\max\{Q(s, \mathbf{x} * \mathbf{e}_k; \theta_Q)\}_{k=0}^{2}$
/* Add noise to $x_k$.                                              */
$x_k \leftarrow x_k + z_k$ where $z_k$ is $k^{th}$-entry of Ornstein noise or Gaussian noise.
return $k, x_k, \mathbf{x}$.
```

Algorithm 4: Pseudocode for `agent.step`$(s, a, r, s', a', terminal, steps)$.

```
1 $k, \mathbf{x} \leftarrow a$.
  /* Add sample to replay memory $D$. */
2 $D \leftarrow D \cup \{(s, [k, x_1, \ldots, x_K], r, s', [k', x'_1, \ldots, x'_K])\}$.
3 if $\#$ of steps $\geq \max(batch\_size, initial\_memory\_threshold)$ then
4     agent.optimize_td_loss().
5     agent.updates $+= 1$.
```

Algorithm 5: Pseudocode for `agent.optimize_td_loss()`.

```
1  $S, A, R, S', Terminals \leftarrow$ batch of samples with $B$ samples.
   /* convert to tensors first. */
2  $K \leftarrow A[:, 0]$.
3  $X \leftarrow A[:, 1:]$.
   /* now optimize $Q$-network. */
4  with torch.no_grad()
5      $X^- \leftarrow \mathbf{x}(S; \theta_x^-)$.
6      $Q^- \leftarrow Q(S, X^-; \theta_Q^-)$.
7      $Q^- \leftarrow \max(Q(S, X^-; \theta_Q^-))$ computed along columns/actions.
8      $Y \leftarrow R + (1 - Terminals)\gamma Q^-$.
9  $Q \leftarrow Q(S, X; \theta_Q)$.
10 $\widehat{Y} \leftarrow Q(:, A[:, 0])$.
11 Compute $L(\theta_Q)$ based on $Y$ and $\widehat{Y}$.
12 $\nabla_{\theta_Q} \leftarrow \nabla_{\theta_Q} L(\theta_Q)$.
13 $\nabla_{\theta_Q} \leftarrow$ clip_gradients$(\nabla_{\theta_Q})$.
14 $\theta_Q \leftarrow \theta_Q - \alpha_Q(\nabla_{\theta_Q})$.
   /* now optimize actor-network */
15 with torch.no_grad()
16     $\widehat{X} \leftarrow \mathbf{x}(S; \theta_x)$.
17 $\widehat{X}$.requires_grad = True.
18 $\widehat{Q} \leftarrow Q(S, \widehat{X}; \theta_Q)$.
19 mean_return $\leftarrow$ mean(sum($\widehat{Q}$, 1)).
20 Run mean_return.backward() to compute $\nabla_{\widehat{X}}$mean_return.
21 $grad =$ deepcopy$(\nabla_{\widehat{X}})$. $X \leftarrow \mathbf{x}(S; \theta_x)$.
22 $grad \leftarrow$ invert_gradients$(grad, X)$.
23 $out \leftarrow grad * X$.
24 Run out.backward$(v = torch.ones_like(out))$ to compute $\nabla_{\theta_Q}$.
25 $\nabla_{\theta_Q} \leftarrow$ clip_gradients$(\nabla_{\theta_Q})$.
26 $\theta_x \leftarrow \theta_x - \alpha_x \nabla_{\theta_x}$.
   /* soft updates of target networks */
27 $\theta_Q^- \leftarrow \tau_Q \theta_Q + (1 - \tau_Q)\theta_Q^-$.
28 $\theta_x^- \leftarrow \tau_x \theta_Q + (1 - \tau_x)\theta_x^-$.
```

Algorithm 6: Pseudocode for `agent.invert_gradients(grad, vals)`.

```
1 assert grad.shape == vals.shape.
2 with torch.no_grad()
3     idx = grad > 0.
4     $grad[idx] = grad[idx] * (idx.float() * \frac{1 - vals}{2}[idx]$.
5     $grad[\sim idx] = grad[\sim idx] * (\sim idx.float() * \frac{vals + 1}{2}[\sim idx]$.
6 return grad
```

Algorithm 7: Pseudocode for forward pass of MultiPass Q-Network.

```
   Input:
      $S_{B\times 9}$
      $X_{B\times K}$
 1 $Z \leftarrow [S, 0_{B\times K-1}]$.
 2 $Z \leftarrow [\underbrace{Z^T, Z^T, \dots, Z^T}_{K \text{ times}}]^T$.
 3 foreach $a \in \{0, 1, \dots, K\}$ do
 4     $Z[aB : (a+1)B, 9+a : 9+(a+1)] = X[:, a : (a+1)]$.
 5 $Q_{all} \leftarrow Q(S, X; \theta_Q)$.
 6 $Q \leftarrow []$.
 7 foreach $a \in \{0, 1, \dots, K-1\}$ do
 8     $Q_a \leftarrow Q_{all}[aB : (a+1)B]$.
 9     $Q$.append($Q_a$).
10 $Q \leftarrow$ torch.cat($Q, dim = 1$).
```

References

[1] C. J. Bester, S. D. James, and G. D. Konidaris, "Multi-pass q-networks for deep reinforcement learning with parameterised action spaces," *CoRR*, vol. abs/1905.04388, 2019.

[2] J. Xiong, Q. Wang, Z. Yang, P. Sun, L. Han, Y. Zheng, H. Fu, T. Zhang, J. Liu, and H. Liu, "Parametrized deep q-networks learning: Reinforcement learning with discrete-continuous hybrid action space," 2018.

[3] J. Ma, Z. Zhao, X. Yi, J. Chen, L. Hong, and E. H. Chi, "Modeling task relationships in multi-task learning with multi-gate mixture-of-experts," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18, (New York, NY, USA), p. 1930–1939, Association for Computing Machinery, 2018.

[4] M. Hausknecht and P. Stone, "Deep reinforcement learning in parameterized action space," *CoRR*, vol. abs/1511.04143, 2016.