Reinforcement Learning

Rainbow - Combining Improvements in Deep Reinforcement Learning

Authors

Clément Hardy
Félix Larrouy

February 5, 2019

Summary

1 Introduction

Deep Q-learning algorithm [4] is a combination of Q-learning with convolutional neural networks and experience replay capable of playing many Atari games at human-level performance and even better. Several extensions have been proposed since the creation of Deep Q-learning in order to improve the speed of convergence and the stability of the algorithm. We are going to dive into the details of each mentionned extension in this paper in section 3.

We have implemented each of these extension separately to compare them individually as well as with the basic Deep Q-learning algorithm. Finally, we have combined these extensions into a single agent called Rainbow.

2 What is Deep Q-learning and why do we need it ?

Recall that in Q-learning, we have to implement a function to create and update a Q-table at each iteration. The Q-table is used to store, for each action at each state, the maximum expected future reward (which has to be computed). Thanks to this table, we'll know what's the best action to take for each state. But now imagine that we want to teach an agent to play an Atari game, such as 'Breakout'.

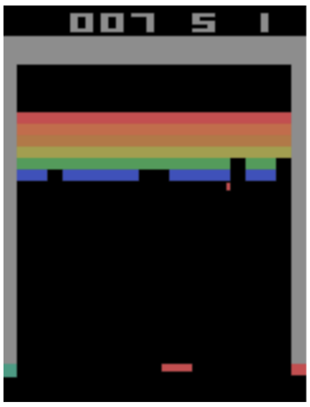

Figure 1: Frame of the Atari arcade game 'Breakout'

Here the number of possible states is huge, and creating a Q-table will be computationally inefficient. Therefore, Q-learning is not scalable for environments like Atari games. This is where Deep reinforcement learning and DQN come in. The various components of reinforcement learning agents, such as policies $\pi(s, a)$ or values $q(s, a)$, are represented with deep neural networks.

In DQN (Mnih et al. 2015) [4], a convolutional neural network has been used to estimate the action values for a given state s_t. Two components have been introduced in the DQN paper, which are experience replay and target networks.

2.1 Experience Replay

Reinforcement learning is known to be unstable (and sometimes to even diverge) when applying non linear functions, such as neural networks, to approximate the Q-value function [4]. One of the reasons is that the observations in the input sequence are highly correlated. For example in an Atari game, a frame at time $t + 1$ is very correlated to the frame at time t. The problem is that doing many iterations of gradient descent on correlated data can lead to overfitting, or falling into a local minimum.

This is why Experience Replay has been introduced. The idea is to store a set of experiences while interacting with the environment. So at each time step t, we store the argen't experience of the form $e_t = (s_t, a_t, r_t, s_{t+1}, done)$ in a replay buffer $e_1, ..., e_n$. Then, during training, we perform Q-learning updates on a minibatch of experiences sampled uniformly at random from the replay buffer. In this way, the data is not correlated anymore because the agent doesn't learn from the latest seen experiences.

2.2 Target Network

Recall the Bellman equation, telling us that if the optimal value $Q*(s', a')$ of the sequence of states a time $t+1$ was know for every actions a', then the optimal strategy is to select the action that maximises the expected value of $r + \gamma Q^*(s', a')$.

$$Q^*(s,a) = \mathbb{E}_{s'}[r + \gamma \max_{a'} Q^*(s', a')|s, a]$$

We can use a neural network to estimate the action-value function. The goal will be to minimize the loss function

$$L(\theta) = \Big(\underbrace{r_{t+1} + \gamma_{t+1} \max_{a'} Q(s_{t+1}, a'; \theta)}_{\substack{\text{maximum possible Q-value for} \\ \text{the next state (Q-target)}}} - \underbrace{Q(s_t, a_t; \theta)}_{\substack{\text{current predicted} \\ \text{Q-value}}}\Big)^2$$

The problem here is that we are using the same parameters θ for estimating the target and the Q-value. Hence, there is a high correlation between the Q-target and the parameters θ that we update at every gradien descent step. This means at every step of training, our predicted Q-values are getting closer to the target value, but the target value is also shifted. To overcome this problem, we can use a separate network with fixed paramters $\bar{\theta}$ for estimating the target. This $\bar{\theta}$ will be periodically updated by copying the parameters of the online network, used to predict the Q-value. The loss function becomes

$$L(\theta) = \big(r_{t+1} + \gamma_{t+1} \max_{a'} Q(s_{t+1}, a'; \theta^-) - Q(s_t, a_t; \theta)\big)^2$$

The use of experience replay and target networks enables relatively stable learning of Q values, and led to super-human performance on several Atari games. But since the emergence of Deep Q-learning, several extensions have been proposed.

3 Extensions to Deep Q-Learning

3.1 Double Q-learning

When we compute the TD target $Q(s, a) = r_{t+1} + \gamma_{t+1} \max_a Q(s', a)$, we can face a problem of overestimation of Q-values. Indeed, here we consider that the best action for the next state is the action with the highest Q-value. But is it always the case ? The answer is: not necessarily. Indeed, at the beginning of the training, the agent hasn't explored enough states to have a good estimation of the the expected rewards. Therefore, taking the action with the maximum Q-value (which can be noisy) can lead to false positives. If the agent takes several noisy actions, this can lead to divergence (or at least a complicated training).

The solution proposed in [7] is to use two neural networks for estimating the Q-value. Actually, the idea is to use the DQN network to select what is the best action to take for the next state (the action with the highest Q value), and to use the target network to compute the target Q value of taking that action at the next state. This gives us the following loss function

$$L(\theta) = \left(r_{t+1} + \gamma_{t+1} Q(s_{t+1}, \arg\max_{a'} Q(s_{t+1}, a'; \theta); \theta^-) - Q(s_t, a_t; \theta)\right)^2$$

This methods was shown to reduce harmful overestimations that were present for DQN, thereby improving performance.

3.2 Prioritized Experience Replay

Experience replay lets a reinforcement learning agent to remember and reuse experiences from the past. In the baseline of deep Q-learning, experiences are uniformly sampled from the replay buffer. Some experiences may be more relevant than others for the training, but they may occur less frequently due to the complete randomness. Prioritized Experience Replay paper [5] proposed a new method to be able to sample more frequently experiences from which there is much to learn.

The idea is to sample in priority experiences where the agent badly performed, i.e. where there is a significant difference between the prediction and the TD target. Past experiences are sample with a probability

$$p_t = |\underbrace{r_{t+1} + \gamma_{t+1} \max_{a'} Q(s_{t+1}, a'; \theta^-) - Q(s_t, a_t; \theta)}_{\text{TD error}}|^{\omega} + \epsilon$$

where ω is a hyper-parameter that determines the shape of the distribution. In practice, we introduce a small constant ϵ such that no experience has a 0 probability to be sampled. In addition, we can't just do greedy prioritization because it will lead to always learn from the same experiences, and the agent will overfit. That is why stochastic prioritization is introduced. The probability of an experience to be sampled becomes

$$P(t) = \frac{p_t^{\alpha}}{\sum_k p_k^{\alpha}}$$

where α is a hyperparameter used to reintroduce some randomness in the experience selection from the replay buffer. If $\alpha = 0$, we have uniform probabilities, if $\alpha = 1$, we perform a full greedy sampling. We also normalize by all the priority values in the replay buffer. We will now get a batch of experiences at each step of training with the above probability distribution $P(t)$. But another problem arises from this method. In classic Experience Replay we update the weights in a stochastic way, which is correct because the experiences are sampled totally randomly from the replay buffer. However, with Prioritized Experience Replay, the probability distribution has changed, and if we update our weights normally, samples that have high priority are likely to be used for training many times, which will introduce bias. We can correct this bias by using importance-sampling (IS) weights

$$w_i = (\frac{1}{N}\frac{1}{P(i)})^{\beta}$$

where β controls how much IS weights affect learning. β can be set close to 0 at the beginning and grow gradually up to 1 during training (1 is reached at the end of training) because these weights are more important in the end of learning when our q values begin to converge.

Lastly, for stability reasons, weights are normalized by $\frac{1}{\max_i w_i}$ so that they only scale the update downwards.

3.3 Dueling Networks

Q-values correspond to how good it is to be at a state s and taking an action a from that state. Therefore, we can decompose this Q-value as

$$Q(s,a) = \underbrace{A(s,a)}_{\text{action advantage}} + \underbrace{V(s)}_{\text{state value}}$$

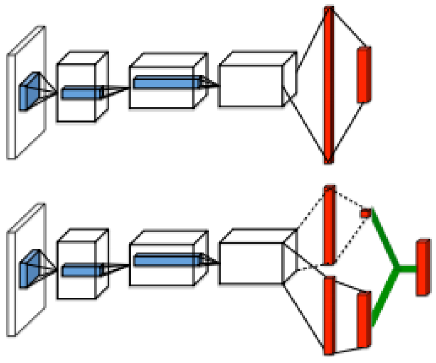

Figure 2: A single stream Q-network (top) and the dueling Q-network (bottom). [8]

As we can see on the figure above, we can then use two separate streams, one for estimating $A(s,a)$ and one for estimating $V(s)$. As stated in [8], the dueling architecture can learn which states are (or are not) valuable, without having to learn the effect of each action for each state. For example, if any action from a specific state leads to death, there is no need to compute the value of each action. Similarly, if we are in a state where the action will not have a big influence on the environment, we don't need to compute action values.

Finally, estimations of $A(s,a)$ and $V(s)$ are combined through the following equation:

$$Q(s,a;\theta) = V(f(s;\theta_{conv});\theta_V) + A(f(s;\theta_{conv}),a;\theta_A) - \frac{\sum_{a'} A(f(s;\theta_{conv}),a';\theta_A)}{N_{\text{actions}}}$$

where θ_{conv}, θ_V, and θ_A are, respectively, the parameters of the shared encoder f, of the value estimator stream V, and of the action advantage stream A; and $\theta = \{\theta_{conv}, \theta_V, \theta_A\}$

is their concatenation.

This method can help us to find much more reliable Q values for each action by decoupling the estimation between two streams.

3.4 Multi-step learning

Instead of updating our estimate with only the next state (one-step TD method), we can use the n consecutive states. This method is called n-step TD method [6]. A multi-step of DQN is then defined by minimizing the alternative loss

$$L(\theta) = \left(r_t^{(n)} + \gamma_t^{(n)} \max_{a'} Q(s_{t+n}, a'; \theta^-) - Q(s_t, a_t; \theta)\right)^2$$

where $r_t^{(n)}$ is the truncated n-step return from a given state s_t defined as

$$r_t^{(n)} = \sum_{k=0}^{n-1} \gamma_t^{(k)} r_{t+k+1}$$

Multi-step learning often leads to faster training with a properly chosen n parameter.

3.5 Distributional RL

In contrast to the common approach where the algorithm predicts the average reward it receives from multiple attempts at a task, and uses this prediction to decide how to act, we can learn the distribution of this random return received by the reinforcement learning agent. [1] shows that it is possible not only to model the average reward, but also the variation of the reward. This method has been introduced from the fact that random perturbations in the environment can alter the agent behaviour by changing the exact amount of reward it receives. A variant of Bellman's equation which predicts all possible outcomes, without averaging them, has been proposed.

Value distribution is modeled using a discrete distribution parametrized by $N_{\text{atoms}} \in \mathbb{N}$ and $v_{\min}, v_{\max} \in \mathbb{R}$, whose support is the vector $z \in \mathbb{R}^{N_{\text{atoms}}}$ where z_i is defined as

$$z_i = v_{\min} + (i-1)\frac{v_{\max} - v_{\min}}{N_{\text{atoms}} - 1}, \qquad i \in \{1, ...N_{\text{atoms}}\}$$

The atom probabilities are given by a parametric model $\theta : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^{N_{\text{atoms}}}$ with probabilities $p_i(s,a;\theta) = \frac{\exp(\theta_i(s,a))}{\sum_j \exp(\theta_j(s,a))}$. Then, the approximating distribution d_t at time t is defined on this support, such that $d_t = (z, p_\theta(s_t, a_t))$. The goal is to update the θ parameter in order to minimize the divergence between the predicted distribution and the actual distribution. The are several algorithms to do that, such as embedding the rewards into a graphical model and apply probabilistic inference to determine the sequence of actions leading to maximal expected reward.

As we said, here the distribution of the returns satisfy a variant of Bellman's equation. As we can see on the illustration below, we can get the nest state distribution under policy π, contract it towards zero according to the discount, and shif it by the reward. The last step is a L2-projection of the target distribution onto the fixed support z.

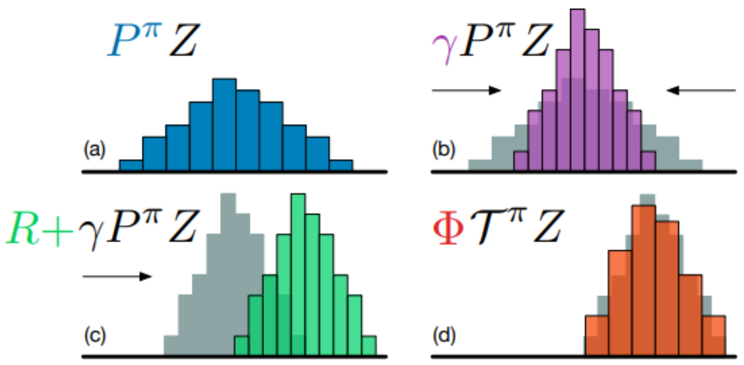

Figure 3: A distributional Bellman operator with a deterministic reward function [1]

We can then minimize the following Kullbeck-Leibler divergence between the distribution d_t and the target distribution

$$D_{KL}(\Phi_z d'_t || d_t) \qquad s.t \quad d'_t = (r_{t+1} + \gamma_{t+1} z, p_{\bar{\theta}})(s_{t+1}, \bar{a}^*_{t+1})$$

where Φ_z is the projection operator and $\bar{a}^*_{t+1} = \arg\max_a Q(s_{t+1}, a; \bar{\theta})$ is the best action to take with respect to the mean action values $Q(s_{t+1}, a; \bar{\theta}) = z^T p(s_{t+1}, a; \theta)$.

This method results in agents that are more accurate and faster to train than previous models. It also add some possibilities such as telling whether and action is safe to take or risky. Indeed, when two choices have the same average outcome, we may favour the one which varies the least.

3.6 Noisy Nets

Two main conventional approaches exist for the exploration/exploitation strategy: entropy reward and ϵ-greedy. The main issue of these two methods is that they don't take into account the current situation the agent is experiencing. [2] proposed a way of overcoming this issue which consists of adding Gaussian noise to the last fully-connected layers of the network. Parameters of this noise will then be learned during training phase along with the agent parameters.

How does it work ?
Let $y = f_\theta(x)$ be a neural network parameterised by the vector of noisy parameters $\theta = \mu + \Sigma \odot \epsilon$. Instead of using the standard linear layer of a neural network $y = b + Wx$, we use a noisy linear layer is defined as:

$$y = (b + Wx) + b_{noisy} \odot \epsilon^b + (W_{noisy} \odot \epsilon^w)x$$

where b_{noisy} and W_{noisy} are learnable parameters, whereas ϵ^b and ϵ^w are noise random variables. There are two types of Gaussian noise. For the Rainbow agent, they used factorised Gaussian noise instead of independent Gaussian noise to reduce the number of independant noise variables. Actually, for independent Gaussian noise, every weight of noisy layer is independent an has it's own mean and variance which have to be learned by the model.

It remains to compute the new loss function, where the usual loss of the neural network is wrapped by expectation over the ϵ for the noisy value function $Q(s, a, \epsilon; \zeta)$ and the noise variable ϵ' for the noisy target value function $Q(y, b, \epsilon'; \zeta^-)$.

$$\begin{aligned}\bar{L}(\mu, \Sigma) &= \mathbb{E}L(\theta) \\ &= \mathbb{E}\left[\mathbb{E}[r_{t+1} + \gamma_{t+1}Q(y, \arg\max_{b\in\mathcal{A}} Q(y, b(y), \epsilon''; \zeta); \epsilon', \zeta^-) - Q(s, a, \epsilon; \zeta)]^2\right]\end{aligned}$$

where $\zeta = (\mu, \Sigma)$. Independent noises are generated to avoid bias due to the correlation between the noise in the target network and the DQN network. Concerning the action choice, we generate another independent sample ϵ'' for the DQN network and we act greedily with respect to the corresponding output action-value function.

This method improves traditionnal methods like entropy reward and ϵ-greedy. It allows the agent to decide when and in what proportion it wants to introduce to explore, where the exploration with be conditionned by the state space.

4 Rainbow agent

All the aforementioned components are integrated into a single agent, which is called **Rainbow**.

Algorithm 1 Training a rainbow agent

```
1: intialize environment env
2: set sum_reward_game = 0
3: for $i = 0, ..., $nb_iter do
4:     action = epsilon-greedy choice
5:     next_state, reward, finish = env.take_action(action)
6:     sum_reward_game += reward
7:     update_epsilon(i)        (we act more and more greedily)
8:     if finish == 1 then
9:         reward = 0
10:    Save the experience in the replay buffer
11:    if finish == 1 then
12:        reset the environment
13:        save the total rewards for this episode
14:        sum_reward_game = 0
15:    if $i >$ batch_size then
16:        compute the loss and backpropagate    (algorithm detailed below)
17:    if $i\%$update_nb_iter == 0 then
18:        update target network with the online network's weights
```

We will now detail how the loss and the backpropagation are computed, which is a very important step in the training.

Algorithm 2 Loss function and backpropagation

```
1: sample (states, actions, rewards, next_states, finish, indices) from replay buffer
2: compute next state probabilities    $(p(s_{t+n}, .; \theta_{\text{online}}))$
3: compute distribution $d_{t+n} = (z, p(s_{t+n}, .; \theta_{\text{online}}))$    (section 3.5)
4: perform the action corresponding to the argmax selection using
   online network: $argmax_a[(z, p(s_{t+n}, a; \theta_{\text{online}}))]$
5: sample new target network noise
6: compute Tz (Bellman operator T applied to z)
7: compute L2 projection of Tz onto fixed support z
8: declare m (distribute probability of Tz)
9: compute cross-entropy loss    (minimises $D_{KL}(m||p(s_t, a_t))$)
10: backpropagate minibatch loss
11: update priorities in replay buffer
```

5 Experiments

We have implemented each of the six extensions indivually in order to compare then to the baseline Deep Q-Network model. We have evaluations our agents on the CartPole-v0 Atari game.

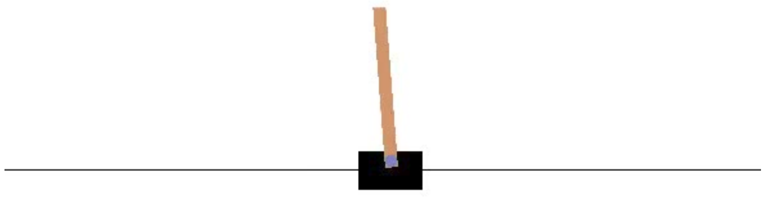

Figure 4: CartPole-v0 Atari game

At the end of each episode, we measure the rewards collected by the agent during this episode. Then the rewards are set to zero to begin a new episode.

5.1 Code structure

This is what you can find in the folder we provide:

- a python file for each baseline model (one for DQN and one for each of its extensions), named `[baseline_name]`.py
- a **config** file with a list of hyperparameters such as the discounting factor, the batch size, the ϵ parameter for ϵ-greedy strategy...
- a file named **model.py** with the neural networks architectures of the different models.
- a file named **buffer.py** with the replay buffer for Experience Replay and the replay buffer for Prioritized Experience Replay.
- a file **layer.py** where we have defined a layer used in the noisy ntetworks.
- a jupyter notebook where all the extensions are trained and compared.

5.2 Results

We have runned the agents for 30,000 iterations. These are the results we have obtained on the CartPole-v0 Atari game.

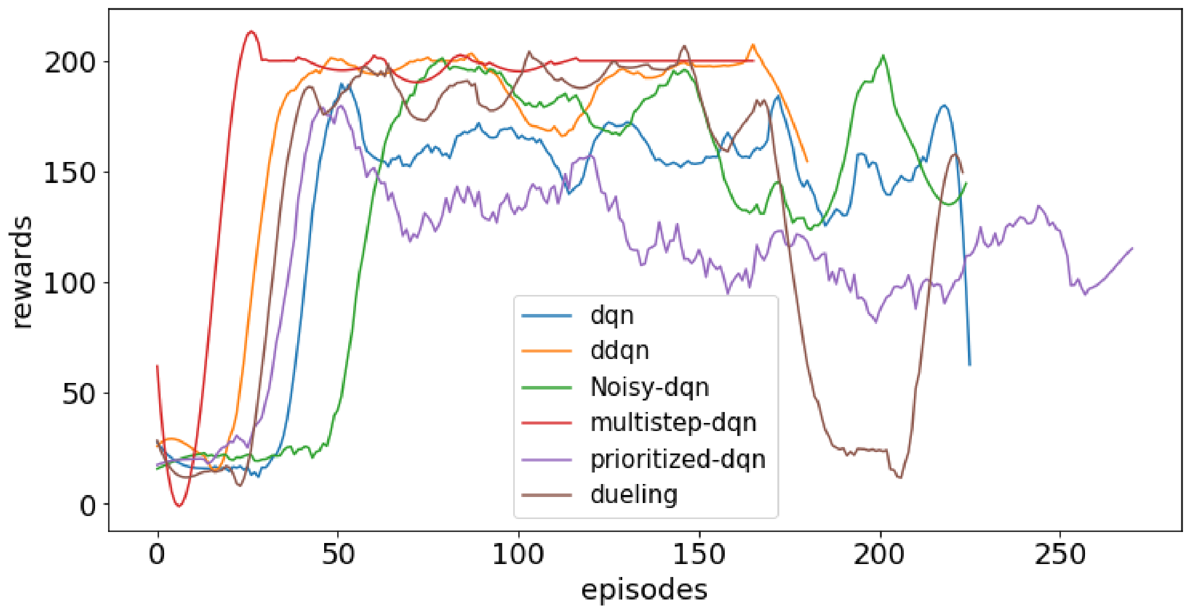

Figure 5: Rewards per episode for each baseline model

As they did in the original Rainbow paper, curves are smoothed but we have chose a moving average over 25 points instead of 5 points for the sake of readability. We have used for that the Savitzky-Golay filter from scipy librairy.

Comments: As we can see on the above figure, as expected, the extensions are performing better than the basic DQN. Except for Noisy-DQN, the extensions are getting more rewards faster than DQN (i.e in a smaller number of episodes). However, at some point, Prioritzed DQN and Dueling Networks are getting less rewards than DQN. But note that the overall goal of the extensions is not to get more rewards faster, but to converge to a better optimum, so it can take a lot more iterations for the extensions to make a notable difference. And we don't have the computational power to reproduce the results exposed in the original paper.

Additional comments: You may notice that distributional DQN and Rainbow do not appear on the plot above. We have implemented both of these methods but were not able to make them converge. It seems that there is a problem with the loss function but we were not able to figure out where it happens. But you can find these implementations in the python files.

References

[1] M. G. Bellemare, W. Dabney, and R. Munos. A distributional perspective on reinforcement learning. *CoRR*, abs/1707.06887, 2017.

[2] M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, C. Blundell, and S. Legg. Noisy networks for exploration. *CoRR*, abs/1706.10295, 2017.

[3] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. G. Azar, and D. Silver. Rainbow: Combining improvements in deep reinforcement learning. *CoRR*, abs/1710.02298, 2017.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb. 2015.

[5] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *CoRR*, abs/1511.05952, 2015.

[6] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[7] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.

[8] Z. Wang, N. de Freitas, and M. Lanctot. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015.