Artificial Intelligence in Super Mario for CS 4100 Artificial Intelligence

Written by Niall Dalton and Samuel Wu-Ochs
Northeastern University Khoury College of Computer Sciences
360 Huntington Ave
Boston, Massachusetts 02115
dalton.n@northeastern.edu, wu-ochs.s@northeastern.edu

Abstract

Super Mario Bros is a popular video game for the Nintendo Entertainment System (NES) in which an agent attempts to complete levels by traversing various obstacles and reaching a goal flag. This paper examines the application of artificial intelligence to create an agent that can effectively play the Super Mario Bros video game. A variety of algorithms were employed, along with a number of enhancements and optimizations to the environment that allowed for improved training speed and test performance. Of the techniques used, the strongest solution was the Deep Q-Learning agent. This performance came at the cost, however, of requiring significant amounts of training time in order to yield satisfactory results. In this paper, we compare the performance of Deep Q-Learning alongside other algorithms, including MuZero, PPO, and AMPED.

Introduction

The primary motivation for solving Super Mario with an AI agent is that it is one of the simpler problems that features a nearly continuous environment. Unlike Pacman, Chess, or some other common problems in AI, the states of Super Mario are not very distinct, and more difficult to quantize as a result. While this introduces new challenges to creating an effective agent for the environment, solving nearly continuous game environments like Super Mario provides insight into solving the commonly continuous problems of the real world. We pose the problem of Super Mario as a reinforcement learning problem with the MDP (S, A, r, P), which are the state, action, reward, and transition probabilities, respectively. The true P is unknown, so a simulator is used. The solution of this problem is to find the optimal policy, i.e. the policy $\pi(s_t)$ that maximizes the expected discounted future reward.

To create the Super Mario environment for our agent to explore, we use the Super Mario AI Gym [3] library. By default, the state of the problem in Super Mario is defined by a 2-dimensional RGB pixel grid of size 256 x 224. These pixels are the portion of the current level visible to the player, and include Mario's location in the level. Formally, the state space is $S = \mathbb{R}^{256x224x3}$, where the 3 represents RGB channels. In practice, this default state offers a far higher level of detail than necessary for the agent, and so several reductions are made for the sake of computation efficiency.

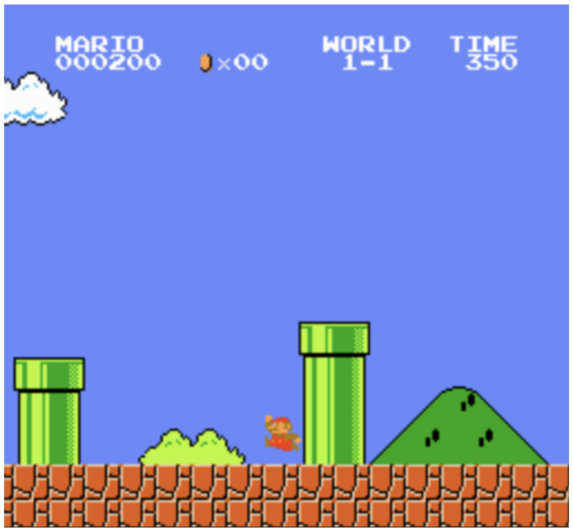

Figure 1: Figure 1: An unfiltered screenshot of the Super Mario Bros. game emulated through the Gym AI library.

The default size of the action space was 256, with one possible action available for each element of the power set of the 8 discrete buttons on the NES. With many of these action combinations being redundant from a gameplay perspective, however, the gym library offers several simplified action lists for agents to use, i.e. $A = \{NOOP, RIGHT, RIGHT + A, RIGHT + B, RIGHT + A + B\}$, where, semantically, right represents movement to the right, A is jumping, and B is running/power-up. Every action is always available to the agent, but it may not have a visual effect on the game depending on the state.

The agent receives rewards for moving to the right and getting the flag at the end of the level. Moving to the left, dying, and doing nothing (time ticks) are penalized. Formally, $r = \Delta x - \Delta t + 25\delta_{ij} - 5\delta_{ai}$, where δ is the Kronecker delta function, and i, j represents the position of the agent and flag, respectively, and a represents the 'death' position of the agent. Δx represents the change in the agent's position, i.e. $\Delta x = x_t - x_{t-1}$.

Our final solution for the agent utilizes the Deep Q-Learning algorithm. This involves using a convolutional neural network (CNN) that analyzes visual features from the state of the problem in order to produce one of the available actions as the output. The CNN is considered the optimal neural network architecture for image analysis, which makes it the ideal fit for our image-based state space of Super Mario. With the CNN receiving a reduced game state as input, we are able to train the network by running episodes of Super Mario through the gym library.

Background

The Deep Q-Learning algorithm is in some ways an advancement on the original Q-Learning algorithm. While the two differ greatly in implementation, the goal of Deep Q-Learning is to utilize the function approximating power of neural networks in order to provide the same policy as that of the optimal Q-Function. The distinction here is that Deep Q-Learning can be used in learning environments where the state-action table would be far too large to accommodate standard Q-Learning. In the environment of Super Mario, this is indeed the case. Taking into account the number of pixels per state multiplied by the number of possible RGB values, the total number of states falls just short of a trillion.

State reductions can help with making Q-Learning feasible, and our initial attempts at making a Super Mario agent are based on this philosophy. Similar to how a Pacman agent can learn to play with decent effectiveness by learning the weights of a few predefined features, we attempt to build a Super Mario agent that determines its actions based on hard-coded features of its current state. Due to the limitations of the gym AI environment, however, only limited information such as Mario's x-y coordinates is easily available to use as features. It would take significantly more advanced image processing to determine high-level features such as obstacles and enemies visible in the state. This loss of information in our Q-Learning implementation leads to the motivation for a Deep Q-Learning approach. The original Q-Learning implementation, although not producing very strong results, provides a algorithmic foundation to build onto in addition to a baseline performance metric for our later agents to compare to.

Related Work

In the environment of Super Mario, one method with the potential to offer increased performance over Deep Q-Learning is the more advanced Double Deep Q-Learning algorithm.

In their implementation of the two algorithms, Grebenisan [1] manages to achieve ten times faster training speed with Double Deep over Single Deep Q-Learning, as well as a significantly higher average reward and win rate by the end of training. This sizable increase in performance results primarily from solving one of the main failures of standard Deep Q-Learning, which is the overestimation of predicted Q-values.

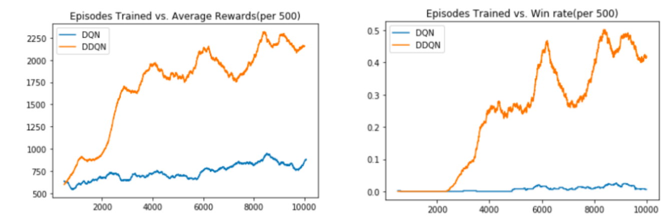

Figure 1: **Left:** Average reward per 500 episodes for 10000 episodes in total, DQN and Double DQN solutions, **Right:** Average percentage of wins per 500 episodes for 10000 episodes in total, DQN and Double DQN solutions

Figure 2: Figure 2: A graph from Grebenisan's article [1] coming their Deep and Double Deep Q-Learning implementations of the Super Mario agent.

Since Q-Learning operates on the fundamental principle of picking the highest expected utility from the available state action pairs, an overestimation of non-optimal actions often causes Deep Q-Learning to choose a non-optimal policy. As Hasselt [2] proves, Double Deep Q-Learning overcomes this failure by introducing a second state-action function that counteracts the overestimation through normalization. This allows the Q-function to converge optimally.

The reasoning for not include the Double Deep Q-Learning algorithm in our final solution is a lack of time to learn the necessary concepts and implement the new features. The Deep Q-Learning algorithm alone is quite complex, and requires long testing periods to yield results. However, with more time the Double Deep Q-Learning would be a promising advancement to our final solution.

Project Description

State Optimizations

Before exploring the algorithms used to train the Super Mario agent, we first discuss the optimizations that allow for improved training speed and testing performance from our agents.

The most immediate issue with the default state of Super Mario for training networks is the large size of the image, which is filled with unnecessary details that create noise in pattern recognition. Conveniently, the Super Mario gym library includes several variations of the standard Super Mario state that help to reduce the amount of details present in the game state. In our Deep Q-Learning solution, we use the rectangle environment, shown in Figure 3, to strip each element of the game down to an easily recognizable monochrome rectangle. To compress the size of the state and reduce computational load, the 256 x 224 grid of RGB pixels is reduced to an 84 x 84 array of grayscale pixels.

As mentioned in the introduction, we are also able to use built-in reductions from the gym library on the action space of Super Mario. For the Deep Q-Learning implementation, we choose the right-only subset of actions. While these 5 actions are a tiny fraction of the total available, this combination proves to be sufficient. At least for the first world of Super Mario, the agent can pass levels by persistently move right while jumping to clear obstacles and enemies. More difficult Super Mario levels may require increasingly com-

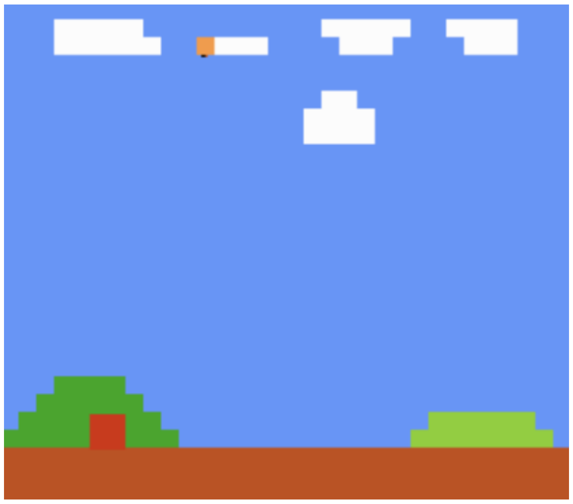

Figure 3: Figure 3: A screenshot of the simplified Super Mario Bros. state to allow for easier pattern recognition by the CNN.

plex action spaces to pass, but the restricted action space allows our agents to learn faster within the scope of this project.

One of the final state optimizations arose in response to some undesirable behavior that occurs in our earlier implementations of the agent. When attempting to clear an obstacle such as a pipe, Mario needs to choose the jump action long enough for him to gain enough altitude to be above the pipe. Due to the nearly continuous environment of Super Mario, the number of consecutive jump actions required numbers about 15-30, depending on the height of the pipe. This makes it extremely unlikely that the agent will ever cross the pipe by randomly choosing to jump the necessary height without any reward to guide it. In response to this learning challenge, we simplify the agent to only choose a new action every 4 in-game ticks. While this sacrifices a degree of our agent's granularity, the resulting agent is still more than capable of responding appropriately to changes in the state.

Using these optimizations, our learning tasks become much simpler and easier to solve.

Deep Q-Learning Description

As mentioned in the introduction, Deep Q-Learning seeks to approximate the state-action Q-values that determine the optimal action to take in any given state. Our implementation approximates the Q-function using a Convolutional Neural Network (CNN) from the Pytorch [4] library that operates over the reduced state image discussed previously.

Shown above is a diagram depicting the training loop of the agent. The loop begins with an initial input state that is passed to the agent's CNN. Composed of 3 convolutional layers, the CNN convolves multiple times over the image input to produce what it determines to be the optimal action for the given state. The Rectified Linear Unit (ReLU) activation function is then applied between each layer. After that action is received and performed in the environment, the agent perceives the resulting next state and reward. It then stores the current state, action, next state, and reward collectively as a single experience in the agent's memory.

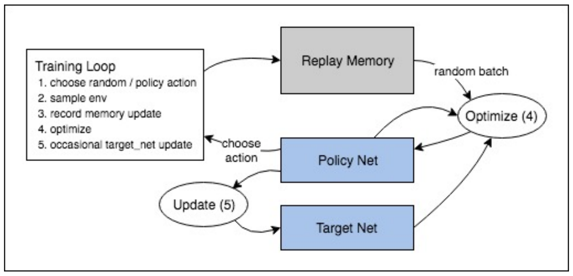

Figure 4: Figure 4: Diagram of the action-feedback loop of the Super Mario agent that optimizes the CNN weights over time using stored experiences.

The final step in the agent loop is to optimize the CNN based on past experiences. A sample of experiences are extracted from the agent memory, and loss is computed for each to determine how well the CNN predicted the actual outcome. To compute loss, we first compute the temporal difference error, δ, using the function below.

$$\delta = Q(s,a) - (r + \gamma \max_a Q(s', a))$$

Next, we compute the loss for the given error.

$$\mathcal{L} = \frac{1}{|B|} \sum_{(s,a,s',r) \in B} \mathcal{L}(\delta)$$

$$\text{where} \quad \mathcal{L}(\delta) = \begin{cases} \frac{1}{2}\delta^2 & \text{for } |\delta| \le 1, \\ |\delta| - \frac{1}{2} & \text{otherwise.} \end{cases}$$

For the Deep Q-Learning implementation, we used the Huber Loss function. This function is effectively mean-squared for small small errors and mean-absolute for large errors, which prevents noisy outliers from causing excessive disruption to the optimization process. The gradient is then computed using the loss, and finally propagated across the CNN parameters to complete the optimization step.

PPO

Proximal Policy Optimization (PPO) was another algorithm we tested against Super Mario [6]. PPO follows a typical policy gradient setup, in which some stochastic policy π_θ and some advantage function estimator $\hat{A}_t$ are used in the following gradient estimator:

$$\hat{g} = \hat{\mathbb{E}}_t \left[\nabla_\theta \log \pi_\theta (a_t \mid s_t) \hat{A}_t \right] \tag{1}$$

The PPO authors then define a ratio of an updated policy to an old policy as $r(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$. Then, the authors define a clipped policy gradient objective as follows:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t, clip\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t \right) \right] \tag{2}$$

This objective is used in conjunction with a squared-error loss on value estimation, alongside an entropy bonus which is thought to encourage exploration. In practice, PPO uses an actor-critic style setup with Generalized Advantage Estimation (GAE) for advantage estimation [6]. We follow this procedure in our implementation.

MuZero

MuZero makes improvement upon the seminal work of AlphaZero by introducing a learned MDP model rather than using a simulator [5]. Specifically, MuZero works by predicting k steps into the future at time t using a set of neural network models parameterized by some parameters θ. The MuZero models predicts three outputs at time t: the policy p_t, the value v_t, and the reward r_t. Note that p is used to denote policy here rather than the traditional π.

MuZero predicts these three quantities through three different sub-network functions. The *representation* function h_θ is used to predict some latent state space s_0 from a given set of observations, i.e. $h_\theta(o_1, ..., o_t) = s_0$. In turn, the *dynamics* function is used as a learned MDP model. That is, the dynamics function computes state transitions on some latent state space rather than the true state space, allowing the flexibility for the whole model to capture only the useful parts of the state during transitions. Specifically, the dynamics function g predicts the reward and subsequent state, i.e. $g_\theta(s^{k-1}, a^k) = r^k, s^k$. The final sub-network is the so called *prediction* function f. The prediction function is used to convert the latent state into a predicted policy and value; this function is equivalent to the actor-critic setup of the likes of PPO except it uses a latent state. Specifically, the prediction function computes as follows: $f_\theta(s^k) = p^k, v^k$; again, note that p^k is some predicted policy at step k.

Like its predecessor AlphaZero, MuZero makes use of Monte-Carlo Tree Search (MCTS) to do empirical prediction of the optimal policy. However, unlike AlphaZero, instead of using a simulator to do rollouts, the dynamics function is used, i.e. the dynamics function predicts the next node (state at step k) using actions as edges. The MCTS produces an empirical estimate of the optimal policy, denoted as π_k.

With all these components, the MuZero objective can now be formulated. It consists of four terms: a reward term, a value term, a policy term, and a weight decay term. It is formulated as follows:

$$L_t^{MU}(\theta) = \sum_{k=0}^{K} l^r \left(u_{t+k}, r_t^k\right) + l^v \left(z_{t+k}, v_t^k\right) + l^p \left(\pi_{t+k}, \mathbf{p}_t^k\right) + c\|\theta\| \tag{3}$$

where u_k and z_k are the true reward and value, respectively. Also note that π_k is, as before, the MCTS estimate of the optimal policy. c represents some arbitrary coefficient to scale the parameters. l^r, l^v, l^p are all individual loss functions, typically taken to be the mean squared error (MSE) or the negative log likelihood (NLL). In practice, we use values of $k = 0..5$ and use MSE for l^r and l^v, and use NLL for l^p.

AMPED

Advantaged Markovian Proxy Evolution Dynamics (AMPED) is an iterative improvement on the MuZero algorithm. There are two important changes relative MuZero: (1) AMPED combines the MuZero objective and the PPO objective; (2) AMPED uses an n-th order Markov evolution dynamics (NOMAD) function instead of the first order Markov dynamics function used in MuZero.

Specifically, the AMPED objective is formulated as follows:

$$L(\theta) = -L^{CLIP} + L^{MU} - L^{ENTROPY} \tag{4}$$

The AMPED objective is minimized using standard gradient descent techniques, i.e. ADAM.

AMPED extends the first order Markov dynamics function used in MuZero by allowing g (the dynamics function) to take in n previous states, $s_{i-n}, ..., s_i$. The n states are initialized to s_0, as generated by the representation function h. Formally, we have: $g_\theta(s_{k-n-1}, ..., s_{k-1}) = r^k, s^k$. This allows AMPED to break the standard Markov assumption (that all states rely only on the previous state). We hypothesize that breaking this assumption by introducing the n-th order Markov chain input will allow the dynamics function to generalize better over time.

Finally, AMPED uses an empirical advantage estimate during MCTS backup phase. This advantage is calculated from the predicted Q-value and the predicted value (from the prediction function f), rather than solely using the Q-value as MuZero does. We refer the readers to [5] for details on the MCTS backup phase.

Results and Experiments

DQN

The main experimental factor of the Deep Q-Learning algorithm is the structure of the CNN used to create the agent policy. While it is simple to verify that various state optimizations such as frame skipping do have an immediately positive impact on agent performance, determining the ideal neural network structure for learning Super Mario is more ambiguous. Therefore, the majority of experimenting for this solution is done on the CNN. The metrics used to measure the effectiveness of a given architecture were the speed of training and the average reward achieved by the agent over the course of its training.

Certain network structures prove to be prohibitively slow for feasible training even with the GPU-aided computations. During testing, we experiment with networks of up to 5 layers. As we increase number of layers, keeping all other aspects the same, training speed slows considerably as expected. At 5 layers, even decreasing the channel sizes, the number of frames processed per second drops to less than 10. Using only a single convolutional layer gives extremely high training speed, but at that cost of more complex pattern recognition capabilities. Following this logic, testing found

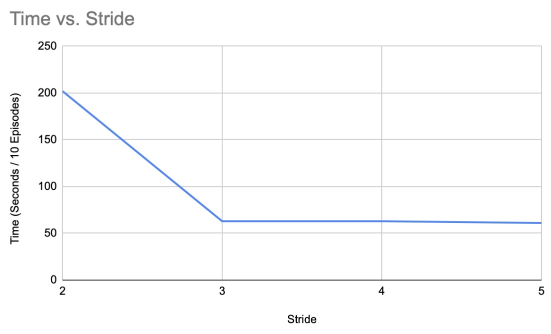

Figure 5: Graph of average training time for different CNN strides. Running time was extremely long for strides less that 3.

that 3 convolutional layers provided the optimal balance of performance and recognition strength.

Having established the number of convolutional layers, the remaining factors tested are channel size, kernel size, and stride. Channel size and stride both have strong performance impacts due to their direct impact on computational load. In our implementation, we find that channel sizes over 100 result in prohibitively slowed performance. For this reason, we choose channel sizes of 32 and 64 for our convolutional layers. The stride of our layers, or the amount of pixels between each iteration of the convolution, can also result in slowed performance. Greater performance costs are found with low strides earlier in the network, since the number of parameters is otherwise reduced in subsequent layers. Due to this behavior in stride experimentation, we choose higher strides in the beginning of the network, and decrease down to lower strides in the second and third layers of the network. Finally, we found in testing that kernel size could be modified with only minor performance consequences. We also did not find significant learning behavior changes with varied kernel sizes of 3 or more. Therefore, the decisions of kernel sizes are fairly arbitrary and unmotivated by any major reasoning.

Using the final network configuration determined earlier, we train our Deep Q-Learning agent and observe its effectiveness as an agent over time. Above is a graph of the average net rewards collected by the agent across a training period of 500 episodes. While the results are somewhat noisy, there is a visible upward trend of performance that shows the agent's capability for learning. One unfortunate aspect found in testing is that the agent takes a considerable amount of training before it can complete the first level of the game, over 2000 episodes. Consistently beating the first level is not something that we found in testing, even after more than 4000 episodes of training. At this point, it seems that we reach the limitations of what we can accomplish with our current implementation. However, the Deep Q-Learning agent does perform with significantly increased effectiveness over our initial implementation of the feature Q-Learning agent. Especially notable is while the Q-Learning agent's reward graph stagnated after only less than 100 episodes, the Deep Q-Learning agent continues to show improvement well beyond that point.

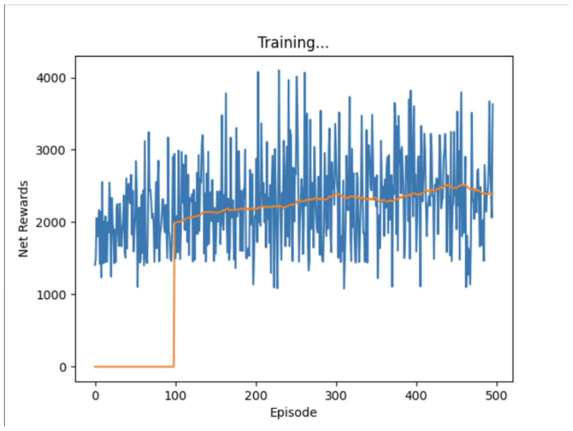

Figure 6: Graph of the net rewards collected by the Deep Q-Learning Super Mario agent across a training period of 500 episodes.

PPO

PPO was implemented in a similar fashion, using PyTorch, as well as the Rllib library. A three layer CNN was used with similar hyper-parameters to the DQN. A entropy coefficient of 0.01 was used on 4 different workers, each with 5 separate environments to run in parallel. GPU training was also used. Batch sizes of 4000 were used and PPO was trained for 100 epochs. The results are shown in Table 1.

AMPED

Likewise, AMPED is implemented in the same fashion as PPO. The n-th order parameter is set as $n = 3$, also with $k = 5$, $sims = 10$ number of simulations, and batch size of 256, which were used to train for 10 epochs. These parameters are likely limiting the performance of AMPED, but time constraints made it impossible to test for larger parameter values. The results are also shown in Table 1.

Algorithm	Maximum Episodic Reward
PPO	631
AMPED	782

Table 1: Results for PPO and AMPED.

While the results for PPO and AMPED are similar, AMPED does achieve a higher overall maximum episodic reward, indicating that it may learn to solve Super Mario Bros better than PPO. Moreover, AMPED trained for 90% fewer iterations than PPO.

Conclusion

In this report, we presented the problem of playing Super Mario Bros. as a reinforcement learning problem. We explained various optimizations to make the problem tractable, as well as several algorithms including DQN and AMPED, that attempt to solve this problem. Finally, we demonstrated the efficacy of these algorithms by training them to solve Super Mario Bros and we illustrated several results. Future work could be done to test these algorithms for more episodes.

References

[1] Andrew Grebenisan. *Building a Deep Q-Network to Play Super Mario Bros*. Sept. 2020. URL: https://blog.paperspace.com/building-double-deep-q-network-super-mario-bros/.

[2] Hado Hasselt. "Double Q-learning". In: *Advances in Neural Information Processing Systems*. Ed. by J. Lafferty et al. Vol. 23. Curran Associates, Inc., 2010, pp. 2613–2621. URL: https://proceedings.neurips.cc/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf.

[3] Christian Kauten. *Super Mario Bros for OpenAI Gym*. GitHub. 2018. URL: https://github.com/Kautenja/gym-super-mario-bros.

[4] Adam Paszke. *Reinforcement Learning (DQN) Tutorial¶*. 2017. URL: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html.

[5] Julian Schrittwieser et al. *Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model*. 2020. arXiv: 1911.08265 [cs.LG].

[6] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG].