Fundamentals of Machine Learning

Final Project Report

Cem Daloglu, Nida Murad, Shourya Verma

Bomberman RL Team - BetaBomber

Github: `https://github.com/cemdaloglu/FML_project`

Heidelberg University

Institute of Computer Science

September 12, 2022

Contents

1 Contributions

1.1 Cem Daloglu

Researched on Q-learning and Q-table methods. Implemented the Q-table model's feature extraction and training. Also worked on custom events used by both Q-table and deep Q-learning methods. Wrote the Introduction, Q-table subsections of each section, and Conclusion of the report.

1.2 Nida Murad

Worked on Q-learning with linear gradient descent. Initially estimated an optimal action-value function using lookup tables. The number of the possible game states in Bomber-man, however, necessitates a pretty good approximation. So, we used linear function approximation in combination with custom-built features to get this estimate as near to the true Q values as possible. Tried using experience replay along with batch gradient descent to minimize divergence. Wrote the Introduction, Q-learning with linear gradient descent subsections of each section, and Conclusion of the report.

1.3 Shourya Verma

Researched on deep reinforcement learning methods and implemented three deep Q-learning agents which included a vanilla linear fully connected model, gated recurrent unit model, and long short-term memory model. Compared the performance of all the different deep Q-learning agents. Wrote the Introduction, Deep Q-learning subsections of each section, and Conclusion of the report.

2 Introduction

This report talks about the task of creating different reinforcement learning agents to play the game Bomberman. Bomberman is a competitive multiplayer action game where the environment is fully observable, but the strategies of the agents are not, a concept similar to chess. The game-play involved strategically placing down bombs, which explode in multiple directions after a certain amount of time, in order to destroy crates and kill other players and possibly self destruct. There was an added ability to collect coins which would randomly appear on the board inside the crates. Collecting the coins would give the player a point.

In this project, we attempted to develop agents which plays the game of Bomberman as efficiently as possible using various techniques. The method we implemented was off-policy Q-learning, with numerical rewards. The initial traditional approach in using Q-learning is using Q-tables to estimate an optimal action-value function, which denotes the expected reward Q(s,a) after taking action a in state s. Secondly, we built a Q-learning agent with linear function approximation. Due to the size of the possible game states in Bomberman, a reasonable feature extraction is required for both models. As the last approach, we did not want to deal with user curated features and hence implemented three different agents which uses Deep Q-learning. The complete code from our team can be found at:

https://github.com/cemdaloglu/FML_project

3 Background of Methods

In this section, firstly Q-learning with Q-table, secondly linear function approximation, and lastly deep Q-learning methods will be discussed. Since the first and the second methods both take the advantage of Q-learning, the following subsection will discuss how the Q-learning works. The math behind the Q-learning will be discussed in the subsequent subsection. Additionally, how each method uses this learning algorithm will also be discussed.

3.1 Q Table

Q-learning is an off-policy reinforcement learning algorithm that aims to find the best action in the current state. A learning algorithm is considered as off-policy when a model can learn which action to take from the data collected by any kind of behavioral policy [1]. Q-learning is an off-policy because the learning function learns from actions that are outside the current policy, like taking random actions, and therefore a policy is not needed. In other words, Q-learning aims to learn how to act according to the maximization of total reward.

In order to perform Q-learning, a lookup table or a Q-table with the same size as the number of states and actions should be constructed and the values should be initialized. Q-table is a reference table for the agent to take the best action in the given state based on the Q-value. Initialized values are dependant to many factors, such as rewards, hyperparameters, and the task. Thus, initial values should be decided by the programmer. Afterwards, Q-values, which refers to the values in the Q-table, are updated regarding to the learning algorithm, which will be discussed in the subsequent paragraphs, and stored after each step.

After constructing the Q-table, an agent is ready to experience the environment and make updates to the state-action pairs in the Q-table. Environment can be experienced in two ways. Either using the Q-table to see all possible actions for a given state and picking the action with the highest Q-value, which is known as exploiting, or action can be chosen randomly, which is known as exploring [2]. Exploring is a crucial part of the learning because it allows the agent to discover different actions' rewards in the current state. In the exploitation process, agent hardly experience various actions depending on the rewards and hyper-parameters. Exploration and exploitation ratio can be balanced via epsilon (ϵ) greedy methods and one can control the frequency of how often the agent would explore or exploit.

The updates of Q-table occurs after each step or action and ends and stored when an episode is done. The agent will not learn much after a single episode, but eventually with enough exploring (steps and episodes) Q-values will converge and Q-table will be it's optimum condition. To summarize [3];

a) Agent starts in a state takes an action and receives a reward.

b) Agent selects action by referencing Q-table with highest Q-value (exploitation) or by random (exploration).

c) Update Q-values.

d) Repeat until the end of game and save the latest version of Q-table.

Update rule for Q-learning can be seen in the below equation;

$$\begin{aligned} Q[state, action] =& Q[state, action] + \\ & \alpha * \Big(reward + \gamma * max\,(Q[new\ state, all\ actions]) - Q[state, action]\Big)[2] \end{aligned} \tag{1}$$

To put the equation 1 another way, new state's Q-value is shared out to current actions to help the agent to select the best return action at any given state. Q-values are adjusted based on the difference between the discounted new values and the old values. New values are discounted by using the discount factor (γ) and step size is adjusted by using the learning rate (α). Variable explanations of the equation 1 can be found below.

Learning Rate (α)**:** This variable can be seen as how much one decides the amount of change of the current state. Reward of the action for the old state is added with the subtraction of the new and old Q-values, the result can increase or decrease according to the contribution of the action taken in the old state. This value then gets multiplied with the learning rate to decide the impact of the contribution that one wants to add to the last action for the old state [4].

Discount Factor (γ)**:** This variable is used to balance the current and future reward. The update rule for the .q-learning algorithm, discount factor is applied to the future reward [4][5].

reward: It is the value received after completing a certain action at a given state. A reward can happen at any given step [4][5].

max(Q[new state, all actions]): This means taking the action with the highest Q-value of the new state. This is a significant part of the Q-learning because the current action is influenced by the future state [5].

We took advantage of some code repositories to comprehend the idea of Q-learning algorithm with Q-table [6]. In brief, Q-learning with Q-table is a reinforcement learning method, which agent takes it's actions regarding to the Q-values of it's current state's actions. In order this method to work, states of the task (Bomberman game for this case) must be compatible with the Q-table's states. Since the Bomberman game has numerous states, we applied feature extraction to the game states, which will be discussed elaborately in further sections.

3.2 Q Learning with Linear Function Approximation

One of the other methods we have tried is, Q-learning using Linear Function Approximation and named the agent as `Bombi_Agent`

This game of Bomberman follows the Markov Decision Process model , where the game state is fully observable but not the strategies of the opponent. It is a model for fully-observable sequential decision making problems in stochastic environments [7].

Let us recall some basic definitions, for easier understanding. We assume, let S be a finite set of states and A be a finite set of actions. The expected reward while transitioning from the state $s \in S$ to a new state $s' \in S$ after completing the action $a \in A$ is denoted by the reward function r : $S \times$ A $\times S \rightarrow \mathbb{R}$

A Q-value $Q(s,a)$ reflects the expected sum of rewards gained after completing action a in state s for each state-activity pair. The likelihood of terminating in state s after selecting action a in state s is given by the transition function P $(s,\ a,\ s')$ [7]. The Q-learning algorithm attempts to directly estimate the optimal action-value function Q^* and can be understood as a sample-based, approximate version of value iteration that produces a sequence of action-value functions $(Q_k$; k $\geqslant 0)$.

It is an off-policy method with an updating rule which means that the algorithm evaluates and improves a target policy that is different from the observational policy used to generate the data, where policy is nothing but a mapping between states and actions.$(\pi : S \rightarrow A$) [8]

source: https://en.wikipedia.org/wiki/Q-learning

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)}^{\text{temporal difference}}$$

The learning rate parameter or α [0, 1] or step size determines to what extent newly acquired information overrides old information whereas, γ ϵ [0, 1] the discount factor determines the importance of future rewards [7]. We generally take $\gamma = 1$ when the MDP is finite.

Q-Learning directly approximates Q^* independent of the policy being followed. Note that for the method to converge to the optimal Q^*, it is required that all pairs $Q(s,\ a)$ continue to be updated. An optimal policy is then given by [9]:

$$\pi(s) = arg_a max Q^*(s,a)$$

We initially use lookup tables to estimate an optimal action-value function, which denotes the expected reward *Q(s, a)* after taking an action a in state *s*. However, using the lookup tables to get the optimal action-value function becomes hard.

We know that the size of the state space S is significant. Every state uses its own *Q-value* for every action and since it needs to explore all actions in all states before being able to conclude which action is the best in a specific space, look-up tables become a bit problematic. We now address the problem of control in Q-learning with function approximation, where the optimal Q-function, Q^*, cannot be represented exactly and, therefore, some form of approximation must be used [10]. Hence, we have chosen linear function approximation. We need the right features along with linear approximation, which requires extensive feature engineering. Here, we have defined just 8 features, ideally to optimise it further many more features should be added. So that features can be added easily in the future we have made a separate class called feat_ex

We then use a weight vector to approximate the Q value with the extracted features,

$$Q(s,a) \approx w_0 + F_1(s,a)w_1 + F_2(s,a)w_2 + \cdots + F_n(s,a)w_n =: Q_w(s,a)$$

with F : $S \times A \to \mathbb{R}^n$ the feature extraction function. Assuming we have far more states than weights, improving the accuracy of one state's estimate necessarily degrades the accuracy of others. In this scenario, employing gradient descent to reduce error on observed samples is a good strategy. We don't know the true value of Q, thus we have to rely on the current value of the weight vector wt at time step t. As a result, we have a bias rather than a proper gradient descent method for the approximation $Q_w t$. The following is the update rule:

$$\begin{aligned} w_{t+1} &= w_t + \alpha_t[r_{t+1} + \gamma \max_{a'} Q_{w_t}(s_{t+1}, a') - Q_{w_t}(s_t, a)]\nabla_w Q_{w_t}(s_t, a_t) \\ &= w_t + \alpha_t \delta_{t+1}(Q_{w_t}) \cdot Q_{w_t}(s_t, a_t) \end{aligned}$$

Algorithm for Linear Function Approximation is as follows[11],

function QLearningLinFApp($X, A, R, Y, w, F, \alpha, \gamma$)
Input: X is the last state, Y is the next state, R the immediate reward associated with this transition, $w \in \mathbb{R}^d$ the parameter vector, $F(S, A)$ the feature extraction, α the learning parameter, and $\gamma \in [0, 1]$ the discount factor.

1. $\delta \leftarrow R + \gamma \cdot \max_{a' \in \mathcal{A}} w^T F(Y, a') - w^T F(X, A)$
2. $w \leftarrow w + \alpha \cdot \delta \cdot F(X, A)$
3. **return** w

3.3 Deep Q learning

Considerable amount of research has been done focusing on combining reinforcement learning and deep learning which has given rise to deep reinforcement learning (DRL) [12]. Deep learning methods are based on using artificial neural networks with representation learning. One of the popular application of DRL in games is AlphaGo [13]. The goal of DRL is to construct networks that are less dependent on user interpretation of the problem and do not require feature engineering to assist the learning process. Q learning [14] is a typical reinforcement learning method. In Q learning, an optimal action policy is obtained after learning an action value function. Using this method in DRL gives rise to terms like Deep Q-learning and Deep Q-Networks [15]. Deep Q-Learning (DQL) replaces the regular Q-table with a neural network. Rather than mapping a state-action pair to a q-value, a neural network maps input states to (action, Q-value) pairs. This approach is of interest, as it does not depend on user specified input to predict how to construct state from the observations, but learns this on its own. Figure 1 shows an example of a deep Q-learning architecture [16].

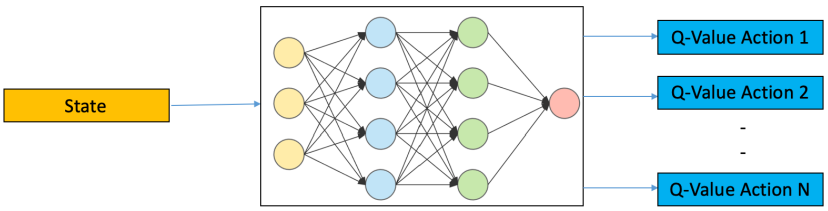
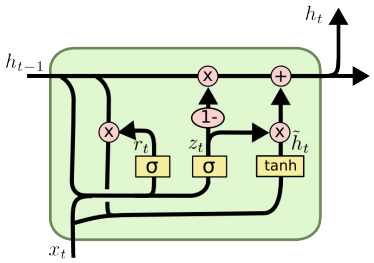


Figure 1: Deep Q-Learning Architecture

A recurrent neural network (RNN) are artificial neural networks whose connections between nodes form a directed or undirected graph along a temporal sequence [17]. This allows it to exhibit temporal dynamic behavior. Derived from feed-forward neural networks, RNNs can use their internal state (memory) to process variable length sequences of inputs. They are increasingly getting popular to build reinforcement learning agents where the model is given hidden states to improve predictions when all the direct observations do not contain information necessary to make suitable predictions.

3.3.1 LSTM Network

The Long Short Term Memory (LSTM) [18] is an artificial RNN architecture used in the field of deep learning. Unlike standard feedforward neural networks, LSTM has feedback connections. It can process not only single data points, but also entire sequences of data. An LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell remembers values over arbitrary time intervals and the three gates regulate the flow of information into and out of the cell. Figure 2 shows an example of an LSTM Network [19].



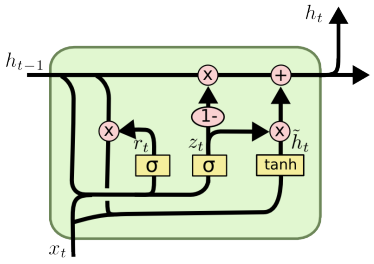
$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figure 2: LSTM Network

Bakkar et. al [20] presented RL with LSTM neural networks using advantage learning and directed exploration to solve non-Markovian tasks with long-term dependencies between relevant events. They demonstrated the performance of an LSTM network in a T-maze task, as well as in a difficult variation of the pole balancing task. Goulart et. al [21] used an LSTM network with proximal policy optimization to play Bomberman through DRL and imitation learning. They investigated the influence of five vector-based state representation and four learning algorithms to make an agent learn how to play Bomberman.

3.3.2 GRU Network

Gated Recurrent Units (GRU) are a gating mechanism in RNN [22]. The GRU is like an LSTM with a forget gate, but has fewer parameters than LSTM, as it lacks an output gate. It aims to solve the vanishing gradient problem which comes with a standard RNN. Figure 3 shows an example of a GRU Network [19].



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figure 3: GRU Network

Xiang Gao [23] used GRU based agents to play a time-series based reinforcement learning game, which achieved the best performance compared to multi-layer, LSTM and CNN networks. This inspired us to use a GRU network to compare the LSTM network we initially implemented.

3.3.3 Fully Connected Network

A fully connected (FC) linear network is a multi-layer neural network used for regression. it takes the features provided to it as input and allows the network to guide the agent to take the best estimate of the optimal action given each input state. The complete neural network provides an estimate of the Q-function from the input features. The final layer then

gives estimated utility of opting different actions. Based on the utilities, the agent decides to take the optimal action, or with a small probability of exploring by taking a random action which is dependent on the epsilon value. Our FC agent consisted of 3 linear layers with rectified linear activation function (ReLU) for the hidden layers and softmax activation for the final layer. Figure 4 shows the example of a fully connected network [24].

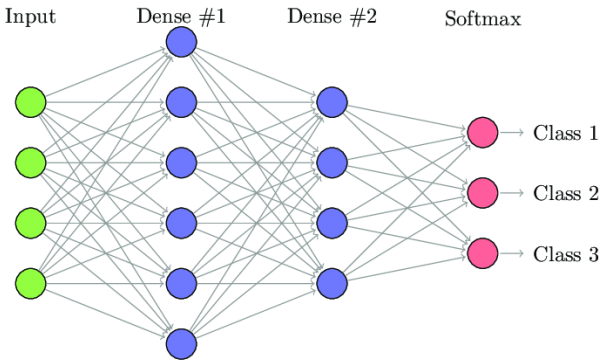


Figure 4: Fully Connected Network

This linear FC agent was used as vanilla DRL to compare the LSTM and GRU agents. LSTM and GRU agents also had two linear fully connected layers to make the Q action value predictions. The specific input and output shapes of each layer are mentioned in the training section of the report which also mentions different hyper-parameters used for the agents.

4 Feature Extraction

The Bomberman game contains numerous states, since the game state is fully observable, as we discussed in the Introduction section. Therefore, it is computationally difficult to construct a lookup table or a Q-table for all the states. However, the agent does not need to know most of the information given in the game state. The features which we have decided to be important are extracted from the raw data (game state).

4.1 Q Table

States are represented by 17 features, specifically, there are four core features for each direction (left, right, up, down) and a feature to check if the agent can drop a bomb or not. Those core features are represented by integer arrays with four elements and every direction is searched for these features separately. Afterwards, four directions' arrays and the agent's bomb dropping capability feature are concatenated and an integer array that contains only zeros and ones (it can be called a Boolean array) with size 17 is obtained. Each element can either have the value 0 or 1, and so each direction has 2^4 states. After the feature extraction, total number of states decreased to 2^{17} which is computationally manageable (Q-table file size became less than 10 Mb). Features will be explained with respect to their index in the

integer array.

4.1.1 Coin Feature

The first core feature, and also the first element of the feature array, is checking if the observed location has a coin or not. The agent only looked at the one block or tile away distance of it's current location. In other words, if the agent is one tile away from a coin towards the left direction, the first element of the feature array for the left direction becomes 1 and other direction arrays' first element becomes 0. After some training, we observed that the agent was not able to collect many coins due to its bounded vision. Then, our agent's range for searching coins is changed from one tile to four tiles. After the modification, agent is able to know if there exists a coin within the three tiles in the searched direction. By doing so, agent's coin collecting skills are increased, which will be discussed in the Results section of this report. The only problem of this feature is, it cannot distinguish if the coin is one or more tiles away from the current location. For instance, if the agent is one tile away in the left direction and three tiles away in the right direction for a coin, the agent may choose to move towards the right direction which is incorrect. We see this as a minor problem and decided not to distinguish close and far coins. We also thought to increase the agent's coin vision range, but this would make problem more complex. Thus, four tiles of range is decided to be the optimum distance for checking coins.

4.1.2 Crate Feature

The second core feature is checking if the given coordinates contain any crate or if it is a free tile. The agent only checked one tile away for crates because bombs achieve the maximum impact when a bomb is placed next to a crate. If we increase the range for looking for crates like the above feature, explosion power of a bomb will also be used for free tiles as well which is unnecessary and inefficient. Therefore, second element of the feature array becomes 1 if the agent is one tile away from a crate for the searched direction and becomes 0 otherwise. During the training with other agents process, we realized that our agent cannot pass over opponents and it is better to explode them. Those two attributes are same for crates and we changed the tile values of the field map, which is given with the game state, in the opponents' coordinates. In other words, our agent sees opponents as crates, so the second feature is actually checking if the observed location has something to explode (crate or opponent) or not.

4.1.3 Danger Feature

The third core feature is checking if the searched coordinate is dangerous or not. Since danger is open to interpretation, not like coin or crate, it needs to be defined. Coordinates which are in the explosion map, and if the timer of a bomb is one and the distance from a bomb in each direction is less than three tiles, or if the timer is zero and the distance from a bomb is less than four tiles in each direction is identified as dangerous. Moreover, walls are also to be avoided to walk on, thus walls are seen as dangerous for our agent. Similar as the previous feature, danger is only checked for one tile away distance for each direction.

Otherwise, it would be difficult for our agent to find safe tiles to move and it may act like trapped when this is not the case. At the beginning of the project, the danger feature was activated when the bomb timer is two and the agents distance to the bomb is less than two tiles. After the training process, we found out that the previously mentioned condition brought more problems than solutions. For instance, in Figure 5, the bomb timer is two and the agent can escape from the bomb by taking actions 'DOWN' and 'RIGHT' respectively. If we did not remove the danger condition when the bomb timer is two, agent would not be able to find that escape route because the danger state of the down direction would be 1. In the current version, agent sees the down direction as safe and in the next step, only the right direction will be seen as safe, since the up and down directions' distance to bomb is less than four tiles, danger state is 1 for those directions and in the left direction crate state becomes 1. The only problem we found was that, in the figure below, the agent would not be in danger state for the down direction even if it had no escape to the right direction. We solved that problem by adding the following core feature.

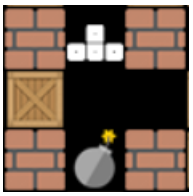


Figure 5: Danger feature sample

4.1.4 Dead-End Feature

The fourth core feature is checking if the observed location is a dead end or not when the agent's distance to a bomb is less than four tiles. As it is mentioned before, this feature aroused out of necessity. When our agent drops a bomb, it can only know if the next tile is dangerous or not and may choose a dead end whose distance to the bomb is less than three tiles, which means agent will die with it's own bomb. After some training without the dead end feature, this type of deaths become reasonably often and we came up with the following solution. If the agent's distance to a bomb is less than four tiles, each directions' next three tiles from the current location of the agent are checked if any of those directions is a dead end or not. In order to be a dead end, there must not be any free tiles among the perpendicular directions of the searched direction and also the searched direction's one axis must be shared with a bomb. For instance, if the agent checks the up direction's dead end state, there should not be any free tiles on the right and left directions for the next three tiles of upper locations. Some examples of the dead end state can be seen in the below figure.

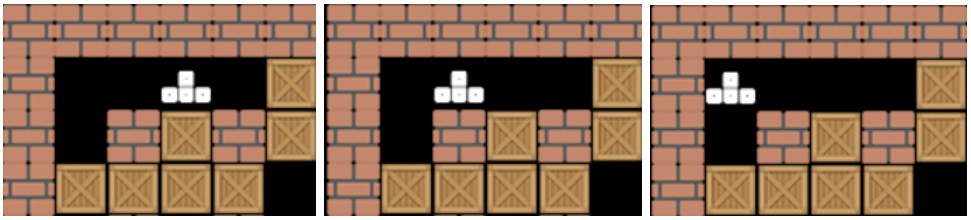


Figure 6: Dead end samples

The image on the left of the above figure, if the agent drops a bomb to it's current location, right direction will be identified as a dead end but the left direction will not. Same for the image on the middle, and for the image on the right, both down and right directions are seen as dead ends if the agent drops a bomb to it's current location. For the first two steps after dropping a bomb, agent will not be able to know if it is in danger or not, but with the help of this feature, it will know, via the rewards, that the non-dead end direction is a better option when the agent does not have any bomb to drop.

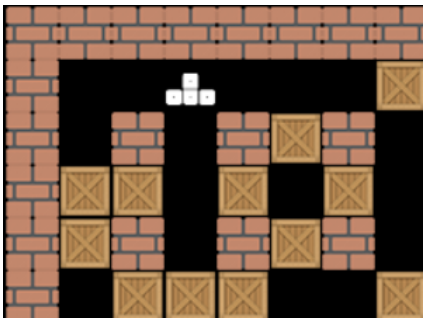


Figure 7: Dead end sample

Another sample for the dead end state can be seen in figure 7. If, somehow an opponent or the agent drops a bomb on it's current location down and right directions will be in dead end state, up direction will be in both danger and dead end state, and left direction will be in non-dead end state. If the agent moves left and drops a bomb, all directions except the up direction will be seen by the agent as non-dead end, since right direction has more than three tiles and left and down directions are not sharing any axis with the bomb.

Before moving into the subsequent sections, chosen directions should be discussed. At first, as a group, we defined the directions as left, right, up, down and the current location. Then, we realized that including the current location is unnecessary. Since the agent chose to be in it's current location in the previous step, because it was the best action to take in that state, the current location has already been checked for the features that are discussed above. Thus, the agent already knows if there exists a coin, crate, danger or a dead end in that location. If the danger state changes for the agent between the steps, at least one other direction's danger state will also change and the agent will be notified that the current location might be dangerous to stay. The only problem of not checking the features of the current location is, if the agent drops a bomb and takes the action 'WAIT' for multiple steps, it cannot know that the 'WAIT' action is dangerous. However, our agent prefers to move

rather than stay because of the given rewards, for this reason, the aforementioned problem never seen in the implementation. Agent's rewards for different actions will be discussed in the Training section of this report.

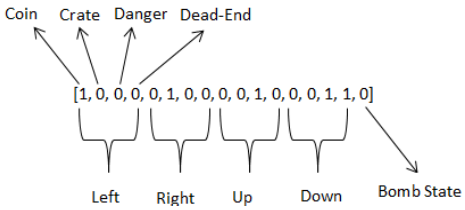


Figure 8: Sample of a State

After defining the features, states are represented as in the figure 8. Actions' Q-values of a state is stored in the Q-table's index of the value after converting the binary state value to it's decimal. In the above case, it is stored in the $10000100001000110 \rightarrow 67654^{th}$ index of the table.

4.2 Q learning with Linear Function Approximation

1. If the agent moves towards the nearest coin- Reward it!

A breadth-first search of the free tiles is performed and if there is a coin for the agent that is reachable, then $F_1(s,a) = 1$ is set for the best possible action a which leads to this coin and $F_1(s,a) = 0$ for $a' \in A \setminus$ a. If there is neither a visible reachable coin for the agent nor any action by which it can be attained, then the feature for all actions is set to zero, that is, $F_1(s,a') = 0$ for all $a \in A$.

2. If the agent moves towards the next crate- Reward it!

The functionality of this feature is same as feature 1, just that in this case it looks for the nearest crate instead of coin. A breadth-first search of the free tiles is performed and if there is a crate for the agent that is reachable, then $F_2(s,a) = 1$ is set for the best possible action a which leads to this crate and $F_2(s,a') = 0$ for a' $\in$ A $\setminus$ a that is, if the agent is already next to a crate or all the crates in the area are already destroyed.

3. Actions that lead to agents' death- Penalize it!

$F_3(s,a) = 1$ is set for every action a leading the agent to cross the path of an exploding bomb. These actions include if the agent moves into a blast range or into an ongoing explosion, either way in both cases it causes death. If the action a does not lead to the death of the agent, then $F_3(s,a) = 0$.

4. Action that leads the agent outside of any blast range- Reward it!

If found in the blast range of any bomb, a breadth first search is performed to find the closest reachable free space outside that blast range. If the action a leads the agent towards this free space, then $F_4(s,a) = 1$, and $F_4(s,a) = 0$ otherwise.

If an agent is endangered by a bomb explosion in the arena, this feature is active; otherwise, $F_4(s,a) = 0$ is set for all an A

5. Action for collecting a coin- Reward it!

This functionality is only active when an agent is next to a coin. If the action a results in the collection of a coin, $F_5(s,a) = 1$ is set; otherwise, $F_5(s,a) = 0$. The key difference between this and F1 is that only if the agent obtains a coin in the next step is the movement rewarded to further encourage the agent about it's role.

6. Action of placing a bomb next to the crate only if it can escape the blast radius later- Reward it!

We do not reward the agent for placing a bomb if it cannot possibly escape it in a later step. All this function does is place a bomb next to a crate, for all other actions $F_6(s,a) = 0$.

7. If the agent can escape from it, the act of of placing a bomb next to an opponent- Reward it!

The functionality of this feature is very similar to F_6. Rewards the action of placing a bomb next to another agent only if it can outrun the blast radius, for all other actions $F_7(s,a) = 0$.

8. Action that leads the agent into getting trapped by it's own bomb- Penalize it!

We added this feature later because one of the most frequent reason for death was it getting stuck in dead ends and bombing itself. To fix this, the agent checks if the last option was a bomb and then penalizes the agent if it goes into a dead-end. $F_8(s,a) = 1$, if the agent gets trapped else $F_8(s,a) = 0$ if the agent does not place a bomb as the last action or is not in a dead-end.

5 Training

Every learning algorithm requires to training processes. These processes are unique to the learning algorithm, features, and task. Therefore, each model went through different training processes.

5.1 Q Table

Training process is separated for the tasks given in the assignment sheet, which are firstly, collecting all the coins in an environment without any crates and opponents, secondly, in a game environment with randomly placed crates yet without any opponents, finding all the hidden coins in the time limit, and lastly, on a game board with crates, fighting for the highest score against one or more opposing agents. Our agent trained for these tasks with custom environment settings respectively.

Before beginning the training process, a Q-table is created and all the Q-values are initialized to be 0.2. Initial value does not make any difference on the learning process.

However, we preferred to observe positive values as a maximum of a given state's best action.

For the first task, the provided 'coin-heaven' environment is used with default settings and the agent is trained for 20 thousand rounds.

For the second task, classic scenario is used but the number of coins is increased to 40. When we used the default coin count, our agent barely faced with coins so the learning process was slow. Most of the states', which includes coins in different directions, Q-values were still in the initialized values. After this change, we trained our agent another 40 thousand rounds.

For the last task, the same environment with the same settings as the previous task is used. To have a faster training process, our agent is trained with only one rule based opponent for 80 thousand rounds. After that, we trained the agent for another 40 thousand rounds with four rule based opponents.

5.1.1 Symmetries

As it is mentioned in the assignment sheet, the Bomberman game world has mirror symmetries. We took advantage of this feature and agent's current state's mirrored equivalent is also updated with the reward of the action taken in the current state. In other words, if the agent takes the action 'LEFT' for a state 'abcd' (where a, b, c, and d refers to the size of four element arrays for the directions left, right, up, and down, respectively), this state-action pair and also the action 'RIGHT' for the mirrored state 'bacd' will be updated. To put it in a more formal explanation, equation 2 can be seen;

$$
\begin{aligned}
&Q['abcd','LEFT'] = Q['abcd','LEFT'] + X \\
&Q['bacd','RIGHT'] = Q['bacd','RIGHT'] + X \\
&\text{where X denotes the below part of the equation 1.}
\end{aligned}
\tag{2}
$$

An example of a state and it's mirrored version can be seen in the figure 9. Consider the left image of the above figure as the original state of the agent. Whatever action the agent takes, both horizontally and vertically flipped states of this state will also be updated with the same amount. Symmetry feature is used for fasten the training process.

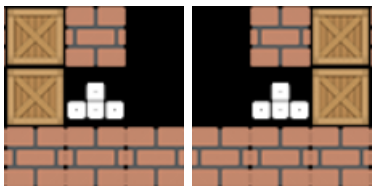


Figure 9: Sample state and it's horizontally mirrored state

5.1.2 Hyperparameter Optimization

There are two hyperparameters that need to be optimized in the equation 1, which are the learning rate (α), and the discount factor (γ). Those parameters are decided according to the figure 10.

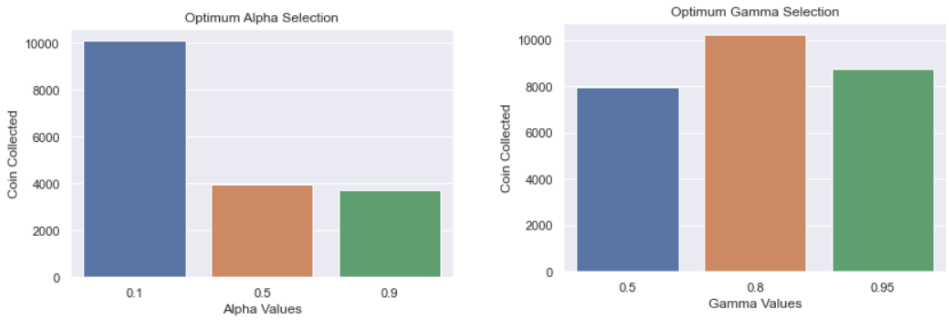


Figure 10: Hyperparameter values vs. collected coin plot

Data in figure 10 are collected after 1000 rounds of training in default coin-heaven scenario. For the plot on the left, discount factor is taken as 0.8, and for the plot on the right, learning rate is taken as 0.1. As it can be seen in figure 10, the optimal values for the hyperparameters are $\alpha = 0.1$ and $\gamma = 0.8$. During the training process these values are applied.

5.1.3 Exploration-Exploitation

Exploitation is taking the action which maximizes the Q-value of a state-action pair. Exploration is taking a random action for the given state-action pair. Exploration is a crucial part of the learning process to find the optimal values for the Q-table.

We used epsilon (ϵ) -greedy method to balance the ratio of actions chosen by exploitation or exploration. This method uses a parameter $0 < \epsilon < 1$ to determine what percentage of actions will be exploration. With the probability of $1 - \epsilon$ of actions will be exploitation. In other words, $\epsilon = 0$ means pick always the best action, and $\epsilon = 1$ means always pick actions randomly. We used $\epsilon = 0.2$ while training the Q-table agnt.

5.1.4 Reward Shaping

We started with the given basic rewards in the assignment sheet's section 5. Since, we do not want our agent to stuck in a loop, we gave small negative rewards for the actions. 'INVALID_ACTION', 'KILLED_SELF', and 'GOT_KILLED' are punished harshly in order our agent to learn the wrong actions for the given states as soon as possible. On the other hand, 'KILLED_OPPONENT', 'COIN_COLLECTED', and 'SURVIVED_ROUND' are rewarded generously because those actions increase our agent's probability to win remarkably. Also the event 'BOMB_EXPLODED' is rewarded positively because we wanted to encourage the escape route of our agent.

We distributed the last round's reward to all state-action pairs taken during the round if the reward is positive. We were also distributing the last round's negative reward to the last three state-action pairs since agent has three steps after dropping a bomb. However, by doing so we were also punishing the correct state-action pairs which makes it more difficult for our agent to escape from it's own bomb. Thus, we abandoned the implementation of distributing the last round's negative reward. The last three events are zero because those

events are caused by at least the three step earlier state-action pair. Rather than giving previous state-action pairs rewards, we preferred to give rewards to current actions.

Table 1: Default Event Rewards used for Q-table and DQL

Events	**Rewards**	**Events**	**Rewards**
MOVED_LEFT	-0.1	INVALID_ACTION	-1.8
MOVED_RIGHT	-0.1	KILLED_OPPONENT	2
MOVED_UP	-0.1	COIN_COLLECTED	0.7
MOVED_DOWN	-0.1	SURVIVED_ROUND	0.5
WAITED	-0.1	BOMB_EXPLODED	0.14
BOMB_DROPPED	-0.1	OPPONENT_ELIMINATED	0
KILLED_SELF	-1	CRATE_DESTROYED	0
GOT_KILLED	-1	COIN_FOUND	0

Even the given events are enough to train our agent to continue the game for a few steps, they are not enough to survive and collect coins. Therefore, we decided to add few custom events. When the agent is within a five tiles distance to a coin, it gets the reward 'DECREASED_DISTANCE' if the distance between the agent and the coin decreases, and our agent gets a punishment 'INCREASED_DISTANCE' when the opposite action taken. When our agent drops a bomb to a corner, it gets a relentless punishment since corners are terrible places to drop a bomb. Instead, near crates are perfect places to place a bomb, and this event is awarded. Dropping a bomb not near a crate is also acceptable so we did not punish this event as much as we award the counter event. Similar as coins, we also give rewards for moving to a bomb and moving away from a bomb within a range of five tiles. With the help of these events, our agent learns the escape route faster. Lastly, choosing or not choosing a dead end route when the agent is within the four tiles range of a bomb are awarded. Absolute value of the reward for the event 'DEAD_END' should be bigger than the event 'MOVED_AWAY_FROM_BOMB' since agent should be punished for choosing the dead end route.

Table 2: Custom Event Rewards used for Q-table and DQL

Custom events	**Rewards**
DECREASED_DISTANCE	0.3
INCREASED_DISTANCE	-0.2
BOMB_DROPPED_CORNER	-0.8
BOMB_NEAR_CRATE	0.6
BOMB_NOT_NEAR_CRATE	-0.3
MOVED_AWAY_FROM_BOMB	0.5
MOVED_TO_BOMB	-0.6
DEAD_END	-0.7
NOT_DEAD_END	0.6

To summarize, we modeled the custom rewards to serve for the features discussed in the aforementioned sections. In this way, our agent learned what the states refers to and mostly takes the proper action.

5.2 Q Learning with Linear Function Approximation

5.2.1 Exploration Methods

The behaviour policy has an impact in Q-learning since it specifies which state-action pairings *(s, a)* are visited and changed. The exploration-exploitation problem arises as a result of this.Exploration is a long-term benefit notion that allows the agent to increase its knowledge of each state-action pair *Q(s,a)* by allowing it to create a better estimate of the optimal Q-function. Exploitation is when an agent takes advantage of it's existing estimated value and selects the greedy method to maximise the payoff

Random-Walk exploration executes a randomly chosen action every time-step. The **Greedy method** assumes the current Q-function is highly accurate and therefore every action is based on exploitation. **ϵ-Greedy exploration** uses the parameter ϵ to determine what percentage of the actions is randomly selected. The parameter falls in the range $0 \leq \epsilon \leq 1$, where 0 translates to no exploration and 1 to only exploration. The action with the highest Q-value is chosen with probability $1-\epsilon$ and a random action is selected otherwise. As a result, $\epsilon = 0$ denotes just exploitation (Greedy) and $\epsilon = 1$ denotes just exploration (Random-Walk). **Diminishing ϵ-Greedy** uses a decreasing value for ϵ, so the agent uses less exploration if the agent played more games assuming the agent is improving its behaviour and thus over time needs less Exploration.[7]

We trained our agent without a graphical user interface and let the game run for 1000 rounds instead of 10 to make the training process more efficient . We adjusted our training system to reset important training parameters, such as total awards in a game, after every 10 rounds to maintain the length of a real game. We have implemented the code in such a way that it facilitates the agent to be trained using any policy(Greedy, ϵ-Greedy,Diminishing ϵ-Greedy), by changing the values in the file constants.py. For training we chose the ϵ -greedy policy with parameters $\alpha = 0.01$,$\gamma = 0.95$ and $\epsilon = 0.15$.

5.2.2 Reward Shaping

Table 3: Default Event Rewards used for Q-Linear Function Approximation

Events	**Rewards**
BOMB_DROPPED	+0.1
KILLED_SELF	-10
GOT_KILLED	-5
INVALID_ACTION	-0.5
KILLED_OPPONENT	7
COIN_COLLECTED	3
SURVIVED_ROUND	2
CRATE_DESTROYED	1
COIN_FOUND	1

We don't really give a big reward (e.BOMB_DROPPED) because though the action is significant, the location where it's dropped and whether or not it can outrun it is more important. Multiple crates are destroyed in one go by placing a bomb in their vicinity .

Destroying a single crate (e.CRATE_DESTROYED) does not give a high reward because in a single blast the rewards are summed up since more than one crate is destroyed. The act of collecting a coin (e.COIN_COLLECTED) and to kill an opponent (e.KILLED_OPPONENT) are some of the main goals and hence is higher comparatively. Getting killed by an opponent is lesser than e.KILLED_SELF because the strategies of the opposing agents are not known. In comparison, e.SURVIVED_ROUND is not the sole purpose of the agent, but collecting coins and killing other agents is and so the reward is lesser.

5.2.3 Feature Matrix

$$\begin{pmatrix} F_1(S,A_1) & F_2(S,A_1) & \dots & F_8(S,A_1) \\ F_1(S,A_2) & F_2(S,A_2) & \dots & F_8(S,A_2) \\ . & . & \dots & . \\ F_1(S,A_6) & F_2(S,A_6) & \dots & F_8(S,A_6) \end{pmatrix}$$

The eight features defined previously F_1 - F_8 are feature functions which actually return a vector V $\in \{0,1\}^6$ where every vector value maps the feature function for an action. Combining all the vectors we form a $8\times$ 6 matrix . A row of this matrix would then translate into a feature vector of an action. Now, with this matrix and the weight vector, the Q-value can be calculated by a simple dot product.

Starting with random weights with our feature matrix was leading to very low rewards for a long time. So, since we had handcrafted the features ourselves, we decided to start with what we thought was the best guess for weights keeping in mind our features and their roles. The best guess values taken for training were, [1, 1.7, -6, 3.5, 1.5, 1, 0.5, -0.8, 0.5]

5.2.4 Batch Gradient Descent with Prioritized Experience Replay

Irrespective of the kind of exploration method we use (that is greedy, -greedy, or diminishing -greedy), linear function approximators can make all weights and values diverge, even though look up tables make sure they converge. We also realised, across different episodes, high negative values were generated as accumulated rewards which in fact was counter-productive, when we ran the game for 2000 rounds using ϵ-greedy learning.

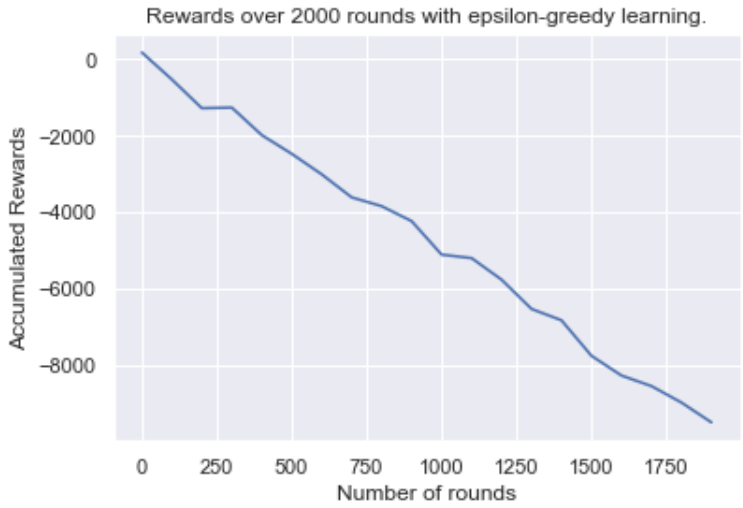


Figure 11: Rewards obtained by our agent using epsilon-greedy method

The Q-learning algorithm only uses the last experience combined with the current Q-function in order to update its function, previous experiences are disregarded. However, In recent years, a technique referred to as experience replay has been suggested as a mechanism to improve Q-learning by allowing the learner to access previous experiences. It has been suggested that using past experiences in such a way might allow Q-learning to better converge to the optimal Q values, by breaking the time and space correlation structure of experiences as they are sampled from the real world, allowing for policy updates not dependent on the current time and location in the markov decision process. Moreover, using experience replay improves the efficiency of data usage, since every experience is used for learning more than once[25].

We do this as it can be advantageous to store the experiences to a buffer memory(training set), and update the current weights by sampling a mini batch from this buffer [26]. The advantage of sampling is to make sure there is no correlation in the same episode between experiences and to include possible experiences that are outliers. The weights are updated using batch gradient descent which performs significantly better than incremental gradient descent, where the weights are updated after every sample but apply the changes only after the batch. For it to converge, we began with small batches so that it learns quickly and then moves up to increase batch sizes. In our model, the starting size of the mini-batch increases by 25 after every generation.

Potential deduced algorithm for Experience Replay with batch gradient descent [30],

function ExperienceReplayLinFApp($B, m, w, F, \alpha, \gamma$)
Input: B is the replay buffer of size N, containing experiences (s, a, r, s'), m is the size of the mini-batch to be sampled from the replay buffer, $w \in \mathbb{R}^d$ is the parameter vector, $F(S, A)$ is the feature extraction, α is the learning parameter, and $\gamma \in [0, 1]$ is the discount value.

1. $b \leftarrow$ RandomSample(B, m)
2. $u \leftarrow 0$
3. for X, A, R, Y in b do:
   $\delta \leftarrow R + \gamma \cdot \max_{a' \in \mathcal{A}} w^T F(Y, a') - w^T F(X, A)$ if Y is not terminal;
   $\delta \leftarrow R$ if Y is terminal.
   $u \leftarrow u + \frac{\alpha}{m} \cdot \delta \cdot F(X, A)$
4. $w \leftarrow w + u$

Just like Schaul et al. [27] and Wang et al. [28], have used prioritised experience replay, prioritising experiences which are important (in some sense) over others rather than replaying all transitions in the same order, to get remarkable results on the well-known Atari benchmark [25], we have tried that too, though a bit differently.

The key assumption is that some transitions teach an agent more efficiently than others. Transitions might be unexpected, redundant, or task-relevant in nature. Some transitions may not be immediately valuable to the agent, but as the agent's competence grows, they may become so. Prioritised replay is nothing but a modification of experience replay where sampling is done from the replay buffer with a probability which depends on the highest rewards or other factors like temporal difference or time step to set the probabilities for which we take samples from a (mini-)batch rather than sample uniformly, which makes the experience algorithm more efficient [29]. After seeing that the most critical occurrences in a round of Bomberman occur in the first 50% of the time steps, we decided to use a more basic approach and have defined the following hyper parameters.

1.	self.replay_buffer_max_steps	$B_1 = 200$	Adding experiences only when t≤ B
2.	self.replay_buffer_update_after_nrounds	$B_2 = 10$	Number of rounds played before sampling the replay buffer
3.	self.replay_buffer_sample_size	$B_3 = 50$	Starting size of the mini-batch increases by 25 after every generation to facilitate convergence.
4.	self.replay_buffer_every_ngenerations	$B_4 = 1$	Prior to replay buffer becoming empty, number of generations played

5.3 Deep Q-learning

Three DQL agents were implemented to play the bomberman game. Initially a linear FC agent and an LSTM agent were tried out. Later a GRU agent was also implemented to compare the performance of LSTM agent, since they both have similar functionality.

The feature states provided as input for these three agents included the complete arena, coins, explosion location, and the location of itself and the enemies on in the arena. The arena state was extracted from `game_state['field']` where field values were equal to 1. The coin state was extracted from `game_state['coins']`. The explosion location was extracted form `game_state['bombs']` and `game_state['explosion_map']`. The location was self and enemies was extracted from `game_state['self']` and `game_state['others']`. Further feature engineering was not performed for these agents as DQL agents are supposed to extract deep features from the states given to it and learn to play the game with minimum user curated input states. The DLQ model also utilized masking the training states and storing the recent 500 steps as experience in a list. The model then took the mini-batches as input and returned the DQL agent actions and state as its output. The FC, LSTM and GRU agents agents also used the same custom events and rewards as the Q-table agent shown in table 2. This allowed us to compare the performance on a uniform reward function for these two agents.

An epsilon-greedy exploration algorithm was also implemented for the DQL agents. This is used because the agent must discover which action gives the most reward through trial and error, or take the best action estimated by the agent. In the epsilon-greedy method, epsilon refers to the probability of choosing to explore, and the agent chooses to exploit most of the time with a small chance of exploring. This pseudo-code for epsilon-greedy method used was:

```
// Epsilon-greedy
epsilon = 0.1
decay = 0.9999
min_epsilon = 0.001
if train and random <= epsilon:
    return random action
else:
    return best action
epsilon = max(min_epsilon, epsilon*decay)
```

Table 4 shows the architecture of all the DQL agents that were used along with their specific input and output shapes, optimizers, loss and activation functions. The LSTM and GRU agent have a similar structure with only a difference between the deep feature extraction RNN layers. This was done to effectively compare the performance of both methods on the same scale.

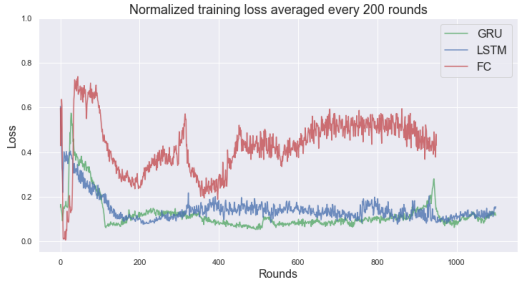
Table 4: DQL Agent Network Architectures

FC agent	**LSTM agent**	**GRU agent**
Linear(intput_states,12)	LSTM(intput_states,12,2)	GRU(intput_states,12,2)
ReLU	MaxPool1D(2,2)	MaxPool1D(2,2)
Linear(12,6)	Linear(6,6)	Linear(6,6)
ReLU	ReLU	ReLU
Linear(6,actions)	Linear(6,actions)	Linear(6,actions)
Softmax	Softmax	Softmax

All the DQL implemented agents used Adam optimizer with learing rate 0.00005 and Smooth L1 loss. Adam optimization [31] is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments. Smooth L1-loss can be interpreted as a combination of L1-loss and L2-loss. It behaves as L1-loss when the absolute value of the argument is high, and it behaves like L2-loss when the absolute value of the argument is close to zero. Training and testing of all DQL agents was done on a machine using Ryzen 5900 CPU and an Nvidia RTX 3060 GPU. The training times for the LSTM and GRU agent reached approximately 4.5 hours for 80K rounds while the FC agent was trained within 6 hours. We realized that the FC agent would have trained much faster on the CPU compared to Cuda. The agents were set to train for 300K rounds as DQL agents take a long time to converge. However due to time limitations the training process was stopped around 220K rounds.

(a) Average Loss 80K Rounds

(b) Average Reward 80K Rounds



(c) Average Loss 220K Rounds

(d) Average Reward 220K Rounds

Figure 12: Metrics Summary for Training DQL Agents

Figure 12 shows the normalized training loss and the training rewards for all the DQL agents. The DQL agents were trained against the `rule_based_agent`. We notice that the rewards for 80K rounds show that the FC agent is trying to maximize the rewards, however the LSTM and GRU are making little progress. The loss for 80K rounds show that the agents are trying to converge. But in the metrics for 220K rounds we notice that the loss for FC agent has gone up, which could be a result of the model being stuck in a local maxima. The rewards for the FC agent has also plateaued right before the 80K round mark and did not rise since. However, the LSTM and GRU agent rewards starts to slowly rise. The GRU agent reward takes a dip at the 200K round mark but it comes back up fairly quickly. If this trend is followed then we can hypothesize that the LSTM and GRU agent rewards will continue to rise as the agent is trained more. The obvious drawbacks we see

from DQL agents is that they learn slowly as they have to extract and learn deep features by themselves. It would be interesting to see what would happen if the agents were left to train for one million rounds.

6 Results

Results obtained after the training processes are discussed individually. At the end of this section, all models are also compared with each other.

6.1 Q Table

After making the discussed adjustments in the Training section, we left our agent for training with different number of rounds to monitor the progress. The agent is trained in the classic scenario without any opponents. Learning progress of our agent can be seen in figure 13.

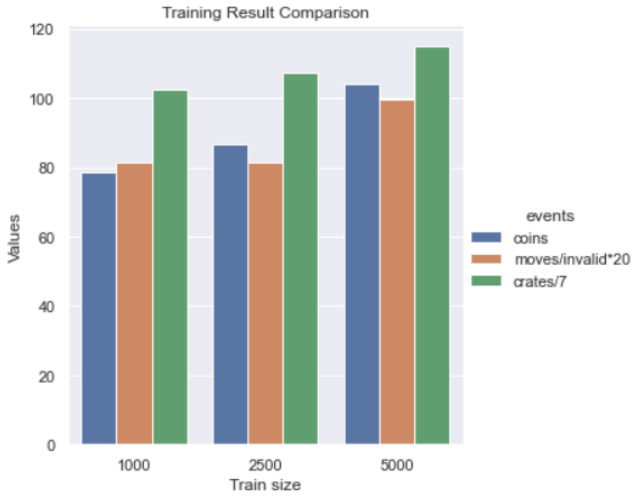


Figure 13: Progress of the Q-table agent

All the data obtained in figure 13 is divided to the training size and multiplied by 100 to make fair comments. After that change, crates and the ratio of moves and invalid actions had an extra operation to scale the values. As it can be seen in the figure, our agent is learning how to play the Bomberman game. Therefore, we trained our agent for 80 thousand rounds in the same environment against with only one opponent.

After the training, we tested our agent in the classic scenario without any opponents. Results of this test can be seen in figure 14.

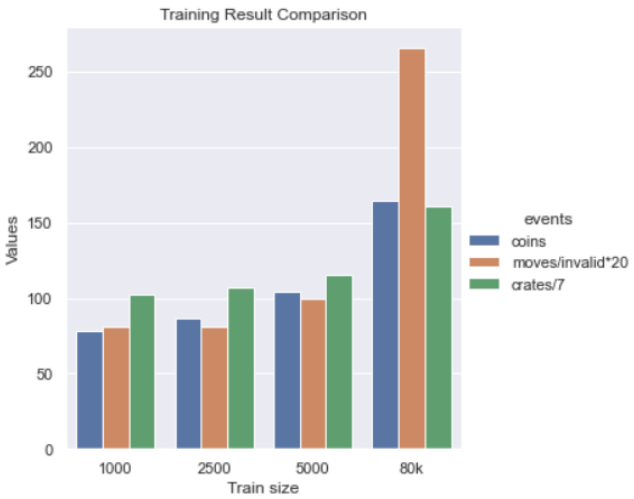


Figure 14: Results after training for Q-table

As it can be seen in the figure, after training our agent learned lots of things about collecting coins, surviving, and destroying crates. We observed an interesting situation after the training. Our agent sometimes tries to go around the left down corner wherever it begins the round. When it reaches around the corner, it gets stuck in a loop around the down left area. Although we got the advantage of the symmetry feature of the game board, we did not observe any loops in other corners.

Later on, we tested our agent with three rule based agents in the classic scenario for 1000 rounds. Specifically for testing, epsilon (ϵ) is changed to 0.1 and the agent took the second best action for the given state-action pair in the exploration part. We did not want to completely ignore the exploration part because we observed that, when no crates, opponents, coins, or bombs are around the agent, it may stuck in a loop. To break that loop, the agent takes the second best action approximately for once in every ten steps. The results of the test can be seen in figure 15.

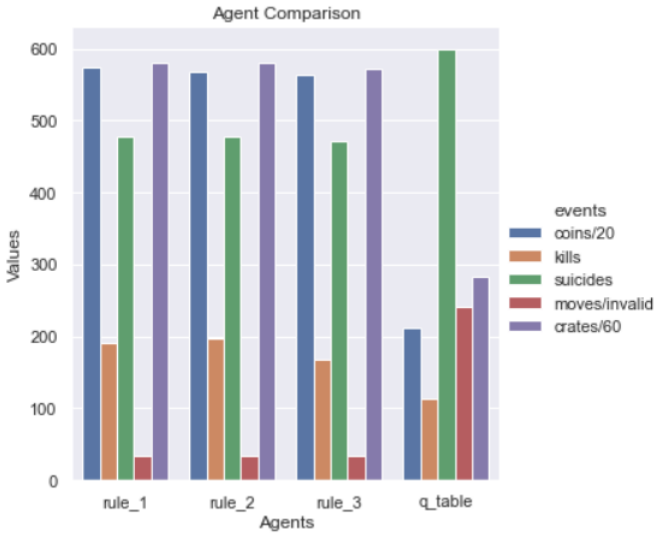


Figure 15: Comparison plot of different agents

Number of coins collected and crates destroyed are scaled to the other values to obtain a visually better image. Our agent collected half as much coins as the other agents collected and destroyed half as much crates as others. It's total moves and invalid actions ratio is better than other agents. By looking at the information we have so far, it can be said that taking the second best action did not completely solved the loop problem. We think that, when our agent destroys the crates around six or seven tiles away from it's corner, sometimes it cannot find the crates in the opposite direction. It begins to make a big loop, for instance, it is stuck in a 3x3 tile sized loop. This loop is mostly created in the down left corner. When our agent begins a round in the up right corner, it does not stuck in a loop and also gets a quite good score. However, when it begins a round in down left corner, it barely discovers the other areas of the map. By looking at the figure 16, it can be said that the agent plays better when it begins the game from upper corners and it most probably gets stuck in a loop when it begins a round in the down left corner.

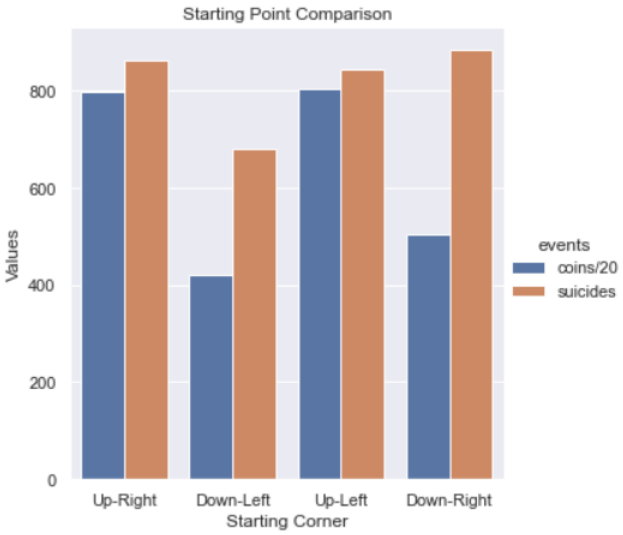


Figure 16: Comparison plot of different starting points for 1000 rounds for Q-table

Moreover, number of kills the Q-table agent made is less than the other opponents but it is acceptable in our point of view. Number of suicides committed is also really close to the rule based agents', which means our agent is mostly learned how to survive. Therefore, it can be said that Q-table agent is learned to survive among three rule based agents but did not learn how to take the lead.

6.2 Q Function Approximation

As we know, for expediting the training process we had used [1, 1.7, -6, 3.5, 1.5, 1, 0.5, -0.8, 0.5] weights as our best guess initially, since we had defined the features on our own we could map suitable weights to it. To further optimize our weights we made our agent starting from the best guess weights and trained it over 5000 rounds (that is 500 rounds of 10 each). The resultant trained weights comparatively gave way better rewards thus, improving our agent's performance(as shown in the graph below).

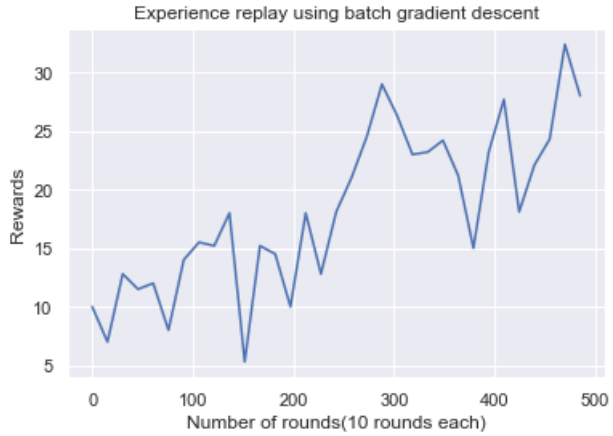


Figure 17: Rewards for Q-Function Approximation using Experience Replay

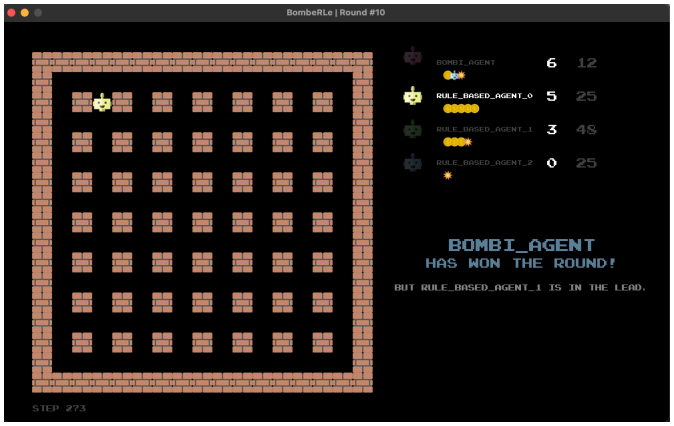


Figure 18: Snapshot of our agent winning against rule based agents

The final trained weights used were [1., 2.13864661, -7.53564286, 3.73366143, 2.61421682, 1.00162857, 0.60588571, -1.17647312, 0.50014563] With these weights our bombi_agent could potentially win a round but never take the lead of entire game of 10 rounds.

6.3 Deep Q learning

The testing results statistics for DQL agents are shown in figure 19. The three agents played against themselves for 10K rounds during testing to compare their performance for a classic match. This was done to observe how many valid and invalid moves each agent would make and the number of bombs they will drop to destroy the crates, kill the enemies and themselves. We notice in the figure that the FC agent is faster in making moves and dropping bombs as the frequency for all actions performed are higher than the LSTM and GRU agents. The LSTM and GRU agents have dropped around 10K bombs only which means they usually ended up committing suicide the first time they dropped the bomb in each round. The number of crates destroyed by all the agents are almost the same as well which means the FC model dropped a lot of bombs which were not near the crates. The ratio of valid to invalid moves performed is also almost the same for all the agents. Which means that the FC agent was just faster than the rest but not more intelligent.

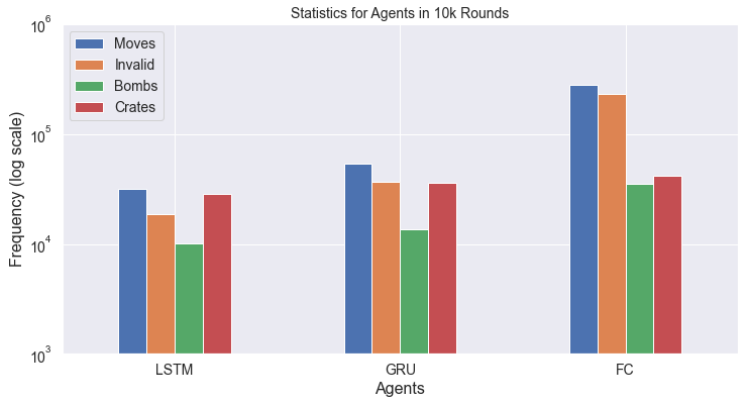


Figure 19: DQL Agent Comparison (playing against each other)

To further compare the DQL agents to Q-table and Q-function-approximation (Q-FA) agents, we tested each agent individually on a classic arena for 10K rounds. This was done to evaluate which agent collects more coins and destroys more crates when it is not competing with other stronger agents. Figure 20 shows the result statistics of all agents' individual performance in 10K rounds. We observe that the DQL agents performed weaker compare to more robust and traditional Q-FA and Q-table agents. the DQL agents barely collected more than 1000 coins combined while the Q-FA and Q-table agents collected upwards of 6000 and 21000 coins, respectively. Similarly, Q-FA and Q-table agents destroyed more crates than the DQL agents. This result shows that the DQL agents either did not get enough training or there were issues with learning the given features to the deep learning models.

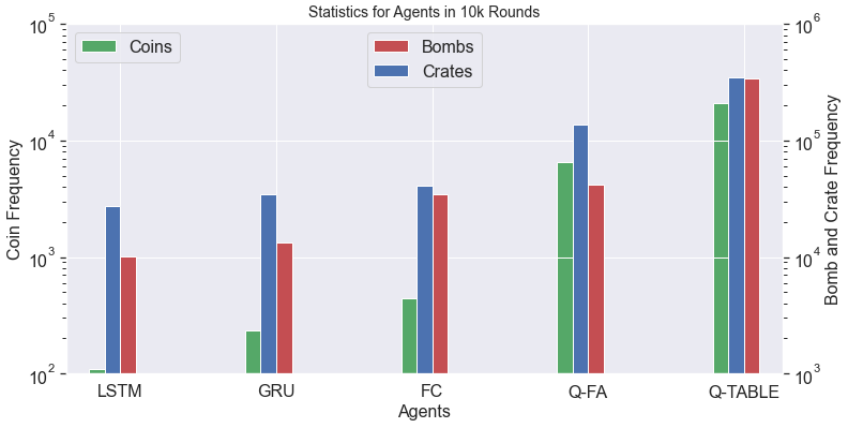


Figure 20: All Agent Comparison (playing individually)

7 Conclusion

Three different models are created for the Bomberman task. The first model is used Q-learning with a Q-table to take an action. It was observed that the Q-table agent worked well enough to survive the round and hide from enemy bombs but not enough to take the lead against the rule based agent. However, it was the strongest agent our team implemented and outperformed all the other agents. The second model is Q-learning with function approximation which has the ability to win a single round but not take the lead. It sometimes has a hard time navigating and mostly die by not escaping the blast radius when bombed by an opponent. Our last model is used three different Deep Q-learning models. The conventional methods for DRL use two dimensional convolutional layers to extract deep features from the given states. We thought that while every other team would be using a convolutional layer to achieve the best performance, we could experiment with other layers for feature extraction. This would allow us to present an argument for advantages and disadvantages of using recurrent neural network DRL methods to play Bomberman, even though the performance of the agent would not be as robust. As expected, the DQL agents learned slower than other methods and also required robust hardware like GPUs for optimal training time. The DQL agents' ability to extract deep features which would help the agent to learn could either be completely wrong or they all were very slow at learning and would require a really

large training time of around 1-2 million rounds.

We learned that reinforcement learning tasks require precise hyperparamter tuning for the agents and we could have experimented more with the different styles of agents. We learned that creating and designing a reward function which would be optimal for this game is very difficult without introducing some kind of bias for a given state. We observed that the model usually waited just to avoid losing rewards in the future.

7.1 Outlook

If we had more time, we would have further improved our features for both the Q-table and function approximation methods. It is obvious that our Q-table agent has a problem with taking the lead, in other words collecting coins. Another problem about the Q-table agent is loops. If iteration times and file size allow, another state for each direction can be added, which will make total of 2^{21} states. This state can hold if the checked direction would make a loop or not. Only this may also solve the coin collecting issue because it is mainly caused by not exploring enough areas on the map. Another solution might be, the unseen areas' direction can be seen as coins by adding another feature. Since our agent is biased to move in the direction of a coin, this might help to break loops.

For the Linear Approximation model, though we were able to make the values converge to some extent, a lot could have been improved. Firstly, since Linear Approximation requires extensive feature engineering, the 8 features were definitely not enough and could have been extended to many more to make it a better model. Furthermore, for prioritised experience replay if we had more time, we could have explored the use of temporal difference error or rewards to set probabilities for sampling from mini-batch instead of the 3 basic hyper parameters we defined. Additionally, we could have also explored other non-linear approximation methods or regression algorithms which does not mandate extensive feature engineering like neural network or decision forest.

If there was more time we would have tried to optimize our DQL agents by fine tuning the hyper-parameters like learning rate and also experimenting with different losses and optimizers like the Huber loss and the SGD optimizer. We would also allow the DQL models to be trained for more time, approximately one million rounds just to see if the reward trend would still keep rising or would it plateau. There was also a possibility to create a convolutional neural network agent to compare it to our RNN agent which would make for an interesting analysis.

For the game setup in next year's lecture, a tutorial session can be done to help the students to understand the environment and the task carefully. We also think that entries in the game state should be in the same type. For instance, field_map returns a numpy array but coins returns a list. Lastly, a test tournament can be done with test uploads to encourage people to send their callbacks file to the beat-my-agent channel.

References

[1] Wu, C., Alomar , A., & Jagwani, S. (2021, April). Reinforcement Learning: Foundations and Methods. Lecture 19: Off-Policy Learning.

[2] Köthe, U. (2022, March). Fml 21/22 Lecture Notes. Model-Free Reinforcement Learning. Heidelberg.

[3] Violante, A. (2019, July 1). Simple reinforcement learning: Q-learning. Medium. Retrieved April 3, 2022, from https://towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56

[4] Jang, B., Kim, M., Harerimana, G., & Kim, J. W. (2019). Q-Learning Algorithms: A comprehensive classification and applications. IEEE Access, 7, 133653–133667. https://doi.org/10.1109/access.2019.2941229

[5] Wikimedia Foundation. (2022, March 2). Q-learning. Wikipedia. Retrieved April 3, 2022, from https://en.wikipedia.org/wiki/Q-learning

[6] Omerbsezer. (n.d.). Omerbsezer/reinforcement_learning_tutorial_with_demo: Reinforcement learning tutorial with demo: DP (policy and value iteration), Monte Carlo, TD Learning (Sarsa, qlearning), function approximation, policy gradient, DQN, imitation, meta learning, papers, courses, etc... GitHub. Retrieved April 3, 2022, from https://github.com/omerbsezer/Reinforcement_learning_tutorial_with_demo

[7] Exploration Methods for Connectionist Q-Learning in Bomberman - Joseph Groot Kormelink, Madalina M. Drugan and Marco A. Wiering, Institute of Artificial Intelligence and Cognitive Engineering, University of Groningen, The Netherlands

[8] Q-learning with linear function approximation by Francisco S. Melo and M. Isabel, Ribeiro Institute for Systems and Robotics Instituto Superior Técnico

[9] https://danieltakeshi.github.io/2016/10/31/going-deeper-into-reinforcement-learning-understanding-q-learning-and-linear-function-approximation/

[10] A new convergent variant of Q-learning with linear function approximation- Diogo S. Carvalho, Francisco S. Melo, Pedro A. Santos, INESC-ID, Instituto Superior Técnico, University of Lisbon Lisbon, Portugal

[11] Reinforcement Learning Algorithms for MDPs, Csaba Szepesv´ari October 3, 200

[12] Francois, Vincent, Henderson, Peter, Islam, Riashat, Bellemare, Marc, Pineau, Joelle. (2018). An Introduction to Deep Reinforcement Learning. 10.1561/2200000071.

[13] Silver, David, Huang, Aja, Maddison, Christopher, Guez, Arthur, Sifre, Laurent, Driessche, George, Schrittwieser, Julian, Antonoglou, Ioannis, Panneershelvam, Veda, Lanctot, Marc, Dieleman, Sander, Grewe, Dominik, Nham, John, Kalchbrenner, Nal, Sutskever, Ilya, Lillicrap, Timothy, Leach, Madeleine, Kavukcuoglu, Koray, Graepel, Thore, Hassabis, Demis. (2016). Mastering the game of Go with deep neural networks and tree search. Nature. 529. 484-489. 10.1038/nature16961.

[14] S.Manju, M.Punithavalli,. (2011). An Analysis of Q-Learning Algorithms with Strategies of Reward Function. International Journal on Computer Science and Engineering.

[15] Jianqing Fan, Zhaoran Wang, Yuchen Xie, Zhuoran Yang Proceedings of the 2nd Conference on Learning for Dynamics and Control, PMLR 120:486-489, 2020.

[16] www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/

[17] Du, K. Swamy, M.N.s. (2014). Recurrent Neural Networks. 10.1007/978-1-4471-5571-3-11.

[18] Hochreiter, Sepp, Schmidhuber, Jürgen. (1997). Long Short-term Memory. Neural computation. 9. 1735-80. 10.1162/neco.1997.9.8.1735.

[19] clay-atlas.com/blog/2020/06/02/machine-learning-cn-gru-note/

[20] Bram Bakker. 2001. Reinforcement learning with long short-term memory. In Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic (NIPS'01). MIT Press, Cambridge, MA, USA, 1475–1482.

[21] Goulart, Í., Paes, A., Clua, E. (2019). Learning How to Play Bomberman with Deep Reinforcement and Imitation Learning. In: van der Spek, E., Göbel, S., Do, EL., Clua, E., Baalsrud Hauge, J. (eds) Entertainment Computing and Serious Games. ICEC-JCSG 2019. Lecture Notes in Computer Science(), vol 11863. Springer, Cham. https://-doi.org/10.1007/978-3-030-34644-7-10

[22] Cho, Kyunghyun, Merrienboer, Bart, Gulcehre, Caglar B,ougares, Fethi, Schwenk, Holger, Bengio, Y.. (2014). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. 10.3115/v1/D14-1179.

[23] Gao, Xiang. (2018). Deep reinforcement learning for time series: playing idealized trading games.

[24] Pelletier, Charlotte, Webb, Geoffrey, Petitjean, François. (2019). Temporal Convolutional Neural Network for the Classification of Satellite Image Time Series. Remote Sensing. 11. 523. 10.3390/rs11050523.

[25] Convergence Results For Q-Learning With Experience Replay- Liran Szlak, Ohad Shamir December 9, 2021

[26] M. Pieters and M. A. Wiering, "Q-learning with experience replay in a dynamic environment," 2016 IEEE Symposium Series on Computational Intelligence (SSCI), 2016, pp. 1-8, doi: 10.1109/SSCI.2016.7849368.

[27] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver- Prioritised experience replay. arXiv preprint arXiv:1511.05952, 2015.

[28] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, and Nando de Freitas- Sample efficient actor-critic with experience replay.arXiv preprint arXiv:1611.01224, 2016.

[29] Prioritized Experience Replay -Tom Schaul, John Quan, Ioannis Antonoglou and David Silver, Google DeepMind https://arxiv.org/pdf/1511.05952.pdf

[30] Python code for Artificial Intelligence: Foundations of Computational Agents- David L. Poole and Alan K. Mackworth

[31] Kingma, Diederik, Ba, Jimmy. (2014). Adam: A Method for Stochastic Optimization. International Conference on Learning Representations.