Progress Report

Annabelle Cloutier

March 21, 2023

1 Paper

1.1 Network Structure

All of the below is based on the work in [1] The network structure is incredibly generic most of the way through, but notes are written here to avoid having to parse through the equations that describe the network structure in the paper. Note that at any step where learned weights are used, the ReLU function is used on the resulting vector. The only exception is for the last set of learned weights in the readout layer.

1.1.1 Node Embedding

The MPNN (Message Passing Neural Network) structure used converts each vertex into a set of m observations and encodes them into a n-dimensional embedding using learned weights. The paper states that they use 64 dimensional embeddings and they do in code, but this could be any size.

The observations in the paper as well as their justifications for them are as follows:

1. Whether the current vertex belongs to the solution set S or not;

   This is a local observation. The reason they state this observation is important is that it provides useful information for the agent to make a decision on whether the vertex should be added or removed from the solution set.

2. The immediate cut change if the current vertex state is changed;

   This is also a local observation. Just as with the above observation, they state this provides useful information for decision making. They also state is that it allows the agent to exploit having reversible actions. This makes intuitive sense, as knowing the difference between having a vertex in the solution or not can allow the agent to make an informed decision on whether to add or remove it, or in this agent's case, possibly reversing an action.

3. The number of steps since the current vertex state was changed;

   Also a local observation. For this observation, they mention that it provides a simple history to the agent to prevent looping decisions. Because this observation gets larger as time goes on and a vertex is unchanged, the agent will be further incentivised to choose other vertices if it gets stuck in a loop choosing the same few nodes over and over. They also state it allows the agent to exploit reversible actions just as with observation 2. This specific observation could be important as the value increases as the vertex remains unchanged, which can entice the agent to choose vertices that have not been chosen in a long time to encourage exploration just as it will discourage loops.

4. The difference of the current cut value from the best observed;

   This and all future observations are global. This observation they state ensures the rewards are Markovian. Just as with observations 2-4, this observation they also mention allows the agent to exploit having reversible actions, which makes intuitive sense. The agent knowing the difference between the current cut value and the best one observed, along with the other observations, can allow the network to make informed decisions on whether the current solution is a promising set to explore.

5. The distance of the current solution set from the best observed;

   This observation as with observation 5 they state ensures the rewards are Markovian. Again, this observation allows the agent to exploit reversible actions.

   The difference between this and observation 5 is that observation 5 compares the cut value only. This observation compares the solution sets and counts the number of vertices that differ between the best observed solution set of vertices and the current one. This is not explained in the paper, but can be seen in the code, specifically in src/envs/spinsystem.py, in the SpinSystemBase class' step function.

   Example for clarity:

   Best seen bitmask of vertices: $[0, 0, 1, 0, 1]$

   Current bitmask of vertices: $[0, 1, 1, 0, 1]$.

   Difference: 1.

   Counting the number of different vertices they do in code by subtracting the two and counting the number of non-zero values. This can also be done using a bitwise XOR and doing a sum on the resulting bitmask.

6. The number of available actions that immediately increase the cut value;

   Once again, this observation ensures Markovian rewards, they also mention that this allows exploitation of reversible actions, which is more easily understandable.

7. The number of steps remaining in the episode

   The only reasoning they have for this observation is that it accounts for the finite time used for solution exploration.

1.1.2 Edge Embedding

The edges for each vertex are also encoded into a separate n-dimensional embedding, same size as for each vertex, once again learned. The input for this step is the set of m observations of the neighboring vertices catenated with the weight on the connecting edge, creating an $m + 1$ dimensional vector for each neighboring vertex. All of these vectors are then summed and passed through a learned layer, creating an $n - 1$ dimensional vector. At this stage, the resulting $n - 1$ dimensional vector is divided by the number of neighbors, catenated with the number of neighbors, and passed through another learned layer, resulting in an n-dimensional embedding representing the edges for a vertex.

The end result of these two steps are an n-dimensional embedding representing a vertex and another representing it's neighbors, all created using the same learned weights for every vertex. Meaning there will be $2|V|$, n-dimensional embeddings.

1. Why 64 dimensions on the node and edge embeddings?

   It's nice that it's 64 but finding the reason why will be important to make informed decisions on modifying the network.

1.1.3 Message Passing

Here there is a message pass layer and an update layer. The message pass per vertex is summing the products of the connected vertices and the weight, divided by the number of neighbors, catenated with the edge embedding for that vertex and then passed through learned weights, resulting in a n-dimensional vector.

The next is the update layer, which is the embedding for that vertex, catenated with the message and passed through another set of learned weights, into an n-dimensional vector, representing the "new" embedding for that vertex.

The message then update is performed K times. Mentioned in the paper and corroborated in the code, this is done 3 times, but can be done however many times necessary.

1. Why have new network layers for these steps?

   As with the decision on the hidden layers sizes, understanding the justification for why certain steps are passed through learned functions before more calculations are performed will help make informed decisions on network changes.

1.1.4 Readout

The readout layer goes through each vertex, summing the embeddings for the neighbors of that vertex, dividing the result by the number of vertices in the entire graph and then passing it through learned weights, resulting in an n-dimensional vector.

The embedding for the vertex itself is then catenated to the resulting vector and passed through another set of learned weights (without applying ReLU), resulting in a single output value. This value represents the Q-value for that node, which is used by the algorithm to determine which vertex will be added/removed from the solution set on that step. The node associated to the maximum Q-value is added to the solution set if it doesn't yet belong to it, or removed from the solution set if it belongs to it.

1.2 Reward Shaping and Training

1.2.1 Q Function, Q values and Training

The Q function is an idea derived from Q-learning [2] which proposes a way for an agent to learn how to behave in an environments. It is defined as the expected value of the discounted sum of future rewards for any state-action pair of an environment. When an optimal Q function is derived, the agent chooses which action to perform in a certain state by selecting the state-action pair which corresponds to the largest Q-value, which is expected to give them to largest reward. The equation following equation represents this idea:

$Q^{\pi}(s,a) = E[\sum_{t=0}^{\infty} \gamma^t R(s_t) | s_0 = s, a_0 = a, \pi]$

where π is a policy, mapping a state to a probability distribution over the actions, $R(s)$ is the reward for a given state and γ^t is a discount given to change whether the agent prefers immediate or future rewards.

This Q function is learned by a Markov decision process. The agent traverses the environment and rewards are given at each step depending on the result of the action the agent performs.

Trying to find this Q function was known to be unstable or even diverge when nonlinear function approximators, like neural networks, were used to try and represent it [3]. In *Mnih et al.*, they propose an approach to Q learning with two main ideas; namely experience replay and an iterative update, adjusting the Q values towards target values only periodically [4]. The agent during training only chooses the actions associated with its current policy with probability $1 - \epsilon$ and otherwise chooses a random action. They demonstrate experimentally that with these ideas, they were able to train a model that had significant performance improvements compared to other existing models on 49 different Atari games.

These ideas are replicated in the training of ECO-DQN. For every episode in the training phase, a random graph is sampled with a random solution set. Then, for each time step in that episode, the agent chooses a random vertex based on the existing Q function with a probability ϵ and a vertex dictated by it's Q function otherwise. Then, the starting state, chosen vertex, reward and resulting state are added to the experience replay memory. Finally, after some fixed set of time steps, the network is updated by stochastic gradient descent on a minibatch sampled from the experience replay memory.

1.2.2 Reward Shaping

The reward for in a certain state is given by the difference between the cut value in that state and the highest cut value seen so far in the episode, divided by the number of nodes. If the difference is negative, it's instead set to 0. The justification behind this choice is that a negative reward will discourage the agent from exploring states that give an immediately worse cut value, even if other cuts including that change later on may give better cut values. If this were to happen, it would discourage the agent from exploring different cut values and cause it to stay in a very small space near a locally maximal cut value. Because previously seen optimal states are stored in memory, it's beneficial to later let the agent explore more states in an attempt to find more locally optimal states, even if not always better than the previously seen local optimum.

They also define a reward for reaching locally optimal states of $\frac{1}{|V|}$. This is once again to encourage exploration. Because the previously seen best cut is stored, it's beneficial for the agent to explore other states, even if other local optimums may be worse than the previously found best cut. They state that local optima in combinatorial problems are typically close to each other and therefore by giving rewards for reaching new local optimums, the agent learns to hop between them during training as it is rewarded for this behaviour. Without this, they state that because there are far too many states to be visited in a finite time, which is typical for combinatorial optimization problems, it is therefore useful to focus on these subsets of states near local optimums.

They demonstrate this behaviour by observing the probability during any given timestep that the agent will either revisit a state, find a locally optimal state or find the maximum cut on the validation set. They show that as the number of timesteps goes up, the probability that the agent revisits a state goes up, as does the probability of finding the maximum cut, while the probability that the agent finds a local optimum goes up very quickly and stabilizes for the rest of the time steps, showing that the agent picks a certain set of local optimum and explores that space by revisiting previously seen states.

1.3 Discussion on Generalization

Most of the internal structure is generic for any graph, however because of the large number of changes that would likely need to be made to observations as well as the interpretation of the output for many other types of problems, it is likely that some changes to the internal structure would have to be made for the agent to make more educated decisions on new problems.

Everything outside of observations (input) and Q-values (output) only propagates information throughout the graph. This means the internal structure likely could remain mostly the same for any problem, so long as good decisions are made for the observations and that the output or it's interpretation are modified to fit new problems.

It is possible for more complex problems that some of the internal structure for the message and update sections of the network may have to change in order to accommodate for extra information. It is also possible that global observations could be embedded somewhere within the internal structure of the network instead of as input observations for every vertex.

The main issue comes with the output and it's interpretation. Each vertex is represented by a single value as the output, and that value is interpreted as adding or removing it from the solution set, in the context where the solution is a subset of the vertices in the graph. More specifically for the Max Cut problem, the vertex associated with the maximum Q-value (output value) is taken and then either added or removed from the solution set, depending on whether it already belongs to it or not.

This approach works fine for a problem like Maximum Cut or Minimum Vertex Cover where the solution can be represented as a set representing chosen vertices for the solution, however any extra constraints forces the output interpretation to be completely redesigned.

For example, the Traveling Salesman Problem where the solution is an ordered list of vertices would not correctly work as the current implementation of simply adding or removing a vertex from the solution could not work.

The Minimum K-Cut Problem which can have an arbitrary number partitions of the graph would also not work with the current model as it would require extra decision making on deciding which set to move a vertex to.

The Minimum Bisection Problem can be represented as a set of chosen vertices, however the extra constraint that the number of chosen vertices has to be the same as the number of unchosen vertices makes the current output interpretation incomplete for solving this problem.

1.4 Benchmarks

The paper displays the performance of the graph in reference to S2V-DQN, a similar paper where the algorithm does not allow for reversing actions as well as a greedy algorithm. It also compares it's performance against modifications of itself, namely where some observations are restricted, intermediate rewards are not given for reaching locally optimal solutions, as well as keeping it from reversing its actions.

They use GSet graphs G1-10 and G22-32 for calculating the approximation ratio, as well as the Physics dataset.

1.5 Graph Generation

They train and test on Erdos-Renyi [5] and Barabasi-Albert [6] graphs.

2 Code

The code holds the capability of running all of the above tests, but they're not explicitly written and must be generated via self-written code.

2.1 Running Code

They very generously provide a README file that specifies the exact commands to run in order to train, test and validate networks. However, there is no code for reproducing their specific tests. These all need to be hand-coded. The only results I've reproduced so far are training and testing.

2.2 Graph Generation

Unsure of the exact implementation, it seems they use NetworkX's ability to generate random graphs in order to do this. It's completely unnecessary though seeing as they have many test, validation and benchmark graphs pre-built within their code that can be used. If absolutely necessary, creating random graphs and storing them within a pickle file as they do would likely lower the workload on inputs as the code for parsing through these types of files is already prebuilt.

The code for generating new random graphs exists in src/envs/utils. This includes the code for generating Erdos-Renyi and Barabasi-Albert graphs, as well as classes for numerous other types of random graphs.

2.3 Input Conversion

The code for converting graphs into observations seems to exist within the file src/envs/spinsystem.py but further code inspection is needed to see exactly how this information is represented and decoded by the agent.

2.4 Benchmarks

They generously provide testing, validation and benchmark graphs in Pickle files, which is a special type of object file for Python, as well as their solutions, which are also stored in Pickle files, but are simply a boolean (technically floats, just 1.0 and 0.0) list for determining whether a node is in the solution set or not.

References

[1] T. D. Barrett, W. R. Clements, J. N. Foerster, and A. I. Lvovsky, "Exploratory combinatorial optimization with reinforcement learning," in *Proceedings of the Thirty-fourth AAAI conference on Artificial Intelligence*, arXiv, 2019.

[2] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, pp. 279–292, 1992.

[3] J. Tsitsiklis and B. Van Roy, "Analysis of temporal-diffference learning with function approximation," in *Advances in Neural Information Processing Systems* (M. Mozer, M. Jordan, and T. Petsche, eds.), vol. 9, MIT Press, 1996.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, “Human-level control through deep reinforcement learning,” *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[5] P. Erdős, A. Rényi, *et al.*, “On the evolution of random graphs,” *Publ. Math. Inst. Hung. Acad. Sci*, vol. 5, no. 1, pp. 17–60, 1960.

[6] R. Albert and A.-L. Barabási, “Statistical mechanics of complex networks,” *Reviews of Modern Physics*, vol. 74, pp. 47–97, jan 2002.