M2Vec: MDP Embeddings through NLP

Ryan Tolsma, Drew Gregory, Kendrick Shen, Daniel Tan
{rtolsma, drewgreg, kshen6, dtch1997}@stanford.edu

June 12, 2019

We present a method for automating feature engineering for deep Q-learning on Markov decision processes (MDPs). We apply Word2Vec, a technique originating from natural language processing, to random walks on MDPs to generate embeddings of state-action pairs into vectors. We show that deep Q-learning performs well using these vectors as features, proving that the embeddings contain meaningful information about the relationships between state-action pairs.

1 Introduction

In recent years, MDPs have gained widespread use as models for stochastic decision tasks. As such, learning optimal policies for MDPs is a field of great interest. Q-learning is a simple tabular algorithm for finding optimal policies. In practice, for MDPs with large numbers of states and actions, Q-learning is often combined with function approximation, where the agent also learns a function $f(s, a) : S \times A \to \mathbb{R}$ which aims to predict the reward using the state and action as features. This allows the algorithm to generalize its training experience to unseen but "similar" states. Often, when f is not known in closed form, deep Q-learning networks (DQNs) are used as function approximators. However, designing appropriate features for f is highly nontrivial and often requires domain expertise and prior knowledge about the MDP in question. If these assumptions turn out to be incorrect, the performance of a DQN is hindered.

To tackle this, we present an approach based on applying a natural langauge processing method to MDPs. We generate random walks on the MDP, which can be thought of as "sentences" where each state-action pair is a word, then apply Word2Vec to generate embeddings of state-action pairs in $\mathbb{R}^m$. We show that this approach automates the process of feature engineering for deep Q-learning.

2 Background

We assume working knowledge of MDPs and provide only a brief exposition. Informally, the environment consists of a set of states S with discrete-time updates. At each time step, the agent can select an action from the set of actions A, upon which the MDP transitions stochastically to a different state and the agent receives some reward. The goal for the agent is to maximize the reward achieved.

Formally, an MDP is defined as a tuple (S, A, T, R, γ), where S is the set of possible states, A is the set of actions, $T : S \times A \times S \to [0, 1]$ is the transition probability function, $R : S \times A \times S \to \mathbb{R}$ is the reward function, and $\gamma \in [0, 1]$ is the discount factor. We restrict our attention to MDPs with finite states and finite actions, such that $S = \{1, \cdots, n\}$ and $A = \{1, \cdots, k\}$ for some integers $n, k \in \mathbb{N}$. Where the meaning is clear from context, we use S, A to denote n, k respectively. The agent's behaviour at each state of the MDP can be described by a policy $\pi : S \to A$, where the agent selects action $a = \pi(s)$ at state s. The optimization objective is the long-term expected discounted reward, which can be described as:

$$L(\pi) = \sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t), s_{t+1})$$

2.1 Tabular Methods for MDPs

For any MDP with $\gamma < 1$ and bounded R, where $|R(s, a, s')| \leq C$ for all $(s, a, s') \in S \times A \times S$ and some $C \in \mathbb{R}$, observe that $|L(\pi)| \leq \frac{C}{1-\gamma}$, i.e. $L(\pi)$ is also bounded. As such, it makes sense to talk about the optimal policy π^* that maximizes $L(\pi)$. Let $Q^*(s, a)$ be the value of action a taken from state s for an agent following π^* and let $V^*(s) = \max_a Q^*(s, a)$ be the value of a state. It is well-known that the values of $Q^*(s, a)$ satisfy the Bellman equations:

$$Q^*(s,a) = \sum_{s' \in S} T(s,a,s')\Big[R(s,a,s') + \gamma V(s')\Big]$$
$$V^*(s) = \max_{a \in A} Q^*(s,a)$$

The optimal policy can then be found by taking $\pi^*(s) = \arg\max_a Q^*(s,a)$ for each $s \in S$. This forms the basis of tabular methods like value iteration, which start with random policies, and then iteratively improve them to eventually converge to the optimal policy. However, such methods have two limitations: (1) storing the values of $Q^*(s,a)$ requires $\theta(|S||A|)$ memory, which may be infeasible for large MDPs, and (2) they need explicit knowledge of the transition probabilities $T(s,a,s')$. (1) is usually addressed by learning a parametric function $f(s,a;\theta) : S \times A \to \mathbb{R}$ that aims to predict $Q(s,a)$, and (2) is addressed by estimating $T(s,a,s')$ from observed sequences of MDP transitions. However, these fixes are not perfect; for (1), the performance of the algorithm depends greatly on the accuracy of the choice of model $f(s,a;\theta)$, and for (2) estimating $T(s,a,s')$ requires $\Theta(|S|^2|A|)$ memory.

2.2 Random Walk Embeddings

We present an alternative solution to the two problems presented above. For each MDP, we generate a corpus of random walks on the MDP following a policy π, then apply Word2Vec to generate embeddings of vectors in $\mathbb{R}^m$. Notably, these embeddings capture syntactic information (which state-action pairs are likely to occur close together) as well as semantic information (which state-action pairs have similar rewards.), effectively representing both T and R.

3 Methods and Experiments

Our methods are split into three main parts: (1) generating random MDPs, (2) embedding each state-action pair into $\mathbb{R}^m$ using Word2Vec, and (3) training DQNs using the generated embeddings to learn the optimal policy.

3.1 Dataset: Synthetically-generated MDPs

For our dataset, we could not find standard benchmark MDPs in the existing literature so we decided to create synthetic MDPs. Our general MDP generator takes parameters S, the number of states, and A, the number of actions. For each value of (S,A), T was generated as a $S \times A \times S$ array with entries drawn independently from a uniform distribution over $[0,1]$ then normalized to a true probability distribution (In practice, we actually generated the entries of T from $[10^{-6},1]$ to avoid dividing by zero in the normalization step). In addition, R was generated as a $S \times A \times S$ array with entries drawn independently from a standard normal distribution.

Three subclasses of MDPs were generated: (1) dense MDPs, where each state-action pair is allowed to transition to every other state; (2) a sparse MDP, where each state-action pair can only transition to a small number of other states; and (3) a bipartite MDP, where the states are split into two equally-sized groups such that actions always go between the two groups. Although this is a limited selection, we hope that the diversity presented here allows us to capture properties of different types of realistic MDPs.

3.2 Word2Vec

For each MDP (with roughly 100 states), a training corpus of 10000 random walks (each of length 100) was obtained, parsed into sequences of (s,a) pairs, and embedded. We ran several experiments to show that our embeddings captured relevant structural information of the MDP because of the inherent difficulties in directly evaluating embedding efficacy. A primary concern was that state-action tuples belonging to the same state s (elements of $E_s = \{(s',a)|s' = s\}$) would not co-occur along our random walks as only one of them could ever be chosen at a time. To see whether our embeddings would accurately recover the similarities between elements of E_s we ran the unsupervised K-Means algorithm while varying k to generate the loss curves plotted below. The elbow point of our clustering loss curves typically occurred where k was equal to the number of states, indicating that elements of E_s for each s were likely clustering together. Comparing the cluster labels among tuples in E_s indicated that roughly $70\% - 80\%$ of the elements of E_s typically clustered together, with the outlier tuples usually indicating extremely high transition probabilities to another state. This implies that by clustering our embeddings, we can recover significant structural information about the MDP state-graph. Another concern was tuning the embedding dimension m to maximize structural information recovery–running the same clustering experiments to graph K-means loss versus dimension, we found that the dimension with the loss curve elbow point closest to the number of states occurred at $d = 15$. Thus, for future experiments we will use this informed choice of embedding in a $\mathbb{R}^{15}$.

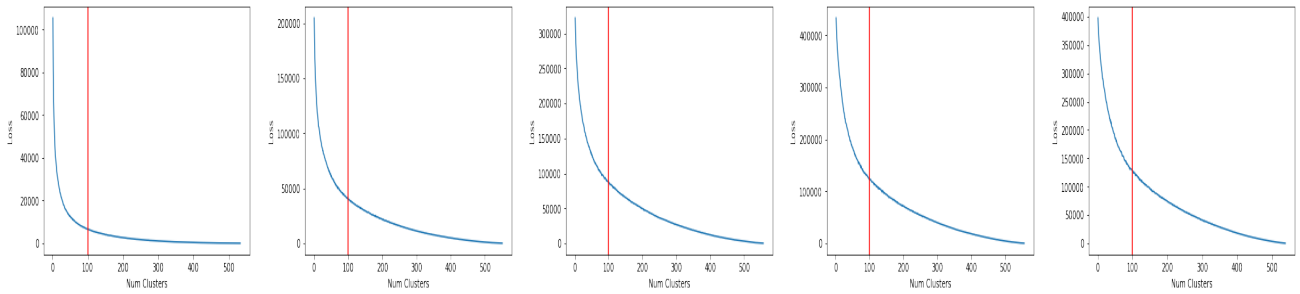

Figure 1: KMeans-Loss for $d = 5, 10, 15, 20, 25$, on a synthetic MDP with 100 states.

3.3 Training DQNs

Once the MDP embeddings were generated, the Q-network was developed iteratively as the reward function approximator through a series of MDP walks. Two networks were trained in tandem: a reward function "emulator" that provided the policy for the walks and was continuously updated, and a stationary target to serve as the ground truth reward function against which to compare the emulator. Each walk began with a randomly-chosen start state and the subsequent learning algorithm comprised three essential steps:

1. From state s, choose an action a according to an epsilon-greedy policy. In other words, with probability ϵ choose a randomly, otherwise choose a to maximize $Q(s, a)$ according to the emulator's reward function. After transitioning to state s', collect reward r and store (s, a, r, s') in replay memory. The randomness parameter ϵ was subjected to exponential decay with respect to the number of iterations.

2. Sample a batch of (s, a, r, s') experience tuples from the replay memory and perform an update step on the emulator's approximation of the various $Q(s, a)$'s using the target as the ground truth and Huber loss to avoid dramatic changes.

3. After every i total steps have been taken, update all of the target's parameters to match those of the emulator.

Because of the sequential nature of MDP episodes, consecutive experience tuples are highly correlated; to work around overfitting to a local state space, we use a replay memory that continuously stores the most recent D pieces of experience. Thus, each mini-batch we collect for training has randomized data drawn from a similar distribution. Additionally, to avoid having a non-stationary target (a training issue that amplifies the effect of recent updates on the network's approximations in the local state space), we must train two separate networks simultaneously.

4 Results

4.1 State Clustering of MDPs

For each MDP, our workflow produces embeddings of each state-action tuple (s, a) into $\mathbb{R}^{15}$. As briefly mentioned earlier, we found that the state-action pairs of each MDP form clusters by state (using Euclidean distance as a metric). To illustrate this, we applied our procedure to the three types of MDP described in section 3.1. After obtaining the embeddings, we applied a t-distributed stochastic neighbor embedding (t-SNE) to reduce the dimension and enable plotting. The resultant plots are shown below:

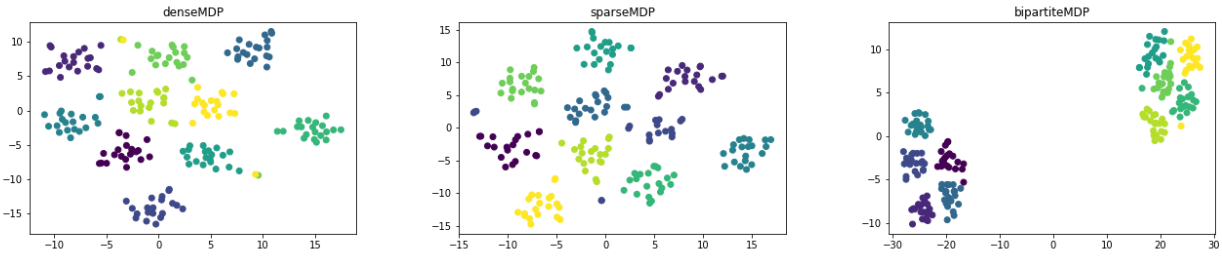

Figure 2: t-SNE of embedded state-action pairs, coloured by state. For this experiment, $S = 10$, $A = 20$. All three types of MDP exhibit clustering by state.

4.2 Comparison of DQN to Value Iteration

After training, the DQN has hopefully learned to correctly predict the expected value of each state under the optimal policy. To evaluate the success of this approach we used value iteration to find the optimal Q^*, V^* for each MDP in our dataset. The learned values of $V(s)$ provide a baseline against which to compare the output of the deep Q-learning network.

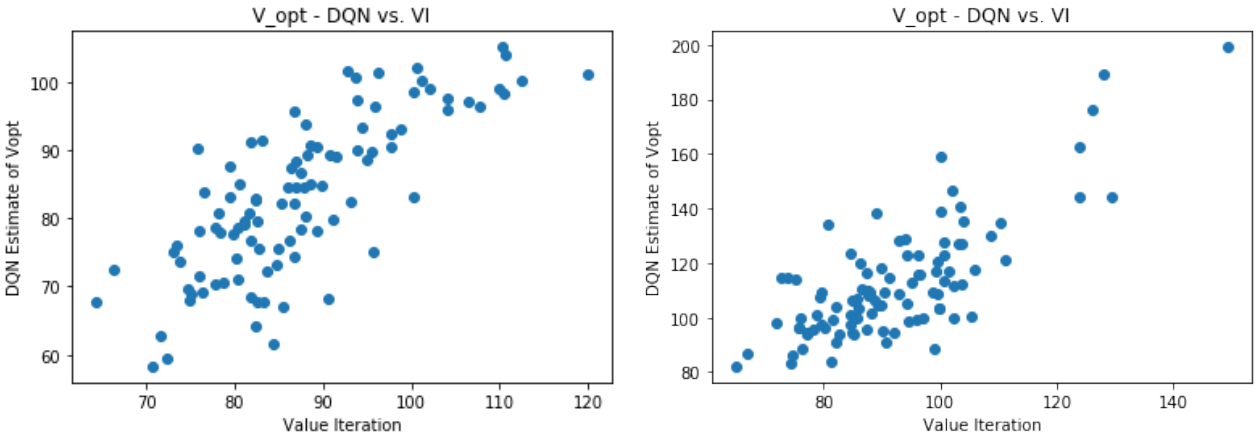

Figure 3: Scatterplot of V_{opt} of 100 states in a "Sparse MDP" learned via deep Q-learning and value iteration with the same horizon of .8:

The DQN estimates and optimal Values are positively correlated with a correlation coefficient of .011309 (left plot) and .05766 (right plot; second trial run). Although the positive association is promising, we are currently far from having embeddings capture state-action pairs well enough to learn an optimal policy.

4.3 DQN and Model-based Monte Carlo

We evaluated our embeddings performance with a DQN against value iteration that had knowledge of the transition probabilities. We realized that there is a time/accuracy tradeoff that can be used to compare this new model against more naive algorithms like Model-Based Monte Carlo, which would require estimating all the transition probabilities. We also noticed that our random walks required to generate these embeddings could also be leveraged to estimate the MDP and give the evaluation function an improved starting target.

Below are our results regarding relative performance as a function of the number of iterations (or walks). Performance is evaluated as Mean Squared Error loss relative to the values estimated in value iteration with full knowledge of the actual MDP possibilities. Note that the tradeoff is almost perfectly inversely proportional for Model-Based Monte Carlo, and that the combination of DQN and Monte Carlo had greater loss but somewhat more optimal asymptotic behavior. The DQN will inherently perform worse than a Model-Based Monte Carlo asymptotically with regards to MSE loss, but making a combination of the two seems to suffer from greater loss as a result of transfer learning difficulties.

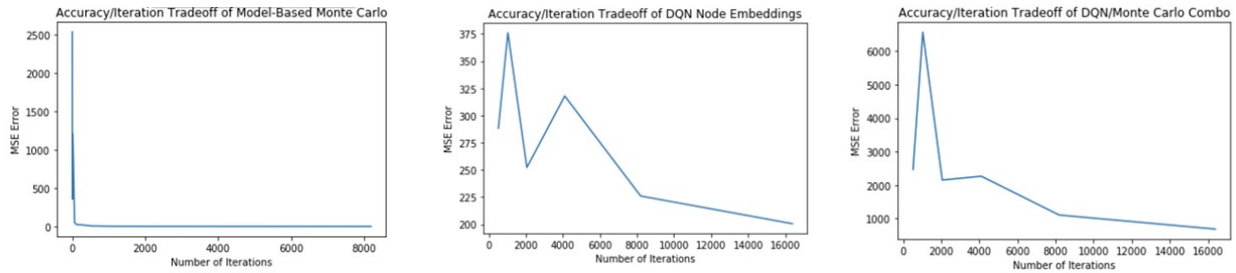

Figure 4: A comparison of performance as a function of number of walk iterations.

5 Conclusion

In conclusion, our proposed algorithm maps state-action pairs of MDPs into $\mathbb{R}^m$, which can then be used as features for deep Q-learning for the MDP. These embeddings cluster by state, indicating that the embeddings capture real information

about the state-action pairs. Additionally, they can be used in deep Q-learning to learn optimal policies of MDPs, negating the need for manual feature engineering.

We were surprised that our combination of a DQN with the model trained on our model-based monte carlo apprpoximation had significantly larger loss than a naive DQN. We believe that this discrepancy is a result of such an aggressively different target.

6 Future Work

Given additional time, we would like to explore the following improvements to our approach:

1. Using our combination of model-based monte carlo and a DQN seems to have all the difficulties and peculiarities of transfer learning. We would like to explore different improvements and tricks developed in that area of machine learning to better allow the DQN and model-based monte carlo target to cooperate.

2. We definitely have not explored all hyperparameters involved in our model. Some hyperparameters to consider include the standard hyperparameters associated with our neural network architecture in our DQN, the length of the walk and the number of walks.

3. This automated feature engineering will undoubtedly produce tradeoffs for different MDP structures. We aim to compare different types of MDP structures (dense MDP or sparse MDPs? MDPs with a central structure or disjoint structure?) and establish which particular classes of MDP's are best suited for this automated feature engineering paradigm.

4. MDP's are somewhat analogous to graphs, and there has been tremendous innovation regarding applying embedding techniques to undirected graphs [1] [2]. Interestingly, graph embeddings have strayed from the traditional skipgram model in favor of convolutions on a graph's adjacency matrix [3] [4]. Although we have currently focused on applying skipgram co-occurrence techniques to produce our embeddings, our underlying MDP architecture has a transition matrix, which we may try to leverage to produce some analogy to GCNs.

5. Unfortunately, skipgrams are agnostic to the ordering of a path. This means that applying deep walks works well in undirected graphs but fails to capture direction of edges in directed graphs. Order matters especially for MDPs because resulting states from actions don't even have a one-to-one mapping (after an action, transitions can map you to an arbitrary number of states). We may try to implement the directional skip-gram techniques found in [5] to encode this directionality and better associate the future rewards of successive states.

7 Individual Contributions

Daniel was responsible for designing, testing, and maintaining the MDP generation architecture. Daniel also ran some experiments and wrote large parts of the poster and final report. Ryan also developed the MDP and embedding architecture, in addition to running K-means to determine the embeddings' retention of MDP structure as well as improving the tradeoff analysis code. Kendrick developed the DQN structure and worked on the final report writeup, and Drew wrote the code for value iteration, the monte-carlo/DQN combo, wrote the accuracy/iteration tradeoff analysis code.

References

[1] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. *CoRR*, abs/1606.09375, 2016.

[2] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. *CoRR*, abs/1607.00653, 2016.

[3] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.

[4] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. *CoRR*, abs/1403.6652, 2014.

[5] Yan Song, Shuming Shi, Jing Li, and Haisong Zhang. Directional skip-gram: Explicitly distinguishing left and right context for word embeddings. pages 175–180, 01 2018.