Vágásgenerálás az egészértékű programozásban gépi tanulással

Becsó Gergely

Önálló Projekt II. beszámoló

1. Bevezető

Az egészértékű programozási feladatokat számos alkalmazásban, modellezési helyzetben használunk. A feladat általánosságban NP-nehéz, így nem ismerünk rá polinomiális algoritmust. Minden olyan megoldás, amivel gyorsítást lehet elérni a jelenlegi megoldóprogramokhoz képest hasznos és jelentős előrelépés. A legjobb jelenleg ismert módszerek a Branch&Cut, amelyben három fajta döntési helyzet van: el kell dönteni, hogy a Branch fának mely ágát építsük tovább; szét kell választani a feladatot két részfeladatra, itt az a kérdés, hogy melyik változó mentén válasszuk szét a feladatot; végül pedig a vágósíkok hozzáadásánál dönteni kell, hogy az adott részfeladatokhoz melyik lehetséges vágósíkot adjuk hozzá.

Az első döntési helyzettel itt nem foglalkozunk. A második döntési helyzet mélytanulással való megtámogatását kutatja az IPDL kutatócsoport Gasse és társai munkássága [Gas+19] nyomán. Az önálló projektem keretében a harmadik döntési helyzettel foglalkozom Yunhao Tang és társai eredményeiből [TAF19] valamint a Columbiai Egyetem IEOR4574 kurzusa anyagaiból indulva.

2. Megerősítéses tanulás

A vágásgenerálás gépi tanulási támogatásához megerősítéses tanulást fogunk használni. A megerősítéses tanulás általános formájában a feladatot valamilyen formális játékként fogalmazzuk meg, amelyben lépéseket kell tenni, és valamilyen célfüggvény szerint minnél jobb eredményt elérni.

A leíráshoz legyen adott S állapottér, D akciótér[1], T átmenetfüggvény, R jutalomfüggvény és egy $\pi : S \to D$ *policy*-nak nevezett leképezés. Egy trajektóriának hívjuk a $\tau = (s_i, d_i, r_i)$ sorozatot, melyet egy játék egy lefutásakor kapunk. A célunk, hogy a $R(\tau) = \sum r_i$ értéket maximalizáljuk π policy függvényében.

A π függvény rendszeritnt egy számos paraméterrel bíró függvény, esetünkben egy neurális háló, amely súlyainak változtatásával szeretnénk maximalizálni. A π súlyaira innentől θ-ként fogok hivatkozni. Egy π_θ policy indukál egy eloszlást a lehetséges trajektóriák felett. A későbbiekben $\mathbf{E}_\pi[\cdot]$ alatt ezen eloszlás szerinti várható értéket értjük, erre az eloszlásra pedig $p(\tau, \theta)$-ként hivatkozunk.

A *policy gradient* módszernél, sok más reinforcement learning módszerhez hasonlóan szükséges további segédfüggvények alkalmazása a feladat kezeléséhez. Itt ez a $Q^\pi : S \times D \to \mathbb{R}$ függvény, mely a várható értéke az s állapotból d döntéssel induló trajektória összjutalmának,

[1] Az irodalomban az elterjedt jelölés az akciótérre az "A", azonban itt összekeveredne a következő fejezet jelölésrendszerével.

azaz

$$Q^{\pi}(s,d) = \max_{\theta} \mathbb{E}_{\pi}\left[R(\tau)|s_0 = s, d_0 = d\right] = \max \mathbb{E}_{\pi}\left[\sum_{i=0}^{\infty} r_i \gamma^i\right].$$

Legyen $J(\theta) = \mathbb{E}_{\theta}[R(\tau)]$. Szeretnénk, hogy ez a célfüggvény maximális legyen. Ehhez gradiens emelkedés módszert alkalmazzuk. A gradienst egyrészt a következő módon tudjuk számolni:

$$\bigtriangledown_{\theta} J(\theta) = \mathbb{E}_{\pi}\left[r(\tau) \bigtriangledown_{\theta} \log p(\tau, \theta)\right].$$

Másrészt a Columbiai egyetem tananyaga ettől eltér:

$$\bigtriangledown_{\theta} J(\theta) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} Q(s_t, d_t) \bigtriangledown_{\pi} log_{\pi}(s_t, d_t)\right].$$

Mindkét esetben Monte Carlo módszerrel tudjuk közelíteni a gradienst.

3. Alkalmazás vágásgenerálásra

A vágások meghatározásának a feladatát a megerősítéses tanulás sémája szerint felírjuk egy egyszemélyes játékként állapotok, akciók és egy jutalomfüggvény segítségével.

- Az állapotok az IP feladatok leírásai: $Ax \leq b$, $\min(cx)$,
- az akciók a lehetséges Gomory vágások: $(-A_i^* + \lfloor A_i^* \rfloor)^T x \leq -b_i^* + \lfloor b_i^* \rfloor$ alakban,
- a jutalomfüggvény pedig $R(t+1) = cx_{LP(t)}^* - cx_{LP(t+1)}^*$, ahol $x_{LP(t)}^*$ a t-edik LP feladat optimális megoldása.

Ezt a játékot fogja játszani, és gyakorlással megtanulni egy ügynök, amely az esetünkben egy neurális hálózat lesz. A játék folyamán csökkenti az egészértékűségi rést, és minél eredményesebben csinálja az egészértékűségi rés csökkentését, annál gyorsabb algoritmust kapunk.

Elkerülendő, hogy minden feladatméretre külön hálót kelljen építeni, tehát csak olyan architektúrák jöhetnek szóba, amelyek tudnak változó méretű feladaton is dolgozni. Ezért a hálózatot érdemes a sorozatokat feldolgozni képes LSTM és attention rétegekre alapozni.

3.1. Módosított Gomory-vágások

Klasszikus esetben a Gomory-vágásokat $Ax = b, \min(cx)$ alakú feladatokra írjuk fel. A következő módon ezt általánosíthatjuk

$$\begin{aligned} Ax &\leq b \\ x &\geq 0 \\ \min(cx) & \end{aligned}$$

alakú feladatokra, ahol $A \in \mathbb{Q}^{m \times n}$, $b \in \mathbb{Q}^m$, $x, c \in \mathbb{Q}^n$. Közismert, hogy az előzővel ekivalens a

$$Ax + \mathbb{I}s = b \tag{1}$$
$$x, s \geq 0 \tag{2}$$
$$\min(cx) \tag{3}$$

feladat, ahol $\mathbb{I} \in \mathbb{Q}^{m \times m}$ egységmátrix és $s \in \mathbb{Q}^m$. Erre a formára már felírható a klasszikus szimplex tábla. A szimplex módszer leállásakor kapott tábla törtváltozókhoz tartozó soraiból kaphatjuk a Gomory-vágásokat a következő formában:

$$e^T x + r^T s \leq d,$$

ahol $e, x \in \mathbb{R}^n$, $r, s \in \mathbb{R}^m$ és $d \in \mathbb{R}$. A slackváltozóktól szeretnénk megszabadulni, ezért (1)-et beszorozzuk r-el,

$$r^T Ax + r^T s = r^T b,$$

majd ezt az egyenletet levonjuk a vágósíkból:

$$(e^T - r^T A)x \leq d - r^T b.$$

Ez az egyenlőtlenség ekvivalens az eredeti vágósíkkal, viszont nem használja a slack változókat.

3.2. Architektúra

Az attention réteg jól boldogul a változó számú bemenet-kimenettel, ha az egyes bemenő vektorok azonos méretűek. Az azonban nem igaz, hogy minden IP feladatban konstans számú változó szerepel, így egy LSTM hálózat segítségével elkódoljuk 64 dimenziós vektorokba az adott feltételeket, egy másik, azonos dimenziójú LSTM háló segítségével pedig elkódoljuk a lehetséges vágásokat szintén 64 dimenziós vektorokba. Az így kapott vektorokat átengedjük egy sűrű rétegen, ebből kapjuk a feltételek és a lehetséges vágások végső beágyazásait, melyeket F_i, V_i jelöl.

Végül az alábbi módon alkalmazunk egy attention jellegű réteget, amely egy P eloszlást eredményez a vágásokon.

$$P = \text{soft}\max(\sum_{sorok} VF^T)$$

A tanulás folyamán úgy generáljuk a trajektóriákat, hogy bizonyos eséllyel felfedezünk (*explore*), ekkor P szerint vételezünk a lépések közül, bizonyos eséllyel pedig S mint pontszám által a legígéretesebb lépést tesszük meg (ezt hívjuk *exploit*-nak). Az inferencia folyamán mindig *exploit*-álunk.

4. Eredmények

Az előző félévben ígéretes pozícióban jelentettem a munkáról. Sikeresen lefutottak az első tesztek a kódon, melyet a Columbiai kurzus példaprojektjeként találtam. Látszólag minden készen állt a továbblépésre. A félév elején azzal foglalkoztam, hogy a Gurobi megoldóprogramot lecseréljem a SCIP megoldóprogramra. Bár a Gurobi felülete sok szempontból kényelmesebb, viszont az egy ipari megoldóprogram, zárt forráskóddal. Így egy nagyobb projekt keretében a használata életszerűtlen, ugyanis további, eddig nem implementált funkciókat kellett beépíteni a programba. A SCIP ezzel szemben egy nyílt forráskódú, sokoldalúan hajlítható akadémiai program, melyet tetszőlegesen lehet módosítani. Az átállás több nehézséggel is járt, a SCIP hiányos dokumentációjával dolgozni komoly programozási kihívás volt.

Amikor a SCIP-re való átállás sikerrel megtörtént, az eddigi mérőszámaimat hiányosnak találtam. Eddig mindössze azt logoltam, hogy abszolútértékben mennyit sikerült csökkenteni az

egészértékűségi résen. Ez nem volt elégséges, ugyanis feladatonként nagyon eltér az egészértékűségi rés: 0,5 javítás nagyon más eredmény 0,65 és 3 értékű kezdeti rés mellett. Az új mérőszám a százalékos javítás lett, amely értelemszerűen a százalékos arányt jelenti. Emellett elkezdtem figyelni, hogy melyik lépésnél melyik indexű döntést hoztuk, illetve ez exploration vagy exploitation volt-e.

Komoly meglepetésemre tapasztaltam, hogy az új információk problémákra utalnak: rendszeresen a maximális rés többszörösét javította a program, amely így hibás működést mutatott. A kód tüzetes átvizsgálása során két jelentős elvi hibát fedeztem fel, amelyek már az eredeti kódbázis szerinti, Gomory-ra támaszkodó implementációban is jelen voltak. Az első a vágások kiszámításánál adódott, a második a háló súlyainak frissítésénél.

4.1. A vágás-számolás problémája

A Gomory-vágások számolásához elkészítjük a szimplex táblát. Ezt olyan formában adjuk vissza, hogy $[x^T|\tilde{\mathbf{A}}]$, ahol $\tilde{\mathbf{A}} = \mathbf{B}^{-1}[\mathbf{AI}]$. Az örökölt implementációban emellett egy olyan első sort is beszúrtak, hogy $[-c(x^*)|RC]$, ahol x^* az optimumhely, RC pedig a változók redukált költségfüggvényeiből alkotott vektor. Ez a szimplex táblának természetes része, véletlenül benne maradt. Ezután ezen kiegészített tábla sorait használták a vágások elkészítéséhez. A fent említett 100% feletti javítások mindig olyankor jelentkeztek amikor a kiegészítő sorból generált vágást választottuk. Ez mutatja a hibát, hiszen ez azt jelenti, hogy a poliéder egy egész pontját is levágtuk. Ugyanakkor a hatalmas javításokat a célfüggvény nagyon jó döntésnek detektálja, így az algoritmus azt tanulja meg, hogy mindig ezt a kiegészítő "vágást" kell választani.

Ezt a hibát orvosoltam.

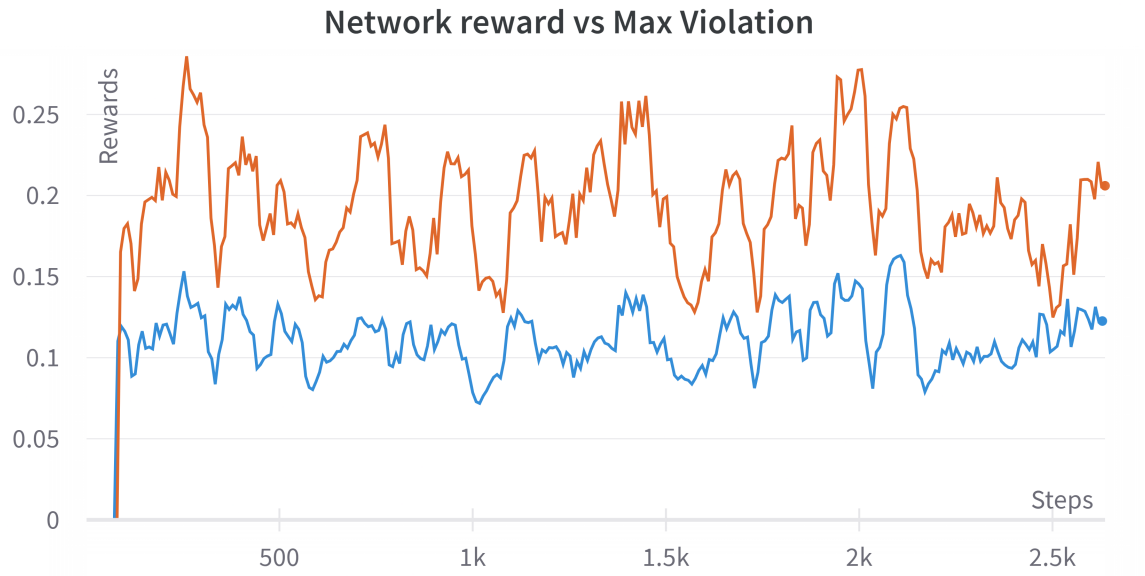
4.2. A súly-frissítés problémája

A következő elvi hibát a háló szúlyainak frissítésénél találtam. Mint feljebb láttuk, a $\sum r_i \gamma^i$ célfüggvényt szeretnénk maximalizálni a tanulás során, ehhez gradienseket számolunk. Azt is láttuk, hogy a gradiens kiszámításához szükség van a π_θ eloszlásra. A kódban eredetileg az szerepelt, hogy végiglépkedtünk a trajektória döntésein, az ott látott vágások alapján legeneráltuk az eloszlást, kiszámoltuk a gradienst és frissítettük a θ súlyokat.

A probléma ott adódott, hogy az első lépés után a frissített háló generálta az eloszlást, mely alapján a későbbi gradiensek alapjaiban véve voltak hibásak. Ezt javítottam: először a trajektória összes elemére eltárolódik gradienst és csak ezután frissülnek a háló súlyai.

4.3. Mérések

A javított hálóval méréseket végeztem. Alább látható háromszáz lejátszott játék után a háló teljesítménye. Összehasonlítási alapnak a *legnagyobb sértés* heurisztika szolgál. Itt legyen x^* az optimális megoldása az *LP* feladatnak, ekkor $x_i^* - \lfloor x_i^* \rfloor$ legnagyobb értékű koordinátáját választjuk.



Az ábrán a kék vonallal a háló teljesítménye látható, összehasonlításul sárga vonallal a *max sértés* heurisztika szolgál. Az ábra jól mutatja, hogy bár irreális javítások nincsenek, a tanulás nem kifejezetten eredményes.

5. További lehetőségek

A kód szerzői a közismert, levezetett gradienstől eltértek. Lehetséges, hogy ez okozza a tanulási nehézségeket, így ennek a lecserélése mindenképp kipróbálandó. A *policy gradient* módszer egy viszonylag kezdetleges módszer, mely zajossága szintén okozhatja az érdemi tanulás hiányát. Az Actor Critic, TRPO és PPO mind olyan algoritmusok, melyek igyekeznek ezen problémát orvosolni, így természetes következő lépést jelentenek a projektben. Mindezek után érdekes és hasznos lenne akár gyökeresen más háló-architektúrákkal is kísérletezni, illetve kipróbálni a ritka mátrixokon való tanulást.

Bár egyenlőre nem született hasznosítható eredmény, komoly előrelépések történtek egy jól működő heurisztika irányába.

Hivatkozások

[Gas+19] Maxime Gasse és tsai. *Exact Combinatorial Optimization with Graph Convolutional Neural Networks*. 2019. DOI: 10.48550/ARXIV.1906.01629. URL: https://arxiv.org/abs/1906.01629.

[TAF19] Yunhao Tang, Shipra Agrawal és Yuri Faenza. *Reinforcement Learning for Integer Programming: Learning to Cut*. 2019. DOI: 10.48550/ARXIV.1906.04859. URL: https://arxiv.org/abs/1906.04859.