Learning Pivot Rule in Simplex Method with Reinforcement Learning

Yongli Peng

Abstract

Large scale linear programming is always an important problem. With the great development of deep learning in these years, people have been thinking about applying these techniques in this classic, or perhaps the originator, topic of the optimization field. We here try to reproduce one of those who attempts to utilize the deep reinforcement learning (RL) to improve the classic results of how to choose a pivot rule in simplex method. We learn a data-driven strategy to determine whether the Danzig or the steepest-edge strategy should be used in each iteration. The numerical result shows this method will make the algorithm more stable (jump out of cycling) while won't impair the efficiency much. The critics of the paper generally thinks the instance in the paper is not large and hard enough. So the author cannot convince others this method can beat traditional methods in real large scale problem. Our work applies this method for relative large problem. Though we have not tuned the parameters much, we think the result is acceptable. And the later experiment shows such a method can really learn a data-driven strategy for a specific type of problem and beat classic means in some sense for those particular applications.

1 Introduction

Organizing the resources in a scientific way to reduce costs or maximize efficiiency is always an important topic in operation research. The network resource allocation problems, as a typical case in this sense, can be formulated in a clear way as a linear programming (LP). So how to solve LP for large scale problems has attracts big companies all along.

As we have pointed out in the abstract, nowadays as we are entering the big data era, the improvement of our ability and efficiency to deal with large data may provide some innovative means to solve LP. And some people are really doing it by using machine learning or deep learning to help us overcome possible obstacles in those canonical methods.

The No-Free Lunch theorem essentially says there are no general-purpose optimization algorithms tha work well on all problems. And this might be a belief for all the scientists in the field of optimization and algorithm. So develop a problem specific algorithm is always natural and desired by people. This is where machine learning can have a kick in. This year there has been a literature review on combining reinforcement learning and combinatorial optimization in arxiv. Khalil et al. learn how to make branching decision on the branch-and-bound tree in mixed-interger programming. Bonami et al try to learn whether to linearize the quadratic objective in mixed-integer quadratic programming. Bertsimas and Stellato uses machine learning to solve online mixed-integer optimization with efficiency. More particular, Bello et al uses RL with a policy gradient method to sovle the TSP. This paper, from another point, tries to use deep Q learning to solve the TSP after relaxing it to a LP and uses simplex method. But the problem scale is too small and this might be the main reason why this paper is rejected at last. So we here try to scale this method and apply it to a larger linear problem with the application of network resources allocation and show its performance compared with traditional methods. Actually as the problem can be converted as a multi-commodity flow problem in the operation research. Our trained strategy actually can be applied to all the problems of this type as well.

2 Background and original problem formulation

The original network resources problem can be modeling and formulated as an optimization problem in the following way. Consider a general network model modeled as a graph with nodes and edges (links). The resources are transported on this graph from source nodes to destination nodes. This requirement urges the commodity to flow on the network and at a fixed time we can assume every link has a specific flow currently, not exceeding the capacity. So basically we want to determine the flow on each link once we know the requirements and storage at each node. This problem can be formulated as solving linear equations naturally. But there might be many solutions (if we have a solution) to the equations. And the producers and the decision makers always want a plan with the minimal costs or the maximal efficiency. So in this way the problem is transformed as a LP. When multiple commodities are transported on the same network, the problem is called the multi-commodity flow (MCF) problem. Our problem of network resources allocation, if you treat the information flow as a resource and all the station as the nodes, is a specific instance for this problem.

Therefore we can regard this problem as a MCF and instead of using the node-arc formulation we can consider the arc-path formulation, which is more close to the real applications. To be precise, assume we have L uni-directional links indexed by ℓ. Each link has its own link capacity $c_l > 0$. Let $\mathbf{c} = (c_1, c_2, \cdots, c_L)$ be the link capacity vecotr. Consider K flows in total moving on the network index by k, i.e., assume we have K commodities to be transported in the network. For each flow k, there is a origin and a target, and there are P_k paths available, which are indexed by p_k. More mathematically, we can use a matrix (of $L \times P_k$) to desribe the relation between links and paths precisely, the so-called arc-path formulation in MCF. Let the routing matrix for flow k be of the following form:

$$\boldsymbol{R}_k = \begin{pmatrix} R^k_{1,1} & R^k_{1,2} & \cdots & R^k_{1,P_k} \\ R^k_{2,1} & R^k_{2,2} & \cdots & R^k_{2,P_k} \\ \vdots & \vdots & \ddots & \vdots \\ R^k_{L,1} & R^k_{L,2} & \cdots & R^k_{L,P_k} \end{pmatrix},$$

where $R^k_{l,p_k} \in \{0,1\}$, with 1 meaning the path p_k transverses the link l and 0 otherwise. Then we align them horizontally and get the network's routing matrix (of the dimension $L \times P$, with $P = \sum_k P_k$):

$$\mathbf{R} = (\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_K).$$

Then we let $\mathbf{x}_k = (x_{k,1}, x_{k,2}, \ldots, x_{k,P_k})$ be the vector whose element $x_{k,p} \geq 0$ indicates the rate allocation of at path p for flow k, with $x_{k,p} = 0$ implies this path is not chosen for the flow k. Now we can formulate the problem as the optimization problem below:

$$\begin{array}{ll} \min_{\mathbf{x}} & \max_l \frac{\mathbf{R}[l]\mathrm{x}}{c_l} \\ \text{s.t.} & \mathrm{Rx} \leq \mathrm{c} \\ & \|\mathrm{x}_k\|_1 = d_k \\ & \mathrm{x} \geq 0. \end{array}$$

The service provider want the flow on the network to be balanced so as to avoid congestions. So the objective function is the maximum of the link utilization ratio. The ratio is the rate on the link, $\mathbf{R}[l]$ divided by the maximal rate endurable c_l for that link to avoid congestion. Here d_k indicates the requirement for the flow k.

Actually we can treat this problem as a MCF, so the later experiment might use the MCF data for testing. A classic MCF problem is

$$\begin{array}{l} \text{Min} \sum_{(i,j)\in A} \sum_k c_{ij}^k x_{ij}^k \\ \text{s.t.} \sum_j x_{ij}^k - \sum_j x_{ji}^k = \begin{cases} d_k & \text{if } i = s_k \\ -d_k & \text{if } i \in t_k \\ 0 & \text{otherwise} \end{cases} \\ \sum_k x_{ij}^k \leq u_{ij} \text{ for all } (i,j) \in A \\ x_{ij}^k \geq 0 \quad \forall (i,j) \in A, k \in K \end{array} \Rightarrow \begin{array}{l} \text{Min} \sum_k \sum_{P\in P^k} c^k(P) f(P) \\ \text{s.t.} \sum_k \sum_{P\in P^k} \delta_{ij}(P) f(P) \leq u_{ij} \quad \forall (i,j) \in A \\ \sum_{P\in P^k} f(P) = d^k \quad \text{for } k = 1 \text{ to } K \\ f(P) \geq 0 \quad \text{for } P \in \bigcup_{k=1}^K P^k. \end{array}$$

The left hand is in the node-arc formulation and the right one is in the arc-path formulation. $c_{ij}^k/c^k(P)$ indicates the cost of transporting the commodity on the link (i,j)/the path P. $x_{ij}^k/f(P)$ is the rate (or flow in this case) on the on the link (i,j)/the path P. d_k is the requirement for the commodity k and u_{ij} is just the capacity in our case. So our problem is just a MCF with arc-path formulation after using a maximal of linear functions as objective. But the constraits are the same and so does the feasible set. So we can just use MCF data as a realistic instance.

3 Linear programming formulation

In this section we set it clear how the above original problem can be transformed as a LP. If we just set $t = \max_l \frac{\mathbf{R}[l]\mathrm{x}}{c_l}$ as a auxiliary variable, then the problem can be written as:

$$\begin{array}{ll} \min_{\mathbf{x},t} & t \\ \text{s.t.} & \mathbf{Rx} \leq \mathbf{c} \\ & tc_l - \mathbf{R}[l]\mathbf{x} \geq 0 \\ & 1^\top \mathbf{x}_k = d_k \\ & \mathbf{x} \geq \mathbf{0}, t \geq 0. \end{array}$$

Evidently, this is a LP when you treat $(\mathbf{x}, t)$ as variables. To use simplex method, we need to convert the problem further as a canonical LP. We add slack variables μ, λ and get:

$$\begin{array}{ll} \min_{\mathbf{x},t} & t \\ \text{s.t.} & \mathbf{Rx} + \boldsymbol{\lambda} = \mathbf{c} \\ & t\mathbf{c} - \mathbf{Rx} = \boldsymbol{\mu} \\ & 1^\top \mathbf{x} = \mathbf{d} \\ & \mathbf{x} \geq \mathbf{0}, t \geq 0. \end{array} \Rightarrow \begin{array}{ll} \min_{\mathbf{x},t} & t \\ \text{s.t.} & \mathbf{Rx} + \boldsymbol{\lambda} = \mathbf{c} \\ & t\mathbf{c} + \boldsymbol{\lambda} - \boldsymbol{\mu} = \mathbf{c} \\ & 1^\top \mathbf{x} = \mathbf{d} \\ & \mathbf{x} \geq \mathbf{0}, t \geq 0. \end{array} \Rightarrow \begin{array}{ll} \min_x & c^\top x \\ \text{s.t.} & Ax = b \\ & x \geq 0, \end{array} \tag{1}$$

where we make a transformation on the equality constrait and get a simpler one. So the parameters in the canonical form is $c = (0, 0, \cdots, 0, 1)^\top$,

$$A = \begin{bmatrix} R & 0 & I_L & 0 \\ 0 & -I_L & I_L & u \\ S & 0 & 0 & 0 \end{bmatrix}, \text{ where } S = \begin{pmatrix} s_{1,1} & s_{1,2} & \cdots & s_{1,P} \\ s_{2,1} & s_{2,2} & \cdots & s_{2,P} \\ \vdots & \vdots & \ddots & \vdots \\ s_{k,1} & s_{k,2} & \cdots & s_{k,P} \end{pmatrix}.$$

To prevent abusing of notation we here use $\mathbf{u}$ to replace $\mathbf{c}$ in the original question. Here I_L is the $L-$dime identity matrix, $s_{k,p} = 1$ indicates the path p is for the flow k and 0 vice versa. Note that every path only transports a single flow, so in each column of S there's only one nonzero element. This property is preserved in our later generation of the training data for this problem in RL. $b = \begin{bmatrix} \mathbf{u} \\ \mathbf{u} \\ \mathbf{d} \end{bmatrix}$. $X = \begin{bmatrix} \mathbf{x} \\ \boldsymbol{\mu} \\ \boldsymbol{\lambda} \\ t \end{bmatrix}$ is the variable after transformation.

We use the above form of canonical LP as a start point to apply the simplex method.

4 Simplex method

4.1 A brief overview

Since for LP the objective as well as the constraints are all linear. Viewed from the geometry we can see the feasible set is in fact a high-dimensional polyhedro. And minimizing a linear function can be think of minimizing its height after making some appropriate translations and rotations. So it's natural to think the extreme will be attained at the vertices and there's a theorem to ensure this. So the basic idea of simplex is just to move from one vertex to another adjacent to it. Since the objective is also convex, the local minimal is just a global one, so we just need to check the adjacent points. If no adjacent point can improve the objetive then we have solved the LP. Moreover, there's theorem proving the extreme point has at most m non-zero entries if we assume A is of $m \times n$, which are called basic variables. So combined all of the above properties, simplex method just aims to move from BFS to another adjacent BFS, where the basic feasible solution (BFS) is defined through a feasible x and a index subset $B \subset \{1, 2, \cdots, n\}$:

1. B contains exactly m elements

2. $i \notin B \Longrightarrow x_i = 0$, indicating x is supported in B
3. The matrix $B = [A_i]_{i \in B}$ is nonsigular.

So at each iteration we just need to add a variable q into the basis B and delete one p (adjacent means the line connecting these two BFS will have at most $m+1$ non-zero elemn). Further analysis shows that p can be chosen in a canonical way, but the choice of q is controversial and there's multiple strategy to choose it, which will greatly affect the performance of simplex method. This rule is called pivot rule and the task of RL is to learn this pivot from the training data and provide a data-driven way in choosing the variable q to enter the basis.

The follwing figure make it more clear the process of simplex method:

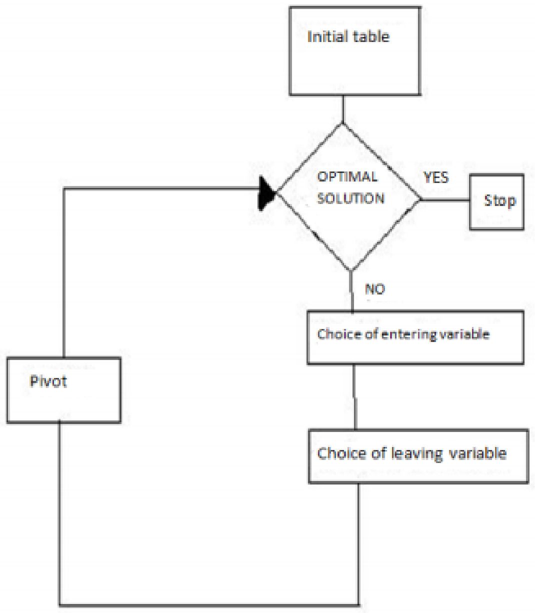


Figure 1: The main process of simplex method

More precisely the algortihm is given below (index set $\mathcal{N} = \mathcal{B}^c$, matrix $N = [A_i]_{i \in \mathcal{N}}$):

1. Start from a BFS x and its associate basis $\mathcal{B}$ with $B = [A_i]_{i \in \mathcal{B}}$, $x_B = B^{-1}b, x_N = 0$.
2. Calculate the reduced costs $s_N = c_N - N^\top (B^{-1})^\top c_B$. If $s_N \geq 0$, the x is optimal and we stop it. Else we choose $q \in \mathcal{N}$ with $s_q < 0$. The choice is made by the pivot rule.
3. Compute $u = B^{-1}A_q$. If $u \leq 0$, the problem is unbounded. We stop it and have $f^* = -\infty$. Else just let $x_q^+ = \min_{\{i|u_i>0\}} \frac{x_{B(i)}}{u_i}$ and $p = \arg\min_{\{i|u_i>0\}} \frac{x_{B(i)}}{u_i}$. Of course $x_p^+ = 0$.
4. Change the basis: $\mathcal{B}^+ = \mathcal{B} + \{q\} - \{p\}$, $\mathcal{N}^+ = \mathcal{N} + \{p\} - \{q\}$.

Although this algorithm is quite simple and easy to implement, there're still some challenges in the way. In each iteration we need to calculate B^{-1}, which is expensive and of the complexity $O(m^3)$. There's a more efficient way to compute it, but in our experiment the result is not that satisfying and we just use the ordinary inversion process. Another problem is cycling, which is observed frequently in large scale LP and also in our example when Danzig or steepest edge is used. One good thing of RL is to help it avoid cycling and hence improve the accuracy. In tradition people use the Bland's rule or Lexicographically rule to avoid it. But the numerical experiment with Bland's rule shows it's very inefficient particularly for large scale LPs. Compared with it, the strategy learned by RL can help improve the accuracy as well as not impair the efficiency much. Actually we observe the algorthm stop descending when it's stuck in the cycling, so we think it's the cycling that mainly affect simplex's accuracy. Later in experiment we just compare their ultimate value and therefore their accuracy to show this aspect of avoid cycling.

4.2 The candidate pivot rule

So the main interests in this report is just the pivot rule, that is, how to choose the variable that will enter the basis. There are many heuristics in past years on this topic, like Bland's rule, Dantzig's rule, steepest-edge rule and greatest improvement rule, etc. The most commonly used ones are Dantzig's rule and the steepest edge rule.

- Dantzig's rule computes the reduced cost $s_N = c_N - N^\top (B^{-1})^\top c_B$ and chooses the most negative component. Its index will enter the basis in the next iteration. They choose $q \in \arg\min_{\bar{c}_q<0} \bar{c}_q$, where $\bar{c}_q = c_q - c_B B^{-1} A_q$.

- Steepest edge rule just makes a normalization based on the Dantzig's rule, using the corresponding updated constraint matrix columns (existing simplex codes always changes b, A, c in each iteration to implement the algorithm). So they just choose $q \in \arg\min_{\bar{c}_q<0} \frac{\bar{c}_q}{||B^{-1}A_q||}$, where we just use vector's 2-norm in this case.

There're sure many other pivot rules available. But as these two are the one used the most frequently, the author just chose these two as the action space for the RL. We merely reproduce their result and just these two rules are considered in RL and experiments below. But we also test for Bland's rule for few cases and the result will be listed.

5 DeepSimplex: using RL to learn the pivot

Now we demonstrate how the reinforcement learning can be applied to learn the pivot, which is called DeepSimplex. The main workflow can be seen from the figure below:

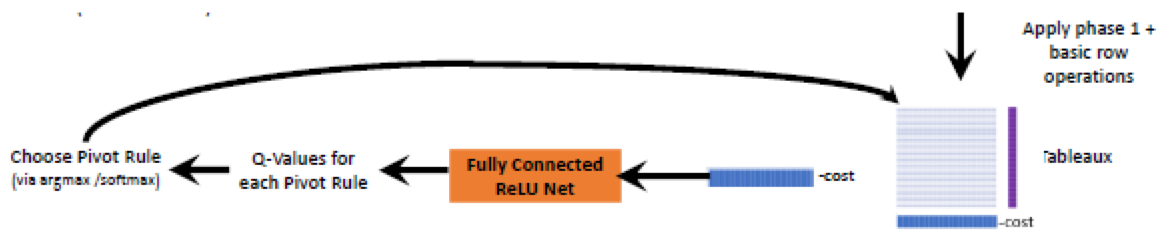


Figure 2: The main process of DeepSimplex, learning the pivot using RL

The main goal in RL is try to learn a strategy that will tell us which one: Dantzig or steepest-edge, we should use in each iteration of simplex method. And we attempt to use deep Q-learning to learn such a strategy, which is just a distribution given the states we are facing. More specifically in each iteration we just use the reduced costs and the objective value to be fed into a DeepNet which approximates the Q-value and choose the pivot with larger Q-value.

Initially we will find a BFS to start with, then we first do nothing and just chosen uniformly from the two pivot. Then after some iteration we use $\epsilon-$greedy methods to choose the pivot and use Q-learning to train the network. After training we used the above way to test our model or just use our model to solve a LP with simplex method, but the LP should have the same dimension $m \times n$.

The following are some key features in the process.

Training set preparation: first describe how the training set is generated. Note later we will feed the reduced cost and the objective into the network, so the dimension is important. In other words, the trained RL can only be applied to solve the LP with the exactly same dimension. This might be another shortcoming for DeepSimplex. In our experiment we generate all the parametric matrix R, S, u, d randomly. The R, S matrices only have 0,1 entries and S has only one nonzero element in each column. We preserve these properties and assume R has i.i.d. entries with Bernoulli distribution, i.e., we let $R_{ij} = 1$ with prob p and 0 with prob $1 - p$. p is pre-determined and fixed in the training and testing process. When we try to use the strategy learned in RL to real data we shall retrain the model with an appropriate p. I use the sparsity of the matrix R in real data as p, i.e. the nonzero ratio in R.

The matrix S is designed in a similar way. To help our initialization (find a BFS at the beginning), we set the first K columns in S to be an identity matrix. The latter columns is chosen uniformly from $\{e_1, e_2, \cdots, e_K\}$, with e_i indicating the $i-th$ entry being 1 and others being 0.

We assume u, d are sample from the same distribution with different center and scale. Since they should be positive in MCF, we assume their entries are from the gamma distribution. Also we try to make $u > d$ for an easy initialization and thus we set $d_k \sim \Gamma(2, 2)$ and $u_l \sim \Gamma(2, 2) * 3 + 20$ and they are also i.i.d sampled.

We trained the model with 1000 instances in total. But we will restricted the simplex in 50 steps for each instance to save time. That is we will terminate the instance after 50 simplex iteration and initialize a new one.

Testing methods: We use several ways to test the model and tries to make a complete description of its performance. We first use 50 instances generated in the same way as the one in the previous section generating training sets. These instances shows our model can reach greater accuracy and prevent from entering cycling easily. Such a mthod is also appied in the training process every 50 epochs to find the best model.

Moreover, as we have observed generally steeepest edge will be more accurate than Dantzig while it sacrifices the efficiency. So we will also generate a special instance, in the same way as above, where Dantzig will outperforms steepest edge and apply our model into this problem and see what happens.

Finally we will apply our model in some real datasets to test its performance. The real data are from some MCF problems and we find missing value is a severe phenomenon in this case. In practical we will delete all the items with some feature missing and the resulting problem is often not large enough. That's why we try to generate big scale LP by ourselves. So when we test our model for real data we will retrain our model with different L, P, k, p and see the performance.

Initial point: First we shall find a BFS to begin with. Thanks to the great form of the MCF problem, we can always find a BFS in the feasible set. Note the coeffcient matrix is in the form:$A = \begin{bmatrix} R & 0 & I_L & 0 \\ 0 & -I_L & I_L & u \\ S & 0 & 0 & 0 \end{bmatrix}$, we can see the dimension is $m = 2L + K, n = P + 2L + 1 > m$. So it's already of full row rank, if every commodity has at least one path to transport it, otherwise we will delete this commodity from the LP. Then for every commodity we choose one path p to enter the basis, and the value $x_p = d_k$ is just the requirement for this commodity.
Then we let $\boldsymbol{\lambda} = \mathbf{u} - \mathbf{Rx}$, note in real application $\boldsymbol{u}$ is the capacity and we here just choose one path to transport the commodity k to meet the requirement d_k, so such a plan should be reasonable and not cause congestion. So generally we will think $\boldsymbol{\lambda} \geq 0$ for such a plan. But in experiment we may met some problems in this way, typically when we randomly generates the training sets. We will try to choose the index p that maximizes $\sum_i u_i \cdot \text{sign}(R_{ip})$ in the implementation, we think such a k will produce the maximal u possible and prevent us confronting any $\lambda_i < 0$. If such a revision also does not produce appropriate λ as well we wil regenerate the LP.

Then what remains are just the equation $\lambda - \mu + tu = u$, that is, $tu = \mu + Rx$. It's impossible to make $t = 0$ here, so we will let exactly one variable in mu to quit the basis. So we just let $t = \max \frac{Rx[i]}{u_i}, j = \arg\max \frac{Rx[i]}{u_i}$. Then $\mu = tu - Rx$ will be positive except for μ_j.

In conclusion we let $x_p = d_k$ and choose path p for every commodity k, the other entries in x is zero. $\lambda = u - Rx$ for appropriate chosen x. $t = \max \frac{Rx[i]}{u_i}$ and $\mu = tu - Rx$. The initial basis $\mathcal{B} = \{p : x_p = d_k\} + \{P+1, \cdots, P+2L, P+2L\} - \{P+L+j\}$ where $j = \arg\max \frac{Rx[i]}{u_i}$ and obviously we have $|\mathbf{B}| = 2L + K = m$.

Action and State space: As we have mentioned we only consider the Dantzig rule and the steepest edge. So the action $a_t = 0$ if the Dantzig's rule is used and 1 otherwise. The state at time t is just the reduced costs and the objective value.

Reward function: We use the reward function from the paper, which is

$$R(s_t, a_t) = \begin{cases} 0 & t > T \text{ or } \ell' = \ell^* \\ 1 - \frac{1}{T} & t \leq T, a_t = 0, \text{ and } \ell'(s_t) > \ell(s_t, a_t) = \ell^* \\ -\frac{1}{T} & t \leq T, a_t = 0, \text{ and } \ell(s_t, a_t) > \ell^* \\ 1 - \frac{1+w}{T} & t \leq T, a_t = 1, \text{ and } \ell'(s_t) > \ell(s_t, a_t) = \ell^* \\ -\frac{1+w^2}{T} & t \leq T, a_t = 1, \text{ and } \ell(s_t, a_t) > \ell^* \end{cases}$$

Here the author uses T as the max number of unweighted iteration (only use Dantzig or steepest edge) to limit the size of Q-values. But here due to our limited computing resources, I just set $T = 50$ as I will cutoff the simplex method after 50 iterations in the training process. The weight w, as the author put it, implies how much more costly is the steepest-edge compared with the Dantzig. It's computed from the ratio of the time used for each iteration for these two methods. We get the ratio is around 0.08 and so set $w = 0.08$ in later experiment. $\ell'(s_t)$ is the objective before the action is taken, $\ell(s_t, a_t)$ is the objective after the action is taken and ℓ^* is the optimal value.

Q-Value function The Q-value function is defined as:

$$Q^\pi(s, a) = \lim_{T\to\infty} E\left[\sum_{t=1}^{T} \gamma^{t-1} r_t \middle| s_1 = s, a_1 = a\right],$$

where s, a is the initial state and action, γ is a discount factor. Since we will terminate it finitely in this case, we just set $\gamma = 1$ in out model. The optimal Q-value function is $Q^*(s, a) = \max_\pi Q^\pi(s, a)$. The Bellman equation is:

$$Q^\pi(s, a) = \underset{s'\sim P}{\mathbf{E}}\left[R(s, a, s') + \gamma \underset{a'\sim\pi}{\mathbf{E}}\left[Q^\pi(s', a')\right]\right]$$

and

$$Q^*(s, a) = \underset{s\sim P}{\mathbf{E}}\left[R(s, a) + \gamma \max_{a'} Q^*(s', a')\right].$$

So the dynamic programming informs us to solve the RL by the Q-value iteration:

$$Q^*_{k+1}(s, a) = R(s, a) + \gamma \sum_{s'} P(s, a, s') \left(\max_{a'} Q^*_k(s', a')\right),$$

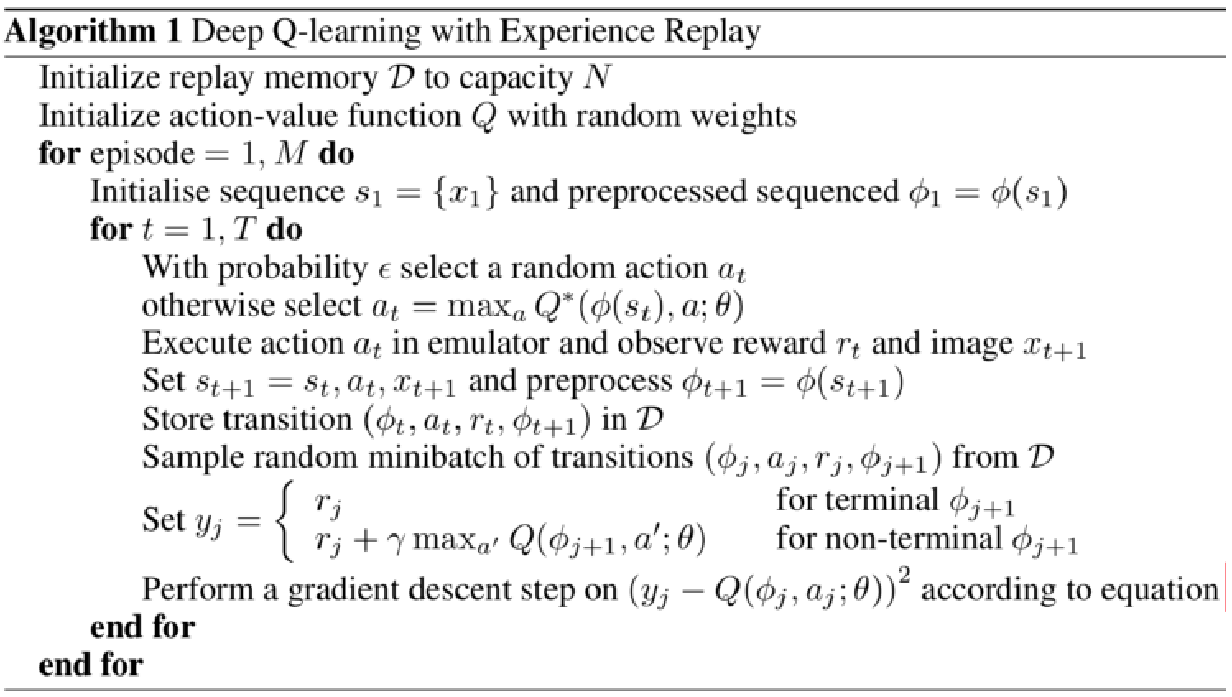
and Q-learning is just combining this iteration with last step's Q-value:

$$Q_{k+1}(s_t, a_t) = (1 - \alpha) Q_k(s_t, a_t) + \alpha \left(r_t + \gamma \max_{a'} Q_k(s_{t+1}, a')\right),$$

which can also be viewed as a gradient step from $Q_k(s_t, a_t)$ when minimizing $||Q - r_t - \gamma \max_{a'} Q_k(s_{t+1}, a')||^2$ and treat Q as the only variable to be minimized. In this way when we try to parametrize the Q-value function with θ we just need to add an additional gradient term $\nabla_\theta Q_\theta(s_t, a_t)$ and we get the update rule:

$$\begin{aligned} &\theta_{k+1} \leftarrow \theta_k + \alpha_k \nabla_{\theta_k} \ell_{\theta_k}(s, a, s'), \text{ where} \\ &\nabla_\theta \ell_{\theta_k}(s, a, s') = \delta \nabla_{\theta_k} Q_{\theta_k}(s, a) \\ &\delta = r + \gamma \max_{a'} Q_{\theta_k}(s', a') - Q_{\theta_k}(s, a). \end{aligned} \tag{2}$$

In deep Q-learning we just use a deep neural net to model the parametrize $Q_\theta(s, a)$. So the standard algorithm can be specifies as:



```
Algorithm 1 Deep Q-learning with Experience Replay
  Initialize replay memory $\mathcal{D}$ to capacity $N$
  Initialize action-value function $Q$ with random weights
  for episode = 1, $M$ do
      Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
      for $t = 1, T$ do
          With probability $\epsilon$ select a random action $a_t$
          otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
          Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
          Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
          Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
          Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
          Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
          Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation
      end for
  end for
```

Figure 3: Algorithm for deep Q-learning

Here we in each episode we initialize a LP and solve it by simplex method. In each simplex iteration we will choose the pivot using the Q-value function at hand and put the transition pair (s_t, a_t, r_t, s_{t+1}) into the pool. Then we also sample a batchsize (assumed 128 from the paper) to update the parameter θ using the gradient descent in (2). When the LP is solved or terminated we will start a new episode and initialize a new LP. update

Moreover the paper assumes update the parameter every 1/2 epoch, but we will update it in every simplex iteration, just following the above algorithm for deep Q-learning. And the ϵ for exploration in our trial is assumed decayed uniformly from 0.99 to 0.01, that is to say, we will decrease it by $(0.99 - 0.01)/50 = 0.0196$ for each episode and reach 0.01 at last.

6 Network details

We further specifies the deep network details here.

Network architecture: The network used here is quite simple, just 8 fully connected hidden layers and we just follow this architecture (since this is a quite new area, there's no much experience on how the network should be formulated). Each layer has a width of 128 and ReLU activation. The input as we mentioned is just the reduced costs and the objective value and the output is of the width 2, which is just the predicted Q-value when choosing Dantzig's rule or the steepest edge rule, respectively. A tanh activation is applied on the outputs. We just follow this architecture in the experiment.

Loss funtcion: As we have mentioned before we are approximating the Q-value function with a deep neural network, and the update follows from a gradient step when solving the least square problem. So we just set the loss as the mean square error between $Q(s_t, a_t)$ and $\max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) + R(s_t, a_t)$, where we need to set addtionally the Q-value at the next step $Q(s_{t+1}, a_{t+i})$ has attribute (require_grad = False) in pytorch. So the derivative will only multiply $\nabla_\theta Q_\theta(s_t, a_t)$ by the chain rule.

Other parameters: Other parameters just follow paper's setting. We set the learning rate be 0.0001, a batch size of 128 and we use the Adam as the optimizer. Moreover we will save the model every 200 episodes, test the model every 50 times and save the best model up to now. This testing process will only test for 10 instances each time.

7 Numerical results

In this section we will show the numerical experiment results. Note in the first two testings with simulated data we want to see what happened for large scale problems. So we set $P = 1024, L = 512, K = 50, p = 0.02$. The A matrix is of the size 1074×2049. And we didn't use GPU, but use 8 core CPU to do all the trainting and testing process. We just list them below (in fact I think I haven't tune the parameter right and the curve we get for reward is quite absurd, so I won't list them here):

7.1 Test for 50 instances

methods	mean cost (50)	mean iter	mean cost (10)
0	0.1844	44.36	0.3149
200	0.227	42.12	0.42
400	0.1874	44.18	0.3324
600	0.227	42.12	0.4161
800	0.1883	44.34	0.3256
1000	0.2346	40.72	0.584
Dantzig	0.0227	42.44	0.4161
Steep	0.1844	43.92	0.3149

Table 1: Here the methods uses Dantzig, Steepest edge and the RL model after 0,200,...,1000 times of iteration. Mean cost (50) is the mean objective value of 50 testing instances (each time we'll cut the simplex for 50 iterations) and mean cost (10) is the mean value of 10 instances. Mean iter is the average of iteration of for the 50 instances.

7.2 Test for a special case

We here test a special case where the Dantzig outperforms the steepest edge rule.

methods	iter	cputime per iter	cost
0	11	0.2296	0.6718
200	357	0.183	0.2138
400	11	0.2149	0.6718
600	357	0.2032	0.2138
800	11	0.2098	0.6718
1000	364	0.2068	0.2085
Dantzig	367	0.1832	0.2138
Steep	21	0.1992	0.6718

Table 2: Here the methods uses Dantzig, Steepest edge and the RL model after 0,200,...,1000 times of iteration. Iter is the total iter number for the method, in this case we didn't cut off the simplex iteration. The algorthms stops when the objective value is not improved for 10 successive iterations. cputime per iter is the cputime for the method in each iteration. Cost is just the final objective value.

7.3 Test on the real data

We here test our model on two real data set in JLF: JLF023 and JLF141. To test on these data we retrain the model with dimension $50, 206, 1, 0.1313$ and $147, 5423, 1, 0.0195627$ separately (after some transformation and delete some items with missing value). Note here we didn;t cut off the simplex iteration after some time and let it go. The algorthm will be assumed converge if the objective value is not improved for 10 succesive iteartions. The Bland's rule is also used for comparison. The result is shown as below:

methods	iter	$\lvert\lvert Ax-b\rvert\rvert_1/\lvert\lvert b\rvert\rvert_1$	cost
0	15	40.949835	0.031418
200	15	0.949835	0.031418
Dantzig	25	0.0002819	0.03989
Steep	17	5.4647e-5	0.03922
Bland	11	1.11923e-5	0.03989

Table 3: Here the methods uses Dantzig, Steepest edge, Bland and the RL model after 0,200,...,800 times of iteration. We find the results for 0,200...800 are almost the same for the above factors. So we just list two of them here. Iter is the total iteration number, note here we didn't cut off the simplex method. $||Ax - b||_1/||b||_1$ quatifies how far our final solution is to the feasible set. Cost is again the ultimate objective value.

methods	iter	$\lvert\lvert Ax-b\rvert\rvert_1/\lvert\lvert b\rvert\rvert_1$	cost
0	21	0.89474	0.2484
800	11	0.8947	0.7453
Dantzig	11	0.005044	0.7453
Steep	21	0.018913	0.2484
Bland	11	0.005044	0.7453

Table 4: Here all the factors are the same as the previous table. But here we find only the model after the 0th episode accomplished performs good and different. All the other models performs exactly the same so we just list one of them here.

8 Conclusions and analysis

From the above results we have seen our model always oscillates between the Dantzig's rule and the steepest rule, For the 6 models saved, some of them are like Dantzig and others are like the steepest edge. The plot of reward (although not listed here) has also shown such a oscillation pattern, with the period approximately 200. So maybe to derive a good model we need to turn to the model in the midst of them. From the experiment we can see model 200,600,1000 acts much like the Dantzig and the others are more like the steepest.

Moreover, in some experiments the model 0 performs better than the others and outperforms Dantzig and the steepest edge. This indicates maybe we don't need much iteration to get a good model. Note the model 0 only updates the parameter for 50 times. This may also indicates our model is overfitting. The author in the paper has add a L2 regularization term in the process and we didn't add it in our modelSo maybe a regularization term is needed for this method.

Furthermore, the cputime per iter shows although our model needs to calculate a Q-value and determine the action, the time consumed additionally in this process is not that much and will not impair the efficiency much.

Finally, although our model have not finely tuned, in some cases it can still improve the accuracy and avoid us getting cycling, like in the 2nd experiment. Although Bland is a good choice for doing this traditionally, the performance of Bland on large scale problem is awful (we didn't list it here because it nearly didn't come down). But our model can improve the accuracy while won't impair the efficiency much. So I think DeepSimplex might be a good choice for large scale LP with parameter finely tuned.